Expressions Arithmétiques

1 Évaluation simple

On désire réaliser quelques fonctions pour traiter les expressions arithmétiques composées de constantes entières $(0, 3, 12 \ldots)$ et d'opérateurs pris parmi $+, -, *, /$ (division entière). Le type suivant sera utilisé :

```
type expr =
| Const of int
| Add of expr * expr
| Sub of expr * expr
| Mul of expr * expr
| Div of expr * expr
```

On manipulera donc des expressions de la forme Add(Const 2, Const 2).

▶ **Question 1** *Écrire une fonction* ecrit *qui affiche une expression arithmétique avec un parenthésage correct.*

```
val ecrit : expr -> string
```

▶ **Question 2** *Écrire une fonction* evalue *qui évalue récursivement une expression arithmétique. Le cas de la division par zéro sera traité par la levée d'une exception.*

```
val evalue : expr -> int
```

2 Gestion des variables locales : les environnements

On souhaite enrichir les expressions arithmétiques par l'ajout de déclarations de variables. Pour cela, ajoutez au type des expressions les constructeurs suivants :

```
type expr =
| ...
| Var of string
| Let of string * expr * expr
```

L'expression **let** x = 4 **in** x + 1 sera ainsi représentée par Let ("x", Const 4, Add(Var "x", Const 1)).

▶ **Question 3** *Modifier la fonction* ecrit *pour gérer les nouveaux cas.*

La modification de evalue est un peu plus complexe : lors de l'évaluation de **let** x = 4 **in** x + 1, le programme doit se souvenir que x vaut 4 pour evaluer $x + 1$. On utilise pour cela un environnement qui associe à un nom de variable (la clé) une valeur. Cet environnement initialement vide sera passé récursivement à chaque appel de evalue.

▶ **Question 4** *Écrivez la définition d'un type* env *pouvant représenter un environnement et écrivez la définition de* env_vide *qui correspond à l'environnement vide et une fonction* etend *qui étend un environnement. Modifier la fonction* evalue *en conséquence.*

```
val env_vide : env
val etend : env -> string -> int -> env
val evalue : env -> expr -> int
```

3 Lexer

L'objet de cette partie est de programmer un analyseur lexical (ou lexer). Son rôle consiste à :

- éliminer les "bruits" du texte source : commentaires, espaces, ...
- reconnaître les opérateurs et les mots-clés : +, let, ...
- reconnaître les chaînes de caractères, les identificateurs et les constantes numériques

Le processus consiste à partir d'une chaîne de caractères, à identifier les éléments (appelé lexèmes) pour les ranger par catégories. Le type de lexèmes que l'on va considérer est le suivant :

```
type lexeme =
| Nombre of int
| Op of (int * string)
| Parg
| Pard
```

On oublie pour l'instant les variables.

- Nombre(n) représente une valeur numérique, on se limite aux valeurs entières ici ;
- Op(p, n) représente une opération binaire, p est la priorité de l'opération (1 pour + et −, 2 pour ∗ et /) et n est le nom de cet opérateur ("+" pour +) ;
- Parg et Pard représentent les parenthèses gauche et droite.

▶ **Question 5** *Écrire une fonction* lex_int *prenant en argument une chaîne de caractère et un entier. Elle doit renvoyer l'entier écrit à partir de la position donnée en argument dans la chaîne de caractère ainsi que la première position après cet entier.*

```
val lex_int : string -> int -> (int * int)
```

Par exemple lex_int *"53*24+1"* 3 *doit renvoyer le couple* (24,5).

▶ **Question 6** *Écrire une fonction qui analyse une chaîne de caractères et la décompose en une liste de lexèmes.*

```
val lexer : string -> lexeme list
```

4 Parser

On va maintenant programmer un analyseur syntaxique. L'analyse syntaxique consiste à mettre en évidence la structure du programme, en pratique on va transformer une liste de lexème en une expression arithmétique de type expr. Cette étape se fait de la manière suivante : en stockant dans une pile les lexèmes rencontrés et en réduisant à chaque étape le haut de la pile lorsque c'est possible selon les règles :

1. remplacer les nombres x par l'expression correspondant à la constante x ;

2. remplacer la séquence x f y g où f et g sont des opérateurs binaires et x, y des expressions par z g si la priorité de f est supérieure ou égale à celle de g, où z est l'expression correspondante à f x y ;
3. remplacer la séquence x f y) où f est un opérateur binaire et x, y des expressions par z) avec z l'expression f x y ;
4. remplacer la séquence (x) où x est une expression par l'expression x.

La troisième règle s'applique également si au lieu d'avoir une parenthèse on a atteint la fin de l'expression. Selon ces règles, si deux opérations binaires ont même priorité alors celle de gauche doit être effectuée en premier, ce qui permet d'évaluer correctement une expression telle que x − y − z.
Pour implémenter cet algorithme on a besoin d'une structure de donnée représentant une pile contenant des éléments de type lexeme et des éléments de type expr.

▶ **Question 7** *Trouvez une manière de représenter ce type de pile et implémentez les fonctions suivantes.*

```
val initialise : lexeme list -> pile
val empile : expr -> pile -> pile
```

▶ **Question 8** *Écrivez une fonction* step *réalisant une étape sur la pile. Elle devra renvoyer une exception dans le cas où il n'y a plus rien à remplacer.*

```
val step : pile -> pile
```

▶ **Question 9** *Écrivez la fonction qui réalise l'analyse syntaxique, on renverra une exception dans le cas où l'expression n'est pas syntaxiquement correcte.*

```
val parser : lexeme list -> expr
```

Vous pouvez tester les fonctions que vous avez programmé pour évaluer des chaînes de caractères du genre *"2+(3∗4)/5"*.

5 Les variables

L'idéal serait maintenant d'autoriser les variables et leurs déclarations. Pour cela on va ajouter ce qu'il faut aux lexèmes.

```
type lexeme =
| ...
| Equals
| Keyword of string
| Name of string
```

Où Keyword contient des mots clés (**let**,**in**) et Name les noms de variables.

▶ **Question 10** *Sur le modèle de* lex_int *, écrivez une fonction* lex_string *(on peut supposer que tout les noms seront écrit en minuscule). Et modifier la fonction* lexer *pour accepter les chaînes de caractères du type* **let** x = 2 **in** 3 ∗ x.

```
val lex_string : string -> int -> (string * int)
```

▶ **Question 11** *Modifier les fonction* step *et* parser *pour accepter les expressions avec des variables.*

Plutôt que d'évaluer forcement les expressions vers des entiers on voudrait pouvoir évaluer les expressions avec des variables libres (qui n'ont pas été initialisé par un **let**). Par exemple que **let** x = 5 **in** y + x s'évalue en y + 5.

▶ **Question 12** *Modifier la fonction évalue de façon à accepter les variables libres.*

```
val evalue : expr -> string
```

6 Manipulation des expressions

▶ **Question 13** *Écrire des fonctions* add, sub, mul, div *qui réalisent les opérations correspondantes sur les arbres d'expressions, en faisant quelques simplifications. Par exemple pour* add (Var *"y"*) (Const 0) *on renverra* Var *"y"*.

```
val add : expr -> expr -> expr
val sub : expr -> expr -> expr
val mul : expr -> expr -> expr
val div : expr -> expr -> expr
```

▶ **Question 14** *Écrivez une fonction* derive *qui dérive une expression par rapport à une variable libre donnée en argument.*

```
val derive : string -> expr -> expr
```

▶ **Question 15** *Ajouter la dérivation aux expressions et aux differents niveaux (lexer, parser, évaluation) de façon à permettre l'évaluation d'expressions avec dérivées. Comme par exemple* **let** y = 4 **in** dif x **of** 3 ∗ y / x.

Expressions Arithmétiques

Un corrigé

▶ Question 1

```
let rec ecrit = function
  | Const x -> string_of_int x
  | Add (x,y) -> "("^ecrit x^"+"^ecrit y^")"
  | Sub (x,y) -> "("^ecrit x^"-"^ecrit y^")"
  | Mul (x,y) -> "("^ecrit x^"*"^ecrit y^")"
  | Div (x,y) -> "("^ecrit x^"/"^ecrit y^")"
```

▶ Question 2

```
let rec evalue = function
  | Const x -> x
  | Add (x,y) -> evalue x + evalue y
  | Sub (x,y) -> evalue x - evalue y
  | Mul (x,y) -> evalue x * evalue y
  | Div (x,y) -> evalue x / evalue y
```

▶ Question 3

```
...
| Var x -> x
| Let (x,y,z) -> "let "^x^" = "^ecrit y^" in "^ecrit z
```

▶ Question 4

```
type env = string -> int

exception UndefinedVariable of string

let (env_vide:env) =
  fun s -> raise (UndefinedVariable s)

let etend (e:env) s v =
  ((fun x -> if x = s then v else e x):env)

let rec evalue env = function
  | Const x -> x
  | Add (x,y) -> evalue env x + evalue env y
  | Sub (x,y) -> evalue env x - evalue env y
  | Mul (x,y) -> evalue env x * evalue env y
  | Div (x,y) -> evalue env x / evalue env y
  | Var x -> env x
  | Let (x,y,z) -> evalue (etend env x (evalue env y)) z
```

▶ Question 5

```
let lex_int s i =
  let rec aux accu j =
    if j < String.length s
    then
      if s.[j] >= '0' && s.[j] <= '9'
      then aux (10*accu + int_of_char s.[j] - int_of_char '0') (j+1)
      else accu, j
    else accu, j
  in aux 0 i
```

▶ Question 6

```
exception LexicalError of int

let lexer s =
  let res = ref [] in
  let i = ref 0 in
    while !i < String.length s
    do
      match s.[!i] with
| ' ' -> incr i
| '(' -> incr i; res := Parg :: !res
| ')' -> incr i; res := Pard :: !res
| '+' -> incr i; res := Op (1,"+") :: !res
| '-' -> incr i; res := Op (1,"-") :: !res
| '*' -> incr i; res := Op (2,"*") :: !res
| '/' -> incr i; res := Op (2,"/") :: !res
| x ->
    if x >= '0' && x <= '9'
    then
      (let y , j = lex_int s !i in i := j ; res := Nombre y :: !res)
    else raise (LexicalError !i)
    done;
    List.rev !res
```

▶ Question 7

```
type pile = Nil | Lex of lexeme * pile | Expr of expr * pile

let rec initialise = function
  | [] -> Nil
  | l :: s -> Lex (l,initialise s)

let empile e p = Expr (e,p)
```

▶ Question 8

```
let expr_of_op n x y =
  match n with
    | "+" -> Add (x,y)
    | "-" -> Sub (x,y)
    | "*" -> Mul (x,y)
    | "/" -> Div (x,y)

let rec step = function
  | Nil -> raise Finished
  | Lex (Nombre x,p) -> empile (Const x) p
  | Expr (x,Lex (Op(pr1,n1),Expr(y,Lex (Op(pr2,n2),p))))
      when pr1 >= pr2 -> Expr (expr_of_op n1 x y,Lex (Op(pr2,n2),p))
  | Expr (x,Lex (Op(pr1,n1),Expr(y,Lex (Pard,p))))
    -> Expr (expr_of_op n1 x y,Lex (Pard,p))
  | Expr (x,Lex (Op(pr1,n1),Expr(y,Nil)))
    -> Expr (expr_of_op n1 x y,Nil)
  | Lex (Parg,(Expr(e,Lex(Pard,p)))) -> Expr(e,p)
  | Lex (x,p) -> Lex (x, step p)
  | Expr (x,p) -> Expr (x, step p)
```

▶ Question 9

```
exception SyntaxError of pile

let parser l =
  let p = initialise l in
  let rec aux x =
    try
      let y = step x in aux y
    with Finished -> x
  in match aux p with
    | Expr (x,Nil) -> x
    | y -> raise (SyntaxError y)

let test = evalue env_vide (parser (lexer "2 + (3 * 4 ) / 5"))
```

▶ Question 10

```
let lex_string s i =
  let rec aux accu j =
    if j < String.length s
    then
      if s.[j] >= 'a' && s.[j] <= 'z'
      then aux (accu ^ String.make 1 s.[j]) (j+1)
      else accu, j
    else accu, j
  in aux "" i

let lexer s =
  ...
  | '=' -> incr i; res := Equals :: !res
  | x ->
    if x >= '0' && x <= '9'
    then
     (let y , j = lex_int s !i in
      i := j ; res := Nombre y :: !res)
    else if x >= 'a' && x <= 'z'
    then
     (let y , j = lex_string s !i in
      i := j ;
      match y with
       | "let" | "in" -> res := Keyword y :: !res
       | x -> res := Name y :: !res)
    else raise (LexicalError !i)
  done;
...
```

▶ Question 11

```
let rec step = function
  ...
 | Lex (Name x,p) -> empile (Var x) p
 | Expr (x,Lex (Op(pr1,n1),Expr(y,Lex (Keyword "in",p))))
     -> Expr (expr_of_op n1 x y,Lex (Keyword "in",p))
 | Lex ( Keyword "let",
    Expr ( Var x,
      Lex (Equals,
        Expr(e ,
          Lex(Keyword "in",
   Expr (f,Nil)))))) ->
              Expr(Let(x,e,f),Nil)
  | Lex ( Keyword "let",
  Expr ( Var x,
 Lex (Equals,
       Expr(e ,
   Lex(Keyword "in",
        Expr (f,Lex(Pard,p))))))) ->
       Expr(Let(x,e,f),Lex(Pard,p))
  | Lex ( Keyword "let",
  Expr ( Var x,
 Lex (Equals,
       Expr(e ,
   Lex(Keyword "in",
        Expr (f,Lex(Keyword "in",p))))))) ->
       Expr(Let(x,e,f),Lex(Keyword "in",p))
  ...
```

▶ Question 12

```
type env = string -> expr

let rec aux_evalue env = function
  | Const x -> Const x
  | Add (x,y) ->
      (match evalue env x, evalue env y with
| Const a , Const b -> Const (a+b)
| e,f -> Add(e,f))
  | Sub (x,y) ->
      (match evalue env x, evalue env y with
| Const a , Const b -> Const (a-b)
| e,f -> Sub(e,f))
  | Mul (x,y) ->
      (match evalue env x, evalue env y with
| Const a , Const b -> Const (a*b)
| e,f -> Mul(e,f))
  | Div (x,y) ->
      (match evalue env x, evalue env y with
| Const a , Const b -> Const (a/b)
| e,f -> Div(e,f))
  | Var x -> (try env x with UndefinedVariable y -> Var x)
  | Let (x,y,z) -> evalue (etend env x (evalue env y)) z
```

▶ Question 13

```
let add x y =
  if x = Const 0 then y
  else if y = Const 0 then x
  else Add (x,y)

let sub x y =
  if y = Const 0 then x else Sub (x,y)

let mul x y =
  if x = Const 0 || y = Const 0 then Const 0
  else if x = Const 1 then y
  else if y = Const 1 then x
  else Mul (x,y)

let div x y =
  if x = Const 0 then Const 0
  else if y = Const 1 then x
  else Div (x,y)
```

▶ Question 14

```
let derive dx =
  let rec aux env = function
    | Const x -> Const 0
    | Var x -> if x = dx then Const 1 else
(try env x with UndefinedVariable y -> Const 0)
    | Add (x,y) -> add (aux env x) (aux env y)
    | Sub (x,y) -> mul (aux env x) (aux env y)
    | Mul (x,y) ->
add (mul (aux env x) y) (mul x (aux env y))
    | Div (x,y) ->
div
  (sub (mul (aux env x) y)
     (mul x (aux env y)))
  (mul y y)
    | Let (x,y,z) ->
if x = dx then Let (x,y,z)
else Let(x,y,aux (etend env x (aux env y)) z)
  in aux env_vide
```

▶ Question 15

```
let evalue e =
 ...
 | Dif (x,y) -> aux env (derive x y)
 ...

let rec step = function
 ...
  | Lex ( Keyword "dif",
     Expr ( Var x,
       Lex(Keyword "of",
         Expr (e,Nil)))) ->
       Expr(Dif(x,e),Nil)
  | Lex ( Keyword "dif",
       Expr ( Var x,
         Lex(Keyword "of",
           Expr (e,Lex(Pard,p))))) ->
       Expr(Dif(x,e),Lex(Pard,p))
  | Lex ( Keyword "dif",
      Expr ( Var x,
       Lex(Keyword "of",
        Expr (e,Lex(Keyword"in",p))))) ->
       Expr(Dif(x,e),Lex(Keyword"in",p))
 ...
```