TP3 - Analyse syntaxique ascendante, ocamllex et ocamlyacc

Raphaël Baron - Paul Chaignon

19 novembre 2013

1 Préparation du TP

1.1 Table SLR

	Ident	←	+	<	and	(	)	$	Inst	E
I0	S2								1	
I1								Acc		
I2		S3								
I3	S6					S5				4
I4			S7/~~R1~~	S8/~~R1~~	S9/~~R1~~		R1	R1		
I5										10
I6			R0	R0	R0		R0	R0		
I7	S6					S5				11
I8	S6					S5				12
I9	S6					S5				13
I10							S14			
I11			~~S7~~/R1	~~S8~~/R1	~~S9~~/R1		R1	R1		
I12			S7/~~R2~~	~~S8~~/R2	~~S9~~/R2		R2	R2		
I13			S7/~~R3~~	S8/~~R3~~	~~S9~~/R3		R3	R3		
I14			R4	R4	R4		R4	R4		

Table 1 – Table SLR avec conflits

1.2 Résolution des conflits

Pour résoudre les confilts, on utilise le fait que le *And* a une priorité moins importante que le $<$ qui a aussi une priorité moins importante que le $+$.

2 Questions

Question 1

L'utilisation d'un crible avec ocamlex permet de réduire le nombre d'états (et de transitions).

Question 2

Il n'y a pas besoin de lever les ambiguités pour une grammaire LR mais l'analyseur ascendant pour la grammaire LR est plus compliqué.

Question 3

Si la colonne d'un terminal est vide, cela signifie que ce terminal ne sera jamais rencontré.

3 Codes source

Listing 1 – grammar.mly

```
%{
  open Definitions
  let parse_error s =
    let pos = Parsing.symbol_start_pos() in
    let (lineNum, colNum) = (pos.Lexing.pos_lnum, pos.Lexing.pos_cnum)
       in
    print_endline ("Parse error a la ligne " ^ (string_of_int lineNum)
       ^ ", caractere " ^ (string_of_int colNum));;


%}

%token <string> IDENT
%token EOF BEGIN PTVIRG END VIRG INT BOOL AFFECT PLUS INF AND PAROUV
   PARFERM ERROR

%left AND
%left INF
%left PLUS

%type <Definitions.arbabstrait> file

%start file

%%
file:
  | bloc EOF {$1};

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

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

decl:
  | typ IDENT {Declaration($1, $2)};

typ:
  | INT {Int}
  | BOOL {Bool};

```

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

inst:
  | bloc {$1}
  | IDENT AFFECT expr {Affectation($1, $3)};

expr:
  | expr PLUS expr {Plus($1,$3)}
  | expr INF expr {Inf($1,$3)}
  | expr AND expr {And($1,$3)}
  | PAROUV expr PARFERM {$2}
  | IDENT {Ident($1)};
```

Listing 2 – lexer.mll

```
{
  open Grammar
}

let id = ['a'-'z'] ['a'-'z' '0'-'9']*

rule scanner = parse
  | "begin" {BEGIN}
  | ";" {PTVIRG}
  | "end" {END}
  | "," {VIRG}
  | "int" {INT}
  | "bool" {BOOL}
  | "<-" {AFFECT}
  | "+" {PLUS}
  | "<" {INF}
  | "and" {AND}
  | "(" {PAROUV}
  | ")" {PARFERM}
  | id as ident {IDENT ident}
  | [' ' '\t' '\n'] {scanner lexbuf}
  | eof {EOF}
  | _ {ERROR}
```

Listing 3 – arbre.ml

```


type decl = Bool | Int

type arbabstrait =
      Ident of string
   |  Bloc of (arbabstrait list) * (arbabstrait list)
   |  Declaration of decl * string
   |  Plus of arbabstrait * arbabstrait
   |  Inf of arbabstrait * arbabstrait
   |  And of arbabstrait * arbabstrait
   |  Affectation of string * arbabstrait
   |  Exception of string;;
```