TP n 5 - Correction

Expressions symboliques

L'obj ctif d c TP st d manipul r d s xpr ssions mathématiqu s à plusi urs *symboles*, pour l s évalu r, l s simplifi r t l s dériv r.

1 Définition

Un *expression à plusieurs symboles*, ou *expression symbolique*, st un obj t mathématiqu qui p ut êtr :
– soit un nombr flottant
– soit l'un d s form s suivant s : $C_1 + C_2$, $C_1 \quad C_2$, $C_1 \quad C_2$, $\cos C_1$, $\sin C_2$, où C_1 t C_2 sont d ux xpr ssions symboliqu s
– soit un *symbole* (x, y, *ab*, tc.)
Par x mpl :

$$((2. \quad 1.) \quad (1. \quad x)) \quad ((19. \quad \cos(0.)) + \sin(theta))$$

st un xpr ssion qui conti nt d ux symbol s, x t *theta*.

Exercice 1 On donn l s typ s suivants pour r prés nt r l s opérat urs binair s t unair s :

```
type binop = Plus | Moins | Fois
type unop  = Cos  | Sin
```

Définir l typ `expr` pour r prés nt r l s xpr ssions symboliqu s, sous la form d'un typ somm à quatr construct urs : `Nombre`, `Binop`, `Unop` t `Symbole` (un symbol étant r prés nté par un chaîn d caractèr s), d sort qu l' xpr ssion e_0 ci-d ssus s ra r prés nté n OCaml par :

```
let e0 = Binop (Fois,
                Binop (Moins,
                       Binop (Fois,  Nombre 2.,  Nombre 1.),
                       Binop (Fois,  Nombre 1.,  Symbole "x")),
                Binop (Plus,
                       Binop (Moins, Nombre 19., Unop (Cos, Nombre 0.)),
                       Unop  (Sin,   Symbole "thet "))) ;;
```

Correction :

```
type expr =
| Nombre  of float
| Binop   of (binop * expr * expr)
| Unop    of (unop  * expr)
| Symbole of string
```

Exercice 2 Écrir un fonction :

```
string_of_expr : expr -> string
```

qui, étant donné un xpr ssion, r nvoi un chaîn d caractèr s r prés ntant l' xpr ssion ntièr m nt par nthésé sous form lisibl par un êtr humain. Par x mpl , pour e_0, c tt fonction r nv rra la chaîn :

```
"(((2.*1.)-(1.*x))*((19.-cos(0.))+sin(thet )))"
```

On pourra écrir au préalabl d s fonctions :
– `string_of_binop : binop -> string`
– `string_of_unop : unop -> string`
qui étant donné un opérat ur, r nvoi nt sa r prés ntation lisibl n chaîn d caractèr s.

Correction :

```
let string_of_binop = function
| Plus  -> "+"
| Moins -> "-"
| Fois  -> "*"

let string_of_unop = function
| Cos -> "cos"
| Sin -> "sin"

let rec string_of_expr = function
| Nombre  n           -> string_of_float n
| Binop   (b, e1, e2) -> "(" ^ string_of_expr e1 ^ string_of_binop b ^ string_of_expr e2 ^ ")"
| Unop    (u, e)      -> string_of_unop u ^ "(" ^ string_of_expr e ^ ")"
| Symbole x           -> x
```

2 É aluation

Exercice 3 Écrir un fonction :

```
ev l_expr : (string * flo t) list -> expr -> flo t
```

t ll qu , si `l` st un *environnement* associant (sous la form d'un list d'association) chaqu symbol à un val ur flottant , t si `e` st un xpr ssion, alors `ev l_expr l e` calcul t r nvoi la val ur d l' xpr ssion e sous l' nvironn m nt l. Par x mpl , pour évalu r l' xpr ssion e_0 ci-d ssus n pr nant 3*.* pour x t 0*.* pour *theta*, on écrira :

```
ev l_expr [("x", 3.); ("thet ", 0.)] e0
```

c qui r nv rra `-18.` (car $\cos 0 = 1$ t $\sin 0 = 0$).
Indication : on pourra utilis r la fonction `List. ssoc : ’ -> (’ * ’b) list -> ’b`.
On pourra au préalabl écrir d s fonctions :
– `v leur_binop : binop -> flo t -> flo t -> flo t`
– `v leur_unop : unop -> flo t -> flo t`
qui ff ctu nt l s opérations binair s t unair s n supposant qu l urs argum nts sont d s val urs flottant s déjà calculé s.

Correction :

```
let valeur_binop = function
| Plus  -> ( +. )
| Moins -> ( -. )
| Fois  -> ( *. )

let valeur_unop = function
| Cos -> cos
| Sin -> sin

let rec eval_expr l = function
| Nombre  n           -> n
| Binop   (b, e1, e2) -> valeur_binop b (eval_expr l e1) (eval_expr l e2)
| Unop    (u, e)      -> valeur_unop  u (eval_expr l e)
| Symbole x           -> List.assoc x l
```

3 Simplification

Exercice 4 Écrir un fonction :

`ev l_sous_expr : expr -> expr`

t ll qu , si e st un xpr ssion, alors `ev l_sous_expr e` r nvoi l' xpr ssion obt nu n r mplaçant tout sous- xpr ssion d e n cont nant pas d symbol , par sa val ur. C tt pass st app lé *évaluation des sous-expressions constantes*.
Par x mpl , dans l' xpr ssion e_0 ci-d ssus, l s sous- xpr ssions $2 : 1 :$ t $19 : \cos 0 :$ n conti nn nt pas d symbol , ll s pourront donc êtr r mplacé s par l urs val urs, t alors `ev l_expr_p rtielle e0` r nv rra l' xpr ssion : $(2 : (1 : x))\ (18 : +\sin(theta))$
Fair d s ssais pour r marqu r qu , si e st un xpr ssion qu lconqu , alors, pour tout nvironn m nt l, `ev l_expr l e` t `ev l_expr l (ev l_sous_expr e)` r nvoi nt l s mêm s val urs.
On dit qu `ev l_sous_expr` st *correcte* vis-à-vis d l'évaluation d s xpr ssions.
R marqu r aussi qu pour tout xpr ssion e :

$$\texttt{ev l\_sous\_expr}(\texttt{ev l\_sous\_expr}\, e) = \texttt{ev l\_sous\_expr}\, e$$

On dit qu `ev l_sous_expr` st *idempotente*.

Correction :

```
let rec eval_sous_expr = function
| Binop (b, e1, e2) -> begin match (eval_sous_expr e1, eval_sous_expr e2) with
                       | Nombre n1, Nombre n2 -> Nombre (valeur_binop b n1 n2)
                       | (e'1, e'2)           -> Binop (b, e'1, e'2)
                       end
| Unop  (u, e)      -> begin match eval_sous_expr e with
                       | Nombre n -> Nombre (valeur_unop u n)
                       | e'       -> Unop (u, e')
                       end
| e                 -> e
```

Exercice 5 Écrir un fonction :

```
ev l_neutres : expr -> expr
```

t ll qu , si e st un xpr ssion, alors `ev l_neutres e` r nvoi l' xpr ssion obt nu n r mplaçant tout sous- xpr ssion d e :
– d la form $0 + e'$ ou $e' + 0$ ou $e' * 1$ par 0
– d la form $1 * e'$, $e' * 1$ ou $e' - 0$ ou $e' + 0$ ou $0 + e'$ par e'
Par x mpl , dans l' xpr ssion e_0 ci-d ssus, la sous- xpr ssion $2 * 1$ (r sp. $1 * x$) pourra êtr r mplacé par 2 (r sp. x), t alors `ev l_neutres e0` r nv rra l' xpr ssion :

$$(2 * x) * ((19 * \cos(0)) + \sin(theta))$$

(l'évaluation d s sous- xpr ssions constant s n'a pas li u)
R marqu r qu `ev l_neutres` st corr ct vis-à-vis d l'évaluation d s xpr ssions, t qu' ll st id mpot nt .

Correction :

```
let rec eval_neutres = function
| Binop (b, e1, e2) ->
  begin match b, eval_neutres e1, eval_neutres e2) with
    | Moins, e'1      , e'2       when e'1 = e'2 -> Nombre 0.
    | Fois , _        , Nombre 0.                -> Nombre 0.
    | Fois , Nombre 0., _                        -> Nombre 0.
    | Fois , e'       , Nombre 1.                -> e'
    | Fois , Nombre 1., e'                       -> e'
    | Moins, e'       , Nombre 0.                -> e'
    | Plus , e'       , Nombre 0.                -> e'
    | Plus , Nombre 0., e'                       -> e'
    | _    , e'1      , e'2                      -> Binop (b, e'1, e'2)
  end
| Unop  (u, e')     -> Unop (u, eval_neutres e')
| e                 -> e
```

Exercice 6 Écrir un fonction :

```
point_fixe : ('  -> ' ) -> '  -> '
```

t ll qu , si f st un fonction t si a st un val ur applicabl à f, alors `point_fixe f` appliqu f sur a t r comm nc sur l résultat jusqu'à c qu'il n chang plus.
On dit qu `point_fixe f` st un *point fixe* d f (n ff t, si x = `point_fixe f` , alors `f` $x = x$). C' st l point fix d f obt nu n partant d a.
Utilis r c tt fonction pour écrir un fonction :

```
simplifier : expr -> expr
```

qui, étant donné un xpr ssion, lui appliqu l s d ux fonctions `ev l_neutres` t `ev l_sous_expr` t continu jusqu'à c qu l' xpr ssion n chang plus.
Pourquoi c tt fonction t rmin -t- ll ?

Correction :

```
let rec point_fixe f a =
 let a' = f a in
 if a' = a
 then a
 else point_fixe f a'

let simplifier = point_fixe (fun e -> let e' = eval_neutres e in eval_sous_expr e')
```

Cette fonction termine car chacune des deux fonctions `eval_neutres` et `eval_sous_expr`, soit laisse l'expression inchangée, soit fait décroître strictement sa taille (nombre de constructeurs, par exemple).

4 Déri ation

Exercice 7 Soit s un symbol . L'*expression dérivée* d'un xpr ssion e par rapport au symbol s, noté $\frac{\partial e}{\partial s}$ ou $@_s e$, st défini par induction sur la structur d e :
– si n st un nombr (xact ou approché), alors sa dérivé st l nombr 0*:*
– si e_1 t e_2 sont d ux xpr ssions, alors $@_s(e_1 + e_2) = @_s e_1 + @_s e_2$ t d mêm pour
– si e_1 t e_2 sont d ux xpr ssions, alors $@_s(e_1 \quad e_2) = (@_s e_1 \quad e_2) + (e_1 \quad @_s e_2)$
– si e st un xpr ssion, alors $@_s(\cos(e)) = 0$*:* $(@_s e \quad \sin(e))$ t $@_s(\sin(e)) = @_s e \quad \cos(e))$
– la dérivé du symbol s par rapport à lui-mêm st l nombr 1*:*
– la dérivé d'un symbol t différ nt d s st l nombr 0*:*
Écrir un fonction :

```
deriv_expr : string -> expr -> expr
```

qui, étant donnés un symbol s t un xpr ssion e, calcul t r nvoi l' xpr ssion dérivé d e par rapport à s.

Correction :

```
let rec deriv_expr s = function
| Nombre _          -> Nombre 0.
| Binop (b, e1, e2) ->
  let e'1 = deriv_expr s e1
  and e'2 = deriv_expr s e2
  in
  begin match b with
  | Fois -> Binop (Plus, Binop (Fois, e'1, e2), Binop (Fois, e1, e'2))
  | _    -> Binop (b, e'1, e'2)
  end
| Unop (u, e)       ->
  let e'  = deriv_expr s e
  in
  begin match u with
  | Cos -> Binop (Moins, Nombre 0., Binop (Fois, e', Unop (Sin, e)))
  | Sin -> Binop (Fois, e', Unop (Cos, e))
  end
| Symbole t         -> Nombre (if t = s then 1. else 0.)
```