```
%{
/* ----------------------------------------------------------------------
   $Id: ptq.l 4 2007-09-20 21:15:01Z pascal.amsili $

   Nom          : ptq.l
   Type         : spécification lex
   Contenu      : Lexeur pour PTQ
   Commentaires :
---------------------------------------------------------------------- */
#include "ptq.h"

char * chaine_cour ;
extern int yylval ;
%}

%%
 /* formules du lambda-calcul typé */

and     { yylval = 1 ; return OP2 ; }
or      { yylval = 2 ; return OP2 ; }
implies { yylval = 3 ; return OP2 ; }
not     { yylval = 0 ; return OP1 ; }

some    { yylval = 1 ; return QUANT ; }
all     { yylval = 2 ; return QUANT ; }
\\      { return LAMBDA ; }

[.()]   { return yytext[0] ; }

  /* déclarations */

prd     { return ID_PRD ; }
cte     { return ID_CTE ; }
var     { return ID_VAR ; }

 /* Autres identificateurs */

[a-zA-Z]+  {
               chaine_cour = malloc(strlen(yytext)+1) ;
               strcpy(chaine_cour, yytext) ;
               yylval = (int) chaine_cour ;
               return recherche_tds(yytext) ; }

[;?] { return yytext[0] ; }

.|\n     /* NOP */ ;
%%
```

```
%{
/* ----------------------------------------------------------------------
   $Id: ptq.y 4 2007-09-20 21:15:01Z pascal.amsili $

   Nom          : ptq.y
   Type         : spécification yacc
   Contenu      : parser pour PTQ
   Commentaires :
      - Consigne : pas d'accents dans les messages de sortie pour éviter
           les problèmes de transcodage
---------------------------------------------------------------------- */
#include "ptq.h"

  int main(void) { return yyparse() ; }
  void yyerror(char * s) { fprintf(stderr, "Erreur : %s\n", s) ; }
%}

%token OP1 OP2 QUANT LAMBDA
%token ID_VAR ID_PRD ID_CTE
%token VAR PRD CTE
%token ID
%token AF
%%

 /* C O N T R O L E */
 /* --------------- */

S: I  S
|  I
;

I: Declaration
|  Requete
;


Declaration: ID_VAR  ID   ';' { insere_tds((char *)$2, VAR) ; }
|            ID_PRD  ID   ';' { insere_tds((char *)$2, PRD) ; }
|            ID_CTE  ID   ';' { insere_tds((char *)$2, CTE) ; }
;

Requete:     F  '?'      { printf("Formule logique reconnue : ") ;
                           affiche_formule((noeud *)$1) ; printf("\n") ;  }
;

/* L A M B D A - C A L C U L */
/* ------------------------ */
/* (la grammaire sur-génère) */

F:  T
|   OP1 F                 { $$ = (int) cree_n(OP1, $1, (noeud *) $2, 0) ; }
|   '(' F OP2 F ')'       { $$ = (int) cree_n(OP2, $3, (noeud *) $2, (noeud *) $4) ; }
|   QUANT VAR '.' F       { $$ = (int) cree_n(QUANT, $1,
                                         cree_n(VAR, val_tds((char *)$2), 0, 0),
                                         (noeud *) $4) ; }
|   LAMBDA VAR '.' F      { $$ = (int) cree_n(LAMBDA, 0,
                                         cree_n(VAR, val_tds((char *)$2), 0, 0),
                                         (noeud *) $4) ; }
|   '(' F F  ')'          { $$ = (int) cree_n(AF, 0, (noeud *) $2, (noeud *) $3) ; }
;

T:  VAR                    { $$ = (int) cree_n(VAR,val_tds((char *)$1),0,0) ; }
|   CTE                    { $$ = (int) cree_n(CTE,val_tds((char *)$1),0,0) ; }
|   PRD                    { $$ = (int) cree_n(PRD,val_tds((char *)$1),0,0) ; }
;

%%
```

```
# ----------------------------------------------------------------------
#   $Id: Makefile 3 2007-09-20 15:36:24Z pascal.amsili $
#
#   Nom           : Makefile
#   Type          : makefile
#   Contenu       : dépendances de l'application ptq
#   Commentaires :
# ----------------------------------------------------------------------
APPLI=ptq

ptq: lex.yy.o y.tab.o ${APPLI}sup.o
        gcc -o ${APPLI} ${APPLI}sup.o lex.yy.o y.tab.o -lfl

y.tab.c: ${APPLI}.y
        yacc -d ${APPLI}.y

y.tab.h: ${APPLI}.y
        yacc -d ${APPLI}.y

lex.yy.o: lex.yy.c y.tab.h
        gcc -c lex.yy.c

lex.yy.c: ${APPLI}.l
        lex ${APPLI}.l

clean:
        rm -f ${APPLI} ${APPLI}sup.o lex.yy.o lex.yy.c y.tab.h y.tab.o y.tab.c


var x ; var y ; var z ;
cte j ; cte m ;
prd chat ; prd dort ; prd homme ; prd mortel ;
(homme j) ?
var P ; var Q ;
\P. \Q. some x . ((P x) implies (Q x)) ?
```

```
// $Id: ptq.h 4 2007-09-20 21:15:01Z pascal.amsili $
// Fichier ptq.h : entête pour PTQ
// -------------------------------
#include <strings.h>
#include <stdlib.h>
#include <stdio.h>

#include "y.tab.h"

void insere_tds(char * s, int val) ;
void affiche_tds() ;
int val_tds(char * s) ;
char * string_tds(int t, int v) ;

struct noeud {
  int type ;
  int val ;
  struct noeud * fg ;
  struct noeud * fd ;
} ;
typedef struct noeud noeud ;

noeud * cree_n(int type, int val, noeud * fg, noeud * fd) ;
void affiche_formule(noeud * racine) ;
```

```
// $Id: ptqsup.c 4 2007-09-20 21:15:01Z pascal.amsili $
// Fichier ptqsup.c : code supplémentaire pour ptq
// --------------------------------------------------------------------
#include <stdlib.h>
#include <strings.h>
#include "ptq.h"

// --------------------------------------------------------------------
// Gestion de la table des symboles
// --------------------------------------------------------------------
struct {
  char nom[22] ;
  int type ;
  int val ;
} tds[30] ;

int next_tds = 0 ;

/*
TDS :
------------------------------
NOM       TYPE     VALEUR
char[22]  int      int

+ le type correspond au token (défini dans y.tab.h) (VAR/PRD/CTE)
+ la valeur est un entier arbitraire, pour distinguer les objets de même
type. On donne une valeur implicite (le premier prédicat vaut 0, le
second 1, etc.).
*/

void insere_tds(char * s, int type)
{
  int i, c=0 ;
  for (i=0 ; i<next_tds ; i++)
    if (tds[i].type == type) c++ ;

  strcpy(tds[next_tds].nom, s) ;
  tds[next_tds].type = type ;
  tds[next_tds].val = c ;
  next_tds++ ;
}

void affiche_tds()
{
  int i ;
  for (i=0 ; i<next_tds ; i++)
    printf("Id: %s ; type : %d, valeur : %d\n", tds[i].nom, tds[i].type, tds[i].val) ;
}

int recherche_tds(char * s)
{
  int i ;
  for (i=0 ; i<next_tds ; i++)
    if (!strcmp(tds[i].nom, s))
      return tds[i].type ;
  return ID ;
}

int val_tds(char * s)
{
  int i ;
  for (i=0 ; i<next_tds ; i++)
    if (!strcmp(tds[i].nom, s)) return tds[i].val ;
  return 0 ;
}

char * string_tds(int t, int v)
{
```

```
  int i ;
  for (i=0 ; i<next_tds ; i++)
    if (tds[i].type==t && tds[i].val == v) return tds[i].nom ;
  return "epsilon" ;
}

// ---------------------------------------------------------------------
// Construction de l'arbre associé à une formule
// ---------------------------------------------------------------------

noeud * cree_n(int type, int val, noeud * fg, noeud * fd)
{
  noeud * x = (noeud *) malloc (sizeof(noeud)) ;
  if (x == (noeud *) NULL) exit(-1) ;
  x->type = type ;
  x->val  = val ;
  x->fg   = fg ;
  x->fd   = fd ;
  return x ;
}

void affiche_formule(noeud * racine)
{
  if (racine == NULL) return ;
  switch (racine->type) {
  case AF:
    printf("(") ; affiche_formule(racine->fg) ; printf("  ") ;
    affiche_formule(racine->fd) ; printf(")") ;
    break ;
  case LAMBDA :
    printf("\\") ; affiche_formule(racine->fg) ; printf(".") ;
    printf(" ") ; affiche_formule(racine->fd) ;
    break ;
  case QUANT :
    printf("%s ", (racine->val==1)?"all":"some") ;
    affiche_formule(racine->fg) ;
    printf(". ") ; affiche_formule(racine->fd) ;
    break ;
  case OP1 : printf("not ") ; affiche_formule(racine->fg) ;
    break ;
  case OP2 :
    printf("("); affiche_formule(racine->fg) ;
    printf(" %s ", (racine->val==1)?" and ":(racine->val==2)?" or ":" implies") ;
    affiche_formule(racine->fd) ; printf(")");
    break ;
  case PRD :
  case VAR   :
  case CTE   : {
    printf("%s", string_tds(racine->type,racine->val)) ;
    break ;
  }
  }
}
```