TP 3 : Introduction à OCaml, Récursivité (Correction)

4 novembre 2023

1 OCaml

*

2 Premiers Programmes, Déclaration de Variables

*

3 Définition Locale de Valeur

*

4 Expressions Conditionnelles

Exercice 3 :

Définir un entier naturel non-nul x, et définir y dont la valeur est le successeur de x dans la suite de Syracuse.

```
let x = 5 ;;
let y =
    if x mod 2 == 0 then x/2 ;;
    else 3*x+1 ;;
```

5 Fonctions

Exercice 5 :

1. Ecrire une fonction `carre` qui renvoie le carré d'un entier.
2. Ecrire une fonction `carre_float` qui renvoie le carré d'un flottant.
3. Ecrire une fonction `cube` qui renvoie le cube d'un entier.
4. Ecrire une fonction `cube_float` qui renvoie le cube d'un flottant.

```
let carre x = x * x ;;
```

```
let carre_float x = x *. x ;;

let cube x = x * x * x ;;

let cube_float x = x *. x *. x ;;
```

Exercice 6 :

1. Définir une fonction `puissance_4` qui prend en entrée un entier x et renvoie la valeur de x^4, en définissant localement à l'intérieur une fonction `carre` qui calcule le carré d'un entier, puis en renvoyant $(x^2)^2$.
2. Définir une fonction `puissance_9` qui prend en entrée un entier x et renvoie la valeur de x^9, en définissant localement à l'intérieur une fonction `cube` qui calcule le cube d'un entier, puis en renvoyant $(x^3)^3$.

```
let puissance_4 x =
    let carre y = y*y
    in carre (carre x) ;;

let puissance_9 x =
    let cube y = y*y*y
    in cube (cube x) ;;
```

5.1 Expressions Conditionnelles dans les Fonctions

Exercice 7 :

Définir une fonction qui prend en entrée un entier naturel non-nul n et renvoie le successeur de n dans la suite de Syracuse.

```
let syracuse n =
    if n mod 2 = 0 then  n/2
    else 3*n+1 ;;
```

Exercice 8 :

1. Définir une fonction qui prend en entrée deux entiers et renvoie le maximum entre les deux.
2. Définir une fonction qui prend en entrée deux entiers et renvoie le minimum entre les deux.

```
let maximum x y =
    if x > y then x
    else y ;;

let minimum x y =
    if x < y then x
    else y ;;
```

Exercice 9 :

Ecrire une fonction qui prend en entrée un flottant et renvoie sa valeur absolue.

```
let valeur_absolue x =
    if x < 0. then -.x
    else x ;;
```

5.2 Récursivité

Exercice 10

Définir une fonction qui prend en entrée un couple d'entiers naturels n et k, avec n non-nul, et renvoie le k-ième successeur de n dans la suite de Syracuse (renvoie n si $k = 0$).

```
let rec successeur_syracuse n k =
    if k = 0 then n
    else
        if n mod 2 = 0 then successeur_syracuse (n/2) (k-1)
        else successeur_syracuse (3*n+1) (k-1) ;;
```

Exercice 11

Définir une fonction qui prend en entrée deux entiers naturels a et n et renvoie a^n, calculé à l'aide de l'exponentiation naïve.

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

Exercice 12

Définir une fonction qui prend en entrée deux entiers naturels a et n et renvoie a^n, calculé à l'aide de l'exponentiation rapide.

```
let rec exp_rapide a n =
    if n = 0 then 1
    else let b = exp_rapide a n/2 in
        if n mod 2 = 0 then b*b
        else a*b*b ;;
```

Exercice 13

Définir une fonction qui prend en entrée trois entiers a, b et c et renvoie le nombre de racines de $ax^2 + bx + c$.

```
let nb_racines a b c =
    let d = b*b-4*a*c in
    if d < 0 then 0
    else if d = 0 then 1
    else 2 ;;
```

Exercice 15

Ecrire une fonction qui prend en entrée deux entiers x et y et renvoie :

1. $x + y$ si $x - y$ est pair
2. $x - y$ sinon.

```
let fonction x y =
```

```
match x-y with
| z with z mod 2 = 0 -> x+y
| _ -> x-y ;;
```

6 Traits Impératifs

Exercice 17

Ecrire une fonction en OCaml qui prend en entrée un entier naturel non-nul n et affiche la phrase "d divise n" pour tous les diviseurs de n.

```
let diviseurs n =
    for k = 1 to n do
        if n mod k = 0 then
            Printf.printf "%d divise %d \n" k n ;
    done;;
```

7 Types Construits

Exercice 18

Définir un type complexe dont la partie imaginaire est donnée par le champ `im` et la partie réelle par le champ `re`.

```
type complexe = {re : float; im : float} ;;
```

im

Exercice 19

Ecrire une fonction qui prend en entrée un nombre complexe et renvoie son conjugué.

```
let conjugue z = {z with im = -. z.im} ;;
```

Exercice 20

Ecrire une fonction qui prend en entrée deux nombres complexes et renvoie leurs produits.

```
let produit z1 z2 =
    {re = z1.re *. z2.re -. z1.im *. z2.im ;
    im = z1.re *. z2.im +. z1.im *. z2.re } ;;
```

7.1 Sommes disjointes

Exercice 21

1) Définir un type `chifoumi` qui comprend les constructeurs Pierre, Feuille et Ciseau et un type `resultat` qui comprend les constructeurs Defaite, Nul et Victoire.

```
type chifoumi = Pierre | Feuille | Ciseau ;;

type resultat = Defaite | Nul | Victoire ;;
```

2) Ecrire une fonction qui prend en entrée un couple de valeur de type `chifoumi * chifoumi`, correspondant aux coups joué par un joueur 1 et un joueur 2, et renvoie un couple de type `resultat * resultat` correspondant aux résultats respectifs du joueur 1 et du joueur 2.

```
let partie (joueur1,joueur2) =
    match (joueur1,joueur2) with
    | (x,y) when x = y -> (Nul, Nul)
    | (Pierre, Feuille) | (Feuille,Ciseau) | (Ciseau,Pierre) -> (Defaite, Victoire)
    | _ -> (Victoire, Defaite)
;;
```

Exercice 22

1) Définir récursivement une fonction `longueur` qui calcule la longueur d'une liste.

```
let rec longueur l =
    match l with
    | [] -> 0
    | t::s -> 1 + (longueur s) ;;
```

2) Définir récursivement une fonction `concat` qui calcule la concaténation de deux listes.

```
let rec concat l1 l2 =
    match l1 with
    | [] -> l2
    | t::s -> t::(concat s l2) ;;
```

3) Définir récursivement une fonction `max` qui calcule la valeur maximale dans une liste d'entiers.

```
let rec max l =
    match l with
```

```
    | [] -> 0
    | t::s -> let m = max s in
            if t > m then t
            else m;;
```

4) Définir récursivement une fonction `somme` qui calcule la somme des éléments dans une liste d'entiers.

```
let rec somme l =
    match l with
    | [] -> 0
    | t::s -> t + somme s ;;
```

5) Définir récursivement une fonction `produit` qui calcule le produit des éléments dans une liste d'entiers.

```
let rec produit l =
    match l with
    | [] -> 1
    | t::s -> t * produit s ;;
```

6) Définir récursivement une fonction `liste_syracuse` qui prend en entrée un entier naturel non-nul a et un entier n et renvoie la liste à n éléments contenant a et ses successeurs dans la suite de Syracuse.

```
let rec liste_syracuse a n =
    let successeur b = if b mod 2 = 0 then b/2 else 3*b + 1 in
    match n with
    | 0 -> []
    | _ -> a::(liste_syracuse (successeur a) (n-1)) ;;
```

7) Définir récursivement une fonction `list_concat` qui prend en entrée une liste de listes et renvoie la concaténation de toutes les listes.

```
let rec liste_concat l =
    match l with
    | [] -> []
    | t::s -> concat t (liste_concat s) ;;
```

8) Définir une fonction `liste_fonction` qui prend en entrée une liste, une fonction et renvoie la liste des $f(x)$ pour tous les éléments x de la liste en entrée.

```
let rec liste_fonction l f =
    match l with
    | [] -> []
    | t::s -> (f t)::(liste_fonction s f) ;;
```

8 Exercices

Exercice 23

1) On considère le matériel (ensemble des pièces restantes) d'un joueur d'échecs dans une partie. Définir un type produit `materiel` pour représenter le matériel d'un joueur, où à

chaque pièce est associée la quantité dont dispose le joueur (entier).

```
type materiel = { pion : int;
    fou : int;
    cavalier : int;
    tour : int;
    dame : int;
    roi : int;
    }
```

2) Définir une fonction qui prend en entrée le matériel d'un joueur représenté sous la forme d'une liste de type `piece` (comme défini précédemment) et renvoie le matériel sous la forme d'un produit nommé (comme défini précédemment).

```
let rec materiel_produit l =
    match l with
    | [] -> { pion = 0; fou = 0; cavalier = 0; tour = 0; dame = 0; roi = 0;}
    | t::s -> let m2 = materiel_produit s in
    match t with
    | Pion -> {m2 with pion = m2.pion + 1}
    | Fou -> {m2 with fou = m2.fou + 1}
    | Cavalier -> {m2 with cavalier = m2.cavalier + 1}
    | Tour -> {m2 with tour = m2.tour + 1}
    | Dame -> {m2 with dame = m2.dame + 1}
    | Roi -> {m2 with roi = m2.roi + 1} ;;
```

3) Aux échecs, la valuation habituelle des pièces est la suivante : 1 pour le Pion, 3 pour le Cavalier et le Fou, 5 pour la Tour et 9 pour la Dame. Définir une fonction qui prend en entrée le matériel d'un joueur (représenté avec le type défini précédemment) et qui renvoie sa valeur totale.

```
let valeur m = m.pion + 3 * m.cavalier + 3 * m.fou + 5 * m.tour + 9 * m.dame ;;
```

4) Ecrire une fonction qui prend en entrée les matériels d'un joueur 1 et d'un joueur 2 dans une partie et renvoie 1 si le joueur 1 a l'avantage matériel, -1 si c'est le joueur 2, et 0 si les deux sont égaux.

```
let comparaison m1 m2 =
    let v1 = valeur m1 and v2 = valeur m2 in
    match (v1 - v2) with
    | x when x > 0 -> 1
    | 0 -> 0
    | _ -> -1
```

Exercice 24

Au Poker, les différents types de mains que l'on peut obtenir sont, dans l'ordre, de la plus faible à la meilleure : Carte Haute, Paire, Double Paire, Brelan, Suite, Couleur, Full, Carré, Quinte Flush, Quinte Flush Royale.

1) Créer un type `main` dont les constructeurs correspondent aux différents types de mains possibles.

```
type main = Carte | Paire | Double | Brelan | Suite | Couleur | Full | Carre | QF | QFR
```

2) Créer une fonction qui prend en entrée les types de mains de deux joueurs et renvoie 1, 0 ou -1 selon que la main du joueur 1 soit supérieure, égale ou inférieure à celle du joueur 2. *On pourra définir une valeur numérique pour chacune des mains possibles et comparer leurs valeurs.*

```
let comparaison_mains m1 m2 =
    let valeur m = match m with
    | Carte -> 0
    | Paire -> 1
    | Double -> 2
    | Brelan -> 3
    | Suite -> 4
    | Couleur -> 5
    | Full -> 6
    | Carre -> 7
    | QF -> 8
    | QFR -> 9
    in let v1 = valeur m1 and v2 = valeur m2 in
    if v1 > v2 then 1
    else if v1 = v2 then 0
    else -1 ;;
```

Exercice 25

Le tri par insertion fonctionne de la façon suivante : soit une liste de valeur non-triée. On prend le premier élément, on trie la suite de la liste, et on insère l'élément dans la liste triée.

1) Ecrire une fonction `insertion` qui prend en entrée un entier et une liste triée d'entiers et renvoie la liste triée à laquelle on a rajouté l'entier.

```
let rec insertion v l =
    match l with
    | [] -> [v]
    | t::s -> if t > v then v::l else t::(insertion v s) ;;
```

2) Ecrire une fonction `tri_insertion` qui prend en entrée une liste et renvoie la liste triée à l'aide de la fonction `insertion` et de la méthode présentée.

```
let rec tri_insertion l =
    match l with
    | [] -> []
    | t::s -> insertion t (tri_insertion s) ;;
```

Exercice 26

Le tri fusion se fait de la façon suivante :

- On divise une liste en deux sous-listes de tailles égales, à une unité près ;
- On trie chacune des sous-listes ;
- On fusionne les listes triées.

1) Ecrire une fonction `division` qui prend en entrée une liste et la divise en deux listes de tailles égales.

```
let rec division l =
    match l with
    | [] -> ([],[])
    | t::[] -> ([t],[])
    | t1::t2::s -> let (l1,l2) = division s in (t1::l1,t2::l2) ;;
```

2) Ecrire une fonction `fusion` qui prend en entrée deux listes triées et renvoie la fusion triée des deux listes.

```
let rec fusion l1 l2 =
    match (l1,l2) with
    | (_,[]) -> l1
    | ([],_) -> l2
    | (t1::s1,t2::s2) -> if t1 < t2 then t1::(fusion s1 (t2::s2))
                         else t2::(fusion (t1::s1) s2) ;;
```

3) Ecrire une fonction `tri_fusion` qui fait le tri d'une liste à l'aide des deux fonctions définies précédemment et de la méthode présentée.

```
let rec tri_fusion l =
  match l with
  | [] -> []
  | t::[] -> l
  | _ -> let (l1,l2) = division l
      in fusion (tri_fusion l1) (tri_fusion l2) ;;
```