Paradigmes de programmation

Il existe de nombreux langages de programmation et des façons différentes de programmer (paradigmes)

En première et Terminale NSI nous avons vu trois façons différentes de programmer :

1. La **programmation impérative** : Un programme est une séquence d'instructions modifiant les états de la mémoire de l'ordinateur (effet de bord) avec le langage Python
2. La **programmation objet** avec le langage Python
3. La **programmation déclarative** avec le langage SQL

Nous ne revenons pas sur les paradigmes précédents, et nous allons plutôt découvrir un nouveau paradigme, la programmation **fonctionnelle** à travers des exemples issus du langage OCaml (Objective Categorical Abstract Machine Language) qui est multi-paradigme avec quand même une forte "coloration" fonctionnelle

1 Programmation Fonctionnelle

Les fonctions forment les constituants élémentaires des programmes en OCaml.

Un programme n'est rien d'autre qu'une collection de définitions de fonctions, suivie d'un appel à la fonction qui déclenche le calcul voulu.

Pour pouvoir essayer les exemples en cours et en TP nous irons sur le site `https://try.ocamlpro.com`

1.1 Données immuables

Une des idées de la programmation fonctionnelle est que chaque donnée est définie une fois pour toutes, et

on ne peut pas la modifier on dit qu'elle est immuable

En Python par exemple on peut définir une variable `x` puis affecter à cette variable une valeur 2 par exemple puis faire l'affectation suivante `x = x + 1`

```
>>> x = 2
>>> x = x + 1;
>>> x
3
```

Même si le langage Python est multi-paradigme , ici on a procédé par programmation impérative

Avec OCaml dans l'évaluateur (après le symbole #) on écrit la phrase

`let x = 2;;` puis on tape sur `entrée`, la phrase est évaluée et on observe

```
# let x = 2;;
val x : int = 2
```

Remarques :

1. `let` permet de définir l'objet `x` comme en mathématiques soit x l'entier ayant la valeur 2 etc...

   Ici le symbole = ne signifie pas une affectation mais une définition

2. le symbole ;; marque la fin de la phrase

3. Le résultat de l'évaluation est la phrase

   `val x : int = 2` ce qui signifie que l'objet `x` est de type `int` et a pour valeur 2

Maintenant si on fait évaluer la phrase `x = x + 1;;`
on observe

```
# x = x + 1;;
- : bool = false
```

Pourquoi ?

1. x est immuable et vaut 2
2. L'évaluation de `x = x + 1;;` est une comparaison, l'évaluateur calcule `x + 1` qui vaut 3 et compare le résultat à 2
3. Voilà pourquoi le résultat de l'évaluation n'est pas défini (symbole -) est de type bool (booléen) et sa valeur est false

1.2 Les fonctions

On veut définir la fonction successeur sur les entiers par

```
let f(x) = x + 1;;
```

on observe

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

`f` a pour type `int -> int` et pour valeur `<fun>`

Si on veut calculer le successeur de 5 on évalue sans le définir

```
f(5) ;;
```

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

Comment faire si on veut définir une fonction sur les réels ?

OCaml est un langage fortement typé si dans la phrase il y a le symbole + cela signifie qu'on additionne des entiers, si on veut additionner des réels dans ce cas on utilise le symbole +.

Si on veut définir une fonction affine sur les réels par exemple on aura

`let g(x) = 0.5*.x -. 3.14`

(Attention ! il y a deux opérations donc il faut bien écrire *. et -. pour signifier qu'on calcule sur les réels)

on observe

```
# let g(x) = 0.5 *. x -. 3.14 ;;
val g : float -> float = <fun>
```

Si on oublie de bien préciser les deux opérations sur les réels on a un message d'erreur

```
# let g(x) = 0.5*.x - 3.14;;
Error: This expression has type float but an expression was expected of type int
```

Ex 1

1. Définir le nombre `pi` par la valeur 3.141592
2. Définir la fonction `aire` d'un disque sur les réels ayant un argument `r` le rayon du disque
3. Calculer l'aire du disque de rayon 1 (faire attention au type)

Ex 2

1. Définir `g` sur les entiers par `g(x) = 1 + x`
2. Evaluer `f = g;;` et/ou `f == g;;`
3. Conclure (la comparaison de deux fonctions est un problème indécidable)

1.3 Fonction anonyme

Comme en Python avec les lambda fonction **on peut utiliser une fonction sans la définir**

Imaginons que l'on veuille calculer l'image d'un nombre par une fonction sans pour autant définir la fonction alors on peut procéder ainsi

`(function x -> x + 1)(5);;`

```
# (function x -> x + 1)(5);;
- : int = 6
```

1.4 Alternative

Supposons que l'on veuille définir la fonction valeur absolue sur les entiers

Ocaml propose une construction

`if ....then .....else ....` permettant de définir des fonctions dans lesquelles il y a une alternative

`let abs(x) = if x >= 0 then x else -x;;`

on observe

```
# let abs(x) = if x >= 0 then x else -x;;
val abs : int -> int = <fun>
# abs(-3);;
- : int = 3
```

Ex 3

Définir la fonction valeur absolue sur les réels

1.5 Polymorphisme

Supposons que l'on veuille définir une fonction `max(x,y)` définie par

`max(x,y) = le plus grand de x et de y`

Si on écrit

`let max(x,y) = if x >= y then x else y;;`

on observe

```
# let max(x,y) = if x >= y then x else y;;
val max : 'a * 'a -> 'a = <fun>
```

Cette fonction qui a pour type `'a * 'a -> 'a`, est **polymorphe** elle s'applique à deux arguments ayant n'importe quel type `'a`

on observe

```
# max(3,14);;
- : int = 14
# max(2.0,3.);;
- : float = 3.
# max("py","thon");;
- : string = "thon"
```

Le polymorphisme est une tendance à l'abstraction observée dans la plupart des langages de haut niveau

1.6 Récursivité

Supposons que l'on veuille définir la fonction factorielle

Si on définit cette fonction ainsi

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

on observe

```
# let fact(x) = if x = 0 then 1 else x*fact(x-1);;
Line 1, characters 37-41:
Error: Unbound value fact
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

L'erreur vient du fait que le nom `fact` souligné (où il y a erreur) n'a pas été défini et n'est pas reconnu

Il y a une définition spéciale pour les fonctions récursives, (on ajoute rec après let)

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

```
# let rec fact(x) = if x = 0 then 1 else x*fact(x-1);;
val fact : int -> int = <fun>
# fact(10) ;;
- : int = 3628800
```

Ex 4

Définir une fonction récursive `fib(n)` pour la suite de Fibonnacci définie par

$F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geqslant 2$

Ex 5

Traduire la fonction récursive suivante en Ocaml

```
def f_91(n):
    if n > 100:
        return n - 10
    else:
        return f_91(f_91(n + 11))
```

1.7 Fonctionnelle

Une **fonctionnelle** est une fonction dont un argument est une fonction et dont le résultat est une fonction

Par exemple définir une fonction `fois_k` qui prend en argument une fonction f et calcule la fonction

kf avec k un entier définie par $kf(x) = \underbrace{k * f(x)}_{\text{multiplication par k dans les entiers}}$

On va matérialiser cette fonction par le symbole $f \to kf$ où k est définie au préalable

On va procéder en plusieurs étapes d'abord on va définir k entier

```
let k = 3;;
```

Puis on va définir la fonctionnelle en utilisant à la fois **l'annotation de type pour dire que f est une fonction sur les entiers** et **une fonction anonyme**

```
let mult_k(f : int -> int) = function x -> k*f(x);;
```

Puis on évalue sur des cas particuliers

```
# let k = 5;;
val k : int = 5
# let mult_k(f : int -> int) = function x -> k*f(x);;
val mult_k : (int -> int) -> int -> int = <fun>
# let g(x) = 3*x - 2;;
val g : int -> int = <fun>
# mult_k(g)(5);;
- : int = 65
```

2 OCaml est multi-paradigme

OCaml n'est pas un qu'un langage purement fonctionnel et prend en compte les **effets de bords** c'est à dire

1. Les entrées-sorties
2. La succession en séquence d'instructions
3. Les boucles for et while