IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP2&3 – Compilation

L'analyse lexicale et syntaxique avec Ocamllex et Menhir

Ingénierie du développement logiciel – F2B304

Correction

1 Des expressions

Cette section revient sur l'évaluateur d'expression arithmétiques vu dans le TP d'introduction à ocaml. Nous allons le doter d'un analyseur lexical qui sera ensuite complèté d'un analyseur syntaxique.

Exercice 1 (*Expression correcte*)

▷ **Question 1.1 :**
Construire un analyseur lexical permettant de reconnaître des expressions arithmétiques basiques composées de réels, des quatre opérateurs de base et de variables.

```
{
 type lexeme =
   | EOF
   | PLUS
   | MINUS
   | DIV
   | TIMES
   | FLOAT of float
   | IDENT of string
}

let letter = ['a'-'z' 'A'-'Z']
```

```
let digit = ['0'-'9']
let real = digit* ('.' digit*)?
let ident = letter (letter | digit | '_')*
let space = [' ' '\t' '\n']

rule nexttoken = parse
  | space+     { nexttoken lexbuf }
  | eof        { EOF }
  | "+"        { PLUS }
  | "-"        { MINUS }
  | "/"        { DIV }
  | "*"        { TIMES }
  | real as nb   { FLOAT (float_of_string nb) }
  | ident as str  { IDENT str }
```

Le code Ocaml ci-dessous vous permet de lire un fichier dont le nom est passé en paramètre de la fonction `compile` :

```
let compile file =
 print_string ("File "^file^" is being treated!\n");
 try
  let input_file = open_in file in
  let lexbuf = Lexing.from_channel input_file in
  (* Travail sur le fichier *)
  close_in (input_file)
 with Sys_error s ->
  print_endline ("Can't find file '" ^ file ^ "'")
```

▷ **Question 1.2 :**
À partir du code précédent, étendez votre programme pour qu'il lise l'expression dans un fichier et affiche tous les lexèmes du fichier.

```
{
 type lexeme =
   | EOF
   | PLUS
   | MINUS
   | DIV
   | TIMES
   | FLOAT of float
   | IDENT of string

 let print_lexeme = function
   | EOF    -> print_string "EOF"
   | PLUS   -> print_string "PLUS"
   | MINUS  -> print_string "MINUS"
   | DIV    -> print_string "DIV"
   | TIMES  -> print_string "TIMES"
```

```
  | FLOAT f -> print_string "FLOAT("; print_float f; print_string ")"
  | IDENT s -> print_string "IDENT("; print_string s; print_string ")"
}

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let real = digit* ('.' digit*)?
let ident = letter (letter | digit | '_')*
let space = [' ' '\t' '\n']

rule nexttoken = parse
 | space+     { nexttoken lexbuf }
 | eof        { EOF }
 | "+"        { PLUS }
 | "-"        { MINUS }
 | "/"        { DIV }
 | "*"        { TIMES }
 | real as nb   { FLOAT (float_of_string nb) }
 | ident as str { IDENT str }

{
 let rec examine_all lexbuf =
  let res = nexttoken lexbuf in
  print_lexeme res;
  print_string " ";
  match res with
  | EOF -> ()
  | _   -> examine_all lexbuf

 let compile file =
 print_string ("File "^file^" is being treated!\n");
 try
  let input_file = open_in file in
  let lexbuf = Lexing.from_channel input_file in
  examine_all lexbuf;
  print_newline ();
  close_in (input_file)
 with Sys_error s ->
  print_endline ("Can't find file '" ^ file ^ "'")

}
```

Exercice 2 (*Gestion des erreurs*)

Il s'agit d'étendre le code précédent pour fournir des messages d'erreur plus parlant. En l'occurrence, il s'agit d'indiquer en quel point du fichier l'erreur (lexicale) se produit. Pour cela, nous

allons utiliser le type `position` défini dans le module `Lexing`. Ce type permet de mémoriser et de manipuler sous forme structurée la position dans un fichier :

```
type position = {
 pos_fname : string; (* nom du fichier *)
 pos_lnum : int;    (* numero de la ligne *)
 pos_bol : int;     (* nb de caracteres entre le debut du fichier et celui de la ligne *)
 pos_cnum : int;    (* nb de caracteres depuis le debut du fichier *)
}
```

Par défaut, l'analyseur lexical généré ne met à jour que le dernier champs (`pos_cnum`). C'est donc aux actions de gérer les autres champs.

Les fonctions `lexeme_start_p` et `lexeme_end_p` du module `Lexing` permettent de récupérer respectivement la position de début et de fin de l'unité lexicale en cours d'analyse.

▷ **Modifier votre code pour atteindre l'objectif fixé.**

```
{
 type lexeme =
  | EOF
  | PLUS
  | MINUS
  | DIV
  | TIMES
  | FLOAT of float
  | IDENT of string

 let print_lexeme = function
  | EOF   -> print_string "EOF"
  | PLUS  -> print_string "PLUS"
  | MINUS -> print_string "MINUS"
  | DIV   -> print_string "DIV"
  | TIMES -> print_string "TIMES"
  | FLOAT f -> print_string "FLOAT("; print_float f; print_string ")"
  | IDENT s -> print_string "IDENT("; print_string s; print_string ")"

 open Lexing
 exception Eof

 type error =
  | Illegal_character of char
  | Illegal_float of string
 exception Error of error * position * position

 let raise_error err lexbuf =
  raise (Error(err, lexeme_start_p lexbuf, lexeme_end_p lexbuf))
```

```
 (* Les erreurs. *)
 let report_error = function
  | Illegal_character c ->
    print_string "Illegal character '";
    print_char c;
    print_string "' "
  | Illegal_float nb ->
    print_string "The float ";
    print_string nb;
    print_string " is illegal "

 let print_position debut fin =
  if (debut.pos_lnum = fin.pos_lnum) then
   begin
    print_string "line ";
    print_int debut.pos_lnum;
    print_string " characters ";
    print_int (debut.pos_cnum - debut.pos_bol);
    print_string "-";
    print_int (fin.pos_cnum - fin.pos_bol)
   end
  else
   begin
    print_string "from line ";
    print_int debut.pos_lnum;
    print_string " character ";
    print_int (debut.pos_cnum - debut.pos_bol);
    print_string " to line ";
    print_int fin.pos_lnum;
    print_string " character ";
    print_int (fin.pos_cnum - fin.pos_bol)
   end

}

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let real = digit* ('.' digit*)?
let ident = letter (letter | digit | '_')*
let newline = ('\010' | '\013' | "\013\010")
let blank = [' ' '\009']

rule nexttoken = parse
  | newline     { Lexing.new_line lexbuf; nexttoken lexbuf }
  | blank+      { nexttoken lexbuf }
  | eof         { EOF }
  | "+"         { PLUS }
  | "-"         { MINUS }
```

```
  | "/"       { DIV }
  | "*"       { TIMES }
  | real as nb   { try FLOAT (float_of_string nb) with Failure "float_of_string" -> raise_error
  | ident as str  { IDENT str }
  | _ as c      { raise_error (Illegal_character(c)) lexbuf }


{
 let rec examine_all lexbuf =
  let res = nexttoken lexbuf in
  print_lexeme res;
  print_string " ";
  match res with
  | EOF -> ()
  | _   -> examine_all lexbuf

 let compile file =
 print_string ("File "^file^" is being treated!\n");
 try
  let input_file = open_in file in
  let lexbuf = Lexing.from_channel input_file in
  try
   examine_all lexbuf;
   print_newline ();
   close_in (input_file)
  with
  | Error (kind,debut,fin) ->
    close_in (input_file);
    report_error kind;
    print_position debut fin;
    print_newline()
 with Sys_error s ->
  print_endline ("Can't find file '" ^ file ^ "'")

 let _ = Arg.parse [] compile ""
}
```

Vous pourrez également utiliser `ocamlbuild` pour produire un exécutable directement. Il convient alors d'ajouter le ligne suivante à votre fichier pour déclenche la fonction `compile` en lui passant en paramètre le contenu de la ligne de commande.

```
let _ = Arg.parse [] compile ""
```

2 Un exercice

Exercice 3 (*Analyse syntaxique des expressions*)

Réaliser l'analyseur syntaxique des expressions arithmétiques et connecter le résultat à l'évaluateur fait dans le TP Ocaml.

lexexpr.mll

```
{
 open Parseexpr
}

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let ident = letter (letter | digit | '_')*
let space = [' ' '\t' '\n']

rule nexttoken = parse
  | space+     { nexttoken lexbuf }
  | eof        { EOF }
  | "+"        { PLUS }
  | "-"        { MINUS }
  | "/"        { DIV }
  | "*"        { TIMES }
  | "%"        { MOD }
  | digit+ as nb  { INT (int_of_string nb) }
  | ident      { IDENT (Lexing.lexeme lexbuf) }
```

parseexpr.mly

```
%{
 open Expr
%}

%token EOF PLUS MINUS TIMES DIV MOD LPAR RPAR
%token <int> INT
%token <string> IDENT

%start expression
%type < Expr.expression > expression


%left PLUS MINUS
%left TIMES DIV MOD
%right UMINUS UPLUS
```

```
%%

expression:
 | e=expr EOF        { e }

expr:
  | LPAR e=expr RPAR
    { e }
  | MINUS e=expr %prec UMINUS
    { Unop(Uminus,e)}
  | PLUS e=expr %prec UPLUS
    { Unop(Uplus,e)}
  | e1=expr o=bop e2=expr
    { Binop(o,e1,e2)}
  | id=IDENT
    { Var id }
  | i=INT
    { Const i }

%inline bop:
  | MINUS    { Bsub }
  | PLUS     { Badd }
  | TIMES    { Bmul }
  | DIV      { Bdiv }
  | MOD      { Bmod }

%%
```

expr.ml

```
type binop =
  | Badd | Bsub | Bmul | Bdiv | Bmod

type unop =
  | Uplus | Uminus

type expression =
  | Const of int
  | Var of string
  | Binop of binop * expression * expression
  | Unop of unop * expression

exception Unbound_variable of string

let get_op_u = function
  | Uplus -> fun x -> x
  | Uminus -> fun x -> -x
```

```
let get_op_b op x y =
 match op with
  | Badd -> x + y
  | Bsub -> x - y
  | Bmul -> x * y
  | Bdiv -> x / y
  | Bmod -> x mod y

let string_of_op_u = function
  | Uplus -> "+"
  | Uminus -> "-"

let string_of_op_b = function
  | Badd -> "+"
  | Bsub -> "-"
  | Bmul -> "*"
  | Bdiv -> "/"
  | Bmod -> "%"

let rec eval env exp =
 match exp with
  | Const c -> c
  | Var v -> (try List.assoc v env with Not_found -> raise(Unbound_variable v))
  | Binop(op,e1,e2) -> (get_op_b op) (eval env e1) (eval env e2)
  | Unop(op,e) -> (get_op_u op) (eval env e)

let rec string_of_expr exp =
 match exp with
  | Const c -> string_of_int c
  | Var v -> v
  | Binop(op, e1, e2) ->
     "(" ^(string_of_expr e1)^ (string_of_op_b op) ^(string_of_expr e2)^ ")"
  | Unop(op, e) -> "(" ^ (string_of_op_u op) ^(string_of_expr e)^ ")"
```