Introduction à la programmation

Le langage Caml

Pierre TELLIER
Sylvain THERY
Cédric WEMMERT

INTRODUCTION À CAML

Objets de base

- constantes entières : 26, -172, ...
- constantes réelles : 12.23, -0.32, ...
- constantes booléennes : true, false
- constantes chaînes de caractères : "oui", "bonjour", ...
- identificateurs : fact, pgcd, succ, ...
- identificateurs des fonctions prédéfinies : *, +, mod, or, *., <, ...

Expressions

Notation classique pour les nombres : 2+3, sqrt(3.71), 3<2, ...

Dialogue avec Caml

```
# 2+2 ;;          # ⇨ invite de Caml       ;; ⇨ fin d'une phrase
```
Réponse de Caml :
```
- : int = 4       - indique qu'une valeur a été calculée, un entier qui vaut 4
```
Remarque : Caml déduit toujours le type du résultat

Définitions

Il s'agit de donner un nom à une valeur ou une expression.

- Définition globale

Une définition globale est permanente (pendant toute la session Caml).

```
# let s=1+2+3 ;;
s : int = 6
# let s2 = s*s ;;
s2 : int = 36
# let s = 1 ;;
s : int = 1
```

- Définition locale

Une définition locale (temporaire) permet de faire un calcul. La définition n'est plus valable à la fin de l'évaluation.

```
# let s = 20 in s*4 ;;  in ⇨ mot-clé signifiant  « dans l'expression »
- : int = 80
# s ;;
s : int = 1             la définition globale reste valable
```

Fonctions

Un programme Caml est une suite de définitions de fonctions, suivie d'un appel à la fonction qui déclenche le calcul voulu.

- Définition globale d'une fonction

Exemple : définition de la fonction succ qui à un entier x associe son successeur x+1.

```
# let succ(x) = x+1 ;;
succ : int -> int = <fun>
# succ ;;
succ : int -> int = <fun>     le nom succ possède une valeur fonctionnelle
```

- Application d'une fonction

```
# succ(2) ;;
- : int = 3
```

- Définition locale d'une fonction

```
# let pred x = x - 1 in (pred 3) * (pred 4) ;;    locale à une expression
- : int = 6
# let puiss4 x = let carre y = y*y in carre (carre x) ;;    locale à une
fonction
puiss4 : int -> int = <fun>
```

- Fonctions à plusieurs arguments

```
# let moy a b = (a+b)/2 ;;
moy : int -> int -> int = <fun>
# let perimetre_rectangle longueur largeur = 2*(longueur + largeur) ;;
perimetre_rectangle : int -> int -> int = <fun>
```

- Notation en style indirect

```
# let perimetre_rectangle =
      function longueur -> function largeur -> 2*(longueur + largeur) ;;
```

Attention ! On ne peut pas écrire cela :

```
# let moyenne = function a b -> (a+b)/2 ;;
```

- Notation complète

```
# let (perimetre_rectangle : int -> int -> int) =
      function longueur ->
      function largeur -> 2*(longueur + largeur) ;;
```

- Fonctions anonymes

Il est possible de définir des fonctions sans leur donner d'identificateur.

```
# (function x -> 2*x + 1) ;;
- : int -> int = <fun>
# (function x -> 2*x + 1) 2 ;;
- : int = 5
```

- Usage des parenthèses

```
# succ(2*3) ;;
- : int = 7
# succ 2*3 ;;           succ 2 est évalué avant la multiplication
- : int = 9
# moy 2 3 ;;            OK
# (moy 2) 3 ;;          ERREUR
# moy (2 3) ;;          ERREUR
# succ -3 ;;            ERREUR
# succ (-3) ;;          OK
# succ succ 1 ;;        ERREUR
# succ (succ 1) ;;      OK
```

- Types quelconques

Dans certains cas, Caml ne peut pas déterminer un type unique pour certains arguments :

```
# let f x = 1 ;;        x peut être de n'importe quel type
f : 'a -> int = <fun>
```

```
# let f x = x ;;          x et le résultat son de n'importe quel type mais le
f : 'a -> 'a = <fun>      résultat est du même type que x

# let f x y = 5 ;;              x et y sont de type quelconque mais pas
f : 'a -> 'b -> int = <fun>     forcément le même

# let f x y = (x=y) ;;          x et y sont de type quelconque, mais le même
f : 'a -> 'a -> bool = <fun>
```

Exercice 1

Définir la fonction `fois4` à partir uniquement de la définition suivante : `let fois2 x = 2*x ;;`

Réponse :

```
# let fois4 x = fois2 (fois2 x) ;;
```

Exercice 2

Discuter :

```
# let moy_e x y = (x+y)/2 ;;
# let moy_r x y = (x +. y) /. 2.0 ;;
```

Réponse :

La première fonction estime la moyenne entière entre deux entiers et la seconde effectue la moyenne entre deux nombres réels et le résultat est un nombre réel.

LE DÉCOUPAGE FONCTIONNEL

Exercice 1

Donner le résultat des expressions Caml suivantes :

```
❶ # (function x -> function y -> 3*(x+y)) 2 5 ;;
❷ # (function x -> 3 * (x + (function y -> y) 2)) 5 ;;
❸ # 3 * (function x -> x * x) 3 ;;
❹ # (function x -> function y -> 2 + (x*y)) (3+1) (1+2) ;;
❺ # (function x -> function y -> function z -> function u -> z) 23 91 89 3 ;;
```

Réponse :

❶ - : int = 21

On remplace x par 2 et y par 5, puis on calcule 3(x+y).*

❷ - : int = 21

Il faut d'abord évaluer (function y -> y) 2. C'est la fonction identité appliquée à 2. Donc le résultat est 2. On remplace alors dans l'expression et on obtient :

```
(function x -> 3 * (x + 2)) 5 ;;
```

❸ - : int = 27

*On évalue d'abord (function x -> x * x) 3, ce qui donne 9. Puis on remplace dans l'expression et on obtient : 3 * 9 ;;*

❹ - : int = 14

Il faut évaluer en priorité les calculs entre parenthèses avant de faire l'affectation :

```
(function x -> function y -> 2 + (x*y)) 4 3 ;;
```

❺ - : int = 89

Cette fonction prend 4 arguments et renvoie le troisième, donc 89.

Exercice 2

Concevoir et coder en Caml des fonctions qui calculent les aires des figures suivantes :carré, rectangle, cercle, triangle et cylindre.

Réponse :

Il suffit d'appliquer les formules «classiques» :

```
# let aire_carré a = a *. a ;;
# let aire_rectangle l L = l *. L ;;
# let pi = 3.14159 ;;
# let aire_cercle r = pi *. r *. r ;;
# let aire_triangle b h = b *. h /. 2 ;;
# let aire_cylindre r h = 2. *. pi *. r *. h ;;
```

Exercice 3

Concevoir et coder en Caml une fonction qui calcule l'aire d'un hexagone régulier sachant que l'aire d'un triangle équilatéral est a*a*racine(3)/4.

Réponse :

Définir tout d'abord la fonction qui calcule l'aire d'un triangle équilatéral :

```
# let aire_triangle_equi a = let racine_3 = 1.732 in a*.a*.racine_3/.4 ;;
```

Puis en déduire la fonction d'aire d'un hexagone régulier qui comporte 6 triangles équilatéraux :

```
# let aire_hexagone a = 6. *. aire_triangle_equi a ;;
```

Exercice 4

Concevoir et coder en Caml des fonctions qui calculent les volumes des objets suivants :
cube, parallélépipède, sphère, prisme hexagonal.

Réponse :

Il suffit d'appliquer les formules « classiques » :

```
# let vol_cube a = a *. a *. a ;;
# let vol_paral l p h = l *. p *. h ;;
# let vol_sphere r = 4. /. 3. *. pi *. r *. r *. r ;;
```

Pour le prisme, utiliser la formule de l'aire d'un hexagone définie plus haut :

```
# let vol_prisme h a = h *. aire_hexagone a ;;
```

Exercice 5

Définir la fonction `renverser` telle que : `renverser 13 = 31`.
Cette fonction ne s'applique qu'aux nombres de 0 à 99.

Réponse :

Un nombre n compris entre 0 et 99 s'écrit avec deux chiffres : l'unité et la dizaine,
*avec n = unité + 10*dizaine. Il suffit donc d'extraire ces deux chiffres du nombre puis de reconstruire un nouveau nombre en les inversant.*

```
# let dizaine n = n / 10 ;;
# let unité n = n mod 10 ;;
# let renverser n = 10 * unité n + dizaine n ;;
```

Exercice 6

Ecrire une fonction qui teste si deux intervalles sont disjoints.

Réponse :

Ecrire tout d'abord une fonction qui teste si deux nombres x et y forment bien un intervalle non-vide :

```
# let est_un_intervalle x y = x < y ;;
```

Puis il suffit de tester les deux cas possibles :

```
# let int_disjoints x1 y1 x2 y2 =
      est_un_intervalle x1 y1 &
      est_un_intervalle x2 y2 &
      ((y1 < x2) or (y2 < x1) ) ;;
```

Exercice 7

Evaluer les expressions suivantes. Préciser un type possible pour chacune des fonctions.

❶ `# (function x -> function y -> x+1) (2*5) (1.13 *. 2.78) ;;`
❷ `# (function x -> not(not x)) false ;;`
❸ `# (function x -> function y -> (not((x*x)=y) or (x<2))) 2 4 ;;`
❹ `# (function x -> function y ->(not((x*x)=y & (x<=2))) 2 4 ;;`

Réponse :

❶ *Il suffit de remplacer x par (2*5) et y par (1.13*.2.78) dans l'expression x+1.*

```
- : int = 11
- : int -> 'a -> int = <fun>
```

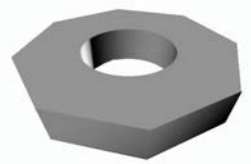
❷ *not (not false) = not (true) = false.*

```
- : bool = false
```

```
- : bool -> bool = <fun>
❸ not((2*2)=4) or (2<2) = not(true) or (false) = false or false = false.
- : bool = false
- : int -> int -> bool = <fun>
❹ not((2*2)=4 & (2<=2)) = not( true & true ) = not( true ) = false.
- : bool = false
- : int -> int -> bool = <fun>
```

Exercice 8

Concevoir et coder en Caml une fonction qui calcule le volume d'un écrou à 8 pans de côtés x, de hauteur h et évidé d'un cylindre de rayon r, comme le montre la figure ci-dessous.



Réponse :

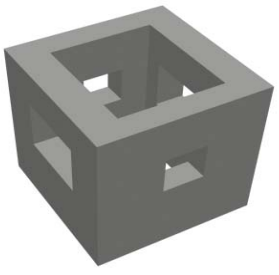
*Le volume de l'écrou se calcule comme le produit entre la hauteur h et la surface de la base (aire de l'octogone – aire du disque), ce qui correspond en fait à 8 fois l'aire d'un triangle isocèle de base x et de hauteur x/(2*tan(pi/8)) moins l'aire du disque de rayon r. On calcule donc la surface couverte par les 8 triangles puis le volume de l'écrou de la manière suivante :*

```
# let vol_ecrou x h r =
      let surf_triangles = 8. *. aire_triangle x (x /. (2. *. tan(pi /. 8.)))
      in h *. (surf_triangles -. aire_cercle r ) ;;
```

Exercice 9

Concevoir et coder en Caml une fonction qui calcule le volume d'un cube troué par 3 parallélépipèdes de côtés différents. Pour cela, écrire tout d'abord trois fonctions, `max`, `min` et `mil` qui renvoie respectivement le maximum, le minimum et la valeur médiane de trois valeurs.

Réponse :



```
# let max x y z =
            if ( x < y ) then
                  if ( y < z ) then z
             else y
            else
                  if ( x < z ) then z
                   else x;;

# let min x y z =
            if ( x > y ) then
                  if ( y > z ) then z
             else y
            else
                  if ( x > z ) then z
                   else x;;
```

```
# let mil x y z =
            if ( x > y ) then
                  if ( x < z ) then x
            else if ( z > y ) then z
            else y
             else
                  if ( x > z ) then x
            else if ( z < y ) then z
            else y;;
```

Ecrivons tout d'abord deux fonctions qui calcule respectivement le volume d'un cube et le volume d'un parallélépipède rectangle :

```
# let volcube c = c*.c *.c ;;
```

```
     # let volpara c l = c*. c*. l ;;
```

On peut maintenant définir la fonction qui calcule le volume du cube troué :

```
# let vol c u v w = let m = max u v w in
                  volcube c -. volpara m c -. volpara (min u v w) (c-.m)
                  -. volpara (mil u v w) (c-.m);;
```

LES BOOLÉENS ET LES CONDITIONNELLES

Exercice 1

Concevoir et coder en Caml la fonction max qui calcule le maximum entre deux nombres entiers.

Réponse :

Il suffit de faire un test pour déterminer si a est supérieur à b ou non.

```
# let max a b = if a > b then a else b ;;
```

Exercice 2

Concevoir et coder en Caml une fonction qui calcule l'aire d'une couronne définie par deux cercles concentriques de rayons r et R.

Réponse :

On définit une fonction calculant l'aire d'un disque et on s'en sert pour calculer l'aire de la couronne comme étant la différence entre l'aire du plus grand disque et celle du plus petit.
ATTENTION ! *r n'est pas forcément plus petit que R.*
Afin de déterminer lequel est le plus grand, on fait un test sur les deux rayons.

```
# let pi = 3.14159 ;;
# let aire_cercle r = pi *. r *. r ;;
# let aire_couronne r R =
      if R >. r then aire_cercle R -. aire_cercle r
      else aire_cercle r -. aire_cercle R ;;
```

Exercice 3

Concevoir et coder en Caml une fonction calculant le prix d'une commande de q litres de vin, dont le prix unitaire au litre est égal à p. Définir la fonction suivant les deux cas suivants :
❶ si la commande vaut au moins 500 F, le port est gratuit, sinon, c'est 10% de la commande.
❷ idem sauf que le port vaut un minimum de 10 F s'il n'est pas gratuit.

Réponse :

❶ *Faire un test pour déterminer si la commande vaut au moins 500 F, c'est-à-dire si p*q est supérieur à 500.*

```
# let facture q p =
      if p *. q > 500. then p*. q
      else 1.1 *. p *. q ;;
```

❷ *Faire le même test plus un second test afin de déterminer si le port vaut moins de 10 F.*

```
# let facture2 q p =
      if p *. q > 500. then p *. q
      else
            if 0.1 *. p *. q < 10. then p *. q + 10.
            else 1.1 *. p *. q ;;
```

Exercice 4

Contrôler le type des fonctions suivantes et expliquer ce qu'elles font :

❶ `# let equal x y = if x=y then true else false ;;`
❷ `# let Entre3et5 x = (x>3) & (x<5) ;;`
❸ `# let Test1 x = if Entre3et5 x = false then 1 else 2 ;;`
❹ `# let Test2 x = if equal (Entre3et5 x) false then 1 else 2 ;;`
❺ `# let Test3 x = if not (Entre3et5 x) then 1 else 2 ;;`

Réponse :

❶ *Fonction d'égalité.*
`- : 'a -> 'a -> bool = <fun>`
❷ *Test si un nombre est strictement compris entre 3 et 5.*
`- : int -> int -> bool = <fun>`
❸ *Renvoie 1 si x est compris entre 3 et 5, et 2 sinon.*
`- : int -> bool = <fun>`
❹ *Renvoie 1 si x n'est pas compris entre 3 et 5 (Entre3et5 x = false), et 2 sinon.*
`- : int -> bool = <fun>`
❺ *Renvoie 1 si x n'est pas compris entre 3 et 5 (not(Entre3et5 x) = true), et 2 sinon.*
`- : int -> bool = <fun>`

Exercice 5

Proposer une expression Caml pour calculer le discriminant d'une équation du second degré, puis une fonction déterminant le nombre de solutions d'une telle équation.

Réponse :

L'expression calculant le discriminant d'une équation de la forme $ax^2 + bx + c = 0$ s'écrit de la manière suivante : $b^2 - 4ac$
Ensuite le nombre de solution s'obtient en testant le signe du discriminant : s'il est strictement positif, il y a 2 solutions ; s'il est strictement négatif, il y a aucune solution ; sinon (c'est qu'il est nul), il y a 1 seule solution.

```
# let nb_sol a b c =
      let discriminant = b *. b -. 4 *. a *. c in
            if discriminant > 0. then 2
            else if discriminant <. 0 then 0
            else 1 ;;
```

LES TYPES

Exercice 1

Définir un type `intervalle` avec les extracteurs : `borne_inf` et `borne_sup` qui renvoient les bornes de l'intervalle ; et les opérateurs : `est_intervalle` qui teste si un intervalle est non vide et non réduit à un singleton ; et `intervalle_disjoints` qui teste si deux intervalles sont strictement disjoints.

Réponse :

On définit le type intervalle comme étant un couple de réels :

```
# type intervalle == float * float ;;
```

Les constructeurs de base renvoient respectivement la borne inférieure et la borne supérieure de l'intervalle :

```
# let (borne_inf : intervalle -> float) = function (inf,sup) -> inf ;;
# let (borne_sup : intervalle -> float) = function (inf,sup) -> sup ;;
```

Pour tester si un intervalle est non vide et non réduit à un singleton, on teste si la borne inférieure est bien strictement inférieure à la borne supérieure :

```
# let (est_intervalle : intervalle -> bool) = function i ->
    (borne_inf i) < (borne_sup i) ;;
```

Pour tester si les deux intervalles a et b sont disjoints, on teste tout si se sont bien des intervalles à l'aide de la fonction est_intervalle. Ensuite on teste si la borne supérieure de a est strictement plus petite que la borne inférieure de b (cas 1), ou si la borne inférieure de a est strictement plus grande que la borne supérieure de b (cas 2) :

```
# let (intervalles_disjoints : intervalle -> intervalle -> bool)
    = function a -> function b ->
      (est_intervalle a) & (est_intervalle b) &
      ( (borne_sup a < borne_inf b) or (borne_inf a > borne_sup b));;
```

Exercice 2

Définir le type nombre réel avec les extracteurs `mantisse` et `caractère`, et les opérateurs : `produit` et `somme` de deux nombres réels.
On définit un nombre réel x par sa mantisse m et son caractère c de la manière suivante : $x = m.10^c$

Réponse :

```
# type reel == int * int ;;

# let (mantisse : reel -> int) = function (a,b) -> a ;;
# let (caract   : reel -> int) = function (a,b) -> b ;;
```

Le produit de deux réels $a.10^b$ et $c.10^d$ est égal à $(a*c).10^{(b+d)}$:

```
# let (prod : reel -> reel -> reel) =
    function x -> function y ->
       let m = mantisse x * mantisse y
       and c = caract x + caract y
       in (m,c);;
```

Afin de définir la somme de deux réels, il faut écrire les deux réels avec le

même caractère, celui des deux qui est le plus petit, puis faire la somme des mantisses obtenues. Pour cela nous définissons une fonction exposant qui élève un entier à une puissance de 10, basée sur la fonction puissance :

```
# let rec puissance x = function
  0 -> 1
 |n -> x * puissance x (n-1);;

# let exposant x = puissance x 10;;

# let (som : reel -> reel -> reel) =
  function x -> function y ->
    if caract y < caract x then
        let m_x = mantisse x * exposant(caract x - caract y)
        in (m_x + mantisse y,caract y)
   else let m_y = mantisse y * exposant(caract y - caract x)
        in (m_y + mantisse x,caract x);;
```

Exercice 3

Définir le type `quotient` avec les extracteurs `numérateur` et `dénominateur`, et les opérateurs : `somme`, `produit`, et `réduction` d'un quotient.

Réponse :

Un quotient est défini comme un couple d'entiers : le numérateur puis le dénominateur.

```
# type quotient == int * int;;
# let (numerateur   : quotient -> int) = function (n,d) -> n ;;
# let (denominateur : quotient -> int) = function (n,d) -> d ;;
```

La somme de deux quotients se calcule de la manière suivante :
$n_1/d_1 + n_2/d_2 = (n_1*d_2 + n_2*d_1) / (d_1*d_2)$

```
# let (somme_quotients : quotient -> quotient -> quotient)
   = function a -> function b ->
     let n = (numerateur a * denominateur b) + (numerateur b * denominateur a)
      and d = (denominateur a * denominateur b)
      in (n,d);;
```

Le produit de deux quotients se calculent de la manière suivante :
$n_1/d_1 * n_2/d_2 = (n_1*n_2) / (d_2*d_1)$

```
# let (produit_quotients : quotient -> quotient -> quotient)
    = function a -> function b ->
      let n = numerateur a * numerateur b
      and d = denominateur a * denominateur b
      in (n,d);;
```

Afin de réduire un quotient, il faut diviser le numérateur et le dénominateur par leur pgcd. La fonction pgcd peut être définie récursivement de la manière suivante (cf exercice 2 TD 5):

```
# let rec pgcd a b =
    if (a = b) or (b=0) then a
    else if (a=0) then b
    else if a < b then pgcd a (b-a)
         else pgcd (a-b) b;;
# let (reduire_quotient : quotient -> quotient)
    = function a ->
      let p = pgcd (numerateur a) (denominateur a)
      in ((numerateur a)/p, (denominateur a)/p);;
```

Exercice 4

Définir le type `vecteur` avec les opérateurs : test vecteur nul, somme, produit scalaire, norme, homothétie, normer.

Réponse :

On définit un vecteur dans l'espace par ses trois coordonnées réelles (x, y, z).

```
# type vecteur == float * float * float ;;
```

Le test du vecteur nul est trivial :

```
# let (testnul : vecteur -> bool) = function v -> v=(0.0, 0.0, 0.0) ;;
```

La somme de deux vecteurs est le vecteurs composé des sommes des différentes coordonnées :

```
# let vecplusvec (x1,y1,z1) (x2,y2,z2) = (x1+.x2, y1+.y2, z1+.z2) ;;
```

Le produit scalaire de deux vecteurs est définit comme suit :

```
# let produitscalaire (x1,y1,z1) (x2,y2,z2) = x1*.x2 +. y1*.y2 +. z1*.z2;;
```

La norme d'un vecteur v est égal à la racine carrée du produit scalaire de v par v :

```
# let norme (x,y,z) = sqrt ( produitscalaire (x,y,z) (x,y,z) ) ;;
```

Le vecteur v', image du vecteur v par l'homothétie de rapport a, se construit en multipliant chaque coordonnée de v par a :

```
# let homothetievec (x,y,z) a = (x*.a, y*.a, z*.a) ;;
```

Pour normer un vecteur, on lui applique une homothétie de rapport égal à l'inverse de sa norme :

```
# let normer (x,y,z) =
    let coeff = 1. /. norme (x,y,z) in homothetievec (x,y,z) coeff ;;
```

Exercice 5

Définir le type `horaire` avec les extracteurs : `heure`, `minute` et `seconde` ; et les opérateurs de tranformation d'un horaire en secondes, d'un nombre de secondes en horaire, de test chronologique, et de différence entre deux horaires.

Réponse :

Le type horaire es constitué de trois entiers : les heures, les minutes et les secondes :

```
# type horaire == int * int * int ;;

# let (heure : horaire -> int) = function (h,m,s) -> h ;;
# let (min :   horaire -> int) = function (h,m,s) -> m ;;
# let (sec :   horaire -> int) = function (h,m,s) -> s ;;
```

Pour transformer un horaire en secondes, il suffit d'appliquer la formule suivante, sachant qu'une heure équivaut à 3600 secondes et une minute équivaut à 60 secondes :

```
# let (horaire_sec : horaire -> int) = function h ->
     (3600 * heure h) + (60 * min h) + sec h ;;
```

L'opération inverse se fait comme suit :

- le nombre d'heures s'obtient en divisant le nombre total de secondes par 3600 ;
- le nombre de minutes est le reste de cette division, divisé par 60 ;
- le nombre de secondes correspond au reste de la division du nombre total de secondes par 60.

```
# let (sec_horaire : int -> horaire) = function s ->
    (s / 3600, (s mod 3600) / 60, s mod 60) ;;
```

Pour le test chronologique entre deux horaires, il suffit de les convertir en nombre de secondes et de comparer les deux résultats :

```
# let (chronologie : horaire -> horaire -> bool)
     = function h1 -> function h2 ->
     horaire_sec h1 < horaire_sec h2 ;;
```

De même que précédemment, on transforme les horaires en nombre de secondes, on effectue la différence puis on retransforme le résultat en horaire :

```
# let (diff_horaire : horaire -> horaire -> horaire)
     = function h1 -> function h2 ->
     sec_horaire (horaire_sec h2 - horaire_sec h1) ;;
```

Exercice 6

Définir le type nombre complexe avec les extracteurs `partie réelle`, `partie imaginaire` et le constructeur d'un imaginaire à partir de deux réels; puis les opérateurs : `somme`, `module`, `produit`, `argument`, `division`, `racine`.

Réponse :

On définit un complexe à partir de deux réels : sa partie réelle et sa partie imaginaire.

```
# type nbcomplexe == float * float ;;

# let (preelle : nbcomplexe -> float) = function (r,i) -> r ;;
# let (pimg    : nbcomplexe -> float) = function (r,i) -> i ;;
# let (faire_nbcomplexe : float -> float -> nbcomplexe) =
        function r -> function i -> (r,i);
```

La somme de deux complexes est le complexe ayant comme partie réelle la somme de leurs parties réelles et comme partie imaginaire la somme de leurs parties imaginaires :

```
# let somme_complexes a b =
        let r = preelle a +. preelle b
        and i = pimg a +. pimg b
        in faire_nbcomplexe r i ;;
```

Le module d'un nombre complexe correspond à la racine carrée de la somme des carrés de sa partie réelle et imaginaire :

```
# let module z =
        let carre x = x *.x in
        sqrt (carre (preelle z) +. carre (pimg z));;
```

Pour le produit, l'argument, la division et la racine, appliquer directement la définition mathématique :

```
# let produit_complexes a b =
        let r1 = preelle a
        and r2 = preelle b
```

```
        and i1 = pimg a
        and i2 = pimg b
        in let x = r1 *. r2 -. i1 *. i2
           and y = i1 *. r2 -. r1 *. i2 ;;

# let argument z =
        let r = preelle z
        and i = pimg z
        in
        if r < 0.0 then atan (i/.r)
                   else
                      if r = 0.0 then if i < 0.0 then (-0.5 *. PI)
                                                 else if i = 0.0 then 0.0
                                                                 else (0.5 *. PI)
                                 else (i/.r) ;;

# let division_complexes z1 z2 =
      let r1 = preelle z1 and i1 = pimg z1 and r2 = preelle z2 and i2 = pimg
z2
      in let a = carre r2 +. carre i2
      in if a = 0.0 then failwith "division complexe par 0"
            else let r = (r1 *. r2 +. i1 *. i2) /. a
            and i = (i1 *. r2 -. r1 *. i2) /. a
            in faire_nbcomplexe r i ;;

# let racine_complexe z =
      let modu = sqrt (module z)
      and argu = (argument z) *. 0.5
      in let r = modu *. sin argu
      and i = modu *. cos argu
      in faire_nbcomplexe r i ;;
```

LA RÉCURSIVITÉ

Exercice 1

Concevoir et écrire une fonction en Caml qui effectue le calcul de la factorielle d'un entier.

Réponse :

*La définition récursive de la factorielle est la suivante : fact(n) = n*fact(n-1) avec fact(0)=1*

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

Exercice 2

Ecrire un programme Caml qui effectue le calcul du pgcd entre deux entiers.

Réponse :

```
# let rec pgcd a b =
    if (a = b) or (b=0) then a
    else if (a=0) then b
    else if a < b then pgcd a (b-a)
         else pgcd (a-b) b;;
```

Exercice 3

Quel est le type de la fonction suivante ? Se termine-t-elle toujours ? Pourquoi ?

```
# let rec Test x y =
    if not ( (x <= y) or (x = y * y) ) then Test x (y + 1)
    else x = y * y ;;
```

Réponse :

```
- : int -> int -> bool = <fun>
```

Cette fonction se termine toujours. Il y a deux cas possibles :
y >= x : la condition est toujours fausse donc si $x=y^2$ le retour est true et sinon false ;
y < x : la condition est vraie tant que x est différent de y^2 et y < x. Donc soit la fonction s'arrête lorsque y >= x et on se retrouve dans le cas ci-dessus, soit elle s'arrête lorsque x=y2 et dans ce cas renvoie true.

Exercice 4

Concevoir et coder en Caml une fonction qui calcule l'élévation à une puissance entière d'un entier.

Réponse :

La définition récursive de x^n est la suivante : $x^n = x*x^{(n-1)}$ et $x^0=1$.

```
# let rec puissance x = function
  0 -> 1
 |n -> x * puissance x (n-1);;
```

Exercice 5

Concevoir et coder en Caml une fonction qui effectue la somme des n premiers entiers.

Réponse :

La définition récursive est assez intuitive : somme(n) = n + somme(n-1) et somme(0) = 0.

```
# let rec somme_premiers_entiers n =
    if n = 0 then 0
    else n + somme_premiers_entiers (n-1) ;;
```

Exercice 6

Concevoir et coder en Caml une fonction qui calcule la somme des cubes des chiffres d'un nombre.

Réponse :

Nous définissons tout d'abord une fonction cube qui calcule le cube d'un nombre.

```
# let cube n = n*n*n;;
```

Ensuite on extrait les chiffres du nombre un à un et on effectue la somme de leurs cubes.

```
# let rec somme_cube_chiffre n =
        if n < 10 then cube n
        else cube (n mod 10) + somme_cube_chiffre (n / 10) ;;
```

Exercice 7

Concevoir et coder en Caml deux fonctions qui renvoient respectivement le quotient et le reste de la division entière de deux entiers en utilisant uniquement l'addition et la soustraction.

Réponse :

En ce qui concerne le quotient, l'idée consiste à retrancher b de a tant que le résultat est supérieur à b, et de compter combien de fois on a pu retrancher b de a :

```
# let rec div a b =
    if a < b or a=0 then 0
    else 1 + div (a-b) b ;;
```

Pour le reste, l'idée est à peu près la même. On retranche b autant de fois que possible, et on renvoie le résultat lorsqu'il inférieur à b :

```
# let rec reste a b =
    if b=0 then (-1)
    else if a=0 then 0
    else if a < b then a
         else reste (a-b) b ;;
```

Exercice 8

Concevoir et coder en Caml une fonction qui calcule la combinatoire C(n,p).

Réponse :

La combinatoire peut s'écrire de manière non récursive :

```
# let combinaison n p =
    if n < p then 0
    else if p=0 then 1
    else (fact n) / ((fact p) * (fact (n-p)));;
```

Sinon on peut l'écrire de manière récursive sans utiliser la factorielle :

```
# let rec combinaison n p =
    if n < p then 0
    else if (p=0 or n=p) then 1
    else combinaison (n-1) (p-1) + combinaison (n-1) p ;;
```

Exercice 9

Concevoir et coder en Caml une fonction qui renverse un nombre quelconque. Exemple : renverser 52423 = 32425

Réponse :

Ecrivons d'abord une fonction qui calcule la puissance de 10 la plus proche de n et inférieure à n, que nous appellerons le rang de n. Exemple : rang(2543) = 1000

```
# let rec rang n =
    if n < 10 then 1
    else 10 * rang (n/10);;
```

Ensuite on définit la fonction renverser de manière récursive en extrayant les chiffres les uns après les autres et en reconstituant au fur et à mesure le résultat, en multipliant le chiffre extrait par le rang de n :

```
# let rec renverser n =
    if n < 10 then n
    else ((n mod 10) * rang n) + renverser (n/10);;
```

Exercice 10

Concevoir et coder en Caml une fonction qui effectue la multiplication par p (p>=0) d'un entier a.

Réponse :

*L'idée consiste à additionner p fois a. On utilise l'appel récursif suivant : a*p = a + a*(p-1).*
Les conditions d'arrêt sont : si a vaut 0 ou p vaut 0 alors le résultat vaut 0 ; si p vaut 1 le résultat vaut a.

```
# let rec mult a p =
    if p = 0 or a = 0 then 0
    else if p = 1 then a
    else a + mult a (p-1);;
```

Exercice 11

Concevoir et coder en Caml une fonction qui teste si un entier n est un palindrome.

Réponse :

Idée simple non récursive, utiliser la fonction renverser : si n = renverser n alors c'est un palindrome.
Version récursive : on extrait les deux « extrémités » de n et on les compare. On continue tant qu'ils sont identiques et tant que n est supérieur à 10.

```
# let rec palindrome n =
    if n < 10 then true
    else let cgauche = n/(rang n) in
         (cgauche = (n mod 10)) & palindrome ((n - (cgauche * (rang n)))/10);;
```

Exercice 12

Concevoir et coder en Caml une fonction qui calcule le $n^{ième}$ terme de la suite de Fibonacci.

Réponse :

La définition de la suite de Fibonacci est déjà récursive. Il n'y a qu'à la transcrire en Caml :

```
# let rec fibo n =
    if n = 0 then 1
    else if n = 1 then 1
    else fibo (n-1) + fibo (n-2);;
```

Version plus rapide :

```
# let rec fibo2 n = match n with
 0 -> (0,1)
|1 -> (1,1)
|_ -> let (i,j) = fibo2 (n-1) in (j,j+i);;

# let fibo_rapide n = let (i,j) = fibo2 n in j;;
```

Exercice 13

Mettre en oeuvre le type date avec les constructeurs de base : lejour (lundi=1, …), jour, mois, année.
Définir les fonctions suivantes :
- vérification que 2 dates sont bien dans l'ordre chronologique ;
- calcul de la veille d'une date ;
- calcul du lendemain d'une date (avec années bissextiles) ;
- calcul du nombre de jours entre deux dates ;
- recherche du prochain vendredi 13 ;
- recherche du jour de sa naissance.

Réponse :

On représente une date par 4 entiers : le jour de la semaine (lundi=1, mardi=2, …), le jour dans le mois, le mois et l'année.

```
# type date == int * int * int * int ;;

# let (lejour: date -> int) = function (day, j, m, a) -> day ;;
# let (jour:   date -> int) = function (day, j, m, a) -> j ;;
# let (mois:   date -> int) = function (day, j, m, a) -> m ;;
# let (annee:  date -> int) = function (day, j, m, a) -> a ;;
```

Une date est chronologiquement antérieure à une autre si :

- *soit son année est strictement inférieure ;*
- *soit son année est égale mais son mois est strictement inférieur ;*
- *soit son année et son mois sont égaux mais son jour est strictement inférieur.*

```
# let (date_avant: date -> date -> bool) = function d1 -> function d2 ->
     if annee d1 < annee d2 then true
     else if annee d1 > annee d2 then false
     else
          if mois d1 < mois d2 then true
          else if mois d1 > mois d2 then false
          else jour d1 < jour d2;;
```

La définition d'une année bissextile (simplifiée) est la suivante :
« Une année est bissextile si c'est un multiple de 4 mais pas un siècle, ou si c'est un multiple de 400. »

```
# let bissextile a = (((a mod 4) = 0) & ((a mod 100) <> 0))
                   ||((a mod 400) = 0) ;;
```

Nous définissons une fonction qui associe à chaque mois le nombre de jours qu'il comporte.

```
# let nbjoursdansmois = function a -> function
    1  -> 31
  | 2  -> if bissextile a then 29 else 28
  | 3  -> 31
  | 4  -> 30
  | 5  -> 31
  | 6  -> 30
  | 7  -> 31
  | 8  -> 31
  | 9  -> 30
  | 10 -> 31
  | 11 -> 30
  | 12 -> 31
  | _  -> -1 ;;
```

Le calcul de la veille se fait en deux étapes. On calcule tout d'abord le jour de la semaine de la veille : il s'agit du jour de la semaine courant moins 1, sauf si on est un lundi (1), dans ce cas on passe au dimanche (7). On calcule ensuite la date de la veille. Il y a trois cas :
- si l'on est un premier janvier, on passe au 31 décembre de l'année précédente ;
- sinon si l'on est le premier jour d'un mois différent de janvier, on passe au dernier jour du mois précédent de la même année ;
- sinon on passe au jour précédent du même mois et de la même année.

```
# let  (veille: date -> date) = function d ->
          let thejour = if lejour d = 1 then 7 else lejour d - 1 in
      if jour d = 1 then
         if mois d = 1 then (thejour, 31, 12, annee d - 1)
         else (thejour, nbjoursdansmois (annee d) (mois d - 1),
               mois d - 1, annee d)
      else (thejour, jour d - 1, mois d, annee d) ;;
```

Pour le lendemain le raisonnement est du même type :
- si l'on est le 31 décembre, on passe au premier janvier de l'année suivante ;
- sinon si l'on est le dernier jour d'un mois, on passe au premier jour du mois suivant de la même année ;
- sinon on passe au jour suivant du même mois de la même année.

```
# let  (lendemain: date -> date) = function d ->
          let thejour = if lejour d = 7 then 1 else lejour d + 1 in
      if jour d = nbjoursdansmois (annee d) (mois d) then
         if mois d = 12 then (thejour, 1, 1, annee d + 1)
         else (thejour, 1, mois d + 1, annee d)
      else (thejour, jour d + 1, mois d, annee d) ;;
```

Pour tester l'égalité entre deux dates, il faut tester si le jour, le mois et l'année sont les mêmes pour les deux dates :

```
# let (meme_date: date -> date -> bool) =
    function d1 -> function d2 ->
    (annee d1 = annee d2) & (mois d1 = mois d2) & (jour d1 = jour d2) ;;
```

La manière la plus simple pour calculer le nombre de jours entre deux dates est de partir de la date la plus ancienne et de l'incrémenter d'un jour jusqu'à arriver à la seconde date. On compte le nombre de jours qui ont été ajoutés et on obtient la différence en jours entre ces deux dates.

```
# let rec (nombredejours: date -> date -> int) =
```

```
     function d1 -> function d2 ->
     if meme_date d1 d2
     then 0
     else
            let dat1 = if chronologie d1 d2 then d1 else d2 in
            let dat2 = if dat1 = d1 then d2 else d1 in
            1 + nombredejours (lendemain dat1) dat2 ;;
```

Afin de déterminer le prochain vendredi 13 à partir d'une date, on teste toutes les dates à partir de cette date jusqu'à tomber sur un vendredi 13.

```
# let rec (vendredi13: date -> date) = function d ->
    if (lejour d = 6) & (jour d = 13) then d
    else vendredi13 (lendemain d) ;;
```

Pour déterminer le jour de la semaine d'une certaine date, il faut utiliser une date de référence don't on connaît le jour de la semaine (par exemple aujourd'hui). Ensuite on décrémente cette date jusqu'à atteindre la date anniversaire.

```
# let rec (journaissance: date -> date -> int) =
     function d1 -> function d2 ->
     if meme_date d1 d2
     then lejour d1
     else journaissance (veille d1) d2 ;;
```

Exercice 14

Définir mutuellement les fonctions pair et impair qui testent respectivement si un nombre positif est pair ou impair.

Réponse :

On utilise la récursivité mutuelle : un nombre n positif est pair si (n-1) est impair. La condition d'arrêt est : 0 est pair.

```
# let rec pair x = match x with
 0 -> true
|_ -> impair (x-1)
and impair x = match x with
 0 -> false
|_ -> pair (x-1);;
```

LES LISTES

Exercice 1

Concevoir et coder en Caml les fonction suivantes sur les listes : test de vacuité, extraction de la tête et du reste d'une liste.

Réponse :

```
# let vide = function
     [] -> true
   | _  -> false;;

# let tete = function
  t::r -> t
 |_ -> failwith "tete";;

# let reste = function
  t::r -> r
 |_ -> failwith "reste";;
```

Exercice 2

Concevoir et coder en Caml une fonction qui effectue la somme des éléments d'une liste d'entiers.

Réponse :

On additionne de manière récursive tous les éléments de la liste.

```
# let rec somme = function
 [] -> 0
| t::r -> t + somme r;;
```

Exercice 3

Concevoir et coder en Caml une fonction qui détermine la longueur d'un liste.

Réponse :

On extrait les éléments un par un jusqu'à ce que la liste soit vide et on les compte.

```
# let rec long = function
      [] -> 0
|t::r -> 1 + long r;;
```

Exercice 4

Concevoir et coder en Caml une fonction qui effectue la concaténation de 2 listes.

Réponse :

On ajoute récursivement les éléments de la seconde liste en queue de la première.

```
# let rec concat l1 l2 =
  match (l1,l2) with
      ([],l2) -> l2
      |(x::r1,l2) -> x::concat r1 l2;;
```

Exercice 5

Concevoir et coder en Caml une fonction qui teste si deux listes ont la même longueur.

Réponse :

On supprime un élément des deux listes simultanément. Si elles sont vides au même moment, c'est qu'elles ont la même longueur.

```
# let rec meme_long l1 l2 =
  match (l1,l2) with
      ([],[]) -> true
      |(_::r1,_::r2) -> meme_long r1 r2
      |_ -> false;;
```

Exercice 6

Concevoir et coder en Caml une fonction qui teste l'égalité entre deux listes.

Réponse :

On compare un à un les éléments des deux listes tant qu'ils sont égaux.

```
# let rec meme_list l1 l2=
 match (l1,l2) with
      ([],[]) -> true
      |(x1::r1,x2::r2) -> if x1=x2 then meme_list r1 r2 else false
      |_ -> false;;
```

Exercice 7

Concevoir et coder en Caml une fonction qui renverse une liste.

Réponse :

On extrait la tête de la liste et on l'insère en queue de la liste résultat.

```
# let rec rev = function
      [] -> []
|x::r -> concat (rev r) [x];;
```

Exercice 8

Concevoir et coder en Caml une fonction qui teste l'appartenance d'un élément à une liste.

Réponse :

On extrait les éléments un à un et on teste si on trouve l'élément recherché.

```
# let rec membre x= function
      [] -> false
|t::r -> if x=t then true else membre x r;;
```

Exercice 9

Concevoir et coder en Caml une fonction qui supprime un élément (une fois) dans une liste.

Réponse :

On recherche l'élément et on reconstruit la liste sans l'inclure.

```
# let rec suppr x= function
      [] -> []
|t::r -> if x=t then r else t::(suppr x r);;
```

Exercice 10

Concevoir et coder en Caml une fonction qui teste si une liste est une permutation d'une autre liste.

Réponse :

On extrait les éléments de la première liste et on les recherche dans la seconde. S'il s'y trouve, on le supprime et on continue.

```
# let rec perm l1 l2=
  match (l1,l2) with
      ([],[]) -> true
      |(x::r1,l2) -> if membre x l2 then perm r1 (suppr x l2) else false
      |_ -> false;;
```

Exercice 11

Concevoir et coder en Caml les opérations ensemblistes suivantes sur des listes : union, intersection, différence, XOR.

Réponse :

Pour les trois opérations d'union, d'intersection et de différence, on extrait les éléments de la première liste, on teste leur appartenance à la seconde liste, et on construit le résultat désiré.

```
# let rec union l1 l2=
 match (l1,l2) with
      ([],l2) -> l2
      |(x::r1,l2) -> if membre x l2 then union r1 l2 else x::(union r1 l2);;

# let rec inter l1 l2=
 match (l1,l2) with
      ([],_) -> []
      |(x::r1,l2) -> if membre x l2 then x::(inter r1 l2) else inter r1 l2;;

# let rec diff l1 l2=
 match (l1,l2) with
      ([],_) -> []
      |(x::r1,l2) -> if membre x l2 then diff r1 l2 else x::(diff r1 l2);;

# let xor l1 l2= diff (union l1 l2) (inter l1 l2);;
```

Exercice 12

Concevoir et coder en Caml une fonction qui renvoie le nombre de multiples de 5 dans une liste.

Réponse :

Idem à la fonction qui donne la longueur d'une liste sauf qu'on ne compte que les multiples de 5.

```
# let rec compte_liste = function
      [] -> 0
|x::l -> if (x mod 5) = 0 then 1 + compte_liste l
                   else compte_liste l;;
```

Exercice 13

Concevoir et coder en Caml une fonction qui supprime tous les multiples de 5 dans une liste.

Réponse :

On teste tous les éléments de la liste et on reconstruit la liste résultat en « oubliant » les multiples de 5.

```
# let rec suppr_liste = function
      [] -> []
|x::l -> if (x mod 5) = 0 then suppr_liste l else x::suppr_liste l;;
```

Exercice 14

Même fonction mais en passant le test en argument : abstraction de la condition.

Réponse :

Idem sauf que le test est passé en argument.

```
# let rec suppri_liste test = function
      [] -> []
|x::l -> if (test x) then suppri_liste test l else x::suppri_liste test l;;

# suppri_liste (function a -> a mod 5=0)  [5;5;10;1000;4;5;0] ;;
```

Exercice 15

Que fait la fonction suivante ?

```
# let rec merp = function
      [] -> true
 |l -> let ll=suppri_liste(function a -> a=list_length l) l in
      (list_length ll=list_length l-1)&(merp ll);;
```

Réponse :

Elle teste si une liste est de la forme [1 ; 2 ; 3 ; 4 ; … ; n].

Exemple :

```
# merp [1 ; 2 ; 3] ;;
- : bool = true
# merp [1 ; 3 ; 4] ;;
- : bool = false
```

Exercice 16

Concevoir et coder en Caml les fonction une fonction de tri d'une liste.

Réponse :

Définissons d'abord une fonction qui recherche le plus petit élément d'une liste :

```
# let rec lepluspetit = function
      []   -> failwith "lepluspetit"
    | [a]  -> a
    | t::r -> if t < lepluspetit r then t else lepluspetit r;;

# let rec lepluspetit = function
      []   -> failwith "lepluspetit"
    | [a]  -> a
    | t::r -> let pt = lepluspetit r in
                  if t < pt then t else pt;;
```

La fonction de tri se définit alors de la manière suivante : on recherche le plus petit élément de la liste et on l'insère en tête de la liste résultat. On continue tant que la liste n'est pas vide.

```
# let rec tri = function
      []   -> []
    | l    -> let pt = lepluspetit l in
                  pt::tri(suppr pt l);;
```

LES DESSINS RÉCURSIFS

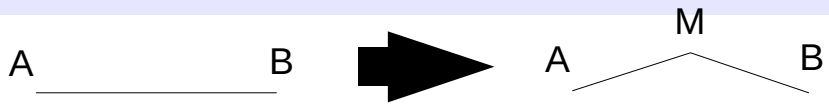
Explications des fonctions à utiliser

Nous considérons un dessin comme étant une liste de points reliés deux à deux par un segment. Construire un dessin de manière récursive revient donc à construire une liste de l'ensemble des points de ce dessin. Pour afficher un résultat, il faut utiliser les fonctions `dessin` et `dessinF` définies ci-dessous :

```
# #open "graphics";;

(* Tracé d'un segment à coordonnées entières *)
# let trace_segment (x1,y1) (x2, y2) =
          moveto x1 y1 ;
          lineto x2 y2 ;;

(* Tracé d'un segment à coordonnées réelles *)
# let trace_segmentF (x1,y1) (x2, y2) =
          moveto (int_of_float (x1 +. 0.5)) (int_of_float (y1 +. 0.5)) ;
          lineto (int_of_float (x2 +. 0.5)) (int_of_float (y2 +. 0.5)) ;;

(* Tracé de l'ensemble des segments d'une liste de points à coordonnées
entières *)
# let rec afficher_liste = function
          [(p1,p2)]  -> trace_segment p1 p2
          | (p1,p2)::r -> (trace_segment p1 p2) ; afficher_liste r ;;

(* Tracé de l'ensemble des segments d'une liste de points à coordonnées
réelles *)
# let rec afficher_listeF = function
          [(p1,p2)]  -> trace_segmentF p1 p2
          | (p1,p2)::r -> (trace_segmentF p1 p2) ; afficher_listeF r ;;

(* Fonction affichant le dessin d'une liste de points à coordonnées entières
*)
# let dessin l =
        open_graph"";
        clear_graph();
        afficher_liste l;
        let k = read_key() in if k=`q` then close_graph();;

(* Fonction affichant le dessin d'une liste de points à coordonnées réelles *)
# let dessinF l =
        open_graph"";
        clear_graph();
        afficher_listeF l;
        let k = read_key() in if k=`q` then close_graph();;
```

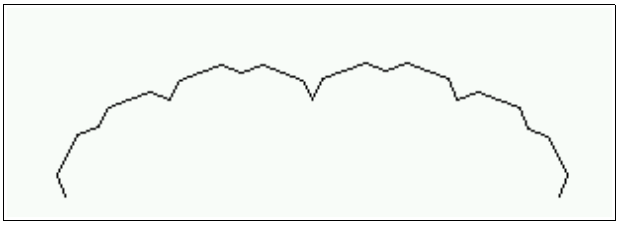
Exercice 1

Il s'agit de créer un dessin récursif de la manière suivante. A partir de 2 points A et B, on construit un nouveau point M comme le montre le schéma ci-dessous. On continue ainsi de manière récursive avec les nouveaux couples de points (A, M) et (M, B). L'altitude du point M par rapport aux points A et B est calculé proportionnellement à la différence d'altitude entre les deux points A et B. Le rapport de proportionnalité est donné en paramètre.



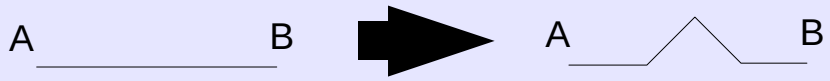
Réponse :

```
# let nouveau_point1 (x1,y1) (x2, y2) alpha =
                let xx1 = float_of_int x1
                and xx2 = float_of_int x2
                and yy1 = float_of_int y1
                and yy2 = float_of_int y2
                in
                let xxi = (xx1+.xx2)/.2. -. (yy2-.yy1)*.alpha
                and yyi = (yy1+.yy2)/.2. +. (xx2-.xx1)*.alpha
                in
                let xi = int_of_float (xxi +. 0.5)
                and yi = int_of_float (yyi +. 0.5)
                in (xi, yi);;

# let rec motif1 p1 p2 niv nivmax alpha=
        if niv >= nivmax then [(p1, p2)]
        else
            let m = nouveau_point1 p1 p2 alpha
            in  (motif1 p1 m (niv+1) nivmax alpha)@(motif1 m p2 (niv+1) nivmax
alpha);;

# let l = motif1 (50,10) (300,10) 1 6 0.2 ;;
# dessin l ;;
```



Exercice 2

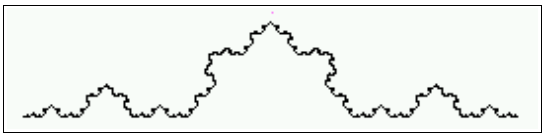
Flocon : On définit cette fois à partir de deux points A et B, trois nouveaux points comme le montre la figure ci-dessous. A nouveau le dessin est construit récursivement à partir des nouveaux segments générés. Comme précédemment la hauteur est donnée par un paramètre.



Réponse :

```
# let nouveau_point3 (x1,y1) (x2, y2) alpha =
                let xx1 = float_of_int x1
                and xx2 = float_of_int x2
                and yy1 = float_of_int y1
                and yy2 = float_of_int y2
                in
                let xxi1 = (2.*.xx1+.xx2)/.3.
                and yyi1 = (2.*.yy1+.yy2)/.3.
                and xxi2 = (xx1+.2.*.xx2)/.3.
                and yyi2 = (yy1+.2.*.yy2)/.3.
                and xxi3 = (xx1+.xx2)/.2. -. (yy2-.yy1)*.alpha
                and yyi3 = (yy1+.yy2)/.2. +. (xx2-.xx1)*.alpha
                in
```

```
                let xi1 = int_of_float (xxi1 +. 0.5)
                and yi1 = int_of_float (yyi1 +. 0.5)
                and xi2 = int_of_float (xxi2 +. 0.5)
                and yi2 = int_of_float (yyi2 +. 0.5)
                and xi3 = int_of_float (xxi3 +. 0.5)
                and yi3 = int_of_float (yyi3 +. 0.5)
                in ((xi1,yi1),(xi2,yi2),(xi3,yi3)) ;;

# let rec flocon p1 p2  nivmax alpha =
        if  nivmax = 0 then [(p1, p2)]
        else
            let (m1,m3,m2) = nouveau_point3 p1 p2 alpha
            in  (flocon p1 m1  (nivmax-1) alpha)@(flocon m1 m2  (nivmax-1)
alpha)@(flocon m2 m3  (nivmax-1) alpha)@(flocon m3 p2 (nivmax-1) alpha);;

# let l = flocon (50,10) (300,10)  6 0.2 ;;
# dessin l ;;
```



```
# let l1 = flocon (450,100) (50,100)   5 0.3 ;;
# let l2 = flocon (50,100)  (250,450)  5 0.3 ;;
# let l3 = flocon (250,450) (450,100)  5 0.3 ;;
# dessin (l1@l2@l3) ;;
```

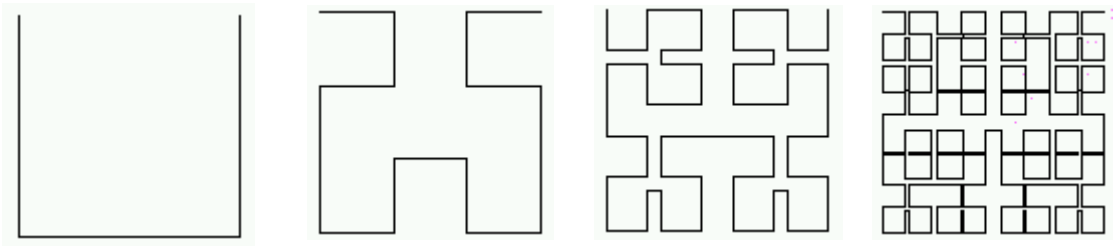
Exercice 3

Hilbert : l'idée est de remplacer chaque motif de la forme l1 par le motif l2. Ensuite chaque motif en forme de U dans l2 est lui-même remplacé par le motif l2, et ainsi de suite récursivement. Il faut donc à partir de quatre points donnés (motif l1) construire douze nouveaux points (motifs l2).

Réponse :

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

# let nouveau_point12 (x1,y1) (x2,y2) (x3,y3) (x4,y4) niv =
    let c2 = 1.0 /. float_of_int ((puissance 2 (niv)) - 1 )
    in  let c1 = 0.5 *.(1.0 -. c2 )
    in  let vx1 = c1 *. (x4-.x1)
        and vy1 = c1 *. (y4-.y1)
        and vx2 = c1 *. (x2-.x1)
        and vy2 = c1 *. (y2-.y1)
        and wx1 = (c1+.c2) *. (x4-.x1)
        and wy1 = (c1+.c2) *. (y4-.y1)
        and wx2 = (c1+.c2) *. (x2-.x1)
        and wy2 = (c1+.c2) *. (y2-.y1)
        in let mx1 = x1+.vx1 and my1 = y1+.vy1
           and mx2 = x1+.wx1 and my2 = y1+.wy1
       in  ( (mx1,my1),           (mx1+.vx2,my1+.vy2),
(x1+.vx2,y1+.vy2),
             (x1+.wx2,y1+.wy2),   (mx1+.(x2-.x1),my1+.(y2-.y1)),
(mx1+.wx2,my1+.wy2),
             (mx2+.wx2,my2+.wy2), (mx2+.(x2-.x1),my2+.(y2-.y1)),
(x4+.wx2,y4+.wy2),
             (x4+.vx2,y4+.vy2),   (mx2+.vx2,my2+.vy2),           (mx2,my2));;
```

```
# let rec hilbert p1 p2 p3 p4 niv nivmax =
        if niv >= nivmax then [(p1, p2);(p2,p3);(p3,p4)]
        else
            let (m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,m11,m12) = nouveau_point12 p1
p2 p3 p4 nivmax
            in (hilbert m3 m2 m1 p1 (niv+1) nivmax)@[(m3,m4)]@(hilbert m4 p2
m5 m6 (niv+1) nivmax)@[(m6,m7)]@
                (hilbert m7 m8 p3 m9 (niv+1) nivmax)@[(m9,m10)]@(hilbert p4 m12
m11 m10 (niv+1) nivmax);;

# let l1 = hilbert (110.,220.) (110.,110.) (220.,110.) (220.,220.) 1 1 ;;
# dessinF l1 ;;

# let l2 = hilbert (110.,220.) (110.,110.) (220.,110.) (220.,220.) 1 2 ;;
# dessinF l2 ;;

# let l3 = hilbert (110.,220.) (110.,110.) (220.,110.) (220.,220.) 1 3 ;;
# dessinF l3 ;;

# let l4 = hilbert (110.,220.) (110.,110.) (220.,110.) (220.,220.) 1 4 ;;
# dessinF l4 ;;
```



l1 l2 l3 l4

Exercice 4

Algorithme de De Casteljau. Cet algorithme permet de calculer de manière récursive les points d'une courbe en fonction de plusieurs points de contrôle. Soit P_1, ..., P_n les points de contrôle de la courbe à calculer. Le point de la courbe, de paramètre u (réel compris entre 0 et 1), se calcule récursivement en interpolant deux à deux les points de contrôle en fonction de u comme le montre la figure ci-contre (u=0.5 sur la figure).

```
# type point == float*float;;

Ecrivons tout d'abord une fonction calculant le point P d'interpolation de 2
points P1 et P2, avec un paramètre u réel compris entre 0 et 1. Si u=0 alors
P=P1 et si u=1 alors P=P2.
# let (interpole :point->point->float->point) =
            function (x1,y1)-> function  (x2,y2)->function u ->
            (x1*.(1.-.u) +. x2*.u , y1*.(1.-.u) +. y2*.u);;

Nous pouvons maintenant définir une fonction interpolant 2 à 2 les points
d'une liste. La liste est initialement composée de n points, et après
interpolation, il reste n-1 points.
# let rec interpole_liste u = function
  [p1] -> []
| p1::(p2::r) -> (interpole p1 p2 u)::(interpole_liste u (p2::r));;
```

Pour calculer la position d'un point par l'algorithme de De Casteljau pour un paramètre u donné, il suffit d'interpoler récursivement la liste des points de départs avec le paramètre u jusqu'à ce qu'elle ne contienne plus qu'un seul point.

```
# let rec casteljau u = function
  [p] -> p
| l -> casteljau u (interpole_liste u l);;
```

Pour le calcul de l'ensemble des points de la courbe nous définissons une fonction ayant 4 paramètres : l = la liste des points de contrôle, [u,u1] = l'espace des paramètres et up = l'incrément du paramètre.

```
# let rec calc l u u1 up = if (u=u1) then [casteljau u l]
                    else (casteljau u l)::(calc l (u +. up) u1 up);;
```

Nous pouvons aussi définir une fonction calculant la courbe en fonction d'un nombre de points demandé.

```
# let Calc_courb l n v0 v1 =
         calc l v0 v1 ((v1-.v0) /. (float_of_int n)) ;;
```

LES ANNALES

LE PETIT GUIDE UNIX

Modification de l'identité

chfn

```
$ chfn
Changing finger information for wemmert
Password:
Name [ ]: WEMMERT Cedric
Office [ ]:
Office Phone [ ]:
Home Phone [ ]:

Finger information changed.
$
```

Changement de mot de passe

passwd

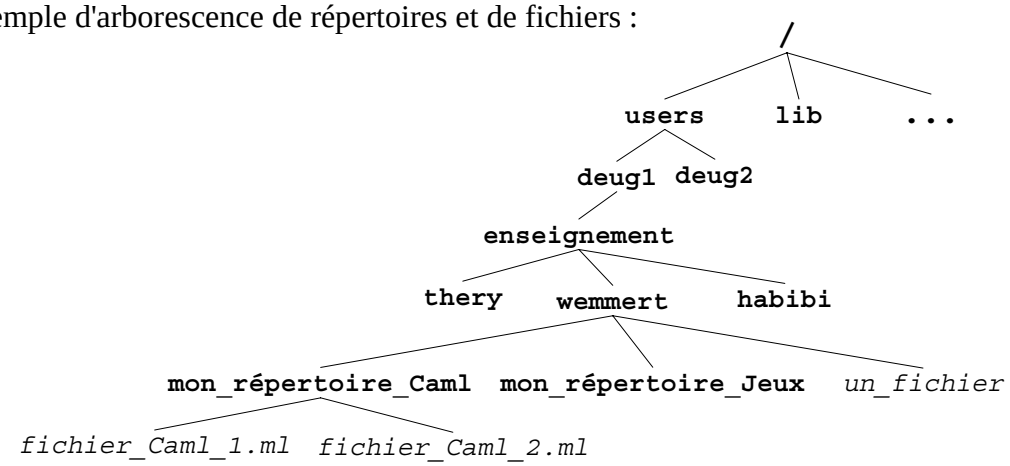
```
$ passwd
Changing password for wemmert
Old password:
New password:
Re-enter new password:
$
```

Qui suis-je ???

whoami

```
$ whoami
wemmert
$
```

Exemple d'arborescence de répertoires et de fichiers :



Référence du répertoire de travail

pwd

```
$ pwd
/users/deug1/enseignement/wemmert
```

Contenu d'un répertoire

ls [-l]

```
$ ls
mon_répertoire_Caml    mon_répertoire_Jeux    un_fichier
$ ls -l
drwxr-xr-x  2  wemmert  users      512   Nov 17 17:45  mon_répertoire_Caml
drwxr-xr-x  2  wemmert  users      512   Jan 23 10:12  mon_répertoire_Jeux
-rw-r--r--  2  wemmert  users      120   Nov 17 17:57  un_fichier
```

Changement de répertoire de travail

cd *nom du répertoire destination*

```
$ pwd
/users/deug1/enseignement/wemmert
$ ls -l
drwxr-xr-x  2  wemmert  users      512   Nov 17 17:45  mon_répertoire_Caml
drwxr-xr-x  2  wemmert  users      512   Jan 23 10:12  mon_répertoire_Jeux
-rw-r--r--  2  wemmert  users      120   Nov 17 17:57  un_fichier
$ cd mon_répertoire_Caml
$ ls
fichier_Caml_1.ml             fichier_Caml_2.ml
$ cd ../../thery
$ pwd
/users/deug1/enseignement/thery
```

Copie d'un fichier

cp *source destination*

```
$ ls
mon_répertoire_Caml    mon_répertoire_Jeux    un_fichier
$ cp un_fichier mon_répertoire_Caml
$ ls mon_répertoire_Caml
fichier_Caml_1.ml             fichier_Caml_2.ml             un_fichier
```

Déplacement d'un fichier

mv *source destination*

```
$ ls
mon_répertoire_Caml    mon_répertoire_Jeux    un_fichier
$ mv un_fichier mon_fichier
$ ls
mon_répertoire_Caml    mon_répertoire_Jeux    mon_fichier
$ mv mon_fichier mon_répertoire_Caml/mon_fichier.ml
$ ls mon_répertoire_Caml
fichier_Caml_1.ml             fichier_Caml_2.ml             un_fichier     mon_fichier.ml
$ ls
mon_répertoire_Caml    mon_répertoire_Jeux
```

Suppression d'un fichier

rm *nom du fichier à supprimer*

```
$ ls
fichier_Caml_1.ml             fichier_Caml_2.ml             un_fichier     mon_fichier.ml
$ rm un_fichier
$ ls
fichier_Caml_1.ml             fichier_Caml_2.ml             mon_fichier.ml
```

Création d'un répertoire

mkdir *nom du nouveau répertoire*

```
   $ ls
   mon_répertoire_Caml    mon_répertoire_Jeux    un_fichier
   $ mkdir un_nouveau_répertoire
   $ ls
   mon_répertoire_Caml    mon_répertoire_Jeux    un_nouveau_répertoire
```

Suppression d'un répertoire

rmdir *nom du répertoire à supprimer*

```
   $ ls
   mon_répertoire_Caml    mon_répertoire_Jeux    un_nouveau_répertoire
   $ rmdir un_nouveau_répertoire
   $ ls
   mon_répertoire_Caml    mon_répertoire_Jeux
   $ rmdir mon_répertoire_Caml
rmdir: mon_répertoire_Caml: Directory not empty
```