TP 5 -Expressions Arithmétiques -

Le but du TP est de réaliser un évaluateur d'*expressions arithmétiques*. La définition de type suivante en Ocaml permet de décrire la syntaxe abstraite de ces expressions :

```
type expr =  Term  of int
           | Sum   of expr * expr
           | Diff  of expr * expr
           | Prod  of expr * expr
           | Quot  of expr * expr;;
```

Exercice 1. Évaluer

Écrire une fonction `eval_expr` de type `expr -> int` qui calcule la valeur entière d'une expression.

```
# eval_expr;;
- : expr -> int = <fun>
# eval_expr (Term 2);;
- : int = 2
# eval_expr (Sum(Term 2, Term 3));;
- : int = 5
# eval_expr (Diff(Term 2, Term 3));;
- : int = -1
# eval_expr (Prod(Term 2, Term 3));;
- : int = 6
# eval_expr (Quot(Term 2, Term 3));;
- : int = 0
# eval_expr (Quot(Term 6, Term 3));;
- : int = 2
# eval_expr (Sum(Quot(Term 6, Term 3), Prod(Term 4, (Diff(Term 2, Term 3)))));;
- : int = -2
```

Exercice 2. Manipuler

1. Écrire la fonction `string_of_expr` qui calcule une représentation sous forme d'une chaîne de caractères d'une expression arithmétique (la chaîne de caractères doit être correctement parenthésée). Vous pourrez utiliser la fonction `string_of_int : int -> string`.
2. Écrire la fonction `simplify_sum_expr` qui simplifie une expression arithmétique en s'appuyant sur $x + 0 = 0$.
3. Écrire la fonction `simplify_diff_expr` qui simplifie une expression arithmétique en s'appuyant sur $x - 0 = x$.
4. Écrire la fonction `simplify_prod_expr` qui simplifie une expression arithmétique en s'appuyant sur $x * 1 = x$ et $x * 0 = 0$.
5. Écrire la fonction `simplify_quot_expr` qui simplifie une expression arithmétique en s'appuyant sur $x/1 = x$.

```
# string_of_expr;;
- : expr -> string = <fun>
# string_of_expr (Sum(Quot(Term 6, Term 3), Prod(Term 4, (Diff(Term 2, Term 3)))));;
- : string = "((6/3)+(4*(2-3)))"
# simplify_sum_expr (Term 0);;
- : expr = Term 0
# simplify_sum_expr (Term 1);;
- : expr = Term 1
# simplify_sum_expr (Sum (Term 1, Term 0));;
- : expr = Term 1
# simplify_sum_expr (Sum (Term 0, Term 0));;
- : expr = Term 0
# simplify_sum_expr (Sum (Sum (Term 0, Term 0), Sum (Term 0, Term 0)));;
- : expr = Term 0
# simplify_sum_expr (Sum (Sum (Term 1, Term 0), Sum (Term 0, Term 0)));;
- : expr = Term 1
# simplify_diff_expr (Diff(Term 1, Term 0));;
- : expr = Term 1
# simplify_diff_expr (Diff(Diff(Term 1, Term 0), Diff(Term 1, Term 0)));;
- : expr = Diff (Term 1, Term 1)
# simplify_prod_expr (Prod(Term 1, Term 2));;
- : expr = Term 2
# simplify_prod_expr (Prod(Term 2, Term 0));;
- : expr = Term 0
# simplify_prod_expr (Prod(Term 1, Term 0));;
- : expr = Term 0
# simplify_prod_expr (Sum(Prod(Term 0, Term 2), Prod(Prod(Term 0, Term 2), Term 3)));;
- : expr = Sum (Term 0, Term 0)
# simplify_quot_expr (Quot(Term 2, Term 1));;
- : expr = Term 2
# simplify_quot_expr (Quot(Term 2, Quot(Term 1, Term 1)));;
- : expr = Term 2
```

Exercice 3. Puissances

Créer le type des *expressions arithmétiques étendues* `eexpr` afin de prendre en charge les puissances entières (entiers naturels) en plus des opérations élémentaires. Écrire la fonction `eval_eexpr` associée.

```
# let test_eval_eexpr e from to_excluded =
    let rec test_eval_eexpr' i acc =
        if i = to_excluded then acc
                           else let x = eval_eexpr (Pow(e, i)) in
                                test_eval_eexpr' (i + 1) (x::acc) in
    List.rev (test_eval_eexpr' from []);;
val test_eval_eexpr : eexpr -> int -> int -> int list = <fun>
# test_eval_eexpr (Sum(Term 1, Term 1)) 0 10;;
- : int list = [1; 2; 4; 8; 16; 32; 64; 128; 256; 512]
```

Écrire la fonction `expr_of_eexpr` permettant de tranformer une expression arithmétique de type `eexpr` en une expression arithmétique équivalente de type `expr`.

```
# expr_of_eexpr;;
- : eexpr -> expr = <fun>
# expr_of_eexpr (Pow(Sum(Pow(Diff(Term 2, Term 1), 2), Pow(Term 3, 3)), 4));;
- : expr =
Prod
 (Sum (Prod (Diff (Term 2, Term 1), Diff (Term 2, Term 1)),
   Prod (Term 3, Prod (Term 3, Term 3))),
 Prod
  (Sum (Prod (Diff (Term 2, Term 1), Diff (Term 2, Term 1)),
    Prod (Term 3, Prod (Term 3, Term 3))),
  Prod
   (Sum (Prod (Diff (Term 2, Term 1), Diff (Term 2, Term 1)),
     Prod (Term 3, Prod (Term 3, Term 3))),
   Sum (Prod (Diff (Term 2, Term 1), Diff (Term 2, Term 1)),
    Prod (Term 3, Prod (Term 3, Term 3))))))
```

Écrire maintenant la fonction `eexpr_of_expr` permettant de tranformer une expression arithmétique de type `expr` en une expression arithmétique équivalente de type `eexpr` (bien sûr `Prod(x, x)` devra se réduire à `Pow(x, 2)`, tout cela récursivement).

```
# eexpr_of_expr;;
- : expr -> eexpr = <fun>
# eexpr_of_expr (Prod(Term 1, Term 1));;
- : eexpr = Pow (Term 1, 2)
# eexpr_of_expr (Prod(Prod(Term 1, Term 1),Prod(Term 1, Term 1)));;
- : eexpr = Pow (Term 1, 4)
# eexpr_of_expr (expr_of_eexpr (Pow(Term 1, 100)));;
- : eexpr = Pow (Term 1, 100)
```

Exercice 4. Pour aller plus loin

Quelques idées pour poursuivre ce TP :

- Simplifier en utilisant des identités remarquables du second degré : $(a+b)^2 = a^2 + 2ab + b^2$, $(a-b)^2 = a^2 - 2ab + b^2$, et $a^2 - b^2 = (a+b)(a-b)$.
- Écrire une fonction de simplification complète.
- Introduire des variables dans les expressions arithmétiques (il est ainsi possible de dériver une expression arithmétique).
- Introduire des opérateurs n-aires (somme Σ, produit $\prod$, ...).