Chapitre 0

Introduction à la programmation fonctionnelle

Option informatique MPSI1 & MPSI2

Dans ce chapitre, on présente les premières caractéristiques de la programmation fonctionnelle. On se restreindra aux manipulations d'entiers.

I Introduction

Voici la définition de l'informatique proposée par la Société Informatique de France (SIF).

Définition 1 - Informatique

L'informatique est la science et la technique de la représentation de l'information d'origine artificielle ou naturelle, ainsi que des processus algorithmiques de collecte, stockage, analyse, transformation et communication de cette information, exprimés dans des langages formels ou des langues naturelles et effectués par des machines ou des êtres humains, seuls ou collectivement.

On peut noter que le premier thème mis en évidence est la représentation de l'information.
L'étude de Python proposée en informatique commune est centrée sur une représentation des données par les listes. Cette structure de donnée est très riche et permet de traiter de nombreux problèmes. Python permet, de plus, d'utiliser des structures de données plus finement adaptée à un problème, en particulier à l'aide de la programmation par objets.
La programmation en Python est la continuation des langages développés à partir de la structure des ordinateur : on utilise un ensemble de fonctions et procédures qui modifient un environnement composés de variables dont les valeurs sont modifiées.
Cette notion d'environnement de travail est la source de l'extraordinaire souplesse de ces langages mais induit des limitations dans le cas de programmes complexes. En particulier il est difficile d'étudier formellement les programmes : leur richesse rend ardue leur traduction dans une forme logique rigoureuse.

Un autre type de programmation

Dès la fin des années 50 d'autres types de langages ont été proposés.
Ils sont structurés autours de concepts indépendants des ordinateurs ; pour les langages qui nous intéressent ils s'appuient sur une formulation de la logique imaginée vers 1930, le **lambda-calcul**.
On peut citer Lisp (1958), ML (1974) dont Caml est une évolution, Miranda, `Haskell` ...
Le principe de ces langages fonctionnels est de considérer la programmation comme une suite de fonctions appliquées à des éléments persistants (dont la valeur ne change pas) qui ont un **type** déterminé.
Cette méthode de programmation s'est avérée très efficace pour développer des programmes surs : les langages modernes contiennent souvent une partie fonctionnelle.
C'est cette méthode de programmation que nous allons étudier avec le langage OCaml.

On verra que OCaml n'est pas seulement un langage fonctionnel ; il contient toutes les fonctions impératives classiques. On peut donc l'utiliser avec la souplesse de ces langages classiques mais on perd alors la rigueur des langages fonctionnels. En particulier, on peut écrire beaucoup de programmes comme en Python sans d'autres modifications que les différences de syntaxe.
Nous allons, dans ce premier chapitre, nous restreindre aux manipulations d'entiers par des fonctions, sous forme fonctionnelle et sans utiliser de boucle.

II Entiers

Le type de données le plus utilisé sera le type entier : `int`.

- Les calculs seront exacts, contrairement aux flottants, à condition de rester dans les limites des nombres exprimables dans la machines : `min_int <= n <= max_int`

```
# max_int;;
- : int = 4611686018427387903
# max_int + 1;;
- : int = -4611686018427387904
```

- Les opérations sont les opérations classiques : +, -, *, /, `mod`
- / désigne la division entière : 7/2 renvoie 3
- `mod` est le reste de la division ; `7 mod 2` renvoie 1.
- Il n'y a pas de fonction puissance.
- On imprime une valeur entière avec `print_int`.

```
# print_int 4;;
4- : unit = ()
```

  `unit` désigne le type du résultat, ici l'absence de résultat donne un résultat vide, `()`.

La syntaxe des définitions ressemble à l'usage mathématique : *soit* $x = 2 + 3$.

```
# let a = 1;;
a : int = 1
```

- La valeur de la variable est donnée au moment de sa création.
- Le type de la variable n'est pas déclaré, c'est le logiciel qui le définit.
- La variable n'est pas très variable : `a` sera toujours associé à 1 sauf si on définit une nouvelle variable avec le même nom ; dans ce cas l'ancienne variable est perdue.

On peut définir une variable pour un usage local : elle n'existe plus en dehors de son domaine d'application.

```
# let x = 2 in x+1;;
- : int = 3
# x;;
Error: Unbound value x
```

Il sera souvent profitable de lire les messages d'erreurs qui peuvent être explicatifs et pertinents. Ici le programme nous dit que `x` n'est pas lié à une valeur.
Si une variable globale existe avec le même nom elle est oubliée dans le domaine d'application de la variable locale puis reprend son existence.

```
# let u = 3;;
val u : int = 3
# let u = 4 in u+1;;
- : int = 5
# u;;
- : int = 3
```

III Fonctions

Un fonction est définie par `fun` :

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

Ce n'est pas très pratique car la fonction est oubliée tout de suite. On peut néanmoins l'appliquer à une variable.

```
# (fun x -> x + 1) 3;;
- : int = 4
```

On notera que les parenthèses sont facultatives.
On souhaitera, le plus souvent donner un nom à la fonction pour pouvoir l'utiliser ensuite.
On reprend alors la syntaxe de la définition d'une variable.

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

#f 5;;
- : int = 6
```

À retenir - Simplification

On peut simplifier la définition sans utiliser le mot `fun`

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

OCaml reconnaît qu'il s'agit bien d'une fonction.
On écrira le plus souvent avec un passage à la ligne

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

Une fonction peut renvoyer plusieurs résultats, il seront séparés par une virgule.

```
# let f x =
    x + 1, x - 1;;
f : int -> int * int = <fun>
# f 7;;
- : int * int = (8, 6)
```

Localité des variables

On peut utiliser une variable définie globalement dans une fonction. On peut toujours utiliser des variables locales

```
# let f x =
      let a = 2 in
      x + a;;
f : int -> int = <fun>

# f 4;;
- : int = 6
```

On peut aussi utiliser des variables globales,

```
# let a = 2;;
# let f x =  x + a;;
f : int -> int = <fun>

# f 4;;
- : int = 6
```

ou bien les masquer.

```
# let a = 2;;
# let f x =
      let a = 3 in
      x + a;;
f : int -> int = <fun>

# f 4;;
- : int = 7
```

Fonctions à plusieurs variables

En mathématiques une fonction n'a qu'une seule variable, celle-ci peut être décomposée en un élément d'un espace produit : soit f définie de $\mathbb{R}^2$ vers $\mathbb{R}$ par $f(x, y) = x + y$. f admet comme variable un couple de réels.
OCaml a le même comportement.

```
# let f (x, y) =
      x + y;;
f : int * int -> int = <fun>

# f (7, -3);;
- : int = 4

# let c = (2, 3);;
# f c;;
- : int = 5
```

Cependant il existe d'autres cas. Par exemple on peut imaginer une fonction qui dépend d'un paramètre, c'est le cas du logarithme en base a : $\log_a(x) = \dfrac{\ln(x)}{\ln(a)}$ pour $x > 0$ et $a > 1$. Ici $\log_a$ est une fonction de $]0; +\infty[$ vers $\mathbb{R}$ mais c'est aussi la valeur en a de la fonction Λ telle que $\Lambda(a) = \log_a$ qui va de $]0; +\infty[$ vers $\mathcal{F}\big(]0; +\infty[, \mathbb{R}\big)$.

On peut aussi écrire la somme sous la forme

```
# let somme = fun x -> (fun y -> x + y);;
somme : int -> int -> int = <fun>
# somme 2 3
- : int = 5
```

C'est une forme **curryfiée**.
Ici il faut lire la suite des flèches sous la forme

```
somme : int -> (int -> int) = <fun>
```

Le sucre syntaxique permet d'éliminer plusieurs fois `fun` :

```
# let somme x y  =
     x + y;;
somme : int -> int -> int = <fun>
# somme 2 3
- : int = 5
```

On peut alors utiliser la fonction partiellement

```
# let ajouter2 = somme 2;;
f : int -> int -> int = <fun>
# ajouter2 5
- : int = 7
```

Branchement conditionnel

On peut différencier le calcul à faire selon un critère avec un branchement conditionnel **`if p then e else e'`**
- `p` doit être une expression de résultat booléen -

```
# let comparer a b =
     if a > b
     then 1
     else if a = b
          then 0
          else -1;;
```

`e` et `e'` doivent produire des résultats de même type.

```
# let f x =
     if x > 0
     then x, x
     else x;;
          ^
Error: This expression has type int but an expression was expected
    of type
       int * int
```

IV Récursivité

Définition 2 - Fonction récursive

Une fonction est récursive si, dans sa définition, elle fait référence à elle-même.
En OCaml une fonction récursive sera déclarée par le préfixe `rec`.

IV.1 Exemples

- On peut définir la factorielle par $0! = 1$ et $n! = n.(n-1)!$ pour $n \geqslant 1$.

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

# fact 10
- : int = 3628800
```

- L'exponentiation est définie aussi par récurrence : $a^0 = 1$ et $a^n = a.a^{n-1}$ pour $n \geqslant 1$.

```
let rec puissance a n =
   if n = 0
   then 1
   else a * (puissance a (n-1));;

# puissance 2 32
- : int = 4294967296
```

- Le PGCD de deux entiers positifs a et b est a si $b = 0$, sinon il vaut la PGCD de b et r où r est le reste de la division de a par b

```
let rec pgcd a b =
   if b = 0
   then a
   else pgcd b (a mod b);;

pgcd 60466176 4782969;;
- : int = 59049
```

Tours de Hanoï

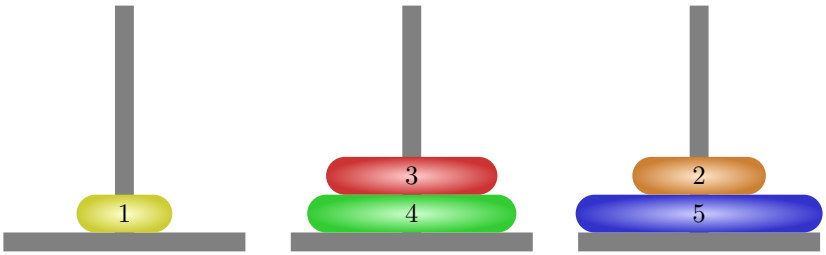
Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas consistant à déplacer des disques de diamètres différents d'une tour de départ à une tour d'arrivée en utilisant une tour intermédiaire tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Au départ les disques sont placés en respectant la seconde règle sur la tour de départ.
Le point de départ est

On veut arriver à

Une position intermédiaire possible serait



La résolution est en fait simple à énoncer.
Pour déplacer n disque de la tour A la tour B il suffit

- de ne rien faire si $n = 0$
- de déplacer les $n - 1$ disques supérieurs de la tour A vers la tour C,
  puis de déplacer le disque restant vers la tour B
  puis enfin de déplacer les $n - 1$ disques supérieurs de la tour C vers la tour B.

On voit encore apparaître un algorithme récursif.

Résolution des tours de Hanoï

```
let rec hanoi n tour1 tour2 tour3
    if n <> 0
    then begin hanoi (n-1) tour1 tour3 tour2;
            print_string (
                        "Déplacer le disque supérieur de "
                       ^tour1^" vers "^tour2);
         hanoi (n-1) tour3 tour2 tour1 end ;;
```

`tour1`, `tour2` et `tour3` sont les noms donnés au 3 tours ; le premier est le nom de la tour de départ, le deuxième celui de la tour d'arrivée et le troisième nom est celui de la tour en plus
`hanoi 4 "A""B" "C"` donne alors un mode d'emploi.

```
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de B vers A
Déplacer le disque supérieur de B vers C
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de C vers A
Déplacer le disque supérieur de B vers A
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
```

On voit ici le caractère un peu magique de la récursivité : on dit très simplement les choses et le programme produit un résultat compliqué.

IV.2 Propriétés de la récursivité

Terminaison

Dans chacun des 3 exemples il y a eu un test pour lequel dans un des cas la fonction récursive ne s'appelait pas elle-même, on parle de **cas terminal**.
Dans le cas des tours de Hanoï, ce cas terminal consistait à ne rien faire.
Cette condition est indispensable car il faut que la suite des appels à la fonction stoppe.

Définition 3 - Cas d'arrêt

Toute fonction récursive doit avoir un (ou des) **cas d'arrêt** : c'est un cas particulier des variables dont le traitement ne fait pas appel à la fonction récursive.

Un des problèmes que nous rencontrerons avec les fonctions récursives sera de s'assurer que toute variable initiale aboutit à un cas d'arrêt après un nombre fini d'appels récursifs.
C'est le problème de la **terminaison** semblable à celui rencontré dans les boucles `while`.
Il est très facile d'écrire des programmes récursifs qui ne terminent pas.

Tracage

On peut suivre les appels récursifs grâce à la directive `trace` :

```
trace "fact";;
fact 3;;

fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
- : int = 24
```

Récursivité et récurrence

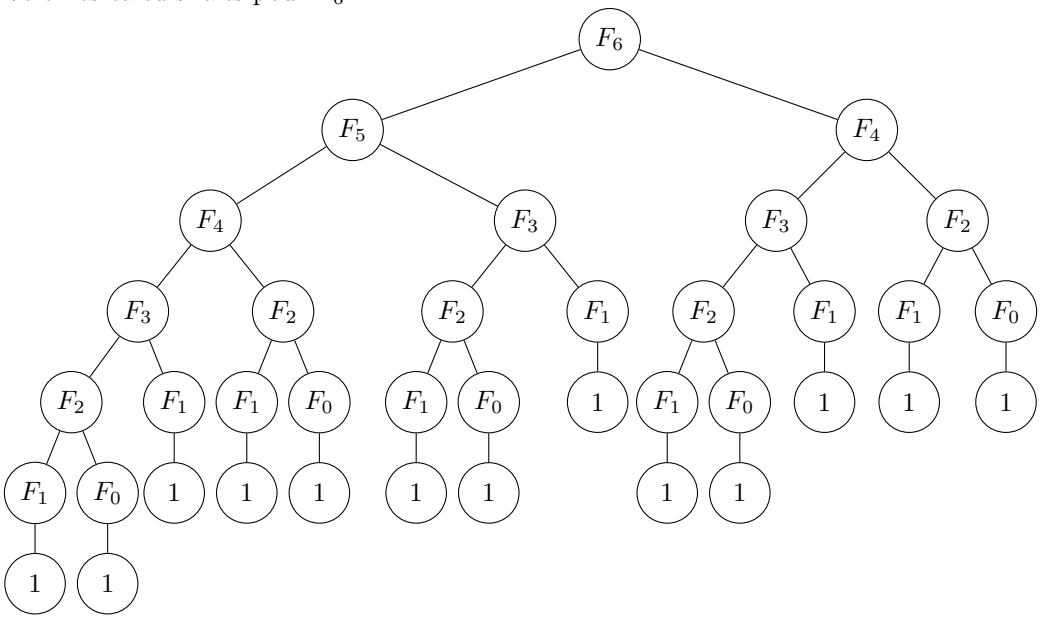
- La récursivité permet parfois d'écrire plus rapidement les algorithmes quand on veut traduire des définitions récurrentes.
- Les programmes récursifs sont souvent plus courts car on délègue à l'ordinateur la tâche d'écrire les boucles (on diminue aussi le nombre de variables référencées).
- La méthode naturelle pour prouver la validité d'une fonction récursive est une démonstration par récurrence.

Explosion des calculs

L'écriture directe de fonctions récursives peut aboutir à des temps de calcul prohibitifs.
Par exemple la suite de Fibonacci définie par $F_0 = F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour $n \in \mathbb{N}$ peut être implémentée sous la forme

```
let rec fibo n =
  if n <= 1
  then 1
  else fibo (n-1) + fibo (n-2);;
```

Le calcul de F_{43} demande plus d'une minute.
Voici les calculs faits pour F_6.



On remarque que le programme fait le même calcul un grand nombre de fois.

Stack overflow

Le principe de la récursivité est de mettre les calculs en attente dans une pile.
Les piles ont une taille limitée et il se peut qu'elle arrive à saturation (stack overflow).

```
let rec add n =
  if n = 0 then 1
           else n + add(n-1);;
add 200000;;
#add : int -> int = <fun>
#Uncaught exception: Out_of_memory
```

Pour éviter cela on peut écrire les fonctions sous forme récursive **terminale** : un algorithme récursif est terminal si l'appel récursif n'apparaît pas comme un argument d'une autre fonction ni comme opérande d'un opérateur.
Les langages évolués reconnaissent alors qu'ils n'ont pas besoin d'empiler les appels.
Cependant un algorithme récursif terminal sera en général moins facile à écrire car il demandera souvent d'ajouter une variable qui servira de résultat partiel ou de compteur.

```
let add_term n =
  let rec aux_add n accu =
    if n = 0  then accu
              else aux_add (n-1) (accu+n) in
  aux_add n 0;;
add_term 200000;;
#- : int = 20000100001
```