Projet de Programmation Fonctionnelle (LU2IN019)

calcul symbolique

Version du 5 juillet 2022

Dans ce projet, on propose de définir en OCaml un langage d'expressions mathématiques simples en notation préfixée. On cherche ensuite à évaluer (calculer la valeur) d'une expression de ce langage. Puis on cherche à calculer la dérivée symbolique d'une expression par rapport à une variable. Enfin, on cherche à simplifier des expressions à partir de règles telles $x + 0 = 0$ et $1 \times x = x$.
On étend ensuite le langage avec des fonctions trigonométriques.

1 Définition de type

Définir le type `exp` des expressions correspondant à la grammaire suivante. Une expression e est soit un entier, soit la constante `pi`, soit une variable x, soit la négation d'une expression, soit une addition, soit une soustraction, soit un produit, soit une division, soit l'exponentiation e^n où n est un entier positif.

$$
\begin{array}{lcl}
e & ::= & n \\
 & | & \texttt{pi} \\
 & | & x \\
 & | & (\texttt{-}\ e) \\
 & | & (\texttt{+}\ e_1\ e_2) \\
 & | & (\texttt{-}\ e_1\ e_2) \\
 & | & (\texttt{*}\ e_1\ e_2) \\
 & | & (\texttt{/}\ e_1\ e_2) \\
 & | & (\texttt{pow}\ e\ n) \\
 & & \\
n, m & ::= & \texttt{<integer>} \\
 & & \\
x, y & ::= & \texttt{<variable>}
\end{array}
$$

Ici, les expressions bien formées ont deux particularités :

1. les applications d'opérateurs sont systématiquement parenthèsées et suivent une notation préfixe (l'opérateur d'abord puis les arguments). L'expression `(- (pow` b `2) (* 4 (*` a c`)))` par exemple représente le discriminant : $b^2 - 4ac$;
2. il n'est pas possible de sur-parenthèser : l'expression (5) par exemple est mal formée.

Les lexèmes du langage d'expressions (parenthèse ouvrante, parenthèse fermante, entiers et noms littéraux) sont représentés en OCaml par des valeurs du type :

```
type token =
| INT of int  (* entier littéral *)
| NAME of tname (* un nom littéral *)
| OPEN  (* parenthèse ouvrante *)
| CLOSE (* parenthèse fermante *)

and tname = string (* noms littéraux, par exemple :
                      +, pi, x, yy, z_43, cos, tan, pow etc. *)
```

On a défini la fonction OCaml `lexer : string -> token list` qui, étant donnée une chaîne de caractères représentant une expression, construit la suite de lexèmes correspondante.
Par exemple, `lexer "(+ (* 2 x) 1)"` retourne :

```
[OPEN; NAME "+"; OPEN; NAME "*"; INT 2; NAME "x"; CLOSE; INT 1; CLOSE]
```

Compléter la fonction `of_string : string -> exp` qui construit une expression (de type `exp`) à partir d'une chaîne de caractères représentant cette expression.
Pour cela, **implémenter les fonctions auxiliaires suivantes** :

- `integer : int -> exp` telle que (`integer` n) construit l'expression (de type `exp`) correspondant à l'entier n ;
- `name : tname -> exp` qui, étant donné un nom littéral, construit une expression (de type `exp`) ;
- `apply : tname -> exp list -> exp` telle que `apply` *op* $[e_1; e_2; \cdots e_n]$ construit l'expression correspondant à l'opérateur 'op' appliqué aux expressions $e_1, e_2, \cdots e_n$.
  Par exemple `apply "+"` $[e_1; e_2]$ est l'expression représentant l'addition des expressions e_1 et e_2.
  On suivra la grammaire définie précédemment, laquelle comporte uniquement des opérateurs unaires ($-$) ou binaires ($+$, $-$, `*`, $/$, `pow`).

2 Evaluation d'expressions

Dans cette question, on veut écrire une fonction OCaml pour calculer la valeur d'une expression symbolique. On propose que la valeur d'une expression soit un nombre flottant OCaml, de type `float`[1].
Ce sera donc une valeur approchée : on peut alors tester l'égalité de deux valeurs à epsilon près.

Pour calculer la valeur d'une expression contenant des variables (par exemple : x, y, z), nous devons connaître l'environnement du calcul, assignant une valeur à chaque variable.

On représente les environnements par le type OCaml :

```
type tenv = (tname * float) list
```

Définir la fonction `eval : tenv -> exp -> float` telle que `eval` *env* e calcule la valeur de l'expression e dans l'environnement *env*.
Par exemple `eval` $[(x, 3.)]$ `(of_string "(+ x 1)")` s'évalue en 4.

3 Dérivation symbolique

La dérivée e' d'une expression e (selon une variable x implicite) peut être obtenue par des règles de réécriture, notamment :

$$
\begin{array}{lll}
n' & \longrightarrow & 0 \\
\texttt{pi}' & \longrightarrow & 0 \\
x' & \longrightarrow & 1 \\
y' & \longrightarrow & y \quad \text{si } 0 \neq x \\
(\texttt{+}\ e_1\ e_2) & \longrightarrow & (\texttt{+}\ e_1'\ e_2') \\
(\texttt{*}\ e_1\ e_2) & \longrightarrow & (\texttt{+ (*}\ e_1\ e_2'\ \texttt{) (*}\ e_1'\ e_2\texttt{))}
\end{array}
$$

Définir de façon similaire les règles de dérivations pour les fonctions (`-` e_1 e_2), (`/` e_1 e_2) où e_2 est non nul, et (`pow` e n).
Définir la fonction `derive : tname -> exp -> exp` telle que `derive` x e dérive (partiellement, suivant la variable x) l'expression e.

1. https://fr.wikipedia.org/wiki/IEEE_754

4 Affichage

Définir la fonction to_string : exp -> string telle que :

$$\texttt{lexer (to\_string (of\_string } s\texttt{)) = lexer } s$$

Par exemple lexer (to_string (of_string "(+ 1 2)")) = lexer "(+ 1 2)".
Indice : On peut utiliser l'opérateur OCaml de concaténation ^ : string -> string -> string. Par exemple : "foo" ^ "bar" s'évalue en "foobar".

5 Simplification d'expressions

définir la fonction simpl : exp -> exp telle que simpl e simplifie l'expression e suivant un ensemble de règles de réécriture donné ci-dessous. On note entre crochets l'évaluation d'une expression ne contenant que des constantes, par exemple : $[1+2] = 3$.

(+ n m) $\longrightarrow$ $[n+m]$
(+ e 0) $\longrightarrow$ e
(+ 0 e) $\longrightarrow$ e
(+ n e) $\longrightarrow$ (+ e n)

(- n m) $\longrightarrow$ $[n-m]$
(- 0 e) $\longrightarrow$ (- e)
(- e 0) $\longrightarrow$ e
(- e_1 (- e_2)) $\longrightarrow$ (+ e_1 e_2)

(- n) $\longrightarrow$ $[-n]$
(- (- e)) $\longrightarrow$ e
(- (- e_1 e_2)) $\longrightarrow$ (- e_2 e_1)
(- (* n e)) $\longrightarrow$ ($[-n]$ e)

(* n m) $\longrightarrow$ $[n \times m]$
(* 1 e) $\longrightarrow$ e
(* e 1) $\longrightarrow$ e
(* 0 e) $\longrightarrow$ 0
(* e 0) $\longrightarrow$ 0
(* n (* m e)) $\longrightarrow$ (* $[n \times m]$ e)
(* e n) $\longrightarrow$ (* e n)

(/ n m) $\longrightarrow$ $[n/m]$ si $[n/m] \in$<integer>
(/ e 1) $\longrightarrow$ e
(/ 0 e) $\longrightarrow$ 0
(/ e_1 (/ e_2 e_3)) $\longrightarrow$ (/ (* e_1 e_2) e_3)
(/ (/ e_1 e_2) e_3) $\longrightarrow$ (/ e_1 (* e_2 e_3))
(pow 0 n) $\longrightarrow$ 0
(pow 1 n) $\longrightarrow$ 1
(pow e 1) $\longrightarrow$ e
(pow e 0) $\longrightarrow$ 1

Implémenter les règles de factorisation suivantes, où $\mathrm{pgcd}(a, b)$ est le plus grand diviseur commun de a et b :

(+ e e) $\longrightarrow$ (* 2 e)
(- e e) $\longrightarrow$ 0

(* e e) $\longrightarrow$ (pow e 2)

`(*` e `(pow` e n`))` $\longrightarrow$ `(pow` e_1 $[n+1]$`)`
`(/` n m`)` $\longrightarrow$ `(/` $[n/k]$ $[m/k]$`)`
si $n \neq 0 \wedge m \neq 0 \wedge k = \mathrm{pgcd}(\mathrm{abs}(n), \mathrm{abs}(m)) \wedge k > 1$

`(+ (*` n e`)` m`)` $\longrightarrow$ `(*` k `(+ (*` $[n/k]$ e`)` $[m/k]$`))`
si $n \neq 0 \wedge m \neq 0 \wedge k = \mathrm{pgcd}(\mathrm{abs}(n), \mathrm{abs}(m)) \wedge k > 1$

`(- (*` n e`)` m`)` $\longrightarrow$ `(* k (- (*` $[n/k]$ e`)` $[m/k]$`))`
si $n \neq 0 \wedge m \neq 0 \wedge k = \mathrm{pgcd}(\mathrm{abs}(n), \mathrm{abs}(m)) \wedge k > 1$

`(-` m `(*` n e`))` $\longrightarrow$ `(*` k `(-` $[m/k]$ `(*` $[n/k]$ e`)))`
si $n \neq 0 \wedge m \neq 0 \wedge k = \mathrm{pgcd}(\mathrm{abs}(n), \mathrm{abs}(m)) \wedge k > 1$

6 Fonctions trigonométriques

Etendre le langage d'expressions avec les applications des fonctions trigonométriques `(cos` e`)`, `(sin` e`)`, `(tan` e`)`.
Mettre à jour vos fonctions OCaml `apply`, `eval`, `derive`, `to_string`, et `simpl`.

7 Extension libre

Etendre le langage d'expressions avec des fonctions et constantes de votre choix, par exemple la fonction racine carrée (`sqrt` e), le logarithme (`ln` e) et le nombre `e`[2].

On donne le type :

```
type assertion =
| Eval of tenv * string * float
| Derive of tname * string * string
| Simpl of string * string
```

`Eval(`env`,`s`,`v`)` dénote que `eval` env `(of_string` s`) =` v (le test d'égalité est à epsilon près).

`Derive(x,s,s')` dénote que `derive x (of_string` s`) = (of_string` s'`)`‘.

`Simpl(s,s')` dénote que `simpl (of_string` s`) = (of_string` s'`)`.

On donne :

```
let assert_eval (env:tenv) (s:string) (v:float) : assertion =
  Eval (env,s,v)

let assert_derive (x:tname) (s:string) (s':string) : assertion =
  Derive(x,s,s')

let assert_simpl (s:string) (s':string) : assertion =
  Simpl(s,s')
```

Testez vos fonctions `eval`, `derive` et `simpl` en définissant une variable `assertions` de type `assertion list` contenant une liste d'assertions vérifiées par votre extension.

2. `https://fr.wikipedia.org/wiki/E_(nombre)`

Voici un tel exemple de jeu de tests (simplifié) :

```
let assertions =
  [ assert_eval [("x",4.)] "(+ x 1)" 5. ;
    assert_derive "y" "(+ y 2)" "(+ 1 0)" ;
    assert_simpl "(+ x 0)" "x" ]
```

8 Bonus : variables locales

Ajouter au langage d'expressions la construction `(let` x `=` e_1 `in` e_2`)`.
On pourra ainsi écrire par exemple : `(let x = (+ 1 2) in (* 2` x`))`.
Mettre à jour votre fonction `eval`.

Indice : on pourra définir une fonction de construction `mk_let : tname -> exp -> exp` et ajouter à la fonction `of_string` le cas suivant :

```
| OPEN :: NAME "let" :: NAME x :: NAME "=" :: r ->
        (* reconnaît la construction [(let x = e1 in e2)].
           Par exemple [(let x = (+ x 1) in (/ x 2))] *)
        (match parse r with
         | Some e1, NAME "in" :: r1 ->
             (match parse r1 with
              | Some e2,CLOSE :: r2 -> Some(mk_let x e1 e2),r2
              | _,_ -> failwith ("syntax error: let ... in ???"))
         | _,_ -> failwith ("syntax error: let ??? in ..."))
```