Compte Rendu de compilation TP3

Théo CHAPON, Alexandre Audinot

INSA de Rennes
4INFO, groupe A

November 13, 2014

TP3 : Analyse syntaxique ascendante, ocamllex et ocamlyacc

1 Question de compréhenion de TP

Question 3.1 Un crible filtre la suite des lexèmes et génère les identifiants. En utilisant ocamllex, on peut générer une hashtable pour stocker les unités lexicales de type string. Ils seront trouver par le générateur d'identifiants mais ne seront pas reconnus comme étant des identifiants mais bien comme des unités lexicales. (exemple : "and", "begin" etc.).

Question 3.2 Les avantages de la grammaire LR part rapport à la LL sont les suivants :

- La classe de grammaire couverte par LR est plus large.
- On peut traiter des grammaires ambiguës.
- La détection d'erreur est faite au plus tôt.

Question 3.3 Une colone vide dans une table SLR correspond à un token inutilisé.

2 Fermeture transitive

2.1 Grammaire

<Inst> –> <Ident> <- <Expr>
<Expr> –> <Expr> "+" <Expr>
<Expr> –> <Expr> < <Expr>
<Expr> –> <Expr> "and" <Expr>
<Expr> –> "(" <Expr> ")"
<Expr> –> <Ident>

2.2 Fermeture

$$Fermeture(I) = [I- > .ident<\text{-}E]$$

$$\begin{aligned} Fermeture(E) = & [E- > .E" + "E] \\ & [E- > .E<E] \\ & [E- > .E"and"E] \\ & [E- > ."("E")"] \\ & [E- > .ident] \end{aligned}$$

2.3 Transition

$$I0 = \{Fermeture(I)\}$$

$$I1 = Tr(I0, ident)- > \{[I- > ident.<\text{-}E]\}$$

$$\begin{aligned} I2 = Tr(I1, <\text{-})- > \{ & [I- > ident<\text{-}.E], \\ & Fermeture(E)\} \end{aligned}$$

$$\begin{aligned} I3 = Tr(I2, E)- > \{ & [I- > ident<\text{-}E.], \\ & [E- > E." + "E], \\ & [E- > E.<E], \\ & [E- > E."and"E]\} \end{aligned}$$

$$I4 = Tr(I2, "(") -> \{[E -> "(".E")"], \\ Fermeture(E)\}$$

$$I5 = Tr(I2, ident) -> \{[E -> ident.]\}$$

$$I6 = Tr(I3, " + ") -> \{[E -> E" + ".E], \\ Fermeture(E)\}$$

$$I7 = Tr(I3, <) -> \{[E -> E<.E], \\ Fermeture(E)\}$$

$$I8 = Tr(I3, "and") -> \{[E -> E"and".E], \\ Fermeture(E)\}$$

$$I9 = Tr(I4, E) -> \{[E -> "(".E.")"]\}$$

$$I10 = Tr(I6, E) -> \{[E -> E" + "E.] \\ [E -> E." + "E], \\ [E -> E.<E], \\ [E -> E."and"E]\}$$

$$I11 = Tr(I7, E) -> \{[E -> E<E.] \\ [E -> E." + "E], \\ [E -> E.<E], \\ [E -> E."and"E]\}$$

$$I12 = Tr(I8, E) -> \{[E -> E\text{"}and\text{"}E.]$$
$$[E -> E.\text{"} + \text{"}E],$$
$$[E -> E.<E],$$
$$[E -> E.\text{"}and\text{"}E]\}$$

$$I13 = Tr(I9, \text{"})\text{"}) -> \{[E -> \text{"}(\text{"}E\text{"})\text{"}.]\}$$

3 Table SLR

État	+	<	and	(	)	<-	Ident	Expr
0							d1	
1						d2		
2				d4			d5	3
3	d6	d7	d8					
4				d4			d5	9
5	r14	r14	r14		r14			
6				d4			d5	10
7				d4			d5	11
8				d4			d5	12
9	d6	d7	d8		d13			
10	d6/r10	d7/r10	d8/r10		r10			
11	d6/r11	d7/r11	d8/r11		r11			
12	d6/r12	d7/r12	d8/r12		r12			
13	r13	r13	r13		r13			

4 Résolution des conflits

On trouve 9 conflits dans la table SLR générés par 3 tokens (+, < et and) : 6 pour les conflits de prédominance entre deux tokens différents (ex : a and b < c) et 3 pour les conflits de réductions entre même tokens (ex : a + b + c).
Le parser ne sait pas s'il doit continuer sur une nouvelle règle ou bien réduire. Il faut donc établir une hiérarchie entre ces 3 tokens. L'ordre de préfominance gardé correspond à celui donné par la grammaire : PLUS > INF > AND. De plus nous avons choisis un décalage à droite pour les conflits entre même tokens.

5 Fichiers de TP

Listing 1: Fichier Lexer et crible

```
(*{
  open Ulex (* Ulex contains the type definition of lexical units *)
}*)
{
   open Parser        (* The type token is defined in parser.mli *)

   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]
}

rule token = parse (* TODO *)
   | [' ' '\t']     { token lexbuf } (* lexbuf : buffer *)
   | ['\n' ] { Lexing.new_line lexbuf; token lexbuf }
   | "/*" ([^'*'] | '*' + [^'/'])* "*/" {token lexbuf} (*commentaires plusieurs
      lignes*)
   | "//" ([^'\n'])* {token lexbuf} (*commentaires 1 ligne*)
   | eof      { EOF }
   | ';'      { SEMI_COLUMN }
   | ','      { COMA }
   | "<-"     { FLECHE }
   | '+'      { PLUS }
   | '<'      { INF }
   | '('      { PAROUV }
   | ')'      { PARFERM }
   | ['A'-'Z' 'a'-'z' '0'-'9' '_']* as ident {try Hashtbl.find keyword_table ident
                                                with Not_found -> IDENT ident}
            (*permet de vérifier si l ident lu n est pas une unité léxicale*)
```

Listing 2: Fichier Parser et grammaire

```
%{open Type %}

%token EOL
%token EOF

%token BEGIN END
%token SEMI_COLUMN COMA
%token INT BOOL
%token FLECHE
%token PLUS INF AND
%token PAROUV PARFERM
%token <string> IDENT

%right AND
%right INF
%right PLUS

%start bloc             /* the entry point */
%type <Type.arbre> bloc
%%

bloc:
    BEGIN sdecl SEMI_COLUMN sinst END EOF { Bloc($2, $4) }
;

sdecl:
    decl                { [$1] }
  | decl COMA sdecl   { $1::$3 }
;


decl:
tipe IDENT {$1,$2}
;


tipe:
  INT {INT}
  | BOOL {BOOL}
;


sinst:
    inst  { [$1] }
  | inst SEMI_COLUMN sinst  { $1::$3 }
;


```

```
49 inst:
50   bloc           { Inst_Bloc $1  }
51   | IDENT FLECHE expr       { Inst($1, $3) }
52 ;
53
54 expr:
55     IDENT                    { Id $1 }
56   | PAROUV expr PARFERM      { Ex $2 }
57   | expr AND expr            { And($1, $3) }
58   | expr INF expr            { Inf($1, $3) }
59   | expr PLUS expr           { Add($1, $3) }
60 ;
```

Listing 3: Type pour l'arbre abstrait

```
type arbre = Bloc of sdecl * sinst and
  sdecl = (tipe * string) list
and
  tipe =
    | INT
    | BOOL
and
  sinst = inst list
and
  inst =
    | Inst_Bloc of arbre
    | Inst of string * expr
and
  expr = Add of expr * expr
    | And of expr * expr
    | Inf of expr * expr
    | Id of string
    | Ex of expr
```

Listing 4: Fichier Main : détection des erreurs et affichage de l'arbre

```
open Parser
open Lexer
open Lexing
open Parsing
open Type

(*On lit dans le fichier test*)
let lexbuf = Lexing.from_channel (open_in "test");;

(*On vérifie si la grammaire du fichier chargé dans lexbuf est respectée*)
let _ = try bloc token lexbuf with
| Parse_error -> failwith (let err = lexeme_start_p lexbuf in
"Erreur dans le fichier "^err.pos_fname
^", a la ligne "^string_of_int(err.pos_lnum)
^", caractere "^string_of_int(err.pos_cnum - err.pos_bol));;

let arbre = bloc token lexbuf;;

(*On affiche les éléments de l arbre abstrait*)
let rec display = function
  | Add (exp1,exp2) -> display exp1;Printf.printf " Plus ";display exp2
  | Inf (exp1,exp2) -> display exp1;Printf.printf " Inf ";display exp2
  | And (exp1,exp2) -> display exp1;Printf.printf " And ";display exp2
  | Id s -> Printf.printf "Ident %s" s
  | Ex exp -> display exp;;

let f = function
  | Inst(s,e) -> Printf.printf "Inst\n";Printf.printf "%s Aff " s;display
     e;Printf.printf "\n"
  | Inst_Bloc a -> failwith "error";;

let _ = match arbre with
  | Bloc (sdecl,sinst) -> Printf.printf "Sdecl\n";
List.iter
  (
  fun (t,s) -> match t with
    | INT -> Printf.printf "Int(%s)\n" s
    | BOOL -> Printf.printf "Bool(%s)\n" s
  )
sdecl;
List.iter
  (
  fun inst -> Printf.printf "Sinst\n";
  f inst
  )
sinst;;

Printf.printf "Success !\n";;
```