Compilation : TP3

2 décembre 2014

Valentin Esmieu
Corentin Nicole
groupe 1.2

1 Code

Listing 1: **main.ml**

```
open Parser
open Lexer
open Lexing
open Parsing

let _ =

let lex = from_string
"begin int lapin , bool bionicle ; begin bool duplo ; lapin <- bionicle and duplo end end"
in
try main token lex with
| Parse_error -> failwith (let err = lexeme_start_p lex in
"Erreur a la ligne "^string_of_int(err.pos_lnum)
^", caractere "^string_of_int(err.pos_cnum - err.pos_bol));;
```

Listing 2: **arbre.ml**

```
(* Construction de l'arbre *)
(* Declaration de types *)

type expr =
        | Node_plus of expr * expr
        | Node_inf of expr * expr
        | Node_and of expr * expr
        | Ident of string

type decl =
        | Node_decl_int of string
        | Node_decl_bool of string


type arbre_bloc =
        | Node_bloc of decl list * inst list
        and
        inst =
        | Node_inst of string * expr
        | Node_inst_bloc of arbre_bloc
```

Listing 3: **lexer.mll**

```
        (* File lexer.mll *)
{
        open Parser
        open Lexing
        let keyword_table = Hashtbl.create 53
        let _ =
        List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
                [
                "begin", BEGIN;
                "end", END;
                "int", INT;
                "bool", BOOL;
                "and", AND
                ]
}

let char  = ['0'-'9' 'a'-'z' 'A'-'Z' '_' '\'' '<' '>']
let comment = '%' [^'\n']* '\n'


rule token = parse
| [' ' '\t']+     { token lexbuf }     (* skip blanks *)
| '\n'            { Lexing.new_line lexbuf ; token lexbuf }
| "<-"                 { FLECHE }
| "("                  { PAROUV }
| ")"                  { PARFERM }
| "+"                  { PLUS }
| "<"                  { INF }
| ";"                  { PTVIRG }
| ","                              { VIRG }
| eof                              { EOF }
| ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_'] * as id
               { try
                   Hashtbl.find keyword_table id
                 with Not_found ->
                   IDENT id }
```

Listing 4: **parser.mly**

```
/* File parser.mly */

%{
  open Arbre
%}

%token PLUS FLECHE
%token PAROUV PARFERM
%token INF AND
%token <string> IDENT
```

```
%token BEGIN
%token END
%token PTVIRG VIRG
%token INT
%token BOOL
%token EOF
%left AND
%left INF
%left PLUS
%start main             /* the entry point */
%type <Arbre.arbre_bloc> main
%%

main:
        bloc EOF { $1 }

bloc:
        BEGIN sdecl PTVIRG sinst END { Node_bloc ($2,$4)  }

sdecl:
        | decl {[$1]}
        | decl VIRG sdecl { $1::$3 }

decl:
        typeuh IDENT {
                match $1 with
                        | "int" -> Node_decl_int $2
                        | "bool" -> Node_decl_bool $2
                        | _ -> failwith "error" }


typeuh:
        | INT { "int" }
        | BOOL { "bool" }


sinst:
        | inst { [$1]}
        | inst PTVIRG sinst { $1::$3 }



inst:
        | bloc { Node_inst_bloc $1 }
    | IDENT FLECHE expr { Node_inst ($1, $3) }


expr:
        | expr PLUS expr { Node_plus ($1, $3) }
        | expr INF expr { Node_inf ($1, $3) }
        | expr AND expr { Node_and ($1, $3) }
        | PAROUV expr PARFERM { $2 }
        | IDENT { Ident $1 }
```

2 Questions

Question 1 *Quel est l'intérêt d'utiliser un crible avec ocamllex ?*
Ceci permet de filtrer l'ensemble des lexèmes pour génerer les Ident correspondants. Dans ce TP, pour limiter le nombre de transitions, nous utilisons une table pour stocker les Ident particuliers tels que *begin*, *end*, etc.

Question 2 *Quelle est la différence, d'un point de vue pratique, entre écrire une grammaire sous forme LL et LR ?*
Dans une grammaire LR, la détection d'erreur se fait plus tôt et permet une couverture plus large. De plus, une grammaire LR permet de traiter des grammaires ambiguës.

Question 3 *Que signifie une colonne vide dans une table SLR ?*
Une colonne vide indique que le token correspondant n'est jamais utilisé.