Expressions arithmétiques

Florian Bourse

On définit en OCaml les expressions littérales comme des arbres binaires, où les nœuds internes sont étiquetés par des opérateurs binaires (ici + et ×), et les feuilles sont étiquettées soit par un nombre entier soit par la variable x :

Expressions littérales

```
type operator = Add | Mul
type expr =
  | Int of int
  | Var
  | Op of expr * operator * expr

let exemple = Op (Int 2, Add, Op (Op (Int 3, Add, Int 7), Mul, Int 4))
let exemple_bis = Op (Op (Int 3, Mul, Var), Add, Int (-126))
```

0 Écrire une fonction `nombres : expr -> int` qui compte le nombre de feuilles (nombres ou variables) dans une expression littérale.

1 Écrire une fonction `operateurs : expr -> int` qui compte le nombre d'opérateurs binaires dans une expression littérale.

2 Que peut-on dire des résultats des deux fonctions précédentes sur une même expression littérale ? le démontrer.

3 Écrire une fonction `eval : int -> expr -> int` qui prend en entrée un entier x et une expression littérale et qui renvoie le résultat de l'expression si la variable vaut x.
On pourra vérifier les valeurs de `eval 0 exemple` et `eval 42 exemple_bis`.

4 Écrire des fonctions `prefixe : expr -> string`,
`postfixe : expr -> string` et `infixe : expr -> string` qui permettent de transformer une expression littérale en une chaîne de caractères qui la représente en notation préfixe, postfixe ou infixe.

5 En réalité, les expressions que l'on manipule ici sont des polynômes (il n'y a pas de division). Écrire une fonction `degree : expr -> int` qui calcule une borne sur le degré d'un polynôme représenté par une expression littérale.
Donner la complexité de votre fonction.
On pourra simplifier les disjonctions de cas en utilisant 0 *comme borne sur le degré du polynôme identiquement nul* $P(X) = 0$.

6 Écrire une fonction `egales : expr -> expr -> bool` qui teste l'égalité entre deux expressions littérales basée sur l'estimation du degré implémentée à la fonction précédente.
Estimer la complexité de votre fonction.

Nous souhaitons à présent développer et réduire nos expressions pour obtenir une forme canonique. Nous allons donc représenter à présent les polynômes par une suite finie de monômes. Remarquons que le polynôme identiquement nul sera représenté par une liste vide, les monômes étant donc de degré positif ou nul. Pour que la représentation canonique soit unique, nous trierons les monômes par degré strictement décroissant.

Polynômes 

```
type monomial = {deg : int; coeff : int}
type polynomial = monomial list
```

7 Écrire une fonction `add_monomial : monomial -> polynomial -> polynomial` qui ajoute un monôme à un polynôme.
En déduire une fonction `sum : polynomial -> polynomial -> polynomial` qui somme deux polynômes.

8 Écrire une fonction `distribute : monomial -> polynomial -> polynomial` qui multiplie un polynôme par un monôme donné en utilisant la distributivité.
En déduire une fonction `prod : polynomial -> polynomial -> polynomial` qui multiplie deux polynômes.

9 Écrire une fonction `expand_simplify : expr -> polynomial` qui développe et réduit une expression littérale pour en déduire sa forme polynomiale.

10 Expliquer comment calculer le degré d'un poylnôme donné sous forme de liste de monôme, ainsi que comment tester l'égalité de deux polynômes.
Estimer les complexités de ces deux opérations.