Programmation impérative et fonctionnelle : TP 4

1. Définition du type `expr` :

```
#type expr =
#    Float of float
#  | Var of string
#  | Som of expr*expr
#  | Prod of expr*expr
#  | Opp of expr
#  | Inv of expr ;;
```

```
#let e1 = Prod (Opp (Inv (Float 2.)), Som (Float 1., Prod (Float 3., Var "x")));;
val e1 : expr =
  Prod (Opp (Inv (Float 2.)), Som (Float 1., Prod (Float 3., Var "x")))
```

2. Fonction permettant d'évaluer une expression ne contenant pas de variables :

```
#let rec eval = fun e ->
#  match e with
#    Var _ -> failwith "variable inconnue"
#  | Float x -> x
#  | Som (x, y) -> eval x +. eval y
#  | Prod (x, y) -> eval x *. eval y
#  | Opp x -> 0. -. eval x
#  | Inv x -> 1. /. eval x;;
val eval : expr -> float = <fun>
```

```
#(* Test *)
#eval e1;;
Exception: Failure "variable inconnue".
```

```
#let e2 = Prod (Opp (Inv (Float 2.)), Som (Float 1., Prod (Float 3., Float 1.)));;
val e2 : expr =
  Prod (Opp (Inv (Float 2.)), Som (Float 1., Prod (Float 3., Float 1.)))
```

```
#eval e2;;
- : float = -2.
```

3. Modification de la fonction précédente pour pouvoir faire une substitution pour les variables :

```
#let rec eval2 = fun subst e ->
#  match e with
#    Var x -> subst x
#  | Float x -> x
#  | Som (x, y) -> eval2 subst x +. eval2 subst y
#  | Prod (x, y) -> eval2 subst x *. eval2 subst y
#  | Opp x -> 0. -. eval2 subst x
#  | Inv x -> 1. /. eval2 subst x;;
val eval2 : (string -> float) -> expr -> float = <fun>
```

```
#let f = fun v -> match v with "x" -> 1. | _ -> failwith "variable inconnue";;
val f : string -> float = <fun>
```

```
#eval2 f e1;;
- : float = -2.
```

4. Fonction qui dérive une expression par rapport à une variable donnée :

```
#let rec derive = fun e var ->
#  match e with
#  | Float _ -> Float 0.
```

```
#  | Var v -> if v = var then Float 1. else Float 0.
#  | Som (x, y) -> Som (derive x var, derive y var)
#  | Prod (x, y) -> Som (Prod (derive x var, y), Prod (x, derive y var))
#  | Opp x -> Opp (derive x var)
#  | Inv x -> Opp (Prod (derive x var, Inv (Prod (x, x))));;
val derive : expr -> string -> expr = <fun>

#derive e1 "x";;
- : expr =
Som
 (Prod (Opp (Opp (Prod (Float 0., Inv (Prod (Float 2., Float 2.))))),
   Som (Float 1., Prod (Float 3., Var "x"))),
 Prod (Opp (Inv (Float 2.)),
  Som (Float 0., Som (Prod (Float 0., Var "x"), Prod (Float 3., Float 1.)))))
```

5. Itérateur pour le type `expr` :

```
#let iterexpr = fun float var somme produit oppose inverse exp ->
#  let rec iter = fun e ->
#    match e with
#      Float x -> float x
#    | Var x -> var x
#    | Som (x, y) -> somme (iter x) (iter y)
#    | Prod (x, y) -> produit (iter x) (iter y)
#    | Opp x -> oppose (iter x)
#    | Inv x -> inverse (iter x) in
#  iter exp;;
val iterexpr :
  (float -> 'a) ->
  (string -> 'a) ->
  ('a -> 'a -> 'a) ->
  ('a -> 'a -> 'a) -> ('a -> 'a) -> ('a -> 'a) -> expr -> 'a = <fun>
```

6.
```
#let eval2 subst = iterexpr (fun x -> x) subst (+.) ( *. ) ((-.) 0.) ((/.) 1.);;
val eval2 : (string -> float) -> expr -> float = <fun>
```

7.
```
#let simplifie =
#  let float = fun c -> Float c
#  and var = fun x -> Var x
#  and somme = fun e1 e2 ->
#    match (e1, e2) with
#      (Float 0., _) -> e2
#    | (_, Float 0.) -> e1
#    | _ -> Som (e1, e2)
#  and produit = fun e1 e2 ->
#    match (e1, e2) with
#      (Float 0., _) -> Float 0.
#    | (_, Float 0.) -> Float 0.
#    | (Float 1., _) -> e2
#    | (_, Float 1.) -> e1
#    | _ -> Prod (e1, e2)
#  and oppose = fun e -> match e with Float 0. -> Float 0. | _ -> Opp e
#  and inverse = fun e -> Inv e in
#  iterexpr float var somme produit oppose inverse;;
val simplifie : expr -> expr = <fun>

#simplifie (derive e1 "x");;
- : expr = Prod (Opp (Inv (Float 2.)), Float 3.)
```