TRAVAUX PRATIQUES XIV

Expressions arithmétiques

Ce TP se fait en OCaml. L'objectif est d'écrire des fonctions de manipulation d'expressions arithmétique.
Dans tout ce TP, il est important de faire des dessins !

A Pré-requis

Allez-lire la partie 1.2.1 et 1.3 et 1.4 du cours sur les Bonnes Pratiques de programmation.

B Formalisation

Formalisons ce qu'est « une expression arithmétique ». Commençons par un exemple simple : les expressions additives. Une expression additive est :

- Soit un entier.
- Soit l'addition de deux autres expressions additives.

Par exemple, « (1+2) +3 » est l'addition de « (1+2) » et « 3 », où « 3 » est un entier et « 1+2 » est l'addition de deux expressions qui sont des entiers.
En OCaml, le type peut s'écrire ainsi :

```
type plus_expr =
  | Int of int
  | Plus of plus_expr * plus_expr
```

Par exemple, ceci est l'expression « (1+2) +3 » :

```
let ud_t = Plus ( Plus (Int 1, Int 2), Int 3 )
```

Dans `plus.ml` , vous trouverez ce type ainsi qu'une fonction d'affichage.

Insistons sur un point : dans le type `plus_expr` , on ne lit pas une expression « de gauche à droite ». On la lit d'addition en addition ! Cf schéma au tableau.

En particulier, pour travailler avec ce type, il faut faire des `match`. Comme pour les listes. En fait, pour n'importe quel type défini *par induction*, il faut raisonner *par induction* en faisant des disjonctions de cas.

0. Écrire une fonction `is_int` qui prend en argument une `plus_expr` et renvoie `true` si elle est de la forme `Int _` est `false` sinon.

1. Écrire une fonction `eval` qui prend en argument une `plus_expr` et renvoie l'entier qu'elle calcule.

C Cas général

Dans `expr.ml` , vous trouverez une version généralisée des expressions additives. On y autorise d'autres expressions.

2. Codez `eval` pour ces expressions-ci.

3. On appelle *complexité* d'une expression le nombre de calculs qu'elle a à faire. Écrivez une fonction `complexité` qui renvoie la complexité d'une expression.

4. On s'intéresse le nombre maximal de parenthèses imbriquées qu'une expression contient. Par exemple, `Cst 3` a 0 parenthèses imbriquées, `Add (Cst 4, Cst 5)` a 1 parenthèses imbriquée et `Mul (Int 1, Add (Cst 4, Cst 5))` a au maximum 2 parenthèses imbriquées.
   Écrivez une fonction `nb_par_max` qui calcule cela.

D Pour occuper les plus rapides : Solver

On s'intéresse ici à résoudre une équation linéaire en 1 inconnue (avec 1 équation). Les expressions que l'on considère sont donc des additions et des multiplications par un scalaire. Une équation est l'égalité entre deux expressions.

Le fichier `solver.ml` contient les types correspondants.

5. Écrire une fonction `solve` qui prend en argument et résout une telle équation.