TP 1 : Corrigé

1 Déclaration, premières fonctions

Exercice 1. Calculez $\frac{1+\sqrt{2}+\sqrt{2}^3}{1+e^{\sqrt{2}}}$ à l'aide d'une affectation locale.

Corrigé.

```
let x=sqrt 2. in (1. +. x +. x**3.) /. (1. +. exp x) ;;
```

Exercice 2. Définir une fonction qui calcule le carré d'un entier.

Corrigé.

```
let f x = x*x ;;
```

Exercice 3. Définir des fonctions prenant en entrée une fonction $f : \mathbb{R} \to \mathbb{R}$ (dont le type est `float -> float =<fun>` a priori) et renvoyant :

- la valeur $\frac{f(0)+f(1)}{2}$,
- la fonction f^2 (carré de f).
- la fonction $f^{(2)}$, c'est à dire $x \mapsto f(f(x))$.
- la fonction $x \mapsto f(x+1)$.

C'est un bon exercice d'essayer de deviner le type de la fonction avant de l'écrire : il peut y avoir polymorphisme.

Corrigé. Les fonctions :

```
let g1 f = (f 0. +. f 1.) /. 2. ;;
let g2 f x = f (x *. x) ;;
let g3 f x = f (f x) ;;
let g4 f x = f (x +. 1.) ;;
```

Le type calculé par Caml :

```
#g1 : (float -> float) -> float = <fun>
#g2 : (float -> 'a) -> float -> 'a = <fun>
#g3 : ('a -> 'a) -> 'a -> 'a = <fun>
#g4 : (float -> 'a) -> float -> 'a = <fun>
```

Explications : lorsqu'il y a une opération flottante sur l'opérateur de la fonction (cas 1, 2, 4), le type d'entrée de `f` est nécessairement `float`. De même à l'arrivée (cas 1). Dans le cas 3, le type d'entrée de `f` est le même que le type de sortie puisqu'on compose `f` avec elle-même.

Exercice 4. *composition de fonctions*. Écrire une fonction `c` telle que `c f g` soit la fonction $f \circ g$. Avant cela donner son type. Objectif : 20 caractères.

```
#c (function x -> x+1) (function x-> x*x) 4 ;;
- : int = 17
```

Corrigé.

```
let c f g x=f(g x);;
```

`f` est `g` sont des fonctions, et le type d'arrivée de `g` est celui d'entrée de `f`. D'où avec `f` de type `'a -> 'b`, `g` est de type `'c -> 'a`, et `c` de type `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`. C'est exactement le type donné par Caml :

```
# c ;;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Exercice 5. *Complexes*. On rappelle que dans le cas particulier des couples (tuples de taille 2), il existe les fonctions `fst` et `snd` pour accéder aux composantes, mais elles ne se généralisent pas à de plus grands tuples. En assimilant un complexe à un couple de floats, écrire les fonctions de manipulation des complexes : partie réelle, partie imaginaire, conjugué, module. Puis rajoutez l'addition et la multiplication. Vous pouvez traiter l'argument également, le plus simple est d'utiliser que $\arg(x+iy) = 2\frac{\Im(z)}{\Re(z)+|z|}$, sauf si π est un réel négatif. Sinon, vous pourrez avoir besoin de $\pi = 4\arctan(1)$, qui s'obtient via `atan` en Caml. Remarque : 0 ne possède pas d'argument, on pourra utiliser `failwith "zero"` pour produire une erreur si le complexe passé en argument est zéro. Voici les types que j'obtiens :

```
pr : 'a * 'b -> 'a
pi : 'a * 'b -> 'b
cj : 'a * float -> 'a * float
modu : float * float -> float
add : float * float -> float * float -> float * float
mult : float * float -> float * float -> float * float
arg : float * float -> float
```

Pour la dernière fonction, on rappelle que `if cond then ... else ...` est la syntaxe d'une instructions conditionnelles, les deux blocs ayant le même type.

Corrigé. On donne parfois plusieurs solutions.

```
let pr z = fst z ;;
let pr (x,y) = x ;;
let pr z = let x,y = z in x ;; (* 3 façons différentes ! *)

let pi z = snd z ;;

let cj z = (pr z, -. pi z) ;;

let modu (x,y) = sqrt (x*. x+.y *. y) ;;

let add (x1,y1) (x2,y2)=
  x1 +. x2 , y1 +. y2 ;;
;;

let mult (x1,y1) (x2,y2)=
  x1 *. x2 -. y1 *. y2 , x1 *.y2 +. x2 *. y1 ;;
;;

let arg (x,y)=
  if x = 0. && y=0. then failwith "zero" else
    if x= 0. && y< 0. then -. 2. *. atan 1.
    else 2. *. y *. (x +. modu (x,y))
;;
```

2 Boucles et références

Exercice 6. *boucle for*. En utilisant une boucle `for`, définir une fonction `puiss` de calcul de puissance sur les entiers.

```
#puiss 3 7 ;;
- : int = 2187
```

Corrigé.

```
let puiss x n=
  let y=ref 1 in
  for i=0 to n-1 do
    y:= !y * x
  done ;
  !y
;;
```

Exercice 7. *boucle while*. En utilisant une boucle `while`, définir une fonction `sdc` qui effectue la somme des chiffres d'un entier. On utilisera des divisions euclidiennes par 10.

```
#sdc 4948353 ;;
- : int = 36
```

Corrigé.

```
let sdc x=
  let y=ref x and s=ref 0 in
  while !y>0 do
    s:= !s + !y mod 10 ;
    y:= !y / 10
  done ;
  !s
;;
```

Exercice 8. *autre boucle while*. Écrire une fonction `miroir` qui renvoie le « symétrique » d'un entier. Par exemple, le symétrique de 123 est 321.

```
#miroir 16163223 ;;
- : int = 32236161
```

Corrigé.

```
let miroir n=
  let s=ref 0 and x=ref n in
  while !x>0 do
    s:= 10 * !s + !x mod 10 ;
    x:= !x / 10
  done ;
  !s
;;
```

Exercice 9. Écrire une fonction `racine x epsilon` qui retourne $\sqrt{\texttt{x}}$ avec une précision de ε, en utilisant l'algorithme suivant, appelé algorithme de Babylone :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{x}{u_n}\right) \end{cases}$$

qui s'arrête lorsque $\left|u_n - \dfrac{x}{u_n}\right| < \varepsilon$. Cette fonction aura pour type `float -> float -> float = <fun>`. On pourra utiliser la fonction `abs_float`.

Corrigé.

```
let racine x epsilon =
  let y=ref x in
  while abs_float ( !y -. x/. !y) >= epsilon do
    y:= (!y +. x/. !y) /. 2.
  done ;
  !y
;;
```

3 Fonctions sur les tableaux

Exercice 10. Écrire une fonction faisant la somme des éléments d'un tableau d'entiers.

Corrigé.

```
let somme t=
  let s=ref 0 in
  for i=0 to Array.length t - 1 do
    s:= !s + t.(i)
  done ;
  !s
;;
```

Exercice 11. Écrire une fonction `inverse t` prenant en entrée un tableau `t` et produisant un nouveau tableau dont les éléments sont dans l'ordre inverse.

Corrigé. Attention à écrire une fonction polymorphe : ici on crée un tableau en se servant du premier élément de `t`, il faut que celui-ci soit non vide.

```
let inverse t=
  let n=Array.length t in
  let u=Array.make n t.(0) in
  for i=0 to n-1 do
    u.(i) <- t.(n-1-i)
  done ;
  u
;;
```

On peut aussi faire une copie :

```
let inverse t=
  let u=Array.copy t and n=Array.length t in
  for i=0 to n-1 do
    u.(i) <- t.(n-1-i)
  done ;
  u
;;
```

Exercice 12. Même question avec une fonction qui modifie le tableau passé en entrée, c'est-à-dire une fonction de type `'a array -> unit`.

Corrigé. Ici, on n'utilise pas de copie du tableau, tout se fait en place.

```
let inverse2 t=
  let n=Array.length t in
  for i=0 to n/2-1 do
    let a=t.(i) in t.(i) <- t.(n-1-i) ; t.(n-1-i) <- a
  done
;;
```

Exercice 13. Écrire une fonction de tri (en place) d'un tableau, de type `'a array -> unit`. Évidemment on utilisera pas `Array.sort`.

Corrigé. Par exemple, le tri par sélection du minimum.

```
let indice_min t i=
  (* indice du min a partir de l'indice i inclus *)
  let n=Array.length t and imin=ref i in
  for j=i+1 to n-1 do
    if t.(j) < t.( !imin) then imin:= j
  done ;
  !imin
;;

let tri_selection t=
  let n=Array.length t in
  for i=0 to n-2 do
    let imin=indice_min t i and a=t.(i) in
    t.(i) <- t.(imin) ; t.(imin) <- a
  done ;;
```