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 la programmation impérative.

Un calcul peut être vu comme l'évolution dans le temps d'une zone de la mémoire.

Par exemple dans la mémoire est stocké un tableau ou une liste de nombres que l'on souhaite trier dans l'ordre croissant.

2.1 Peut on le faire uniquement en programmation fonctionnelle ?

Oui, à la manière du langage LISP (List Processing) on peut écrire une fonction récursive de tri sur les **listes**.

C'est l'occasion de voir comment sont définies les listes en Ocaml.

Ex 6

1. Ecrire en Python une fonction récursive `insertion(elt,liste)` qui va insérer `elt` dans la liste **triée** `liste`
   (Indications : Le cas de base est la liste vide, dans ce cas on renvoie la liste contenant `elt` ensuite on compare `elt` à la tête de liste `liste[0]` et on appelle `insertion` sur le reste de la liste `liste[1:]`)
2. En déduire une fonction récursive en Python `tri_insertion(liste)`

2.1.1 Qu'est ce qu'une fonction ?

En Mathématiques une fonction f est définie comme une relation entre un élément x d'une ensemble E et son image $f(x)$

On parle alors de la fonction f et non de la fonction $f(x)$.

Ocaml permet de mettre en avant le nom de la fonction dans sa définition en utilisant les fonctions anonymes (voir 1.31)

Par exemple la fonction f définie par $f(x) = x^2 + 1$.

Au lieu d'être définie ainsi

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

sera à partir de maintenant définie par

```
let f = function x -> x*.x +. 1.;;
```

ce qui correspond mieux à la phrase "Soit f définie par ..."

Ex 7

2.1.2 Qu'est ce qu'une liste en Ocaml

On définit une liste par extension de la manière suivante

`let liste = [1;4;1;5;9;2];;`

Presque du Python mais on utilise des points-virgule à la place des virgules.

Attention ! Ocaml fait la distinction entre tableau et liste :

1. Une liste (comme une liste chaînée) est une structure de données dynamique qui peut évoluer (on peut ajouter ou enlever des éléments) alors qu'un tableau est statique
2. On ne peut pas accéder à l'élément d'indice i dans une liste alors que c'est possible dans un tableau

On définit un tableau par extension de la manière suivante
`let tab = [| 3;1;4;1;5;9;2 |];;`
Pour avoir accès à l'élément d'indice 2 :
`tab.(2);;`
On obtient alors 4 qui est l'élément du tableau tab d'indice 2

Revenons aux listes
Si on veut ajouter 3, un élément **de même type** entier à la liste d'entiers `liste` , l'ajout se fait en tête de la liste avec le symbole : : de la manière suivante
`3::liste;;`

```
# let liste = [1;4;1;5;9;2] ;;
val liste : int list = [1; 4; 1; 5; 9; 2]
# 3::liste;;
- : int list = [3; 1; 4; 1; 5; 9; 2]
```

Les listes se prêtent bien à la programmation récursive.

2.1.3 Filtrage ou pattern matching

La suite de Fibonnacci est définie mathématiquement ainsi en listant les cas des cas les plus simples au cas général (cette façon de procéder est appelée filtrage ou pattern matching)

```
soit f une fonction définie par:
 0 -> 0
 1 -> 1
```

```
 n -> f(n-1) + f(n-2)
```

ce qui en Ocaml donne

```
let rec f= function
| 0 -> 0
|1 -> 1
|n -> f(n-1) + f(n-2)
```

```
# let rec fib = function
    |0 -> 0
    |1 -> 1
    |n -> fib(n-1) + fib(n-2) ;;
val fib : int -> int = <fun>
```

On va maintenant utiliser le filtrage pour définir des fonctions récursives sur les listes.

On va écrire en Ocaml la fonction `insertion` de l'exercice 6

```
let rec insertion elt = function
| [] -> [elt]
|tete::reste -> if elt <= tete then elt::tete::rest
else tete::insertion elt (reste)
```

Ex 8

Ecrire en Ocaml la fonction `insertion`

Ex 9

Ecrire en Ocaml les fonctions suivantes :

1. qui renvoie la tête d'une liste
2. qui renvoie le reste d'une liste
3. qui renverse l'ordre des éléments d'une liste
4. qui renvoie la somme des entiers contenu dans la liste

5. qui renvoie Vrai si une liste d'entiers est un palindrome et Faux sinon

2.2 Peut on le faire en programmation impérative ?

Maintenant on va utiliser un tableau pour mémoriser les valeurs et **parcourir** le tableau est une expression qui fait partie de la programmation impérative.

2.2.1 mutable vs immutable

Les éléments d'un tableau sont mutables mais le tableau en soi (son adresse mémoire, sa référence est immutable)

```
let tab = [|2;7;1;8;2;8;1;8;2;8;4;5|];;
```

On veut remplacer 5 par 6
on entre dans la console `tab.(11) <- 6`

```
# let tab = [|2;7;1;8;2;8;1;8;2;8;4;5|];;
val tab : int array = [|2; 7; 1; 8; 2; 8; 1; 8; 2; 8; 4; 5|]
# tab.(11) <- 6 ;;
- : unit = ()
# tab ;;
- : int array = [|2; 7; 1; 8; 2; 8; 1; 8; 2; 8; 4; 6|]
```

2.2.2 Référence

On veut obtenir la même souplesse pour tous les types.

Imaginons que l'on veuille calculer la somme des entiers du tableau tab

On a besoin pour cela d'un variable `somme` initialisé à 0.

```
let compteur = ref 0;;
```

Si on veut augmenter la valeur de la variable `somme` on a besoin de l'opérateur ! pour avoir accès à la valeur d'une variable à partir de sa référence

`!somme` est la valeur de la référence `somme`

Le symbole d'affectation n'est pas le = comme dans le langage Python mais le symbole :=

Ainsi si on veut augmenter de 1 la valeur de la variable `somme` on écrit :

```
somme := !somme + 1;;
```

```
# let somme = ref 0;;
val somme : int ref = {contents = 0}
# somme := !somme + 1;;
- : unit = ()
# !somme ;;
- : int = 1
```

2.2.3 Séquences

La programmation impérative consiste à exécuter en **séquences** des actions dans le temps.

On fait telle action, puis telle autre etc ...

Pour construire une séquence d'actions on utilise le séparateur ;

Par exemple

```
somme := !somme + 1;somme := !somme + 2;
somme := !somme + 3;;
```

calcule la somme 1 + 2 + 3

Dans l'éditeur on a écrit

```
let somme = ref 0;;
somme := !somme + 1;
somme := !somme + 2;
somme := !somme + 3;
somme;;
```

A l'évaluation on obtient :

```
# let somme = ref 0 ;;
val somme : int ref = {contents = 0}
# somme := !somme + 1;
  somme := !somme + 2;
  somme := !somme + 3;
  somme ;;
- : int ref = {contents = 6}
```

2.2.4 Boucles

On généralise la répétition précédente avec une boucle for Dans l'éditeur on a écrit

```
4 let somme = ref 0;;
5 for i = 1 to 3 do somme := !somme + i done;
6 !somme;;
```

A l'évaluation on obtient :

```
# let somme = ref 0 ;;
val somme : int ref = {contents = 0}
# for i = 1 to 3 do somme := !somme + i done;
  !somme ;;
- : int = 6
```

Ex 10

Calculer la somme des carrés des 100 premiers entiers naturels

Pour échanger deux valeurs dans un tableau

```
let echange tab i j =
let temp = tab.(i) in
tab.(i) <- tab.(j);
tab.(j) <- temp;
```

Le mot réservé in permet de faire une définition locale uniquement à l'intérieur de la fonction échange

On utilise cette fonction pour faire un tri par sélection d'un tableau d'entiers

```
# let echange tab i j =
    let temp = tab.(i) in
    tab.(i) <- tab.(j);
    tab.(j) <- temp ;;
val echange : 'a array -> int -> int -> unit = <fun>
# let tri t =
    for i = 0 to Array.length t - 2 do
      let min = ref i in
      for j = i + 1 to Array.length t - 1 do
        if t.(j) <= t.(!min) then min := j
      done;
      echange t i !min
    done ;;
val tri : 'a array -> unit = <fun>
```

Ex 11

1. Ecrire une fonction `selection` qui renvoie l'indice de la première occurrence du minimum pour les éléments d'indice $k \geqslant i$
2. En déduire une fonction `tri_selection` qui trie dans l'ordre croissant les éléments d'un tableau d'entiers

La fonction Python suivante insère la valeur tab[i] dans la partie déjà triée tab[:i]

```
def insertion(tab: list, i:int)->None:
    """
    insère la valeur tab[i] à sa place dans la
    partie triée tab[:i]
    """
    temp = tab[i]
    j = i
    while j > 0 and tab[j - 1] > temp:
        tab[j] = tab[j - 1]
        j -= 1
    tab[j] = temp
```

On va traduire cette fonction dans le langage OCaml en utilisant la programmation impérative

```
let insertion  tab i =
   let temp = tab.(i) in
   let j =  ref i in
   while !j > 0 && tab.(!j - 1) > temp do
        tab.(!j) <- tab.(!j - 1);
        j := !j -1;
   done;
    tab.(!j) <- temp
```

```
# let insertion  tab i =
    let temp =  tab.(i) in
    let j =  ref i in
    while !j > 0 && tab.(!j - 1) > temp do
      tab.(!j) <- tab.(!j - 1);
      j := !j -1;
    done;
    tab.(!j) <- temp ;;
val insertion : 'a array -> int -> unit = <fun>
# let tab = [|1;2;4;3|] ;;
val tab : int array = [|1; 2; 4; 3|]
# insertion tab 3 ;;
- : unit = ()
# tab ;;
- : int array = [|1; 2; 3; 4|]
```

Ex 12

Mettre au point le tri fusion avec OCaml