AP1 - TP 5 - Manipulation d'expressions arithmétiques

Le langage des expressions arithmétiques (avec variables) a été défini en cours de la façon suivante :

```
type expr =   Var of string | Nb of int
            | Plus of expr * expr | Mult of expr * expr;;
```

1. Représenter l'expression $x + 2y$.

   Corrigé : `Plus (Var "x", Mult (Nb 2, Var "y"))`

   Elle est nommée ex dans la suite

2. Ecrire une fonction `vars` qui calcule la liste des noms des variables qui apparaissent dans une expression. La liste pourra être avec ou sans doublons. Par exemple `vars (Plus (Var "x", Mult ("y", Nb 1)))` retourne la liste `["x";"y"]`.

   Corrigé :

```
#
(*interface vars
type : expr -> string list
args e
precondition : true
postcondition : calcule la liste des variables qui apparaissent dans e
tests : vars (Var "x") (résultat attendu = ["x"])
        vars (Plus (Var "x", Mult (Var "y", Var "y"))) (résultat attendu =
        ["x";"y";" y"] ou dans un ordre ordre)
        vars (Nb 3) (résultat attendu  [])
*)
let rec vars e = match e with
    Var x -> [x]
   | Nb _ -> []
   | Plus(e1,e2) -> (vars e1) @ (vars e2)
   | Mult(e1,e2) -> (vars e1) @ (vars e2);;
val vars : expr -> string list = <fun>

ou solution récursive terminale :

#(*interface vars_aux
type : string list -> expr -> string list
args l e
precondition : true
postcondition : calcule la liste des variables qui apparaissent dans e
concaténée  à la liste l
*)
let rec vars_aux l e = match e with
    Var x -> x::l
   | Nb _ -> l
   | Plus(e1,e2) -> (vars_aux (vars_aux l e1) e2)
   | Mult(e1,e2) -> (vars_aux (vars_aux l e1) e2);;
val vars_aux : string list -> expr -> string list = <fun>

#(*interface vars
type : expr -> string list
```

```
args e
precondition : true
postcondition : calcule la liste des variables qui apparaissent dans e
tests : vars (Var "x") (résultat attendu = ["x"])
        vars (Plus (Var "x", Mult (Var "y", Var "y"))) (résultat attendu =
        ["x";"y";" y"] ou dans un ordre ordre)
        vars (Nb 3) (résultat attendu  [])
*)
let vars = vars_aux [];;
val vars : expr -> string list = <fun>
```

Attention à la postcondition de `vars_aux` *est :* `vars_aux l e = liste des variables de e concaténée avec l.`

3. Ecrire une fonction `evaluer` qui évalue une expression. Cette fonction prend en argument un environnement qui associe des valeurs aux variables. Le calcul échouera si l'une des variables n'était pas valuée dans l'environnement.

   On représentera un contexte par une liste d'association c'est-à-dire une liste de couples de la forme (nom de variable, valeur).

   Commencer par écrire la fonction assoc qui prend en paramètre un nom de variable et un environnement et retourne la valeur associée à cette varaible dans l'environnement. Elle échouera si la variable n'est pas présente dans l'environnement. Par exemple `assoc("x", [("y",4);("x",5)])` retourne la valeur 5. Mais `assoc("z", [("y",4);("x",5)])` n'est pas définie.

   Puis écrire la fonction évaluer.

```
(*interface assoc
arg (s,l)
preconsition : s est le premier composant d'un couple de l
postcondition : retourne y tel que (s,y) apparteint à l
raises : échoue avec le message "assoc : non trouve" si il n'existe
pas de couple (s, _) dans l
tests : assoc (3, [(1,2) ; (3, 4) ; (5;6)] ) (résultat attendu = 4)
        assoc (2, [(1,2) ; (3, 4) ; (5;6)] ) (échec)
*)
let rec assoc (s ,l) = match l with
  [] -> failwith "assoc : non trouve"
| (x,y)::r -> if s=x then y else assoc (s, r);;

(*interface evalue
type : expr * string*int list -> int
arg e (une expression) c (un contexte)
precondition : les varaibles de e sont toutes valuées dans c
postcondition : retourne la valeur entière de e
raises :  échoue avec le message "assoc : non trouve" si une des
variaables de e n'est pas valuée dans c
tests : tester avec une expression de chaque sorte et une expression
où une variable au moins ne figure pas dans c
*)
let rec evalue (e,c) = match e with
   Nb n -> n
 | Var s -> assoc (s, c)
```

```
 | Plus (e1,e2) -> (evalue (e1, c)) + (evalue (e2, c))
 | Mult (e1,e2) -> (evalue (e1, c)) * (evalue (e2, c));;
```

4. Ecrire la fonction simplifie qui simplifie une expression. Ces simplifications consistent à effectuer toute opération sur deux nombres et prennent en compte le fait que 0 est élément neutre pour l'addition et élément absorbant pour la multiplication, et que 1 est élément neutre pour la multiplication. Ces simplifications sont bien sûr faites autant que faire se peut : par exemple, $(3 - 2) \times x + (y \times 0)$ se simplifie en appliquant les remarques précédentes sur chaque sous-expression, en $(1 \times x) + 0$. La première sous-expression se simplifie encore en x, donnant l'expression $x + 0$ qui se simplifie en x.

```
(* interface simplifier
type expr -> expr
precondition : true
postcondition : simplifie au maximun l'expresson e en appliquant les
regles de l'enoncé
tests : passer au moins une fois partout
*)
let rec simplifier e = match e with
  (* Opérations sur deux nombres *)
  Plus (Nb n1, Nb n2) -> Nb (n1+n2)
| Mult (Nb n1, Nb n2) -> Nb (n1*n2)
  (* 0 neutre pour Plus *)
| Plus (Nb 0, e) -> simplifier e
| Plus (e, Nb 0) -> simplifier e
  (* 0 absorbant pour Mult *)
| Mult (Nb 0, _) | Mult (_, Nb 0) -> Nb 0
  (* 1 neutre pour Mult *)
| Mult (Nb 1, e) -> simplifier e
| Mult (e, Nb 1) -> simplifier e
  (* Simplification de sous-expressions: une simplification de
  l'expression peut être  nécessaire après simplification des deux
  sous-expressions; pour éviter que la fonction  ne boucle, on
  teste si au moins une des sous-expressions a été modifiée sinon on
  exécute exactement le même appel à la fonction simplification et on
  boucle *)
| Plus (e1, e2) -> let e'1 = simplifier e1 and e'2 = simplifier e2 in
    if e'1 <> e1 || e'2 <> e2 then simplifier (Plus (e'1, e'2))
    else Plus (e'1, e'2)
| Mult (e1, e2) -> let e'1 = simplifier e1 and e'2 = simplifier e2 in
    if e'1 <> e1 || e'2 <> e2 then simplifier (Mult (e'1, e'2))
    else Mult (e'1, e'2)
| e -> e;;
```

5. Ecrire une fonction qui transforme l'expression en une chaîne de caractères, i.e. une fonction de type `expr -> string`.
   Rappel : la fonction `string_of_int` transforme un entier en une chaîne de caractères. Mettez le minimum de parenthèses.

   1ère solution :

```
let rec string_of_expr  e = match e with
       Nb n -> string_of_int n
```

```
   |  Var s -> s
   | Plus(e1,e2) -> (string_of_expr e1)^"+"^(string_of_expr e2)
   | Mult(e1,e2) -> (string_of_expr e1)^"*"^(string_of_expr e2);;
```

Ne convient pas car elle produit des expressions ambiguës. Ainsi `string_of_expr (Mult (Plus (Var "x", Nb 3), Nb 5))` produit le résultat "x+3*5". Est-ce l'expression x+(3x5) ou l'expression (x+3)x5 ?

2ème solution : paraenthéser chaque expression binaire.

```
let ouv = "(" and ferm = ")";;

let rec string_of_expr  e = match e with
       Nb n -> string_of_int n
   |  Var s -> s
   | Plus(e1,e2) -> ouv^(string_of_expr e1)^"+"^(string_of_expr e2)^ferm
   | Mult(e1,e2) ->  ouv^(string_of_expr e1)^"*"^(string_of_expr e2)^ferm;;

# string_of_expr (Mult (Plus (Var "x", Nb 3), Nb 5));; - : string
= "((x+3)*5)"
```

C'est correct mais laid.

3ème solution : on utilise les règles de priorités usuelles et on ne parenthèse que lorsque c'est nécessaire.

* est plus prioritaire que +. On pose priorité de + = 1 et priorité de * = 2.

La fonction `prio_expr` donne la priorité d'une expression : par convention la priorité d'une variable ou d'une constante est 3 (la plus grande), la priorité d'une expression où l'opérateur principal est + est 1 et celle d'une expression où l'opérateur principal est * est 2.

```
let prio_expr e = match e with
  Plus(_,_) -> 1
| Mult(_,_) -> 2 | _ -> 3;;

let rec string_of_expr e =
  match e with
   Var s -> s
  | Nb n -> string_of_int n
  | Plus(e1,e2) -> let p = prio_expr e in
                   (if prio_expr e1 <= p then
                     ouv^(string_of_expr e1)^ferm
                    else string_of_expr e1)
                    ^"+"^
                   (if prio_expr e2 <= p then
                     ouv^(string_of_expr e2)^ferm
                    else string_of_expr e2)
  | Mult(e1,e2) -> let p = prio_expr e in
                   (if prio_expr e1 <= p then
                     ouv^(string_of_expr e1)^ferm
                    else string_of_expr e1)
                    ^"*"^
                   (if prio_expr e2 <= p then
                     ouv^(string_of_expr e2)^ferm
```

```
                    else string_of_expr e2);;


# string_of_expr (Mult (Plus (Var "x", Nb 3), Nb 5));;
 - : string= "(x+3)*5"
 # string_of_expr (Plus (Var "x", Mult (Nb 3, Nb 5)));;
- : string = "x+3*5"
 # string_of_expr (Plus (Var "x", Plus (Nb 3, Nb 5)));;
 - : string = "x+(3+5)"
 # string_of_expr (Mult (Var "x", Plus (Nb 3, Nb 5)));;
 - : string = "x*(3+5)"
 # string_of_expr (Plus (Mult (Var "x", Nb 3), Nb 5));;
 - : string = "x*3+5"
```

On peut certes écrire la fonction d'une autre façon mais la solution est ici la technique générale qui marche chaque fois qu'un jeu d'opérateurs avec priorité est utilisée.