

Programmation impérative et fonctionnelle avec OCaml

TP 4

Objectifs :

- Types algébriques :
 - type somme ;
 - type produit ;
 - type récursif.
- Ordre supérieur :
 - passage de fonction en paramètre ;
 - itérateur.
- Calcul symbolique

Manipulation d’expressions arithmétiques

On souhaite effectuer des calculs symboliques sur des expressions arithmétiques (comme peuvent le faire Maple ou Mathematica).

Pour ce TP, on utilisera uniquement l’interpréteur `ocaml` afin de s’épargner l’écriture d’une fonction d’impression des expressions. On éditera donc son code avec Emacs puis on utilisera le copier-coller pour passer les fonctions à l’interpréteur¹.

1. Définir le type `expr` pour la représentation d’expressions arithmétiques. On pourra considérer qu’une expression arithmétique peut être :
 - un nombre : constante de type `float` ;
 - une variable : un identificateur représenté par une `string` ;
 - une somme de deux expressions ;
 - un produit de deux expressions ;
 - l’opposé d’une expression ;
 - l’inverse d’une expression.
2. Représenter avec ce type l’expression suivante :

$$-\frac{1}{2}(1 + 3x)$$

Ne pas hésiter à utiliser des valeurs intermédiaires :

```
let trois_x = ... ;;
let un_plus_trois_x = ...;;
let e = ...;;
```

3. Écrire une fonction

```
eval : expr -> float
```

permettant d’évaluer une expression ne contenant pas de variable. Si une variable est rencontrée, on déclenchera une erreur avec la fonction `failwith`².

1. Il est vivement conseillé de configurer l’alias utilisant `ledit` préconisé sur e-Campus pour pouvoir éditer facilement les expressions saisies dans l’interpréteur.

2. La fonction `failwith : string -> 'a` est une fonction qui prend une chaîne de caractères en argument (typiquement un message d’erreur) et qui lève une *exception*. Cette exception stoppe l’exécution du programme en cours et la chaîne de caractères est affichée.

4. Modifier (en la recopiant) la fonction précédente pour écrire une fonction

```
eval2 : (string -> float) -> expr -> float
```

prenant en argument une fonction d'évaluation pour les variables (une *substitution*) et une expression et évaluant cette dernière pour cette substitution.

```
let subst = fun v ->
  match v with
  | "x" -> 1.
  | _   -> failwith "subst: variable inconnue";;
```

eval2 subst évalue $-\frac{1}{2}(1 + 3x)$ en -2.0 .

5. Écrire une fonction

```
derive : expr -> string -> expr
```

qui dérive une expression par rapport à une variable donnée.

6. Écrire l'itérateur pour le type `expr`.
 7. Réécrire la fonction `eval2` en utilisant l'itérateur.
 8. En utilisant l'itérateur, écrire une fonction

```
simplifie: expr -> expr
```

qui simplifie une expression donnée en utilisant (au moins) les règles suivantes :

$$\forall e \ 0 + e = e + 0 = e, \ \forall e \ 1 * e = e * 1 = e, \ \forall e \ 0 * e = e * 0 = 0, \ -0 = 0$$