▶ **MP – Option Informatique**
2ème TP Caml
sl@jachiet.com
http://jachiet.com/tps
Vendredi 11 octobre 2013

Interprétation

1 Expressions arithmétiques

1.1 Expression simples

On souhaite manipuler des expressions arithmétiques entières, pour cela je vous propose le type suivant :

```
type expr =
  | Int of int
  | Plus of expr*expr
  | Minus of expr*expr
  | Mult of expr*expr
  | Div of expr*expr
```

Ce type représente un arbre où les feuilles sont les "Int" et les nœuds sont les opérateurs binaires Plus, Minus, Mult, Div.
On remarque que les arbres représentent des calculs. À chaque arbre on peut associer un entier (la valeur [1] du résultat du calcul de cet arbre). Votre objectif c'est de calculer cette valeur. On dit aussi évaluer une expression.

▶ **Question 1** *Écrire une fonction récursive*

```
compute : expr -> int
```

qui calcule la valeur entière d'un arbre.

1.2 Expressions avec variables

La puissance de calcul de ces arbres est très limité. On peut avoir envie de faire beaucoup plus de choses avec. On considère donc maintenant une liste de paire $(string * int)$. Si la paire $("x", 3)$ fait partie de cette liste cela signifie que la valeur de la variable "x" c'est l'entier 3. On appelle généralement une telle liste un environnement. Si un index est présent deux fois dans l'environnement alors seule la première variable compte. Si un index non présent dans l'environnement est recherché vous devez lancer une erreur "unbound variable"

▶ **Question 2** *Écrire une fonction*

```
lookup : 'a -> ('a * 'b) list -> 'b
```

qui prend un argument index et une liste environnement et qui cherche dans l'environnement la valeur (de type 'b) associée à l'index (de type 'a) donné.

Le nouveau type est donc :

```
type expr =
  | Int of int
  | Plus of expr*expr
  | Minus of expr*expr
  | Mult of expr*expr
  | Div of expr*expr
  | Var of string
```

▶ **Question 3** *Ré-écrire la fonction*

```
compute : (string * int) list -> expr -> int
```

pour évaluer les expressions avec des variables et un environnement.

1.3 Expressions Let

On rajoute au type la construction

```
let a = v in b
```

où a est de type String et v,b sont de type expr.
Le type devient donc :

```
type expr =
  | Int of int
  | Plus of expr*expr
  | Minus of expr*expr
  | Mult of expr*expr
  | Div of expr*expr
  | Var of string
  | Let of string*expr*expr
```

1. Si l'on voulait définir de façon précise l'entier représenté, il faudrait définir ce qui s'appelle une sémantique sur les arbres, c'est à dire un sens mathématiquement défini de ce que représente chaque arbre. Par exemple, si l'on note S la sémantique on aurait $S(Plus(a,b)) = S(a) + S(b)$, $S(Div(a,b)) = \lfloor \frac{S(a)}{S(b)} \rfloor$, etc. J'aurais pu proposer un type où le Int (ou n'importe quel autre opérateur) serait remplacé par Zog_Zog, dans ce cas, la sémantique serait moins naturelle mais ce serait mathématiquement équivalent.

▶ **Question 4** *Ré-écrire*

```
compute : (string * int) list -> expr -> int
```

pour gérer cette nouvelle construction. Je rapelle que si on a une construction

```
let a = v in b
```

cela veut dire qu'il faut d'abord transformer v en une valeur et ensuite on ajoute à l'environnement l'association entre la variable a et la valeur (que l'on vient de calculer) de v.

1.4 Expression if-then-else

On rajoute maintenant la construction if-then-else et des expressions booléennes. Le type expression devient :

```
type expr =
    | Int of int
    | Plus of expr*expr
    | Minus of expr*expr
    | Mult of expr*expr
    | Div of expr*expr
    | Var of string
    | Let of string*expr*expr
    | If of cond*expr*expr
and cond =
    | Not of cond
    | And of cond*cond
    | Or of cond*cond
    | Equal of expr*expr
    | Less of expr*expr
```

Un élément $\text{If}(c, a, b)$ a la valeur de a quand c s'évalue à vrai et de b sinon.

▶ **Question 5** *Écrire les fonctions*

```
compute : (string * int) list -> expr -> int
computeBool : (string * int) list -> cond -> bool
```

2 Fonctions

On souhaite rajouter des fonctions à notre langage. Par ailleurs, ces fonctions peuvent aussi être des valeurs (avant seul les entiers étaient des valeurs). Il faut donc être capable de gérer plusieurs types de valeurs. Les types utilisés sont :

```
type val =
    | Ival of int
    | SomethingElse
and expr =
    | Int of int
    | Plus of expr*expr
    | Minus of expr*expr
    | Mult of expr*expr
    | Div of expr*expr
    | Var of string
    | Let of string*expr*expr
    | If of cond*expr*expr
and cond =
    | Not of cond
    | And of cond*cond
    | Or of cond*cond
    | Equal of expr*expr
    | Less of expr*expr
```

▶ **Question 6** *Écrire une fonction val_to_int qui transforme les Ival en int et qui lève une exception quand elle reçoit autre chose qu'un Ival.*

▶ **Question 7** *Ré-écrire les fonctions qu'il faut pour calculer les termes sur le nouveau type.*
On rajoute maintenant vraiment des fonctions à notre langage et ces fonctions sont des valeurs. Les arbres d'expressions s'évaluent maintenant soit en des valeurs entières, soit en des fonctions (comme en caml où une fonction peut elle même renvoyer une fonction).
Par ailleurs, il y a quelques détails à propos desquels il faut être vigilant. En effet, si on considère la valeur :

```
(let x = Int 42 in (fun y -> Var y*Var x)))
```

alors il faut faire bien attention à ne pas oublier l'environnement dans lequel la fonction $fun\ y -> y * x$ a été définie sinon quoi on ne sait plus ce que x signifie. Ce qui est parfois fait, par exemple en caml, c'est de stocker l'environnement de la fonction pour pouvoir ensuite appeler cette fonction avec l'environnement dans lequel elle a été crée. On appelle ce stockage une clôture (ou closure en anglais) de la fonction.
Par ailleurs les valeurs retournées n'étant plus nécessairement des entiers, il faut donc définir un comportement si on essaye de comparer ou d'additionner la valeur 3 avec $fun\ x \to x$. Le plus simple c'est d'échouer avec un message d'erreur.
Types utilisés dans cette partie :

```
type val =
    | Ival of int
    | Lam of string*expr*(string*val) list
and expr =
    | Int of int
    | Plus of expr*expr
    | Minus of expr*expr
    | Mult of expr*expr
    | Div of expr*expr
    | Var of string
    | Let of string*expr*expr
    | Fun of string*expr
    | App of expr*expr
    | If of cond*expr*expr
and cond =
    | Not of cond
    | And of cond*cond
    | Or of cond*cond
    | Equal of expr*expr
    | Less of expr*expr
```

▶ **Question 8 *** *Écrire les fonctions mutuellement récursives :*

```
compute : (string * val) list -> expr -> val
computeBool : (string * val) list -> cond -> bool
```

Il peut être intéressant pour factoriser le code des opérateurs Plus, Minus, Div, Mult d'écrire une troisième fonction mutuellement récursive qui prend deux expressions, un environnement et la fonction[2] *à appliquer sur les valeurs entières pour renvoyer le Ival qui faut.*

▶ **Question 9 *** *Programmer la fonction factorielle dans notre langage.*

2. donc parmi $fun\ x\ y \to x + y$, $fun\ x\ y \to x - y$,$fun\ x\ y \to x \times y$,$fun\ x\ y \to x/y$

▶ **MP – Option Informatique**
2ème TP Caml
sl@jachiet.com
http://jachiet.com/tps
Vendredi 11 octobre 2013

Interprétation

Un corrigé

▶ Question 1

```
let rec compute = function
  | Int(i) -> i
  | Plus(a,b) -> (compute a) + (compute b)
  | Minus(a,b) -> (compute a) - (compute b)
  | Mult(a,b) -> (compute a) * (compute b)
  | Div(a,b) -> (compute a) / (compute b)

let a = compute (Plus (Div(Int 5,Int 2), Mult(Int 5,Int 6)))
```

▶ Question 2

```
let rec lookup s = function
  | [] -> failwith ''unbound variable''
  | (a,v)::q -> if s=a then v else lookup s q
```

▶ Question 3

```
let rec compute env = function
  | Int(i) -> i
  | Plus(a,b) -> (compute env a) + (compute env b)
  | Minus(a,b) -> (compute env a) - (compute env b
  | Mult(a,b) -> (compute env a) * (compute env b)
  | Div(a,b) -> (compute env a) / (compute env b)
  | Var(s) -> lookup s env
```

▶ Question 4

```
let rec compute env = function
  | Int(i) -> i
  | Plus(a,b) -> (compute env a) + (compute env b)
  | Minus(a,b) -> (compute env a) - (compute env b)
  | Mult(a,b) -> (compute env a) * (compute env b)
  | Div(a,b) -> (compute env a) / (compute env b)
  | Var(s) -> lookup s env
  | Let(s,a,b) -> let v = compute env a in compute ((s,v)::env) b

let a = compute [] (Let(''a'',Int 42,Mult(Var(''a''),Var(''a'')))) ;;
```

▶ Question 5

```
let rec compute env = function
  | Int(i) -> i
  | Plus(a,b) -> (compute env a) + (compute env b)
  | Minus(a,b) -> (compute env a) - (compute env b)
  | Mult(a,b) -> (compute env a) * (compute env b)
  | Div(a,b) -> (compute env a) / (compute env b)
  | Var(s) -> lookup s env
  | Let(s,a,b) -> let v = compute env a in compute ((s,v)::env) b
  | If(c,a,b) -> if computeBool env c
    then compute env a
    else compute env b

and computeBool env = function
  | Not b -> not computeBool env b
  | And(b1,b2) -> computeBool env b1 && computeBool env b2
  | Or(b1,b2) -> computeBool env b1 || computeBool env b2
  | Equal(a,b) -> (compute env a)=(compute env b)
  | Less(a,b) -> (compute env a)<(compute env b)
```

▶ Question 8

```
let rec compute env = function
  | Int(i) -> Ival i
  | Plus(a,b) -> bin_int (fun x y -> x+y) a b env
  | Minus(a,b) -> bin_int (fun x y ->x-y) a b env
  | Mult(a,b) -> bin_int (fun x y -> x*y ) a b env
  | Div(a,b) -> bin_int (fun x y ->x/y) a b env
  | Var(v) -> lookup v env
  | Let(s,a,b) -> let v = compute env a in compute ((s,v)::env) b
  | Fun(s,a) -> Lam(s,a,env)
  | App(f,e) ->
    let arg = compute env e in
    (match compute env f with
      | Lam(s,v,clos) ->compute ((s,arg)::clos) v
      | _ -> failwith ''int'')
  | If(b,ife,elsee) ->
    if computeBool env b
    then compute env ife
    else compute env elsee
and bin_int f x y env =
  (match (compute env x,compute env y) with
    | (Ival a,Ival b) -> Ival(f a b)
    | _ -> failwith ''cast int failed'')
and computeBool env = function
  | Not b -> not computeBool env b
  | And(b1,b2) -> computeBool env b1 && computeBool env b2
  | Or(b1,b2) -> computeBool env b1 || computeBool env b2
  | Equal(e1,e2) -> (match (compute env e1,compute env e2) with
      | Ival a, Ival b -> a=b
      | _ -> failwith ''='')
  | Less(e1,e2) -> (match(compute env e1,compute env e2) with
      | Ival a,Ival b -> a < b
      | _ -> failwith ''<'')
```

▶ Question 9

```
let a = compute [] (Let(''f'',Fun(''fac'', Fun(''n'', If( Equal (Var
  ''n'',Int 1), Int 1, Mult(Var ''n'',App(App(Var ''fac'',Var
  ''fac''),Minus(Var ''n'',Int (1)))) ))),App(App(Var ''f'',Var
  ''f''),Int 6)));;
```