TP de Langages Formels

Younesse Kaddar

- Énoncé
- Version PDF
- http://younesse.net/Langages-formels/TP_LangagesFormels/

I.

NB : dans la suite, on a modifié la fonction `pp_expr` dans `expr.ml` pour afficher le type des constantes/variables :

```
(** Afficheur d'expressions. *)
let rec pp_expr chan = function
  | Var v -> Format.fprintf chan "Var %s" v
  | Int i -> Format.fprintf chan "Int %d" i
  | String s -> Format.fprintf chan "String %s" s
  | Any -> Format.fprintf chan "Any"
```

- cela nous sera utile pour distinguer l'affichage de `String "_"` et de `Any`, à partir de la question 12.

Question 1.

```
let x=42 in x and y
```

provoque une `parse error`, car `and` est pris pour une variable.

Mettre les opérateurs avant les variables dans le lexer règle le problème :

```
rule token = parse

  (* [...] *)

  | "and"              { AND }
  | "or"               { OR }
  | "not"              { NOT }

  | varname as v       { VAR v }

  (* [...] *)
```

Question 2.

```
let varname = ['a'-'z' '_'](['a'-'z' 'A'-'Z' '0'-'9' '_' '\''])*
```

dans

```
rule token = parse
  | varname as v       { VAR v }
```

Question 3.

```
let int = ['0'-'9']['0'-'9' '_']* | '0'['x' 'X']['a'-'f' 'A'-'F' '0'-'9']['a'-'f' 'A'-'F' '0'-'9' '_']*
```

dans

```
rule token = parse
  | int as n               { INT (int_of_string n) }
```

Question 4.

```
let string = '"'([^'"']|'\"')*'"'
```

dans

```
rule token = parse
  | string as s   { STRING s }
```

Question 5.

```
let op_cmp = ['>' '=' '<' '^']
let op_mult = ['*' '/' '%']
let op_plus = ['+' '-']
let op_other = ['!' '$' '?' '.' ':' ';']
let remainder = (op_cmp | op_mult | op_plus |op_other)*
```

dans

```
rule token = parse
  | op_plus remainder as o { BIN_PLUS o }
  | op_mult remainder as o { BIN_MULT  o }
  | op_cmp remainder  as o { BIN_CMP o }
```

Question 6.

Le retour à la ligne est représenté par

- `\n` sur Linux
- `\r` sur Mac
- `\r\n` sur Windows

On remplace donc, dans la lexer, la règle :

```
| [' ' '\t' '\r' '\n']+   { token lexbuf }
```

par les règles

```
| [' ' '\t']+   { token lexbuf }
| ['\r' '\n'] | "\r\n"   { lexbuf.lex_curr_p
                          <- { lexbuf.lex_curr_p with
                                                pos_bol = lexbuf.lex_curr_p.pos_cnum ;
                                                pos_lnum = 1 +
                                                           lexbuf.lex_curr_p.pos_lnum };
                          token lexbuf }
```

```
./minirien "1=12

 2 3"

Line 3, char 2, before "2": parse error
```

Question 7.

Dans la documentation officielle, il est indiqué, à la section 12.2.5 :

> ***entrypoint [exp1… expn] lexbuf :***
>
> (Where entrypoint is the name of another entry point in the same lexer definition.) Recursively call the lexer on the given entry point. Notice that lexbuf is the last argument. Useful for lexing nested comments, for example.

Ainsi, si on voulait supporter les commentaires *pas nécessairement imbriqués bien parenthésés*, on pourrait procéder de la sorte :

```
rule token = parse

  (* [...] *)

  | "(*"   { comment_section lexbuf }


and comment_section = parse
  | "*)"   { token lexbuf }
  | _      { comment_section lexbuf }
```

Mais pour ce que l'on souhaite (*i.e* le support de commentaires imbriqués bien parenthésés), cela ne suffit pas.

Utilisons l'indication de l'énoncé : on peut stocker l'état "nombre de parenthèses ouvrantes rencontrées non refermées" comme paramètre (`num`) de l'analyseur lexical :

```
rule token = parse

  (* [...] *)

  | "(*" { comment_section 1 lexbuf }

and comment_section num = parse
  | "(*" { comment_section (num+1) lexbuf }
  | "*)" { if num>1 then comment_section (num-1) lexbuf
           else token lexbuf }
  | eof { failwith "Nested comments are not closed"}
  | _ { comment_section num lexbuf }
```

```
./minirien "1=2 (* (* test

 *)

 3=4 *) and 4=5"
>> and(=(Int 1,Int 2),=(Int 4,Int 5))


./minirien "1=2 (* (* test

3=4 *) and 4=5"
Line 1, char 32: Nested comments are not closed
```

Si on est tatillon et qu'on veut améliorer le message d'erreur pour les entrées multi-lignes avec commentaires imbriqués, on écrira plutôt :

```
rule token = parse

  (* [...] *)

  | "(*" { comment_section 1 lexbuf }

and comment_section num = parse
  | "(*" { comment_section (num+1) lexbuf }
  | "*)" { if num>1 then comment_section (num-1) lexbuf
           else token lexbuf }
  | ['\r' '\n'] | "\r\n"   { lexbuf.lex_curr_p
                             <- { lexbuf.lex_curr_p with
                             pos_bol = lexbuf.lex_curr_p.pos_cnum ;
                             pos_lnum = 1 + lexbuf.lex_curr_p.pos_lnum };
                             comment_section num lexbuf }
  | eof { failwith "Nested comments are not closed"}
  | _ { comment_section num lexbuf }
```

```
./minirien "1=2 (* (* test


3=4 *) and 4=5"
Line 4, char 15: Nested comments are not closed
```

II.

Question 8.

Comme la règle

```
| "="                { EQUALS }
```

est prioritaire (puisqu'elle arrive avant) sur la règle

```
| op_cmp remainder  as o { BIN_CMP o }
```

le `=` de `1=2` est associé au lexème `EQUALS`, lequel est associé à la production

```
| LET VAR EQUALS expr IN expr    { Let ($2,$4,$6) }
```

d'où le message d'erreur `before "=": parse error`

> Notons que le problème ne se pose pas pour les autres opérateurs de comparaison, pour lesquels c'est bien la règle `op_cmp remainder as o { BIN_CMP o }` qui s'applique.

On ajoute la règle

```
| expr EQUALS expr               { App ("=",[$1;$3]) }
```

à la grammaire

```
/minirien "1+1+1"
>> +(+(Int 1,Int 1),Int 1)

./minirien "1+1*1"
>> +(Int 1,*(Int 1,Int 1))

./minirien "1*1+1"
>> +(*(Int 1,Int 1),Int 1)
```

Le comportement est bien conforme à la grammaire, mais n'est pas satisfaisant : on aimerait établir un ordre de priorité sur les opérateurs (en l'occurrence : `BIN_MULT` devrait être prioritaire sur `BIN_PLUS`, lui-même prioritaire sur `BIN_CMP`).

Question 9.

```
%left BIN_PLUS
%left BIN_MULT
```

Question 10.

On ajoute

```
%left OR
%left AND
%left NOT
%left BIN_CMP EQUALS
```

avant les déclarations d'associativité de `BIN_PLUS` et `BIN_MULT`

⟶

```
./minirien "not 1+1 < 2 and 1 = 1 or foo"
>> or(and(not(<(+(Int 1,Int 1),Int 2)),=(Int 1,Int 1)),Var foo)
```

Question 11.

On a des conflits shift/reduce

```
31: shift/reduce conflict (shift 15, reduce 6) on BIN_MULT
31: shift/reduce conflict (shift 16, reduce 6) on BIN_PLUS
31: shift/reduce conflict (shift 17, reduce 6) on BIN_CMP
31: shift/reduce conflict (shift 18, reduce 6) on AND
31: shift/reduce conflict (shift 19, reduce 6) on OR
31: shift/reduce conflict (shift 20, reduce 6) on EQUALS
state 31
    expr : LET VAR EQUALS expr IN expr .  (6)
    expr : expr . BIN_PLUS expr  (7)
    expr : expr . BIN_MULT expr  (8)
    expr : expr . BIN_CMP expr  (9)
    expr : expr . EQUALS expr  (10)
    expr : expr . AND expr  (11)
    expr : expr . OR expr  (12)
```

causés par la règle réductible

```
| LET VAR EQUALS expr IN expr    { Let ($2,$4,$6) }
```

En effet :

Par exemple, l'entrée

```
let x=1 in x+2*2
```

est ambiguë : doit-elle se lire comme

```
(let x=1 in x+2)*2 (* = 6 *)
```

ou

```
let x=1 in (x+2*2) (* = 5 *)
```

?

On aimerait que ce soit la deuxième option.

On ajoute la déclaration

```
%nonassoc IN
```

dans `parse.mly`, avant toutes les déclarations précédentes (ajoutées en **9)** et en **10)**).

```
./minirien "let x=1 in x+2*2"
>> let x = Int 1 in +(Var x,*(Int 2,Int 2))
```

Question 12.

1.

On ajoute au parser les tokens

```
%token CASE OF PIPE GIVES
```

et au lexer les règles

```
| "case"            { CASE }
| "of"              { OF }
| "=>"              { GIVES }
| "|"               { PIPE }
```

2.

Puis, on modifie les règles grammaticales de la sorte :

```
expr:
  | const                       { $1 }
  | VAR                         { Var $1 }
  | LPAR expr RPAR              { $2 }
  | CASE expr OF pmatch         { Case ($2,List.rev $4) }
  (* [...] *)

pmatch:
  | prule                       { [$1] }
  | pmatch PIPE prule           { $3::$1 }

prule:
  | pat GIVES expr              { ($1,$3) }

pat:
  | VAR                         { if $1="_" then Any else Var $1 }
  | const                       { $1 }

const:
  | INT                         { Int $1 }
  | STRING                      { String $1 }
```

3.

On a les conflits suivants :

```
44: shift/reduce conflict (shift 18, reduce 18) on BIN_MULT
44: shift/reduce conflict (shift 19, reduce 18) on BIN_PLUS
44: shift/reduce conflict (shift 20, reduce 18) on BIN_CMP
44: shift/reduce conflict (shift 21, reduce 18) on AND
44: shift/reduce conflict (shift 22, reduce 18) on OR
44: shift/reduce conflict (shift 23, reduce 18) on EQUALS
state 44
    expr : expr . BIN_PLUS expr  (6)
    expr : expr . BIN_MULT expr  (7)
    expr : expr . BIN_CMP expr  (8)
    expr : expr . EQUALS expr  (9)
    expr : expr . AND expr  (10)
    expr : expr . OR expr  (11)
    prule : pat GIVES expr .  (18)
```

- *en effet* :
  - l'entrée suivante est ambiguë :

```
case x of 1 => 1 | _ => 2 * 2

(* se lit-elle

  (case x of 1 => 1 | _ => 2) * 2

  ou

  case x of 1 => 1 | _ => (2 * 2) : ce qu'on voudrait

  ? *)
```

    - de même qu'en **11)**, on résout le problème avec la déclaration (avant les autres déclarations d'associativité) :

```
%nonassoc GIVES
```

```
36: shift/reduce conflict (shift 40, reduce 4) on PIPE
state 36
    expr : CASE expr OF pmatch .  (4)
    pmatch : pmatch . PIPE prule  (17)

    PIPE  shift 40
    EOF  reduce 4
    BIN_MULT  reduce 4
    BIN_PLUS  reduce 4
    BIN_CMP  reduce 4
    AND  reduce 4
    OR  reduce 4
    EQUALS  reduce 4
    IN  reduce 4
    OF  reduce 4
    RPAR  reduce 4
```

- *en effet* :
    - l'entrée suivante est ambiguë :

```
case x of 1 => case y of 1 => 1 | _ => 2
(* se lit-elle

  case x of 1 => (case y of 1 => 1) | _ => 2

  ou

  case x of 1 => (case y of 1 => 1 | _ => 2) : ce qu'on voudrait (associativité à droite)

  ? *)
```

    - on résout le problème avec la déclaration (avant les autres déclarations) :

```
%right CASE OF PIPE
```

```
./minirien "case x+y of 0 => 0 | _ => 1"
>> case(+(Var x,Var y),[(Int 0,Int 0),(Any,Int 1)])

./minirien "case 1 of 0 => case 2 of 2 => 2 | 1 => 1"
>> case(Int 1,[(Int 0,case(Int 2,[(Int 2,Int 2),(Int 1,Int 1)]))])
```

Question 13.

1.

Dans `lex.mll` :

```
let op_plus = ['+' '-']
```

n'a plus de raison d'être, puisqu'il nous faut maintenant particulariser l'opérateur `-`.

On supprime donc l'identificateur `op_plus` et on remplace la règle

```
| op_plus remainder as o  { BIN_PLUS o }
```

par

```
| "+" remainder as o      { PLUS o }
| "-" remainder as o      { MINUS o }
```

Dans `parse.mly` :

On remplace

- le token

```
%token <string> BIN_PLUS
```

par

```
%token <string> PLUS
%token <string> MINUS
```

- la déclaration

```
%left BIN_PLUS
```

par

```
%left PLUS MINUS
```

- la règle

```
| expr BIN_PLUS expr            { App ($2,[$1;$3]) }
```

par

```
| expr PLUS expr                { App ("+",[$1;$3]) }
| expr MINUS expr               { App ("-",[$1;$3]) }
```

et on ajoute la règle

```
| MINUS expr                    { App ("-",[$2]) }
```

2.

Tout semblait aller pour le mieux dans le meilleur des mondes :

```
./minirien "x * -y"
>> *(Var x,-(Var y))

/minirien "-x + y"
>> +(-(Var x),Var y)
```

mais il y a un manque de symétrie :

```
./minirien "-x+-y"
>> +(-(Var x),-(Var y)) # comme on s'y attendait

./minirien "-x*-y"
>> -(*(Var x,-(Var y))) # on préférerait *(-(Var x),-(Var y)), pour garder la symétrie entre x et y
```

et bien pire encore :

```
./minirien "-1%3"
>> -(%(Int 1,Int 3)) # on voulait plutôt %(-(Int 1),Int 3)
```

3.

Pour remédier à cela, on crée un nouveau token

```
%token MMINUS
```

non associatif, auquel on assigne la plus grande priorité :

```
%right CASE OF PIPE

(* [...] *)

%left BIN_MULT

%nonassoc MMINUS
```

Puis, on modifie la règle concernée de la sorte :

```
| MINUS expr %prec MMINUS        { App ("-",[$2]) }
```

Bingo !

```
./minirien "-x*-y"
>> *(-(Var x),-(Var y))
```

Question 14.

1.

On modifie les règles de l'analyseur syntaxique de la sorte :

```
expr:

(* [...] *)

| expr PLUS expr                 { match ($1, $3) with
                                    | (Int n, Int m) -> Int (n+m)
                                    | _ -> App ("+",[$1;$3]) }
| expr MINUS expr                { match ($1, $3) with
                                    | (Int n, Int m) -> Int (n-m)
                                    | _ -> App ("-",[$1;$3]) }
| MINUS expr %prec MMINUS        { match $2 with
                                    | Int n -> Int (-n)
                                    | _ -> App ("-",[$2]) }

| expr BIN_MULT expr   { let op_list = ["*", ( * ); "/", (/); "%", (mod)] in
                         match ($1, $3) with
                         | (Int n, Int m) when (try
                                                  let _ = List.assoc $2 op_list in
                                                  true
                                                with Not_found -> false)
                           -> let op = List.assoc $2 op_list in
                              Int (op n m)
                         | (Int n, Int m) when $2="/" -> Int (n/m)
                         | (Int n, Int m) when $2="%" -> Int (n mod m)
                         | _ -> App ($2,[$1;$3]) }
```

```
./minirien "1+(2*3)+x"
>> +(Int 7,Var x)

./minirien "6/2-5+4%3*x"
>> +(Int -2,*(Int 1,Var x))
```

2.

Pour les variables muettes, j'ai tenté cette approche, mais en vain :

```
%{
    open Expr
    module VarSet = Set.Make(struct
                               type t = string
                               let compare = Pervasives.compare
                             end)
    let vars = ref VarSet.empty
    let adding_var = ref false
%}

/* [...] */

%%

(* [...] *)

expr:
  (* [...] *)
  | VAR                             { if VarSet.mem $1 !vars || !adding_var
                                      then Var $1
                                      else failwith ("Variable "^$1^" not defined") }
  (* [...] *)
  | LET VAR EQUALS expr IN expr     { let expr_equals = $4 in
                                      (
                                      adding_var := true;
                                      vars := VarSet.add $2 !vars;
                                      adding_var := false;
                                      Let ($2,expr_equals,$6)
                                      )
                                    }
```

3.

Je n'ai pas le temps d'implémenter cette dernière amélioration, mais l'idée serait d'utiliser une référence pour compter le nombre de parenthèses ouvrantes déjà rencontrées :

- à chaque nouvelle parenthèse ouvrante (resp. fermante), on incrémente (resp. décrémente) le compteur de 1 (et on lève une exception si le compteur devient strictement négatif).
- à chaque crochet fermant :
  - si le compteur est négatif (ou nul), on renvoie un erreur
  - sinon, on remet le compteur à zéro