Cours de programmation fonctionnelle et logique
Cours 3 : programmation fonctionnelle *versus* impérative

Table des matières

1 Principes des deux paradigmes

1.1 Principe de la programmation impérative

Modifications successives de l'état de la machine par des commandes, des effets.
À chaque étape de l'écriture d'un algorithme « impératif », il faut :

- déterminer quelles valeurs sont nécessaires au calcul ;
- associer des emplacements mémoires aux données ;
- décrire les transformations à appliquer à la mémoire.

In fine

- l'état final **contient** le résultat de l'algorithme.

1.2 Principe de la programmation fonctionnelle (pure)

- Évaluation d'expressions (par application de fonctions).
- Le résultat d'une évaluation n'est pas (explicitement) écrit en mémoire : il est argument d'une autre expression ou fonction.
- Un programme fonctionnel spécifie **ce qui** doit être calculé, et non pas **comment** le calculer.
- La mémoire est exclusivement géré par l'interpréteur ou le compilateur, et non par le programmeur.

1.3 Exemple : la factorielle

Définition mathématique

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

Version impérative

```
#let fact_imp n =
   let a =ref(1) in
     for i=2 to n
       do a := i * !a done;
     !a;;
         val fact_imp : int -> int = <fun>
```

Version fonctionnelle

```
#let rec fact_rec = function
   0 -> 1
 | n -> n * (fact_rec (n-1));;
     val fact_rec : int -> int = <fun>
```

Version impérative récursive

```
#let rec fact_imp n =
   let a =ref(0)
   in match n with
        0 -> a := 1
      | _ -> let b = ref(fact_imp (n-1))
             in a := n;
   print_int !a;;
             val fact_imp : int -> unit = <fun>
```

2 Propriétés des langages fonctionnels

- **Inférence de type automatique** :
  - garantie la correction du programme (typage fort) ;
  - pas de typage obligatoire par le programmeur mais inférence du type le plus général possible et **polymorphisme**. Le polymorphisme est la faculté d'accepter des arguments de types différents.

    ```
    #let fonction_un x = 1;;
     val fonction_un : 'a -> int = <fun>
    ```

- Les **fonctions** sont des **individus de première classe** : pas de différence fondamentale entre un entier et une fonction, les deux objets se manipulent de la même manière. Les fonctions peuvent être : membre de structures de données ; passées comme argument à une fonction ; retournées comme résultat d'une fonction.

  Dans les deux derniers cas, on parle de **fonctions d'ordre supérieur** ou de **fonctionnelles**.

  ```
  #let identite x = x;;
   val identite : 'a -> 'a = <fun>
  ```

  ```
  #(identite identite) 3;;
   - : int = 3
  ```

  Maximum sur une liste :

  ```
  #let rec max_sur_liste = function
      [] -> failwith "liste vide"
    | [a] -> a
    | t::r -> max t (max_sur_liste r);;
         val max_sur_liste : 'a list -> 'a = <fun>
  ```

```
# max_sur_liste [4;7;3];;
 - : int = 7
```
Minimum sur une liste :
```
# let rec min_sur_liste = function
    [] -> failwith "liste vide"
  | [a] -> a
  | t::r -> min t (min_sur_liste r);;
      val min_sur_liste : 'a list -> 'a = <fun>
```
```
# max_sur_liste [4;7;3];;
 - : int = 7
```
Fonctionnelle générique :
```
# let rec opt_sur_liste f = function
    [] -> failwith "liste vide"
  | [a] -> a
  | t::r -> f t (opt_sur_liste f r);;
      val opt_sur_liste : ('a -> 'a -> 'a) -> 'a list -> 'a = <fun>
```
```
# opt_sur_liste max [4;7;3];;
 - : int = 7
```
```
# opt_sur_liste min [4;7;3];;
 - : int = 3
```
```
# opt_sur_liste (function x -> function y -> x+y) [4;7;3];;
 - : int = 14
```
- **Récursivité**.
- Définition par **filtrage**. *switch* élaboré.
- **Évaluation paresseuse** (parfois) : un argument n'est évalué que quand c'est nécessaire. Un argument peut donc n'être jamais évalué.
- **Transparence référentielle** si pas d'effets de bords, si une variable n'est définie qu'une seule fois (pas d'affectations). Autrement dit, « des égaux peuvent être remplacés pas des égaux ».

  `(f a) + (f a)` $\Leftrightarrow$ `let x = f a in x + x`

  Avec la fonction suivante, il n' y a pas équivalence :
  ```
  # let f x = print_int x; x;;
   val f : int -> int = <fun>
  ```
  Il est alors possible de raisonner sur les programmes, de les analyser, de prouver des propriétés et de les garantir.

3 Notion de variable et de définition

3.1 En mathématiques

Variable = inconnue. On obtient la valeur de la variable par la résolution d'une équation. La définition est donc implicite.

$$\left\{\begin{array}{rcl} x+4 & = & x-2 \\ 2x-3 & = & 1 \end{array}\right. \quad \Rightarrow \quad x=2$$

$$\left\{\begin{array}{rcl} x+2 & = & 4x-1 \\ 2x-3 & = & 1 \end{array}\right. \quad \Leftrightarrow \quad \left\{\begin{array}{rcl} x & = & (2+1)/3 \\ x & = & (1+3)/2 \end{array}\right. \quad \Rightarrow \quad \text{pas de solution.}$$

En informatique, les « équations » ci-dessous ont un sens.

$$\begin{array}{rcl} x & = & (2+1)/3; \\ x & = & (1+3)/2 \end{array}$$

3.2 En informatique

Variable = emplacement mémoire.

Définition = affectation = écriture d'une valeur en mémoire.

Dans un langage fonctionnel, une variable n'est « affectée » qu'une seule fois. Une nouvelle définition ne change pas la valeur de la variable, mais crée une nouvelle variable de même nom.

Premier exemple :

```
#let f x = x + 1;;

#let g = f;;

#let f x = x + 2;;

#g 0;;
 - : int = 1
```

Deuxième exemple :

```
#type entier = Integer of int;;
 type entier = | Integer of int

#let a = Integer 3;;
 val a : entier = Integer 3

#type entier = Integer of int;;
 type entier = | Integer of int

#let f = function Integer x -> x;;
 val f : entier -> int = <fun>

#f a;;
 Characters 2-3:
This expression has type entier but is here used with type entier
```

On a donc bien cohabitation de deux définitions *différentes* du type `entier`.