Corrigé de la feuille d'exercices n°1.

1. a) if 2 > 3 then 4;;
Le résultat est :

```
#if 2 > 3 then 4;;
Entrée interactive:
>if 2 > 3 then 4;;
>              ^
Cette expression est de type int,
mais est utilisée avec le type unit.
```

En Caml le «`else`» est obligatoire et le résultat obtenu avec le `then` doit être du même type que le résultat obtenu par le `else` sauf si on quitte la fonction avec un message d'erreur ou bien si le résultat du «`else`» est de type `unit`.
b) 1>2 and 5>3;;
L'opérateur booléen «et» est & et non `and`.

```
#1>2 and 5>3;;
Entrée interactive:
>1>2 and 5>3;;
>    ^^^
Erreur de syntaxe.
```

L'expression correcte est la suivante :

```
#1>2 & 5>3;;
- : bool = false
```

c) 3.5 + 4.0;;

```
#3.5 + 4.0;;
Entrée interactive:
>3.5 + 4.0;;
>^^^
Cette expression est de type float,
mais est utilisée avec le type int.
```

L'opérateur d'addition sur les réels est «+.» et non «+».

```
#3.5 +. 4.0;;
- : float = 7.5
```

d) 1 + 4.0;;
L'expression est incorrecte car on ne peut pas additionner un entier et un réel.
e) 8. /. 3.;;
L'expression est correcte et le résultat est de type réel.

```
#8. /. 3.;;
- : float = 2.66666666667
```

f) 8 / 3;;
L'expression est correcte et le résultat qui est de type entier, est le quotient de la division euclidien de 8 par 3.

```
#8 / 3;;
- : int = 2
```

g) 1.0 < 2.0 or 3>4;;
L'expression est correcte (on rappelle au passage que les opérateurs de comparaison sont polymorphes).

```
#1.0 < 2.0 or 3>4;;
- : bool = true
```

2. Fonction «valeur absolue» définie sur les entiers :

```
#let val_abs_int n =
if n >= 0
  then n
  else -n;;
val_abs_int : int -> int = <fun>
```

Fonction «valeur absolue» définie sur les réels :

```
#let val_abs_float n =
if n >= 0.
  then n
  else -. n;;
val_abs_float : float -> float = <fun>
```

3. Fonction «signe» sur les entiers.

```
#let sgn_int n =
if n > 0
  then 1
  else if n < 0 then -1
              else 0;;
sgn_int : int -> int = <fun>
```

Fonction «signe» sur les réels.

```
#let sgn_float n =
if n >. 0.
  then 1
  else if n <. 0. then -1
                else 0;;
sgn_float : float -> int = <fun>
```

ou bien

```
#let sgn_float n =
if n >. 0.
  then 1.
  else if n <. 0. then -. 1.
                  else 0.;;
sgn_float : float -> float = <fun>
```

suivant le choix de l'ensemble d'arrivée.

4.

```
#let solve (a,b,c) =
let delta = b*.b -. 4. *. a *. c in
if a = 0.
   then failwith "Erreur : le polynôme n'est pas de degré 2"
   else
     if delta < 0.
        then failwith "Erreur : discriminant strictement négatif"
        else ((-.b +. sqrt(delta))/.(2. *.a),(-.b -. sqrt(delta))/.(2.*.a))
;;
solve : float * float * float -> float * float = <fun>
#solve (1., -.3., 2.);;
- : float * float = 2.0, 1.0
```

5.

```
#let somme n =
let s = ref 0 in
if n < 0
   then failwith "Erreur : entier négatif"
   else for i = 1 to n do s := !s + i done;
        !s;;
somme : int -> int ref = <fun>
```

6.

```
#let somme_carré n =
let s = ref 0 in
if n < 0
   then failwith "Erreur : entier négatif"
   else for i = 1 to n do s := !s + i*i done;
        !s;;
somme_carré : int -> int ref = <fun>
```

7.

```
#let fact n =
let p = ref 1 in
if n < 0
   then failwith "Erreur : entier négatif"
   else for i = 1 to n do p := !p * i done;
        !p;;
fact : int -> int = <fun>
#fact 10;;
- : int = 3628800
#fact 30;;
- : int = -738197504
```

Le dernier calcul est là pour rappeler que les opérations sur les entiers sont faites dans $\mathbb{Z}\,|_{2^{32}\mathbb{Z}}$.

8.

```
#let puiss x n =
let r = ref 1. in
if n >= 0
  then
    for i=1 to n do
        r := !r *. x
        done
  else
    for i=1 to abs n do
        r := !r /. x
        done;
!r;;
puiss : float -> int -> float = <fun>
#puiss 2. 10;;
- : float = 1024.0
#puiss 2. (-10);;
- : float = 0.0009765625
```

9.

```
#let racine x epsilon =
let r = ref 1. in
while abs_float(!r -. x /. !r) > epsilon do
    r := (!r +. x /. !r) /. 2.
    done;
!r;;
racine : float -> float -> float = <fun>
#racine 2. 0.00001;;
- : float = 1.41421568627
```

10.

```
#let symetrie n =
let a = ref n and b = ref 0 in
while !a <> 0 do
  b := 10 * !b + !a mod 10;
  a := !a / 10;
  done;
!b;;
symetrie : int -> int = <fun>
#symetrie 123456;;
- : int = 654321
```

11.

```
#let max t =
let long = vect_length t in
if long = 0
   then
      failwith "Erreur : tableau vide"
   else
      let m = ref t.(0) in
        for i=1 to (long-1) do
            if t.(i) > !m then m := t.(i)
        done;
!m;;
max : 'a vect -> 'a = <fun>
#max [|5;6;1;2;7;-1|];;
- : int = 7
#max [|5.8;6.4;1.6;2.5;7.;-10.;11.|];;
- : float = 11.0
```

12. Les polynômes sont représentés par des tableaux de réels $P = a_0 + a_1X + \ldots + a_nX^n$ est représenté par $[||a_0; a_1; \ldots; a_n|]$.
«imprime_monone» imprime un monôme donné par son coefficient et par son degré.

```
#let imprime_monone coeff degre =
    if degre = 0 then print_float coeff else
    if coeff <> 0.0 then
        begin
           if coeff > 0.0
             then
                begin
                print_string " + ";
                if coeff<>1.0 then
                              begin print_float coeff; print_string " *" end;
                print_string " x";
                if degre <> 1 then begin print_string "^"; print_int degre end
                end
             else
                begin
```

```
                print_string " - ";
                if coeff <> (-. 1.) then
                                    begin print_float (abs_float coeff);
                                    print_string " *" end;
                print_string " x";
                if degre <> 1 then begin print_string "^"; print_int degre end
                end
      end;;
imprime_monone : float -> int -> unit = <fun>

#let affiche p1 =
      for i = 0 to vect_length p1 - 1 do imprime_monone p1.(i) i done;
       print_string " ";;
affiche : float vect -> unit = <fun>
```

«somme p1 p2» calcule la somme de deux polynômes $p1$ et $p2$.

```
#let somme p1 p2 =
     let somme = make_vect (max (vect_length p1) (vect_length p2) ) 0.0 in
            for i = 0 to vect_length p1 - 1 do somme.(i) <- p1.(i) done;
            for i = 0 to vect_length p2 - 1 do somme.(i) <- somme.(i) +. p2.(i) done;
     somme;;
somme : float vect -> float vect -> float vect = <fun>
```

Différence de deux polynômes : calcule $(p1 - p2)$ *)

```
#let difference p1 p2 =
   let somme = make_vect (max (vect_length p1) (vect_length p2) ) 0.0 in
       for i = 0 to vect_length p1 - 1 do somme.(i) <- p1.(i) done;
       for i = 0 to vect_length p2 - 1 do somme.(i) <- somme.(i) -. p2.(i) done;
   somme;;
difference : float vect -> float vect -> float vect = <fun>
```

produit de deux polynômes :

```
#let produit p1 p2 =
      let degre1 = (vect_length p1 -1) and
           degre2 = (vect_length p2 - 1)  in
               let produit = make_vect  (degre1 + degre2 + 1) 0.0 in
      for i = 0 to degre1 do
            for j = 0 to degre2 do
                  produit.(i+j) <- produit.(i+j) +. p1.(i) *. p2.(j)
                                        done
                                done;
     produit;;
produit : float vect -> float vect -> float vect = <fun>
```

Produit d'un polynome P par un scalaire x. On multiplie tous les coefficients de P par x.

```
#let scalproduit x p =
let p1 = make_vect (vect_length p) 0.0 in
      for i = 0 to (vect_length p - 1) do
```

```
            p1.(i) <- x *. p.(i) done;
      p1;;
scalproduit : float -> float vect -> float vect = <fun>
```

Simplification d'un polynôme : supprime les zéros des termes de plus haut degré

```
#let rec simplifie p =
let degre = vect_length p -1 in
if (degre = 0) or (p.(degre) <> 0.0)
    then p
    else begin
           let p1 = make_vect degre 0.0 in
                for i = 0 to degre - 1 do
                      p1.(i) <- p.(i) done;
                simplifie p1
            end;;
simplifie : float vect -> float vect = <fun>
```

«derive» retourne le polynome dérivé de P

```
#let derive p =
    let degre = vect_length p - 1 in
        let res = make_vect degre 0.0 in
            for i = 0 to degre-1 do
                res.(i) <- p.(i+1) *. (float_of_int (i+1))
                done;
            res;;
derive : float vect -> float vect = <fun>
```

«derive P n» retourne la dérivée n-ième du polynome P.

```
#let derive_nième P n =
    let Q = ref(copy_vect P) in
        for i = 1 to n do
            Q := derive !Q
            done;
    simplifie (!Q);;
derive_nième : float vect -> int -> float vect = <fun>
```

La fonction «eval» permet de calculer $p(x)$ la valeur du polynôme p au point x d'après l'algorithme de Hörner.

```
#let eval p x =
    let res = ref(0.) and degre = vect_length p - 1 in
        for i = degre downto 0 do res := !res *. x +. p.(i) done;
    !res;;
eval : float vect -> float -> float = <fun>
```

«div_eucl» retourne le couple (Q, R) où :

Q est le quotient de la division euclidienne de A par B

R est le reste de la division euclidienne de A par B

```
#let div_eucl a b =
```

```
    let degA = vect_length a - 1 and degB = vect_length b - 1 in
        if degA < degB
           then ([|0.0|],a)
           else begin
               let degQ = degA - degB in
               let Q = make_vect (degQ+1) 0.0 and R=ref(a) and b_n = b.(degB) in
                    for i = degA downto degB do
                        let x = !R.(i) /. b_n in
                        Q.(i-degB) <- x;
                        for j = 0 to degB do
                            !R.(i+j-degB) <- !R.(i+j-degB) -. b.(j) *. x
                        done;
                    done;
               (Q,simplifie(!R));
               end;;
div_eucl : float vect -> float vect -> float vect * float vect = <fun>
```

`PGCD` de 2 polynômes
si $\deg(p1) < \deg(p2)$ on permutte $p1$ et $p2$
si $p2$ est le polynôme nul alors le `PGCD` est $p1$
si $p1 = a_n * x^n + \ldots$ et $p2 = b_p * x^p + \ldots$, on calcule :

$R = b_n * p1 - a_n * x^{n-p} * p2$ et on remarque que `PGCD`(p1,p2) = `PGCD`(p2,R)
à chaque étape on a : $\deg(R) < \deg(p2)$ ce qui nous assure qu'en un nombre fini d'étapes, on aura $P = 0$.

```
#let rec PGCD p1 p2 =
  let degp1 = vect_length p1 - 1      (* degré du polynôme p1 *)
 and degp2 = vect_length p2 - 1 in    (* degré du polynôme p2 *)
  if degp2 > degp1
      then
         PGCD p2 p1                         (* cas où on permute p1 et p2 *)
      else
         if p2 = [|0.0|]
             then p1                        (* nous avons le PGCD *)
             else                           (* voir explications ci dessus *)
                let Q = make_vect ((degp1 - degp2) + 1) 0.0 in
                  Q.(degp1 - degp2) <- 1.0;
                  let QP = produit Q (scalproduit p1.(degp1)  p2) in
                  let R = simplifie(difference (scalproduit p2.(degp2)  p1) QP)
                  in  PGCD  p2 R ;;
PGCD : float vect -> float vect -> float vect = <fun>
```