Travaux Pratiques ML No.7

Cours d'Informatique de Deuxième Année

—Deug SSM M4—

Aujourd'hui...
Le but de ce TP est de travailler sur un petit langage fonctionnel. On va ainsi se définir une "syntaxe abstraite" pour ce langage que nous appellerons désormais `PLF` et on pourra alors "évaluer" les expressions `PLF`.

Le langage PLF

On associe à notre langage `PLF`, une syntaxe abstraite qui le décrit : on explicite ainsi ce qu'est une expression pour notre langage :

```
# type Expr =                         (* syntaxe abstraite du langage *)
#           Const of int
#         | Var of string
#         | Function of string * Expr
#         | App of Expr * Expr
#         | Let of string * Expr * Expr
#         | BinOp of Op * Expr * Expr
#         | Opp of Expr
# and Op = Plus | Minus | Times | Div ;;
```

A partir de cette syntaxe, on peut donner des expressions en `PLF`.

▸ **Exercice 2**. Donnez les expressions en `PLF` correspondant aux expressions *Caml-Light* suivantes :

```
# let f = function x -> x + 1 in f (2*3) ;;
# let g x y = x + y in g 2 3 ;;
# let comp f g x = f (g x) in 2 + comp (function x -> x + 1) (function x -> x - 1) (-2) ;;
# let x = 2 in (function x -> x / 2) x ;;
```

. . . .

. . . .

. . . .

On associe maintenant à chaque expression `PLF` une valeur c'est-à-dire on choisit de donner à telle ou telle expression telle ou telle valeur. Par exemple, intuitivement, on peut associer à l'expression `BinOp (Plus,a,b)`, la valeur correspondant à la somme des évaluations de `a` et `b`.

Une expression n'a de valeur que par rapport à un environnement. En effet, considérons l'expression suivante :

```
# let x = 2 in x + 1 ;;
```

L'évaluation de l'expression `x + 1` ne peut se faire que si `x` représente une certaine valeur : c'est à l'environnement de mémoriser cette valeur. Dans l'exemple précédent, le `let x = 2` augmente l'environnement en lui ajoutant le couple `(x,2)`, et l'évaluation de l'expression `x + 1` se fait sous l'environnement `[(x,2)]` ce qui explique que l'expression `x + 1` est évaluée à `3`. L'exemple suivant montre comment un environnement peut être utile dans l'évaluation d'une expression complexe :

```
# let x = 2 in (function y -> let x = 3 in x / y) x ;;
```

Cette évaluation se fait grossièrement en deux étapes : la première consiste à évaluer l'expression `(function y -> let x = 3 in x / y) x` sous l'environnement `[(x,2)]`. La seconde étape évalue `x / y` sous l'environnement `[(y,2);(x,3);(x,2)]`.

On définit donc les types `Env` et `Value` suivants :

```
# type Env = ENV of (string * Value) list    (* environnement *)
#
# and Value =                                (* valeur retournee apres evaluation *)
#          VConst of int
#        | Fun of string * Expr * Env
#        | Error ;;
```

▸ **Exercice 3**. Ecrire une fonction d'insertion `add_env` d'un couple `(variable,valeur)` dans un environnement.

. . . .

▸ **Exercice 4**. Ecrire une fonction `env_value` qui retourne la valeur d'une variable d'environnement.

. . . .

On fournit ci-dessous les règles utilisées pour l'évaluation. Il faut lire chacune des règles de la façon suivante: `value` représente l'évaluation de l'expression `expr` sous l'environnement `env`, ce que l'on écrit aussi

$$\mathtt{env} \vdash \mathtt{expr} \Rightarrow \mathtt{value}$$

Une règle, telle qu'elle est écrite ci-dessus, s'appelle un *jugement*. On a en particulier

$$[\,] \vdash \mathtt{Const\ n} \Rightarrow \mathtt{VConst\ n} \qquad \mathtt{env} \vdash \mathtt{Var\ x} \Rightarrow \mathtt{v}\ ((\mathtt{x},\mathtt{v}) \in \mathtt{env}) \qquad \mathtt{env} \vdash \mathtt{Function\ x}- > \mathtt{a} \Rightarrow \mathtt{Fun}(\mathtt{x},\mathtt{a},\mathtt{env})$$

le cas de l'application est à peine plus complexe:

$$\frac{\mathtt{env} \vdash \mathtt{a1} \Rightarrow \mathtt{Fun}(\mathtt{x},\mathtt{a},\mathtt{env'}) \qquad \mathtt{env} \vdash \mathtt{a2} \Rightarrow \mathtt{v} \qquad (\mathtt{x},\mathtt{v}) :: \mathtt{env'} \vdash \mathtt{a} \Rightarrow \mathtt{r}}{\mathtt{env} \vdash \mathtt{a1\ a2} \Rightarrow \mathtt{r}}$$

qu'il faut comprendre de la manière suivante: pour évaluer `a1 a2` sous un environnement `env`, il faut que `a1` s'évalue en une valeur fonctionnelle (il est évident qu'on ne peut traiter l'application que si `a1` est une fonction), que `a2` s'évalue en une valeur quelconque, et le résultat de l'évaluation de cette application sera le même que l'évaluation de la fonction dans laquelle le paramêtre `x` est remplacé par la valeur de `a2`.

Le cas du `Let` est plus simple :

$$\frac{\mathtt{env} \vdash \mathtt{e1} \Rightarrow \mathtt{v} \qquad (\mathtt{x},\mathtt{v}) :: \mathtt{env} \vdash \mathtt{e2} \Rightarrow \mathtt{r}}{\mathtt{env} \vdash \mathtt{Let}(\mathtt{x},\mathtt{e1},\mathtt{e2}) \Rightarrow \mathtt{r}}$$

▸ **Exercice 5**. Complétez l'évaluateur suivant afin qu'il puisse gérer le cas du `Let`.

```
# (* l'evaluateur lui-meme *)
# let rec eval env = eval_exp
#          where rec eval_exp = function
#          Const n -> VConst n
#        | Var x -> env_value x env
#        | Function (x,a) -> Fun(x,a,env)
#        | Let(x,exp1,exp2) -> ...
#
#
#
#
#        | App (exp1,exp2) -> let second_V = eval_exp exp2 in
#          begin
#          match second_V with Error -> Error | v2 -> let first_V = eval_exp exp1 in
#                 begin
#                 match first_V with
#                        Fun(x,exp,env2) -> eval (add_env (x,v2) env2) exp
#                      | _ -> Error
#                 end
#          end
#          ;;
```

▸ **Exercice 6**. Rajoutez l'évaluation des expressions arithmétiques à cet évaluateur.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .