TP n°5

XML et expressions symboliques

L'objectif de ce TP est double. En première partie comprendre la structure de document XML et les manipulations de base à l'aide de la bibliothèque `Xml-Light`. En deuxième partie, manipuler des expressions mathématiques à plusieurs *symboles*, pour les évaluer et les simplifier.

1 Définition de XML

XML est en fait un méta-langage[1] à balise permettant de définir des formats de documents et de créer des documents respectant ces formats. Ainsi, et c'est important, XML n'est pas un format de document comme tel. Les documents XML se composent d'unités de stockage appelées *entités*, qui contiennent des données analysables ou non. Les données analysables se composent de caractères, certains formant les données textuelles, et le reste formant le *balisage*. Le balisage décrit les structures logiques et de stockage du document. XML fournit un mécanisme pour imposer des contraintes à ces structures.

Exemple de structure XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogue>
    <!--ceci est un commentaire -->
    <famille n="f1">
        <ligne n="l11">
             <coef>2</coef>
             <GRAPH>av+ ro0 av+</GRAPH>
        </ligne>
    </famille>
</catalogue>
```

Remarques :
- `catalogue` est l'élément racine
- dans `<famille n="f1">`, `n` est un attribut de valeur "f1"
- dans `<coef>2</coef>`, 2 est du contenu texte
- un élément = balise d'ouverture + contenu + balise de fermeture
- le contenu d'un élément = un nom + ensemble d'attributs + ensemble d'éléments.

Parseur XML : Le parseur permet d'une part d'extraire les données d'un document XML (on parle d'analyse du document ou de parsing) ainsi que de vérifier éventuellement la validité du document.

1. c'est-à-dire qui permet de créer de nouveaux langages, comme par exemple XHTML

2 Xml-Light

Xml-Light est un parseur XML pour OCaml. Il fournit des fonctions pour transformer un document XML en une structure de données OCaml, la manipuler, et éventuellement générer un document XML en sortie. Il peut également aider à vérifier la validité d'un document vis-à-vis d'une DTD.

Structure xml en OCaml : un nœud XML est soit un Element avec des arguments de la forme $(tag\_name, attributes, children)$, soit un PCData suivi d'un texte.

```
type xml =
 | Element of (string * (string * string) list * xml list)
 | PCData of string
```

Fonctions de base :

```
(* Transforme un fichier XML en données XML. *)
val parse_file : string -> xml

(* Transforme en données XML une chaîne contenant un document XML. *)
val parse_string : string -> xml

(* Affiche les données XML en chaine de caractères de façon compacte. *)
val to_string : xml -> string
```

Exemple :

```
#directory "+xml-light";;
#load "xml-light.cma";;
let x = Xml.parse_file "agents.xml" in
    print_endline (Xml.to_string x)
```

Les deux premières instructions chargent la librairie Xml-Light. Ce programme parse le document *agents.xml* en type OCaml `Xml.xml` et affiche ensuite le résultat en chaines de caractères.

3 Définition d'une expression symbolique

Une *expression à plusieurs symboles*, ou *expression symbolique*, est un objet mathématique qui peut être :
- soit un nombre flottant
- soit l'une des formes suivantes : $C_1 + C_2$, $C_1 - C_2$, $C_1 \times C_2$, $\cos C_1$, $\sin C_2$, où C_1 et C_2 sont deux expressions symboliques
- soit un *symbole* (x, y, ab, etc.)

Par exemple :

$$((2. \times 1.) \ - (1. \times x)) \times ((19. - \cos(0.)) + \sin(theta))$$

est une expression qui contient deux symboles, x et $theta$.

Exercice 1 On donne les types suivants pour représenter les opérateurs binaires et unaires :

```
type binop = Plus | Moins | Fois
type unop  = Cos  | Sin
```

Définir le type `expr` pour représenter les expressions symboliques, sous la forme d'un type somme à quatre constructeurs : `Nombre`, `Binop`, `Unop` et `Symbole` (un symbole étant représenté par une chaîne de caractères), de sorte que l'expression e_0 ci-dessus sera représentée en OCaml par :

```
let e0 = Binop (Fois,
                Binop (Moins,
                       Binop (Fois,  Nombre 2.,  Nombre 1.),
                       Binop (Fois,  Nombre 1.,  Symbole "x")),
                Binop (Plus,
                       Binop (Moins, Nombre 19., Unop (Cos, Nombre 0.)),
                       Unop  (Sin,   Symbole "theta"))) ;;
```

Exercice 2 Écrire une fonction :

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

qui, étant donnée une expression, renvoie une chaîne de caractères représentant l'expression entièrement parenthésée sous forme lisible par un être humain. Par exemple, pour e_0, cette fonction renverra la chaîne :

```
"(((2.*1.)-(1.*x))*((19.-cos(0.))+sin(theta)))"
```

On pourra écrire au préalable des fonctions :
– `string_of_binop : binop -> string`
– `string_of_unop : unop -> string`
qui étant donné un opérateur, renvoient sa représentation lisible en chaîne de caractères.

Exercice 3 Un intérêt d'utiliser XML est de structurer des expressions symboliques de type `expr` en document XML pouvant être stocké, transmis entre deux programmes, envoyé à travers un réseau, etc. Écrire une fonction :

```
expr_of_xml : Xml.xml -> expr
```

qui lit une expression sous forme XML (correspondant par exemple au document *expr.xml* ci-dessous) et retourne une expression de type `expr`.

```
<?xml version="1.0"  encoding="UTF-8" ?>
<Binop>
  <Moins/>
  <Binop>
    <Fois/>
    <Nombre>2</Nombre>
    <Nombre>1</Nombre>
  </Binop>
  <Binop>
    <Fois/>
    <Nombre>1</Nombre>
    <Symbole>"X"</Symbole>
  </Binop>
</Binop>
```

4 Évaluation d'une expression symbolique

Exercice 4 Écrire une fonction :

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

telle que, si `l` est un *environnement* associant (sous la forme d'une liste d'association) chaque symbole à une valeur flottante, et si `e` est une expression, alors `eval_expr l e` calcule et renvoie la valeur de l'expression e sous l'environnement l. Par exemple, pour évaluer l'expression e_0 ci-dessus en prenant 3. pour x et 0. pour *theta*, on écrira :

```
eval_expr [("x", 3.); ("theta", 0.)] e0
```

ce qui renverra `-18.` (car $\cos 0 = 1$ et $\sin 0 = 0$).

Indications : On pourra utiliser la fonction `List.assoc : 'a -> ('a * 'b) list -> 'b`.
On pourra aussi écrire au préalable des fonctions :
– `valeur_binop : binop -> float -> float -> float`
– `valeur_unop : unop -> float -> float`
qui effectuent les opérations binaires et unaires en supposant que leurs arguments sont des valeurs flottantes déjà calculées.

5 Simplification d'une expression symbolique

Exercice 5 Écrire une fonction :

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

telle que, si e est une expression, alors `eval_sous_expr e` renvoie l'expression obtenue en remplaçant toute sous-expression de e ne contenant pas de symboles, par sa valeur. Cette passe est appelée *évaluation des sous-expressions constantes*.
Par exemple, dans l'expression e_0 ci-dessus, les sous-expressions $2.\times 1.$ et $19.-\cos 0.$ ne contiennent pas de symboles, elles pourront donc être remplacées par leurs valeurs, et alors `eval_expr_partielle e0` renverra l'expression : $(2. - (1. \times x)) \times (18. + \sin(theta))$
Faire des essais pour remarquer que, si e est une expression quelconque, alors, pour tout environnement l, `eval_expr l e` et `eval_expr l (eval_sous_expr e)` renvoient les mêmes valeurs.
On dit que `eval_sous_expr` est *correcte* vis-à-vis de l'évaluation des expressions.
Remarquer aussi que pour toute expression e :

$$\texttt{eval\_sous\_expr}(\texttt{eval\_sous\_expr}\ e) = \texttt{eval\_sous\_expr}\ e$$

On dit que `eval_sous_expr` est *idempotente*.

Exercice 6 Écrire une fonction :

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

telle que, si e est une expression, alors `eval_neutres e` renvoie l'expression obtenue en remplaçant toute sous-expression de e :
– de la forme $0. \times e'$ ou $e' \times 0.$ ou $e' - e'$ par $0.$
– de la forme $1 \times e'$, $e' \times 1$ ou $e' - 0$ ou $e' + 0.$ ou $0. + e'$ par e'

Par exemple, dans l'expression e_0 ci-dessus, la sous-expression 2×1 (resp. $1 \times x$) pourra être remplacée par 2 (resp. x), et alors `eval_neutres e0` renverra l'expression :

$$(2. - x) \times ((19. - \cos(0.)) + \sin(theta))$$

(l'évaluation des sous-expressions constantes n'a pas lieu)
Remarquer que `eval_neutres` est correcte vis-à-vis de l'évaluation des expressions, et qu'elle est idempotente.

Exercice 7 Écrire une fonction :

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

telle que, si f est une fonction et si a est une valeur applicable à f, alors `point_fixe f a` applique f sur a et recommence sur le résultat jusqu'à ce qu'il ne change plus.
On dit que `point_fixe f a` est un *point fixe* de f (en effet, si $x =$ `point_fixe f a`, alors `f` $x = x$). C'est le point fixe de f obtenu en partant de a.
Utiliser cette fonction pour écrire une fonction :

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

qui, étant donnée une expression, lui applique les deux fonctions `eval_neutres` et `eval_sous_expr` et continue jusqu'à ce que l'expression ne change plus.
Pourquoi cette fonction termine-t-elle ?