Université Paris Diderot — Programmation Fonctionnelle
L2 Math-Info/L3 Informatique/M1 Linguistique — Année 2013-2014

TP n 5

Expressions et évaluation

L'objectif de ce TP est de construire, manipuler, évaluer et simplifier des expressions arithmétiques simples construites avec des nombres et des variables. Les expressions considérées se définissent de la manière suivante :

- un nombre (0, 1.5, 3.14, etc.) est une expression,
- une variable (x, y, $theta$, etc.) est une expression,
- si e et e' sont des expressions, alors $(e + e')$ $(e \quad e)$ $(e \quad e')$ $\sin(e)$ $\cos(e)$ sont des expressions.

Dans les dernières formes, cos et sin seront appelés *opérateurs unaires*, et $+$, , des *opérateurs binaires*. Par exemple, $(\cos(x+0.5) \quad (theta \quad 2))$ est une expression contenant les deux variables x, $theta$, les nombres 0.5 et 2, l'opérateur unaire cos et les opérateurs binaires $+$, , .

1 Représentation d'expressions en Caml

Pour représenter les opérateurs et les expressions en Caml, on se servira des types suivants :

```
type unop  = Cos  | Sin;;
type binop = Plus | Moins | Fois;;
type expr  =   Num of float
             | Var of string
             | Unop of  unop  * expr
             | Binop of binop * expr  * expr;;
```

Par exemple, l'expression $(\cos(x + 0.5) \quad (theta \quad 2))$ sera représentée par :

```
Let e0 = Binop (Fois,
                Unop (Cos,   Binop(Plus, Var "x", Num 0.5)),
                Binop(Moins, Var "theta", Num 2.));;
```

Noter que tous les nombres, entiers ou non, se représentent à l'aide du type `float`.

Exercice 1. Ecrire :

```
string_of_unop  : unop  -> string
string_of_binop : binop -> string
```

renvoyant, pour chaque sorte d'opérateur, sa représentation sous forme de chaîne de caractères (*i.e.*, littéralement `"cos"`, `"sin"`, `"+"`, etc).

Exercice 2. A partir des fonctions précédentes et de la fonction prédéfinie `string_of_float`, écrire

```
string_of_expr : expr -> string
```

renvoyant, étant donnée une expression, une chaîne de caractères représentant cette expression écrite avec toutes ses parenthèses. Par exemple, appliquée à l'expression ci-dessus, cette fonction renverra `"(cos(x + 0.5) * (theta - 2.))"`.

2 Évaluation d'expressions

Appelons *environnement* toute liste d'association de type `(string * float) list`. Un environnement permet de spécifier les valeurs des variables d'une expression avec la convention suivante : s'il contient le couple `("x", f)`, alors `f` est la valeur choisie pour `"x"`.

La *valeur d'une expression* `e` *dans l'environnement* `l` est celle obtenue en remplaçant dans `e` les variables par les valeurs spécifiées par `l`, ainsi que tous les opérateurs par les opérations "réelles" qui leur sont naturellement associées : sinus, cosinus, addition, etc.

Exercice 3. Ecrire deux fonctions

```
val_unop  : unop  -> (float -> float)
val_binop : binop -> (float -> float -> float)
```

telles que `val_unop u` (resp. `val_binop b`) renvoie la fonction Caml naturellement associée à `u` (resp. à `b`) – les fonctions `sin` et `cos` sont prédéfinies. Par exemple, `val_binop Plus` renverra la fonction `(fun x y -> x +. y)`, ou mieux encore, la fonction `(+.)`.

Exercice 4. Rappelons que `List.assoc v l` renvoie la partie droite `f` du premier couple de `l` de la forme `(v, f)` s'il existe, et déclenche une exception si ce couple est introuvable. A partir des deux fonctions précédentes (*i.e.* en laissant `val_unop` et `val_binop` associer à chaque opérateur la fonction appropriée) et de la fonction `List.assoc`, écrire

```
eval_expr : (string * float) list -> expr -> float
```

telle que, si `e` est un expression et `l` est un environnement, `eval_expr l e` renvoie la valeur de `e` dans l'environnement `l`. Par exemple, en choisissant pour `"x"` la valeur `-0.5` et pour `"theta"` la valeur `4.` dans l'expression `e0` ci-dessus, on aura

```
eval_expr [("x", -.5); ("theta", 4.)] e0 = 2.
```

car cos($0.5 + 0.5$) (4 2) = 1 2 = 2.

3 Simplification d'expressions

Une expression peut ne contenir aucune variable – elle est alors appelée *expression constante*. La valeur d'une telle expression est directement calculable, sans environnement. Même lorsque une expression contient des variables, certaines parties de cette expression peuvent être constantes : dans ce cas, l'expression peut être simplifiée en remplaçant ces sous-expressions par leurs valeurs.

Exercice 5. Ecrire une fonction

```
eval_sous_expr : expr -> expr
```

telle que `eval_sous_expr e` renvoie l'expression obtenue en remplaçant chaque sous-expression constante de `e` par sa valeur. On notera que si `e` est elle-même une expression constante, la valeur renvoyée doit être la forme `Num f`. D'autre part, si `e1` et `e2` sont constantes, alors `Binop(b,e1,e2)` et `Unop(u,e1)` sont aussi des expressions constantes, et peuvent être remplacées par leur valeur.

Exercice 6. Même lorsqu'elles sont non constantes, certaines parties d'une expression peuvent malgré tout être simplifiées au moyen des règles suivantes. Etant donnée une sous-expression d'une expression quelconque, on peut la remplacer :

- par 0 si elle est de la forme $(0 \quad e)$, $(e \quad 0)$, $(e \quad e)$,
- par e si elle est de la forme $(1 \quad e)$, $(e \quad 1)$, $(e \quad 0)$, $(e + 0)$, $(0 + e)$.

L'expression résultant aura la même valeur dans tout environnement.

Écrire une fonction

```
eval_neutres : expr -> expr
```

telle que `eval_neutres e` renvoie l'expression obtenue en appliquant à `e`, en partant des sous-expressions les plus profondes et en remontant vers la racine, les règles de transformation données ci-dessus.

4 Point fixe [⋆] et dérivation [⋆⋆]

Exercice 7. [⋆] Etant donnée une fonction `f` et une valeur `a`, le *point fixe de* `f` *obtenu en partant de* `a` est la première valeur `r` dans la suite de valeurs

```
a
(f a)
(f (f a))
⋮
(f ... (f (f a))...)
⋮
```

telle que l'on ait `(f r) = r`, et donc, `(f (f r)) = (f r) = r`, etc. Autrement dit, il s'agit du premier élément de cette suite à partir de laquelle elle devient constante. Écrire une fonction :

```
point_fixe : ('a -> 'a) -> 'a -> 'a
```

telle que `point_fixe f a` cherche (à l'aide de la récurrence) et renvoie, s'il existe, le point fixe de `f` obtenu en partant de `a`. A partir de cette fonction, écrire une fonction

```
simplifier : expr -> expr
```

qui, étant donnée une expression `e`, itère sur `e` l'application de `eval_neutres` et `eval_sous_expr` jusqu'à ce que l'expression ne soit plus modifiée. Pourquoi cette fonction termine-t-elle ?

Exercice 8. [⋆⋆] Soit s un symbole. L'*expression dérivée* d'une expression e par rapport au symbole s, notée $\frac{\partial e}{\partial s}$ ou $\partial_s e$, est définie par induction sur la structure de e :

- si n est un nombre (exact ou approché), alors sa dérivée est le nombre 0.
- si e_1 et e_2 sont deux expressions, alors $\partial_s(e_1 + e_2) = \partial_s e_1 + \partial_s e_2$ et de même pour
- si e_1 et e_2 sont deux expressions, alors $\partial_s(e_1 \quad e_2) = (\partial_s e_1 \quad e_2) + (e_1 \quad \partial_s e_2)$
- si e est une expression, alors $\partial_s(\cos(e)) = 0. \quad (\partial_s e \quad \sin(e))$ et $\partial_s(\sin(e)) = \partial_s e \quad \cos(e))$
- la dérivée du symbole s par rapport à lui-même est le nombre 1.
- la dérivée d'un symbole t différent de s est le nombre 0.

Écrire une fonction :

```
deriv_expr : string -> expr -> expr
```

qui, étant donnés un symbole s et une expression e, calcule et renvoie l'expression dérivée de e par rapport à s.