Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 5 – Corrigé
Automates, expressions arithmétiques

1 Simulation d'automates

a. L'automate suivant reconnait le langage $L_1 = a(a|b)^*b$. Pour ne pas avoir de transition non définie en fonctionnement "normal" (en partant de l'état 0 et en n'utilisant que les lettres a et b), on définit un état puits 9 auxquel mènent les transitions non initialement définies.

```
let arcs1 = fun
      | 0 'a' -> 1 | 0 'b' -> 9
      | 1 'a' -> 1 | 1 'b' -> 2
      | 2 'a' -> 1 | 2 'b' -> 2
      | 9 'a' -> 3 | 9 'b' -> 3
      | _ -> failwith "Transition non définie" ;;
```

L'automate complet s'écrit alors :

```
let automate1 =
      { initial = 0 ;
        arcs    = arcs1 ;
        finals  = [ 3 ] } ;;
```

b.
- L'état initial q_0 se trouve par `auto.initial`
- Les transitions successives $q_{i+1} = \delta(q_i, \alpha_i)$ s'écrivent `etat := auto.arcs!etat mot.[i]`
- L'acceptation ou non du mot $q_{n-1} \in \mathcal{F}$ s'écrit `appartient etat auto.finals`

```
let rec appartient x = function
  | [] -> false
  | t::q -> if (t = x) then true
                       else appartient x q ;;

let reconnaitre auto mot =
  let etat = ref auto.initial
  in for i = 0 to (string_length mot) do
       etat := auto.arcs !etat mot.[i]
     done ;
     appartient !etat auto.finals ;;
```

c.* Les automates indéterministes peuvent se simuler en raisonnant sur des listes d'états Q_i. Les transitions deviennent alors $Q_{i+1} = \bigcup_{q \in Q_i} \delta(q, \alpha_i)$, ce qui se traduirait en Caml par une expression du type `etats := union (map auto.arcs!etats)`. L'acceptation finale du mot demande alors une intersection entre la liste d'états et celles des états finals de l'automate.

2 Expressions arithmétiques

2.1 Évaluation simple

d.

```
let rec ecrit = function
   | Const n -> string_of_int n
   | Add (a,b) -> "(" ^ (ecrit a) ^ ")+(" ^ (ecrit b) ^ ")"
   | Sub (a,b) -> "(" ^ (ecrit a) ^ ")-(" ^ (ecrit b) ^ ")"
   | Mul (a,b) -> "(" ^ (ecrit a) ^ ")*(" ^ (ecrit b) ^ ")"
   | Div (a,b) -> "(" ^ (ecrit a) ^ ")/(" ^ (ecrit b) ^ ")"  ;;
```

Le code ci-dessus produit un parenthésage suffisant mais un peu exagéré. Pour diminuer le nombre de parenthèses “inutiles”, il faut examiner différents cas selon que les termes d’un produit soient des additions ou non.

```
let rec parenth = function
   | Const n -> string_of_int n
   | p -> "(" ^ (ecrit p) ^ ")"
and parplus p = match p with
   | Mul _ | Div _ -> ecrit p
   | _ -> parenth p
and ecrit = function
   | Const n -> string_of_int n
   | Add (e,f) -> (parplus e) ^ "+" ^ (parplus f)
   | Sub (e,f) -> (parplus e) ^ "-" ^ (parplus f)
   | Mul (e,f) -> (parenth e) ^ "*" ^ (parenth f)
   | Div (e,f) -> (parenth e) ^ "/" ^ (parenth f) ;;
```

e.

```
exception Division_par_zero ;;

let rec evalue = function
    | Const  n -> n
    | Add (a,b) -> (evalue a) + (evalue b)
    | Sub (a,b) -> (evalue a) - (evalue b)
    | Mul (a,b) -> (evalue a) * (evalue b)
    | Div (a,b) -> let bb = evalue b
                   in if (bb = 0)
                      then raise Division_par_zero
                      else (evalue a) / bb  ;;
```

2.2 Gestion des variables locales : les environnements

f. On peut par exemple ajouter à la fonction `ecrit` les cas suivants :

```
   | Var s -> s
```

```
   | Let (cle, valeur, expr) ->
       "let " ^ cle ^ " = " ^ (ecrit valeur) ^ " in " ^ (ecrit expr)
```

... avec des listes d'association

g.

```
exception Variable_non_definie ;;

let rec associe l x = match l with
  | [] -> raise Variable_non_definie
  | (cle,valeur)::q -> if (cle = x) then valeur
                                    else associe q x ;;
```

h.

```
let rec aux1 env = function
    | Const  n -> n
    | Add (a,b) -> (aux1 env a) + (aux1 env b)
    | Sub (a,b) -> (aux1 env a) - (aux1 env b)
    | Mul (a,b) -> (aux1 env a) * (aux1 env b)
    | Div (a,b) -> let bb = aux1 env b
                   in if (bb = 0)
                      then raise Division_par_zero
                      else (aux1 env a) / bb
    | Var cle -> associe env cle
    | Let (cle, valeur, expr) -> aux1 ((cle, (aux1 env valeur))::env) expr ;;

let evalue1 = aux1 [] ;;
```

Vous avez peut-être commencé par étendre l'environnement en écrivant seulement `(cle, valeur)::env`. Or la valeur est une expression qui doit aussi être évaluée. Dans ce cas il faudrait le faire lors de l'appel de la variable (`Var cle -> evalue env (associe env cle)`), mais cette évaluation se fera à chaque appel, ce qui est très couteux.

... avec des environnements fonctionnels

i. Un appel de `associe` sur un environnement vide fait lever une exception. On fait la même chose pour définir l'environnement fonctionnel vide :

```
let env_vide x = raise Variable_non_definie ;;
```

Pour étendre un environnement fonctionnel `env` avec une nouvelle paire `cle, valeur`, il faut expliciter le comportement du nouvel environnement lorsqu'on lui passe un argument `x` : il renvoie `valeur` si `x` vaut la nouvelle `cle`, et sinon il se comporte comme l'ancien `env`.

```
let etend2 env cle valeur x =
          if (cle = x) then valeur
                       else env x ;;
```

j.

```
let rec aux2 env = function
    | Const  n -> n
    | Add (a,b) -> (aux2 env a) + (aux2 env b)
    | Sub (a,b) -> (aux2 env a) - (aux2 env b)
    | Mul (a,b) -> (aux2 env a) * (aux2 env b)
    | Div (a,b) -> let bb = aux2 env b
                   in if (bb = 0)
                      then raise Division_par_zero
                      else (aux2 env a) / bb
    | Var cle -> env cle
    | Let (cle, valeur, expr) -> aux2 (etend2 env cle (aux2 env valeur)) expr ;;

let evalue2 = aux2 env_vide ;;
```

2.3 S'il vous reste du temps...

k. Une façon brutale mais simple pour vérifier que chaque variable est définie est de tenter d'évaluer l'expression et de rattraper éventuellement l'exception `Variable_non_definie`. Cette technique ne rattrapera pas les autres exceptions (comme une `Division_par_zero`). Cela permet justement de gérer proprement plusieurs erreurs différentes.

```
let correct expr = try
                      evalue1 expr ; true
                   with
                      | Variable_non_definie -> false ;;
```

Pour les trois dernières questions, voici quelques indices... n'hézsitez pas à me demander plus de précisions ou à me montrer vos œuvres.

l. On peut mettre en place un compteur (utilisant une référence) pour faire semblant d'évaluer le coût de l'évaluation d'une expression; ce compteur sera incrémenté selon le coût de chaque opérateur. On peut avoir l'impression que l'évaluation reste linéaire en la taille de l'expression... mais c'est inexact : avec les listes d'association, l'évaluation d'une expression avec n variables différentes peut conduire à $\mathcal{O}(n)$ appels de `associe` qui sont à chaque fois linéaires !

m.* L'écriture et la lecture de variables globales sont plus faciles que les variables locales en utilisant par exemple un tableau global `memoire` avec des instructions telles que :

```
| Ecrire (i,e) -> let ee = (evalue e) in memoire.[i] <- e ; ee
| Lire(i) -> memoire.[i]
```

Il est alors possible de créer l'instruction `Sequence (e1, e2)` qui exécute l'expression `e1` puis l'expression `e2` (cela n'a de sens que si `e1` a des *effets de bord* comme `Ecrire`).

n.* Deux expressions égales avec des variables libres peuvent être vues comme une équation. Vous pouvez essayer de résoudre ce type d'équations, mais c'est une autre histoire...