Implémentation de quelques algorithmes en raisonnement non-monotone

(annexe à la thèse UPS numéro 2217)

Marie-Christine Lagasquie-Schiex

Rapport IRIT / 95-22-R

Décembre 1995

Implémentation de quelques algorithmes en raisonnement non-monotone

Marie-Christine Lagasquie-Schiex
Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier
118 route de Narbonne
31062 Toulouse Cedex
FRANCE

e-mail : {lagasq}@irit.fr

Résumé

Ce document est une annexe de la thèse [LS95], dans laquelle est présentée une comparaison de certains processus d'inférence non-monotone sur une base de croyances E ordonnée ou pas et pouvant être initialement inconsistante.

Ces processus sont construits par la combinaison d'un mécanisme de génération d'ensembles de croyances consistants issus de E et pouvant être ordonnés ou pas à l'aide de préférences (ce mécanisme est noté m) et d'un principe d'inférence p.

Dans cette thèse, ont été exposées trois méthodes algorithmiques pour traiter l'une de ces relations d'inférence, celle correspondant au mécanisme de génération utilisant l'ordre lexicographique et au principe d'inférence définissant la conséquence universelle. Cette relation est notée UNI-LEX.

Ces trois méthodes sont : une méthode naïve reposant sur l'emploi d'un prouveur traitant le problème SAT, une méthode nécessitant un prouveur pour le problème MAX-SAT et enfin une méthode utilisant les BDD (Binary Decision Diagram).

L'objet de ce document annexe est la présentation de l'implémentation en langage C de ces trois méthodes, et de la partie concernant les tests aléatoires de ces diverses techniques.

Table des matières

Chapitre 1

Programmes pour Uni-Lex : méthode naïve et méthode utilisant Max-Sat

Dans cette annexe, sont donnés les fichiers source en langage C des implémentations correspondant à la méthode naïve et à la méthode utilisant MAX-SAT.

Pour le cas de la méthode naïve, la fonction principale est UNI-LEX donnée dans le fichier `lexico.c`.

Pour la seconde méthode utilisant MAX-SAT, la fonction principale est UNI-LEX2 donnée dans le fichier `lexico.c`.

Les autres fichiers contiennent les fonctions et procédures utiles au développement de UNI-LEX et UNI-LEX2.

```
/*******************
        Fichier : lexico.h

        Contient les types de donnees :

        Contient les #define de :

        Contient les declarations en externe des fonctions et procedures :
                . rech_pref_lex,
                . rech_lex_par_strate,
                . inference_lex,
                . UNI_LEX,
                . UNI_LEX2.

        Utilise les packages :
                . table_poids,
                . max_sat (de Thierry Castell),
                . prouveur (de Thierry Castell),
                . mclperso.
********************/

#include <prouveur.h>
#include <max_sat.h>
#include "base_strate.h"
#include "mclperso.h"



/****** les procedures et fonctions ******/
liste_base_strate * rech_pref_lex (base_strate *, clause *, int, int, int, elem_tab_poids *);

void rech_lex_par_strate (strate *, clause *, liste_base_strate **,
        liste_base_strate **, int *, int, int, int, elem_tab_poids *);

bool inference_lex (liste_base_strate *, clause *, int, int, int, elem_tab_poids *);

bool UNI_LEX (base_strate *, clause *, int, int, int, elem_tab_poids *, bool);

bool UNI_LEX2 (base_strate *, clause *, int, int, int, elem_tab_poids *);
```

```
/********************
        Fichier : lexico.c

        Contient les types de donnees :

        Contient les #define de :

        Contient les declarations et le corps des fonctions et procedures :
                . rech_pref_lex,
                . rech_lex_par_strate,
                . inference_lex,
                . UNI_LEX,
                . UNI_LEX2.

        Utilise les packages :
                . lexico.
********************/

#include "lexico.h"



/**  fonction rech_pref_lex
***        but : trouver les sous-bases preferees
***                lexicographiquement d'une base de strates,
***        parametres : la base de strates, la clause a inferer,
***                les informations permettant la creation
***                de la base par Thierry Castell (nombres
***                maximum de symboles, des clauses,
***                de litteraux par clause), la table des poids,
***        retourne : la liste des sous bases preferees
***                lexicographiquement, et le booleen mis a jour
***                (vrai si recherche arretee car la sous-base
***                infere non Phi, faux sinon),
***/

liste_base_strate * rech_pref_lex (base_strate * E, clause * Phi,
                int nb_max_symb, int nb_max_cl, int nb_max_lit_cl,
                elem_tab_poids * t_b)
{
liste_base_strate * Vliste;
liste_base_strate * Nliste ;
int cpt ;
int Max ;
liste_strate * ls ;
base_strate * b ;

Vliste = init_liste_base_strate();
b = init_base_strate();
Vliste = ajout_liste_base_strate(Vliste,b);
cpt = 1 ;
ls = E->les_strates;
while (cpt <= E->nb_strates)
        {
        Max = 0 ;
        Nliste = init_liste_base_strate();
        rech_lex_par_strate(ls->un_ens,Phi,&Vliste,&Nliste,&Max,
                        nb_max_symb,nb_max_cl,nb_max_lit_cl,t_b);
        Vliste = Nliste ;
        ls = ls->ens_suiv;
        cpt++ ;
        }
return(Vliste);
}
```

```
/**  procedure rech_lex_par_strate
***        but : determiner pour une strate donnee et a partir
***              d'une liste de sous-bases preferees
***              lexicographiquement pour les strates precedentes
***              quelles sont les clauses a rajouter
***              a ces sous-bases pour obtenir les sous-bases
***              preferees lexicographiquement jusqu'a
***              la strate donnee,
***        parametres : la strate, la clause a inferer, la liste
***              des sous-bases preferees
***              lexicographiquement pour les strates
***              precedentes, la liste des sous bases preferees
***              lexicographiquement en cours de construction
***              pour la strate donnee, un pointeur sur un
***              entier donnant le nombre maximum des
***              clauses de la strate apparaissant en meme
***              temps dans une sous-base preferee
***              lexicographiquement,
***              les informations permettant la creation de la
***              base par Thierry Castell
***              (nombres maximum de symboles, des
***              clauses, de litteraux par clause), la table des poids,
***        retourne : la liste des sous bases preferees
***              lexicographiquement jusqu'a la strate donnee,
***              le booleen mis a jour (vrai si recherche
***              arretee car la sous-base infere non Phi,
***              faux sinon) et l'entier mis a jour avec le
***              nombre maximum des clauses de la strate
***              apparaissant en meme temps dans une
***              sous-base preferee lexicographiquement,
***
***    ATTENTION = la liste de bases Vliste donnee en parametre
***              est detruite !
***/

void rech_lex_par_strate (strate * S, clause * Phi, liste_base_strate ** PVliste,
                          liste_base_strate ** PNliste, int * Max,
                          int nb_max_symb, int nb_max_cl, int nb_max_lit_cl,
                                elem_tab_poids * t_b)
{
base_strate * b;
strate * dsb ;
int n ;
bool res ;
clause * contradiction ;
liste_base_strate * l ;

if (vide_ens(S) == faux)                                          /* strate non vide */
    {
    *PVliste = intro_strate_ds_liste (*PVliste,S);
    contradiction = creer_clause_vide(nb_max_symb);
    while ( vide_liste_base_strate(*PVliste) == faux)
        {
        *PVliste = suppr_liste_base_strate (*PVliste, &b);
        dsb = (b->les_strates)->un_ens ;
        n = dsb->nb_clauses ;
        if (n >= *Max)                        /* nb de formules de la derniere strate */
            {                                  /* de la base >= au max deja trouve */
            res = base_plus_clause_consistante (b,
                        contradiction, nb_max_symb,
                        nb_max_cl, nb_max_lit_cl,t_b);
            if (res == faux)                               /* base est consistante */
                {
                if (base_appartient_a_liste(b,*PNliste) ==
                        faux)
                            /* base n'appartient pas
                                deja a Nliste */
```

```
                        {
                        *PNliste = ajout_liste_base_strate(*PNliste,b);
                        *Max = n ;
                        }
                    else                        /* base appartient deja a Nliste */
                        {
                        b = destruc_plus_base_strate(b);
                        }
                    }
                else                                        /* base inconsistante */
                    {
                    if (n > *Max)                     /* on tente un developpement */
                        {
                        l = develop_base_strate (b,n);
                        *PVliste = concat_liste_base_strate
                                            (*PVliste,l);
                        }
                    else                     /* developpement inutile car resultat */
                        {                              /*trop petit en cardinalite */
                        b = destruc_plus_base_strate(b);
                        }
                    }
                }
            else                                  /* base trop petite en cardinalite */
                {
                b = destruc_plus_base_strate(b);
                }
            }
        clause_free(contradiction);
        }
else                                                                  /* strate vide */
        {
        *PNliste = *PVliste ;
        }
}


/**  fonction inference_lex
***         but : verifier que toutes les sous-bases preferees
***                 lexicographiquement inferent la formule
***                 Phi,
***         parametres : la liste des sous-bases preferees
***                 lexicographiquement, la clause a
***                 inferer, les informations permettant la
***                 creation de la base par Thierry Castell
***                 (nombres maximum de symboles, des
***                 clauses, de litteraux par clause), la table des poids
***         retourne : un booleen a vrai si toutes les
***                 sous-bases inferent la formule Phi, a faux sinon.
***/

bool inference_lex (liste_base_strate * Liste, clause * Phi,
                        int nb_max_symb, int nb_max_cl, int nb_max_lit_cl,
                                elem_tab_poids * t_b)
{
base * base_th ;
bool res ;
clause * c0;
clause * contradiction ;
rapport * r ;
rapport * rp ;
base_strate * b ;
liste_clause * lc ;

if (vide_liste_base_strate(Liste) == faux)
        {
        contradiction = creer_clause_vide(nb_max_symb);
```

```
        lc = negation_clause(Phi) ;
        while (vide_liste_base_strate (Liste) == faux)
                {
                b = duplicat_base_strate(Liste→une_base);
                b = ajout_liste_clause_a_base (b, lc);
                base_th = recopie (b,
                                nb_max_symb + clause_longueur(Phi),
                                nb_max_cl + clause_longueur(Phi),
                                MAX(nb_max_lit_cl,
                                clause_longueur(Phi)),t_b) ;
                destruc_plus_base_strate(b);
                r=rapport_creer();
                rp = rapport_creer();
                res = !(prouveur(nom_prouveur,base_th,rp,r));
                base_free(base_th);
                rapport_free(r);
                rapport_free(rp);
                if (res == faux)
                        {
                        clause_free(contradiction);
                        destruc_compl_liste_clause(lc);
                        return(faux);
                        }
                Liste = Liste→base_suiv ;
                }
        clause_free(contradiction);
        destruc_compl_liste_clause(lc);
        return(vrai);
        }
else
        {
        return (faux);
        }
}


/**  fonction UNI_LEX
***       but : calculer les sous-bases preferees lexicogra-
***             phiquement et verifier qu'elles inferent
***             toutes la formule,
***       algorithme utilise : algo naif a base d'appel de SAT,
***       parametres : la base a etudier, la clause a inferer,
***             les informations permettant la creation
***             de la base par Thierry Castell
***             (nombres maximum de symboles, des
***             clauses, de litteraux par clause), la table des poids,
***                     une commande d'affichage
***       retourne : un booleen a vrai si la base UNI-LEX-
***             infere la formule Phi, a faux sinon.
***/

bool UNI_LEX (base_strate * E, clause * Phi,
                        int nb_max_symb, int nb_max_cl,
                        int nb_max_lit_cl,
                          elem_tab_poids * t_b, bool aff)
{
  liste_base_strate * sous_bases_pref ;
  bool res;

  sous_bases_pref = rech_pref_lex(E, Phi, nb_max_symb, nb_max_cl, nb_max_lit_cl,t_b);

  if (aff == 1)
    {
      printf("les sous-bases preferees sont = \n");
      affich_liste_base_strate(sous_bases_pref);
    }
```

```
  res = inference_lex(sous_bases_pref , Phi, nb_max_symb, nb_max_cl, nb_max_lit_cl, t_b);

  sous_bases_pref = destruc_liste_base_strate(sous_bases_pref);

  return(res);
}



/**  fonction UNI_LEX2
***         but : calculer les sous-bases preferees lexicogra-
***                 phiquement et verifier qu'elles inferent
***                 toutes la formule,
***         algorithme utilise : algo moins naif avec appel a
***                 MAX_SAT,
***         parametres : la base a etudier, la clause a inferer,
***                 les informations permettant la creation
***                 de la base par Thierry Castell
***                 (nombres maximum de symboles, des
***                 clauses, de litteraux par clause),
***             la table des poids
***         retourne : un booleen a vrai si la base UNI-LEX-
***                 infere la formule Phi, a faux sinon.
***/

bool UNI_LEX2 (base_strate * b, clause * c,
                         int nb_max_symb, int nb_max_cl,
                         int nb_max_lit_cl,
                         elem_tab_poids * t_b)
{
  base_strate * baux;                          /* base + negation de c en derniere st. */
  liste_clause * negc ;                                   /* negation de la clause c */
  int k[nbmaxstrate] ;                      /* tab: strate -> nb max de formules sat. */
  unsigned int pmax;                           /* somme des poids des formules sat. */
  unsigned int pres;                         /* reste de la somme des poids formules */
  unsigned int max_pprec;                     /* somme poids formules ds strates prec. */
  base * base_th ;                           /* base au format Castell pour MAX_SAT */
  bool fini ;                                            /* booleen d'arret de boucle */
  bool res ;                                   /* resultat manip sur tables de poids */
  bool resfin ;              /* resultat UNI_LEX (vrai si b uni-lex-infere c, faux sinon) */
  int ns ;                                                 /* indice numero de strate */
  int i ;                                                             /* indice de k */
  elem_tab_poids * t_baux ;                                 /* table des poids de baux */
  strate * s ;                                 /* strate a rajouter (negation de c) */
  liste_clause * lc ;                           /* liste des clauses (negation de c) */
  liste_strate * ls ;                                     /* liste de strates courante */
  rapport * r ;                                  /* rapport des resultats de MAX_SAT */
  rapport * rp ;                                /* rapport des parametres de MAX_SAT */

  if (verif_tab_poids (t_b, b->nb_strates))
    {
      /* initialisations */
      for (i=0; i < nbmaxstrate ; i++)
        k[i]=0 ;
      baux = duplicat_base_strate(b);
      s = init_ens();
      lc = negation_clause (c);
      s->les_clauses = lc ;
      s->nb_clauses = clause_longueur(c) ;
        /* construction base contenant negation de la clause en derniere strate */
      baux = ajout_fin_base_strate(baux,s);
      t_baux = creer_init_tab_poids() ;
      res = ajout_fin_strate_tab_poids(t_baux, t_b, b->nb_strates - 1, s->nb_clauses);
      if (res)
        {
          /* calcul somme des poids max */
          pmax = 0 ;
```

```
        for (i=0 ; i < baux→nb_strates; i++)
            pmax = (t_baux[i].poids * t_baux[i].nb_cl_ds_st) + pmax ;
        /* parcours de de la base pour mettre a jour
              la somme des poids max */
        base_th = recopie(baux, nb_max_symb, nb_max_cl,
                                              nb_max_lit_cl,t_baux);
        r = rapport_creer();
        rp = rapport_creer ();
        max_sat (base_th, pmax,rp,r) ;
        rapport_affecte_int(r,"MAX_SAT", &pmax);
        base_free(base_th) ;
        rapport_free (r) ;
        rapport_free(rp);
        /* test du resultat sur la derniere strate (negation de c) :
              si la formule fait partie des max-satisfaites
           alors la base b n'infere pas la clause c, sinon b infere c */
        for (i = 0 ; i < (baux→nb_strates - 1); i++)
              {
              pres = pmax % t_baux[i].poids ;
              pmax = pres ;
              }
        if (pmax ≠ clause_longueur(c))
          resfin = vrai ;
        else
          resfin = faux ;
      }
    else
      {
        resfin = faux ;
        printf("Pb lors du rajout de la negation de la clause
                      en derniere strate \n");
      }
    /* nettoyage des structures creees */
    while (!(vide_liste_clause(lc)))
      {
        clause_free(lc→une_clause);
        lc= lc→clause_suiv ;
      }
    baux = destruc_plus_base_strate(baux) ;
    t_baux = destruc_tab_poids (t_baux);
  }
else
  {
    printf("La base est incorrecte du point de vue des poids \n");
    resfin = faux ;
  }
return(resfin);
}
```

```
/********************
        Fichier : mclperso.h

        Contient les types de donnees :

        Contient les #define de :
                . bool (type booleen exprime sous la forme d'un entier : 1 pour vrai, 0 pour faux),
                . vrai (l'entier 1),
                . faux (l'entier 0),
                . nbmaxstrate (nb max de strates autorise),
                . nbmaxclause (nb max de clauses autorise),
                . nbmaxlit (nb max de litteraux autorise),
                . nb_car_max (nb max de car ds une string),
                . nb_lit_par_cl (nb max de litteraux par clause),
                . path_bdd (chemin d'acces a bdd),
                . et (et logique),
                . ou (ou logique),
                . new (allocation de memoire),
                . newerr (detection d'une erreur d'allocation),
                . dispose (liberation memoire).

        Contient les declarations en externe des fonctions et procedures :

        Utilise les packages :
                . stdio,
                . stdlib.
********************/

#include <stdio.h>
#include <stdlib.h>

#define bool int
#define vrai 1
#define faux 0

#define nbmaxstrate 100
#define nbmaxclause 500
#define nbmaxlit 5000
#define nb_car_max 100
#define nb_lit_par_cl 3

#define path_bdd "../cmubdd/Lab/mykbdd"
#define path_anal "../benchmark/"
#define et &&
#define ou ||

#ifndef CLK_TCK
#define CLK_TCK 60
#endif

#define new(x) (x *)malloc(sizeof(x))
#define newtab(x,y) (x *)malloc(sizeof(x)*(y))
#define newerr(p,x) if (p==NULL) {printf(x); printf("\n"); exit(1);}
#define dispose(x) free(x)

#define nom_prouveur "pdp"

#define READ_D(c,x) {printf(c);fflush(stdout);scanf("%lf",x);}
```

```c
/*******************
        Fichier : clause_mcl.h

        Contient les types de donnees :
                . liste_clause

        Contient les #define de :

        Contient les declarations en externe des fonctions et procedures :
                . init_liste_clause,
                . affich_liste_clause,
                . ajout_liste_clause,
                . suppr_liste_clause,
                . tete_liste_clause,
                . destruc_liste_clause,
                . destruc_compl_liste_clause,
                . vide_liste_clause,
                . duplicat_liste_clause,
                . negation_clause,
                . creer_clause_vide,
                . egal_clause.

        Utilise les packages :
                . base (Thierry Castell),
                . litteral (Thierry Castell),
                . mclperso.
********************/

#include <bc.h>
#include <litteral.h>
#include "mclperso.h"



/***** les types  *******/
typedef struct liste_clause { struct liste_clause * clause_suiv ;
                              clause * une_clause;} liste_clause;



/****** les procedures et fonctions ******/
liste_clause * init_liste_clause();

liste_clause * ajout_liste_clause (liste_clause *, clause *);

void affich_liste_clause (liste_clause *);

bool vide_liste_clause(liste_clause *);

liste_clause * suppr_liste_clause (liste_clause *, clause **);

clause * tete_liste_clause (liste_clause *);

liste_clause * destruc_liste_clause (liste_clause *);

liste_clause * destruc_compl_liste_clause (liste_clause *);

liste_clause * duplicat_liste_clause (liste_clause *);

liste_clause * negation_clause (clause *);

clause * creer_clause_vide(int);

bool egal_clause (clause *, clause *);
```

```
/*******************
        Fichier : clause_mcl.c

        Contient les types de donnees :

        Contient les #define de :

        Contient les declarations et le corps des fonctions et procedures :
                . init_liste_clause,
                . affich_liste_clause,
                . ajout_liste_clause,
                . suppr_liste_clause,
                . tete_liste_clause,
                . destruc_liste_clause,
                . destruc_compl_liste_clause,
                . vide_liste_clause,
                . duplicat_liste_clause,
                . negation_clause,
                . creer_clause_vide,
                . egal_clause.

        Utilise les packages :
                . clause_mcl.
********************/

#include "clause_mcl.h"


/**  fonction init_liste_clause
***        but : initialiser une variable de type liste_clause avec la valeur NULL,
***        parametres : neant,
***        retourne : la liste_clause a NULL.
***/

liste_clause * init_liste_clause()
{
return(NULL);
}


/**  fonction ajout_liste_clause
***        but : ajouter une clause a une liste de clauses,
***        parametres : la liste de clauses, la clause a rajouter,
***        retourne : le pointeur sur la liste de clauses modifiee.
***
***        ATTENTION : La clause ajoutee doit deja exister physiquement.
***
***/

liste_clause * ajout_liste_clause(liste_clause * lc, clause * c)
{
liste_clause * lcaux ;

lcaux = lc ;
lc = new(liste_clause);
newerr(lc, "ajout_liste_clause : plus de memoire");

lc->une_clause = c ;
lc->clause_suiv = lcaux ;
return(lc);
}



/**  procedure affich_liste_clause
***        but : afficher une liste de clauses,
***        parametres : la liste de clauses.
```

```
***/

void affich_liste_clause(liste_clause * lc)
{
liste_clause * lcaux ;
int cpt ;

if (vide_liste_clause(lc) == faux)
        {
        lcaux = lc ;
        cpt = 1 ;
        while (lcaux ≠ NULL)
                {
                printf("clause numero %d = ",cpt);
                clause_print(lcaux→une_clause);
                printf("\n");
                lcaux = lcaux→clause_suiv;
                cpt = cpt + 1 ;
                }
        }
else
        {
        printf("La liste de clauses est vide \n");
        }
}



/**  fonction vide_liste_clause
***         but : verifie si la liste_clause est vide ou pas,
***         parametres : la liste de clause,
***         retourne : la valeur vrai si la liste_clause est vide, faux sinon.
***/

bool vide_liste_clause(liste_clause * lc)
{
if (lc == NULL)
        {
        return(1);
        }
else
        {
        return(0);
        }
}



/**  fonction suppr_liste_clause
***         but : supprimer la premiere clause d'une liste de clauses,
***         parametres : la liste de clauses, l'adresse de la clause qui a ete supprimee,
***         retourne : la nouvelle liste de clauses modifiee et la clause supprimee.
***
***     ATTENTION : La clause n'est pas supprimee physiquement
***/

liste_clause * suppr_liste_clause(liste_clause * lc, clause ** pc)
{
liste_clause * lcaux ;

if (vide_liste_clause(lc) == faux)
        {
        *pc = lc→une_clause ;
        lcaux = lc→clause_suiv ;

        dispose((liste_clause *)lc) ;
```

```
        return(lcaux);
        }
else
    {
    return(lc);
        }
}



/**  fonction tete_liste_clause
***          but : renvoyer la premiere clause d'une liste de clauses,
***          parametres : la liste de clauses,
***          retourne : la premiere clause de la liste de clauses.
***/

clause * tete_liste_clause(liste_clause * lc)
{

if (vide_liste_clause(lc) == faux)
        {
        return(lc→une_clause);
        }
else
    {
    return(NULL);
    }
}



/**  fonction destruc_liste_clause
***          but : detruire une liste de clauses,
***          parametres : la liste de clauses,
***          retourne : la liste de clauses a NULL.
***
***      ATTENTION : Les clauses ne sont pas supprimees physiquement
***/

liste_clause * destruc_liste_clause(liste_clause * lc)
{
liste_clause * lcaux ;
clause * c ;

while (vide_liste_clause(lc) == faux)
        {
        lcaux = lc→clause_suiv ;
        c = lc→une_clause ;
        dispose((liste_clause *)lc) ;
        lc = lcaux ;
        }
return (lc);
}



/**  fonction destruc_compl_liste_clause
***          but : detruire une liste de clauses en detruisant aussi les clauses,
***          parametres : la liste de clauses,
***          retourne : la liste de clauses a NULL.
***
***      ATTENTION : Les clauses sont supprimees physiquement
***/

liste_clause * destruc_compl_liste_clause(liste_clause * lc)
```

```
{
liste_clause * lcaux ;
clause * c ;

while (vide_liste_clause(lc) == faux)
        {
        lcaux = lc→clause_suiv ;
        c = lc→une_clause ;
        clause_free(c);
        dispose((liste_clause *)lc) ;
        lc = lcaux ;
        }
return (lc);
}


/**  fonction duplicat_liste_clause
***         but : duplique la liste de clauses,
***         parametres : la liste de clauses a dupliquer,
***         retourne : la liste de clauses dupliquee.
***
***    ATTENTION : Les clauses ne sont pas dupliquees
***/

liste_clause * duplicat_liste_clause (liste_clause * lc)
{
liste_clause * lcaux ;

        liste_clause * duplicat_liste_clause_recur (liste_clause * lc, liste_clause * lcaux)
        {
        clause * ci ;

        if (vide_liste_clause(lc) == faux)
                {
                lcaux = duplicat_liste_clause_recur (lc→clause_suiv, lcaux) ;
                ci = lc→une_clause ;
                lcaux = ajout_liste_clause (lcaux,ci) ;
                return(lcaux);
                }
        else
                {
                return (NULL);
                }
        }


lcaux = init_liste_clause() ;
lcaux = duplicat_liste_clause_recur (lc,lcaux) ;
return(lcaux);
}




/**  fonction negation_clause
***         but : calculer la negation d'une clause et la mettre sous la forme
***            d'une liste de clauses,
***         parametres : la clause dont il faut trouver la negation,
***         retourne : la liste de clauses correspondant a la negation de la clause.
***
***     ATTENTION : il y a creation physique des clauses correspondant a la
***                 negation.
***/

liste_clause * negation_clause (clause * c)
{
liste_clause * l ;
```

```
clause * c1;
int n ;
int cpt ;
int lit ;

l = init_liste_clause() ;
n = clause_longueur(c) ;
for (cpt =1; cpt ≤ n; cpt++)
          {
          c1 = clause_creer() ;
          lit = clause_element(c);
          lit = lit_compl(lit);
          clause_ajouter(c1,lit);
          l = ajout_liste_clause (l,c1);
          }
return(l);
}


/**  fonction creer_clause_vide
***         but : creer la clause vide,
***         parametres : le nombre maximal de symboles dans la base,
***         retourne : la clause vide.
***
***      ATTENTION : il y a creation physique de la clause vide.
***/

clause * creer_clause_vide(int nb_max_symb)
{
clause * c ;

c = clause_creer();
clause_init(c);
return(c);
}



/**  fonction egal_clause
***         but : compare deux clauses,
***         parametres : les deux clauses a comparer,
***         retourne : un booleen a vrai si les deux clauses sont egales, a faux sinon.
***/

bool egal_clause(clause * c1, clause * c2)
{
bool res ;
int cpt ;

if (clause_longueur(c1) ≠ clause_longueur(c2))
          {
          res = faux ;
          }
else
          {
          res = vrai ;
          cpt = 1 ;
          while ( (cpt ≤ clause_longueur(c1)) && (res == vrai))
                    {
                    if (clause_appartient(c2, clause_element(c1)) == faux)
                              {
                              res = faux ;
                              }
                    else
                              {
                              cpt++ ;
```

```
                  }
              }
          }
      return(res);
      }
```

```
/********************
        Fichier : strate.h

        Contient les types de donnees :
                . ens,
                . liste_ens.

        Contient les #define de :
                . strate,
                . liste_strate.

        Contient les declarations en externe des fonctions et procedures :
                . init_ens,
                . affich_ens,
                . ajout_ens,
                . suppr_ens,
                . tete_ens,
                . destruc_ens,
                . destruc_compl_ens,
                . destruc_plus_ens,
                . destruc_compl_plus_ens,
                . vide_ens,
                . duplicat_ens,
                . egal_ens,
                . init_liste_ens,
                . affich_liste_ens,
                . ajout_liste_ens,
                . ajout_fin_liste_ens,
                . suppr_liste_ens,
                . tete_liste_ens,
                . destruc_liste_ens,
                . destruc_compl_liste_ens,
                . vide_liste_ens,
                . duplicat_liste_ens.

        Utilise les packages :
                . clause_mcl,
                . clause (Thierry Castell),
                . mclperso.
********************/

#include <clause.h>
#include "clause_mcl.h"
#include "mclperso.h"


/***** les types  *******/
typedef struct ens { struct liste_clause * les_clauses ;
                     int nb_clauses;} ens ;

typedef struct liste_ens { struct liste_ens * ens_suiv ;
                     struct ens * un_ens;} liste_ens;



/***** les define *******/
#define strate ens
#define liste_strate liste_ens



/****** les procedures et fonctions ******/
ens * init_ens();

ens * ajout_ens (ens *, clause *);

void affich_ens (ens *);
```

```
bool vide_ens(ens *);

ens * suppr_ens (ens *, clause **);

clause * tete_ens (ens *);

ens * destruc_ens (ens *);

ens * destruc_compl_ens (ens *);

ens * destruc_plus_ens (ens *);

ens * destruc_compl_plus_ens (ens *);

ens * duplicat_ens (ens *);

bool egal_ens (ens *, ens *);


liste_ens * init_liste_ens();

liste_ens * ajout_liste_ens (liste_ens *, ens *);

liste_ens * ajout_fin_liste_ens (liste_ens *, ens *);

void affich_liste_ens (liste_ens *);

bool vide_liste_ens(liste_ens *);

liste_ens * suppr_liste_ens (liste_ens *, ens **);

ens * tete_liste_ens (liste_ens *);

liste_ens * destruc_liste_ens (liste_ens *);

liste_ens * destruc_compl_liste_ens (liste_ens *);

liste_ens * duplicat_liste_ens (liste_ens *);
```

```
/********************
        Fichier : strate.c

        Contient les types de donnees :

        Contient les #define de :

        Contient les declarations et le corps des fonctions et procedures :
                . init_ens,
                . affich_ens,
                . ajout_ens,
                . suppr_ens,
                . tete_ens,
                . destruc_ens,
                . destruc_compl_ens,
                . destruc_plus_ens,
                . destruc_compl_ens,
                . vide_ens,
                . duplicat_ens,
                . egal_ens,
                . init_liste_ens,
                . affich_liste_ens,
                . ajout_liste_ens,
                . ajout_fin_liste_ens
                . suppr_liste_ens,
                . tete_liste_ens,
                . destruc_liste_ens,
                . destruc_compl_liste_ens,
                . vide_liste_ens,
                . duplicat_liste_ens.

        Utilise les packages :
                . strate.
********************/

#include "strate.h"



/**  fonction init_ens
***        but : initialiser une variable de type ens,
***        parametres : neant,
***        retourne : l'ens initialise avec le champ les_clauses a NULL et nb_clauses a 0.
***/

ens * init_ens()
{
ens * e ;

e = new(ens);
newerr(e, "init_ens : plus de memoire");

e->les_clauses = NULL ;
e->nb_clauses = 0 ;
return(e);
}



/**  fonction ajout_ens
***        but : ajouter une clause a un ens,
***        parametres : l'ens, la clause a rajouter,
***        retourne : le pointeur sur l'ens modifie.
***
***        ATTENTION : La clause ajoutee doit deja exister physiquement.
***
***/
```

```
ens * ajout_ens(ens * e, clause * c)
{

e->les_clauses = ajout_liste_clause(e->les_clauses, c);
e->nb_clauses = e->nb_clauses + 1;
return(e);
}



/**  procedure affich_ens
***          but : afficher un ens,
***          parametres : l'ens.
***/

void affich_ens(ens * e)
{

if (vide_ens(e) == faux )
    {
        printf("L'ensemble contient %d clause(s) : \n",e->nb_clauses);
        affich_liste_clause(e->les_clauses);
    }
else
        {
        printf("L'ensemble est vide \n");
        }
}




/**  fonction vide_ens
***          but : verifie si l'ens est vide ou pas
***                  (un ens est vide s'il n'a pas ete cree ou
***                  si il vient d'etre initialise),
***          parametres : l'ens,
***          retourne : la valeur vrai si l'ens est vide, faux sinon.
***/

bool vide_ens(ens * e)
{
if (e == NULL)
        {
        return(1);
        }
else
        {
        if (e->nb_clauses == 0)
                {
                return(1);
                }
        else
                {
                return(0);
                }
        }
}



/**  fonction suppr_ens
***          but : supprimer la premiere clause d'un ens,
***          parametres : l'ens, l'adresse de la clause qui a ete supprimee,
***          retourne : le nouvel ens modifie et la clause supprimee.
***
```

```
***          ATTENTION : la clause n'est pas supprimee physiquement
***/

ens * suppr_ens(ens * e, clause ** pc)
{

if (vide_ens(e) == faux)
        {
        e->les_clauses = suppr_liste_clause (e->les_clauses,pc);
        e->nb_clauses = e->nb_clauses - 1 ;
        return(e);
        }
else
    {
    return(e);
    }
}



/**  fonction tete_ens
***          but : renvoyer la premiere clause d'un ens,
***          parametres : l'ens,
***          retourne : la premiere clause de l'ens.
***/

clause * tete_ens(ens * e)
{

if (vide_ens(e) == faux)
        {
        return(tete_liste_clause(e->les_clauses));
        }
else
    {
    return(NULL);
    }
}



/**  fonction destruc_ens
***          but : detruire un ens, c'est-a-dire detruit la liste de clauses et
***                  remet le nb de clauses a 0,
***          parametres : l'ens,
***          retourne : l'ens reinitialise.
***
***          ATTENTION : les clauses ne sont pas supprimees physiquement
***/

ens * destruc_ens(ens * e)
{

if (vide_ens(e) == faux)
        {
        e->les_clauses = destruc_liste_clause(e->les_clauses);
        e->nb_clauses = 0;
        }
return (e);
}


/**  fonction destruc_compl_ens
***          but : detruire un ens, c'est-a-dire detruit la liste de clauses en
***                  en detruisant les clauses et
***                  remet le nb de clauses a 0,
```

```
***        parametres : l'ens,
***        retourne : l'ens reinitialise.
***
***        ATTENTION : les clauses sont supprimees physiquement
***/

ens * destruc_compl_ens(ens * e)
{

if (vide_ens(e) == faux)
        {
        e->les_clauses = destruc_compl_liste_clause(e->les_clauses);
        e->nb_clauses = 0;
        }
return (e);
}



/**  fonction destruc_plus_ens
***        but : detruire un ens completement, c'est-a-dire libere la
***                place occupee par cet ens,
***        parametres : l'ens,
***        retourne : l'ens a NULL.
***
***        ATTENTION : les clauses ne sont pas supprimees physiquement
***/

ens * destruc_plus_ens(ens * e)
{

e = destruc_ens(e);
dispose((ens *)e) ;
return (NULL);
}


/**  fonction destruc_compl_plus_ens
***        but : detruire un ens completement, c'est-a-dire libere la
***                place occupee par cet ens y compris par ses clauses,
***        parametres : l'ens,
***        retourne : l'ens a NULL.
***
***        ATTENTION : les clauses sont supprimees physiquement
***/

ens * destruc_compl_plus_ens(ens * e)
{

e = destruc_compl_ens(e);
dispose((ens *)e) ;
return (NULL);
}




/**  fonction duplicat_ens
***        but : duplique un ensemble,
***        parametres : l'ensemble a dupliquer,
***        retourne : l'ensemble duplique.
***
***        ATTENTION : les clauses ne sont pas dupliquees physiquement
***/

ens * duplicat_ens (ens * e)
{
```

```
ens * eaux ;

eaux = init_ens() ;
eaux→nb_clauses = e→nb_clauses;
eaux→les_clauses = duplicat_liste_clause (e→les_clauses);
return(eaux);
}
```

```
/**  fonction egal_ens
***        but : compare deux ensembles,
***        parametres : les deux ensembles a comparer,
***        retourne : le booleen resultat (vrai si les deux ensembles sont
***                 egaux, faux sinon).
***/

bool egal_ens (ens * e1, ens * e2)
{
bool res ;
liste_clause * lc1;
liste_clause * lc2 ;

if (e1→nb_clauses ≠ e2→nb_clauses)
        {
        res = faux ;
        }
else
        {
        res = vrai ;
        lc1 = e1→les_clauses ;
        lc2 = e2→les_clauses ;
        while ((vide_liste_clause(lc1) == faux) && (res== vrai))
                {
                if (egal_clause(lc1→une_clause, lc2→une_clause) == vrai)
                        {
                        lc1 = lc1→clause_suiv;
                        lc2= lc2→clause_suiv;
                        }
                else
                        {
                        res = faux ;
                        }
                }
        }
return(res);
}
```

```
/***********************************************************/
```

```
/**  fonction init_liste_ens
***        but : initialiser une variable de type liste_ens avec la valeur NULL,
***        parametres : neant,
***        retourne : la liste_ens a NULL.
***/

liste_ens * init_liste_ens()
{
return(NULL);
}
```

```
/**  fonction ajout_liste_ens
***         but : ajouter un ens a une liste d'ens,
***         parametres : la liste d'ens, l'ens a rajouter,
***         retourne : le pointeur sur la liste d'ens modifiee.
***
***         ATTENTION : l'ensemble ajoute doit deja exister.
***
***/

liste_ens * ajout_liste_ens(liste_ens * le, ens * e)
{
liste_ens * leaux ;

leaux = le ;
le = new(liste_ens);
newerr(le, "ajout_liste_ens : plus de memoire");

le->un_ens = e ;
le->ens_suiv = leaux ;
return(le);
}
```

```
/**  fonction ajout_fin_liste_ens
***         but : ajouter un ens a la fin d'une liste d'ens,
***         parametres : la liste d'ens, l'ens a rajouter,
***         retourne : le pointeur sur la liste d'ens modifiee.
***
***         ATTENTION : l'ensemble ajoute doit deja exister.
***
***/

liste_ens * ajout_fin_liste_ens(liste_ens * le, ens * e)
{
liste_ens * leaux;
liste_ens * leaux2 ;

leaux = le ;
if (vide_liste_ens(leaux))
  le = ajout_liste_ens(le,e);
else
  {
    while (! (vide_liste_ens(leaux->ens_suiv)))
      leaux = leaux->ens_suiv ;
    leaux2 = new(liste_ens);
    newerr(le, "ajout_liste_ens : plus de memoire");
    leaux2->un_ens = e ;
    leaux2->ens_suiv = NULL ;
    leaux->ens_suiv = leaux2 ;
  }
return(le);
}
```

```
/**  procedure affich_liste_ens
***         but : afficher une liste d'ens,
***         parametres : la liste d'ens.
***/

void affich_liste_ens(liste_ens * le)
{
liste_ens * leaux ;
int cpt ;
```

```
if (vide_liste_ens(le) == faux)
        {
        leaux = le ;
        cpt = 1 ;
        while (leaux ≠ NULL)
                {
                printf("ensemble numero %d = ",cpt);
                affich_ens(leaux→un_ens);
                printf("\n");
                leaux = leaux→ens_suiv;
                cpt = cpt + 1 ;
                }
        }
else
        {
        printf("La liste d'ensembles est vide \n");
        }
}
```

```
/**  fonction vide_liste_ens
***        but : verifie si la liste_ens est vide ou pas,
***        parametres : la liste d'ens,
***        retourne : la valeur vrai si la liste_ens est vide, faux sinon.
***/

bool vide_liste_ens(liste_ens * le)
{
if (le == NULL)
        {
        return(1);
        }
else
        {
        return(0);
        }
}
```

```
/**  fonction suppr_liste_ens
***        but : supprimer le premier ens d'une liste d'ens,
***        parametres : la liste d'ens, l'adresse de l'ens qui a ete supprime,
***        retourne : la nouvelle liste d'ens modifiee et l'ens supprime.
***
***        ATTENTION : l'ensemble n'est pas supprime physiquement
***/

liste_ens * suppr_liste_ens(liste_ens * le, ens ** pe)
{
liste_ens * leaux ;

if (vide_liste_ens(le) == faux)
        {
        *pe = le→un_ens ;
        leaux = le→ens_suiv ;

        dispose((liste_ens *)le) ;
        return(leaux);
        }
else
    {
    return(le);
    }
}
```

```
/**   fonction tete_liste_ens
***          but : renvoyer le premier ens d'une liste d'ens,
***          parametres : la liste d'ens,
***          retourne : le premier ens de la liste d'ens.
***/

ens * tete_liste_ens(liste_ens * le)
{

if (vide_liste_ens(le) == faux)
        {
        return(le->un_ens);
        }
else
    {
    return(NULL);
    }
}



/**   fonction destruc_liste_ens
***          but : detruire une liste d'ens,
***          parametres : la liste d'ens,
***          retourne : la liste d'ens a NULL.
***
***          ATTENTION : les clauses de l'ens ne sont pas supprimees
***                  physiquement
***/

liste_ens * destruc_liste_ens(liste_ens * le)
{
liste_ens * leaux ;
ens * e ;

while (vide_liste_ens(le) == faux)
        {
        leaux = le->ens_suiv ;
        e = le->un_ens ;
        e = destruc_plus_ens(e);
        dispose((liste_ens *)le) ;
        le = leaux ;
        }
return (le);
}




/**   fonction destruc_compl_liste_ens
***          but : detruire une liste d'ens,
***          parametres : la liste d'ens,
***          retourne : la liste d'ens a NULL.
***
***          ATTENTION : les clauses de l'ens sont supprimees
***                  physiquement
***/

liste_ens * destruc_compl_liste_ens(liste_ens * le)
{
liste_ens * leaux ;
ens * e ;
```

```
while (vide_liste_ens(le) == faux)
          {
          leaux = le->ens_suiv ;
          e = le->un_ens ;
          e = destruc_compl_plus_ens(e);
          dispose((liste_ens *)le) ;
          le = leaux ;
          }
return (le);
}




/**   fonction duplicat_liste_ens
***          but : duplique la liste d'ensembles,
***          parametres : la liste d'ensembles a dupliquer,
***          retourne : la liste d'ensembles dupliquee.
***
***          ATTENTION : les clauses de l'ens ne sont pas dupliquees
***                    physiquement
***/

liste_ens * duplicat_liste_ens (liste_ens * le)
{
liste_ens * leaux ;

          liste_ens * duplicat_liste_ens_recur (liste_ens * le, liste_ens * leaux)
          {
          ens * e ;

          if (vide_liste_ens(le) == faux)
                    {
                    leaux = duplicat_liste_ens_recur (le->ens_suiv,leaux);
                    e = duplicat_ens(le->un_ens) ;
                    leaux = ajout_liste_ens (leaux,e) ;
                    return(leaux);
                    }
          else
                    {
                    return(NULL);
                    }
          }


leaux = init_liste_ens() ;
leaux = duplicat_liste_ens_recur (le,leaux);
return(leaux);
}
```

```
/********************
        Fichier : table_poids.h

        Contient les types de donnees :
                . elem_tab_poids

        Contient les #define de :

        Contient les declarations en externe des fonctions et procedures :
                . creer_init_tab_poids,
                . creer_init_a_un_tab_poids,
                . destruc_tab_poids,
                . affich_tab_poids,
                . verif_tab_poids,
                . ajout_fin_strate_tab_poids.
                . ajout_debut_strate_tab_poids.

        Utilise les packages :
                . base (Thierry Castell),
                . mclperso.
*********************/

#include "mclperso.h"



/***** les types  *******/
typedef struct elem_tab_poids {
                unsigned int poids ;
                int nb_cl_ds_st ;} elem_tab_poids;
                        /* l'indice 0 represente la premiere strate */



/****** les procedures et fonctions ******/
elem_tab_poids * creer_init_tab_poids();

elem_tab_poids * creer_init_a_un_tab_poids();

elem_tab_poids * destruc_tab_poids(elem_tab_poids *);

void affich_tab_poids (elem_tab_poids *);

bool verif_tab_poids(elem_tab_poids *, int);

bool ajout_fin_strate_tab_poids(elem_tab_poids *, elem_tab_poids *, int, int);

bool ajout_debut_strate_tab_poids(elem_tab_poids *, elem_tab_poids *, int, int);
```

```
/********************
        Fichier : table_poids.c

        Contient les types de donnees :

        Contient les #define de :

        Contient les declarations et le corps des fonctions et procedures :
                . creer_init_tab_poids,
                . creer_init_a_un_tab_poids,
                . destruc_tab_poids,
                . affich_tab_poids,
                . verif_tab_poids,
                . ajout_fin_strate_tab_poids.
                . ajout_debut_strate_tab_poids.

        Utilise les packages :
                . table_poids.
********************/

#include "table_poids.h"


/**  fonction creer_init_tab_poids
***        but : creer et initialiser une variable de type table d'elem_tab_poids
***                        avec la valeur 0,
***        parametres : neant,
***        retourne : la table d'elem_tab_poids mise a zero.
***/

elem_tab_poids * creer_init_tab_poids()
{
  elem_tab_poids * t ;
  int i ;

  t = newtab(elem_tab_poids, nbmaxstrate);
  newerr(t, "creation table des poids : plus de memoire \n");

  for (i = 0 ; i < nbmaxstrate ; i++)
    {
      t[i].poids = 0 ;
      t[i].nb_cl_ds_st = 0 ;
    }
  return (t) ;
}




/**  fonction creer_init_a_un_tab_poids
***        but : creer et initialiser une variable de type table d'elem_tab_poids
***                        avec la valeur 1 dans le champ poids,
***        parametres : neant,
***        retourne : la table d'elem_tab_poids mise a jour.
***/

elem_tab_poids * creer_init_a_un_tab_poids()
{
  elem_tab_poids * t ;
  int i ;

  t = newtab(elem_tab_poids, nbmaxstrate);
  newerr(t, "creation table des poids : plus de memoire \n");

  for (i = 0 ; i < nbmaxstrate ; i++)
    {
      t[i].poids = 1 ;
```

```
      t[i].nb_cl_ds_st = 0 ;
    }
  return (t) ;
}



/**  fonction destruc_tab_poids
***        but : liberer la place occupee par une variable de type table d'elem_tab_poids,
***        parametres : la table d'elem_tab_poids,
***        retourne : la table d'elem_tab_poids mise a NULL.
***/

elem_tab_poids * destruc_tab_poids(elem_tab_poids * t)
{

  if (t ≠ NULL)
    dispose(t) ;
  return (NULL) ;
}



/**  procedure affich_tab_poids
***        but : afficher une table de poids,
***        parametres : la table de poids.
***/


void affich_tab_poids (elem_tab_poids * t)
{
  int i ;

  if (t == NULL)
    printf("\n table des poids n'existe pas \n");
  else
    {
      printf("\n Table des poids : \n");
      i = 0 ;
      while ((i < nbmaxstrate) et (t[i].nb_cl_ds_st ≠ 0))
        {
          printf(" Strate %d : poids = %d, nb de clauses ds la strate = %d \n",i, t[i].poids,
t[i].nb_cl_ds_st) ;
          i ++;
        }
    }
}



/**  fonction verif_tab_poids
***        but : verifier que la base de strates est correcte du point de vue des poids,
***        parametres : la table a verifier, et le nb de cases
***                 significatives de la table
***        retourne : un booleen a vrai si la verification est correcte,
***                 a faux sinon.
***/

bool verif_tab_poids(elem_tab_poids * t, int nb)
{
  int i ;
  bool pb ;
  unsigned int pprec ;
  int nbprec ;

      pb = faux ;
```

```
    pprec = 1 ;
    nbprec = 0 ;
    i= nb -1;
    while ((i ≥0) et (! pb))
      {
        if (t[i].poids ≠ pprec * (nbprec +1))
          pb = vrai ;
        else
          {
            pprec =t[i].poids ;
            nbprec = t[i].nb_cl_ds_st ;
            i -- ;
          }
      }
  if (pb)
      printf("Probleme ds la verification de la table des poids \n");
  return(! pb) ;
}
```

```
/**  fonction ajout_fin_strate_tab_poids
***      but : mettre a jour la table d'elem_tab_poids t2 avec la table t1 a la suite de
***              l'ajout d'une strate en fin de base (donc avec un poids a 1)
***                contenant nb clauses,
***      parametres : la table des poids a mettre a jour, la table des poids initiale,
***                l'indice du dernier element significatif de la table initiale
***                et le nombre de clauses consitutant la strate rajoutee
***      retourne : un booleen a vrai si la mise a jour s'est bien passee,
***               a faux sinon.
***/

bool ajout_fin_strate_tab_poids(elem_tab_poids * t2, elem_tab_poids * t1, int der_t1, int nb)
{
  int i;
  bool res ;

  if (der_t1 == nbmaxstrate - 1)
    res = faux;                          /* il n'y a pas la place pour une nouvelle strate */
  else
    {
      t2[der_t1 + 1].poids = 1 ;
      t2[der_t1 + 1].nb_cl_ds_st = nb ;
      for (i = der_t1; i ≥0; i--)
        {
          t2[i].nb_cl_ds_st = t1[i].nb_cl_ds_st ;
          t2[i].poids = t2[i+1].poids * (t2[i + 1].nb_cl_ds_st + 1) ;
        }
      res = vrai ;
    }
  return(res);
}
```

```
/**  fonction ajout_debut_strate_tab_poids
***      but : mettre a jour la table d'elem_tab_poids t2 avec la table t1 a la suite de
***              l'ajout d'une strate en debut de base contenant
***              nb clauses,
***      parametres : la table des poids a mettre a jour, la table des poids initiale,
***                l'indice du dernier element significatif de la table initiale
***                et le nombre de clauses consitutant la strate rajoutee
***      retourne : un booleen a vrai si la mise a jour s'est bien passee,
***               a faux sinon.
***/
```

```
bool ajout_debut_strate_tab_poids(elem_tab_poids * t2, elem_tab_poids * t1, int der_t1, int nb)
{
  int i;
  bool res ;

  if (der_t1 == nbmaxstrate - 1)
    res = faux;                                  /* il n'y a pas la place pour une nouvelle strate */
  else
    {
      for (i = der_t1; i ≥0; i--)
        {
          t2[i+1].nb_cl_ds_st = t1[i].nb_cl_ds_st ;
          t2[i+1].poids = t1[i].poids ;
        }
      if (der_t1 < 0)
        t2[0].poids = 1 ;
      else
        t2[0].poids = t2[1].poids * (t2[1].nb_cl_ds_st + 1) ;
      t2[0].nb_cl_ds_st = nb ;
      res = vrai ;
    }
  return(res);
}
```

```
/********************
        Fichier : base_strate.h

        Contient les types de donnees :
                . base_strate,
                . liste_base_strate.

        Contient les #define de :

        Contient les declarations en externe des fonctions et procedures :
                . init_base_strate,
                . affich_base_strate,
                . ajout_base_strate,
                . ajout_fin_base_strate,
                . suppr_base_strate,
                . tete_base_strate,
                . destruc_base_strate,
                . destruc_plus_base_strate,
                . destruc_compl_base_strate,
                . destruc_compl_plus_base_strate,
                . vide_base_strate,
                . recopie,
                . duplicat_base_strate,
                . develop_base_strate,
                . ajout_liste_clause_a_base,
                . base_plus_clause_consistante,
                . egal_base-strate,
                . init_liste_base_strate,
                . affich_liste_base_strate,
                . ajout_liste_base_strate,
                . suppr_liste_base_strate,
                . tete_liste_base_strate,
                . destruc_liste_base_strate,
                . destruc_compl_liste_base_strate,
                . vide_liste_base_strate,
                . concat_liste_base_strate,
                . intro_strate_ds_liste,
                . base_appartient_a_liste.

        Utilise les packages :
                . strate,
                . table_poids,
                . prouveur (Thierry Castell),
                . mclperso.
********************/

#include <prouveur.h>
#include "strate.h"
#include "mclperso.h"
#include "table_poids.h"

/***** les types *******/
typedef struct base_strate { struct liste_strate * les_strates ;
                             int nb_strates;} base_strate ;

typedef struct liste_base_strate { struct liste_base_strate * base_suiv ;
                             struct base_strate * une_base;} liste_base_strate;



/****** les procedures et fonctions ******/
base_strate * init_base_strate();

base_strate * ajout_base_strate (base_strate *, strate *);

base_strate * ajout_fin_base_strate (base_strate *, strate *);
```

```
void affich_base_strate (base_strate *);
bool vide_base_strate(base_strate *);
base_strate * suppr_base_strate (base_strate *, strate **);
strate * tete_base_strate (base_strate *);
base_strate * destruc_base_strate (base_strate *);
base_strate * destruc_plus_base_strate (base_strate *);
base_strate * destruc_compl_base_strate (base_strate *);
base_strate * destruc_compl_plus_base_strate (base_strate *);
base * recopie (base_strate *, int, int, int, elem_tab_poids *);
base_strate * duplicat_base_strate (base_strate *);
base_strate * ajout_liste_clause_a_base (base_strate *, liste_clause *);
liste_base_strate * develop_base_strate (base_strate *, int);
bool base_plus_clause_consistante(base_strate *, clause *, int, int, int, elem_tab_poids *);
bool egal_base_strate (base_strate *, base_strate *);


liste_base_strate * init_liste_base_strate();
liste_base_strate * ajout_liste_base_strate (liste_base_strate *, base_strate *);
void affich_liste_base_strate (liste_base_strate *);
bool vide_liste_base_strate(liste_base_strate *);
liste_base_strate * suppr_liste_base_strate (liste_base_strate *, base_strate **);
base_strate * tete_liste_base_strate (liste_base_strate *);
liste_base_strate * destruc_liste_base_strate (liste_base_strate *);
liste_base_strate * destruc_compl_liste_base_strate (liste_base_strate *);
liste_base_strate * concat_liste_base_strate (liste_base_strate *, liste_base_strate *);
liste_base_strate * intro_strate_ds_liste (liste_base_strate *, strate *);
bool base_appartient_a_liste (base_strate *, liste_base_strate *);
```

```
/********************
        Fichier : base_strate.c

        Contient les types de donnees :

        Contient les #define de :

        Contient les declarations et le corps des fonctions et procedures :
                . init_base_strate,
                . affich_base_strate,
                . ajout_base_strate,
                . ajout_fin_base_strate,
                . suppr_base_strate,
                . tete_base_strate,
                . destruc_base_strate,
                . destruc_plus_base_strate,
                . destruc_compl_base_strate,
                . destruc_compl_plus_base_strate,
                . vide_base_strate,
                . recopie,
                . duplicat_base_strate,
                . develop_base_strate,
                . ajout_liste_clause_a_base,
                . base_plus_clause_consistante,
                . egal_base_strate,
                . init_liste_base_strate,
                . affich_liste_base_strate,
                . ajout_liste_base_strate,
                . suppr_liste_base_strate,
                . tete_liste_base_strate,
                . destruc_liste_base_strate,
                . destruc_compl_liste_base_strate,
                . vide_liste_base_strate,
                . concat_liste_base_strate,
                . intro_strate_ds_liste,
                . base_appartient_a_liste.

        Utilise les packages :
                . base_strate.
********************/

#include "base_strate.h"



/**  fonction init_base_strate
***         but : initialiser une variable de type base_strate,
***         parametres : neant,
***         retourne : la base_strate initialisee avec le champ les_strates a
***                 NULL et nb_strates a 0.
***/

base_strate * init_base_strate()
{
base_strate * b ;

b = new(base_strate);
newerr(b, "init_base_strate : plus de memoire");

b->les_strates = NULL ;
b->nb_strates = 0 ;
return(b);
}
```

```
/**  fonction ajout_base_strate
***          but : ajouter une strate a une base_strate,
***          parametres : la base_strate, la strate a rajouter,
***          retourne : le pointeur sur la base_strate modifiee.
***
***          ATTENTION : La base et la strate a ajouter doivent deja exister physiquement.
***
***/

base_strate * ajout_base_strate(base_strate * b, strate * s)
{

  if (b->nb_strates == nbmaxstrate)
    exit(1);                                    /* plus de place pour rajouter une strate */
  else
    {
      b->les_strates = ajout_liste_ens(b->les_strates, s);
      b->nb_strates = b->nb_strates + 1;
      return(b);
    }
}



/**  fonction ajout_fin_base_strate
***          but : ajouter une strate a la fin d'une base_strate,
***          parametres : la base_strate, la strate a rajouter,
***          retourne : le pointeur sur la base_strate modifiee.
***
***          ATTENTION : La strate ajoutee doit deja exister physiquement.
***
***/

base_strate * ajout_fin_base_strate(base_strate * b, strate * s)
{
  if (b->nb_strates == nbmaxstrate)
    exit(1);                                    /* plus de place pour rajouter une strate */
  else
    {
      b->les_strates = ajout_fin_liste_ens(b->les_strates, s);
      b->nb_strates = b->nb_strates + 1;
      return(b);
    }
}



/**  procedure affich_base_strate
***          but : afficher une base_strate,
***          parametres : la base_strate.
***/

void affich_base_strate(base_strate * b)
{

if (vide_base_strate(b) == faux )
    {
        printf("La base de strates contient %d ensemble(s) : \n",b->nb_strates);
        affich_liste_ens(b->les_strates);
    }
else
        {
        printf("La base est vide \n");
        }
}
```

```
/**  fonction vide_base_strate
***        but : verifie si la base_strate est vide ou pas (une base_strate est vide
***                si elle n'a pas ete creee
***                ou si elle vient d'etre initialisee),
***        parametres : la base_strate,
***        retourne : la valeur vrai si la base_strate est vide, faux sinon.
***/

bool vide_base_strate(base_strate * b)
{
if (b == NULL)
        {
        return(1);
        }
else
        {
        if (b->nb_strates == 0)
                {
                return(1);
                }
        else
                {
                return(0);
                }
        }
}



/**  fonction suppr_base_strate
***        but : supprimer la premiere clause d'une base_strate,
***        parametres : la base_strate, l'adresse de la strate qui a ete supprimee,
***        retourne : la nouvelle base_strate modifiee et la strate supprimee.
***
***        ATTENTION : la strate n'est pas supprimee physiquement
***/

base_strate * suppr_base_strate(base_strate * b, strate ** ps)
{

if (vide_base_strate(b) == faux)
        {
        b->les_strates = suppr_liste_ens (b->les_strates,ps);
        b->nb_strates = b->nb_strates - 1 ;
        return(b);
        }
else
    {
    return(b);
    }
}



/**  fonction tete_base_strate
***        but : renvoyer la premiere strate d'une base_strate,
***        parametres : la base_strate,
***        retourne : la premiere strate de la base_strate.
***/

strate * tete_base_strate(base_strate * b)
{

if (vide_base_strate(b) == faux)
```

```
        {
        return(tete_liste_ens(b->les_strates));
        }
else
    {
    return(NULL);
    }
}



/**  fonction destruc_base_strate
***       but : detruire une base_strate, c'est-a-dire detruit la liste de
***             strates et remet le nb de strates a 0,
***       parametres : la base_strate,
***       retourne : la base_strate reinitialisee.
***
***       ATTENTION : les clauses de la base ne sont pas supprimees
***             physiquement
***/

base_strate * destruc_base_strate(base_strate * b)
{

if (vide_base_strate(b) == faux)
        {
        b->les_strates = destruc_liste_ens(b->les_strates);
        b->nb_strates = 0;
        }
return (b);
}



/**  fonction destruc_plus_base_strate
***       but : detruire une base_strate completement,
***           c'est-a-dire libere la place occupee par
***       cette base_strate,
***       parametres : la base_strate,
***       retourne : la base_strate a NULL.
***
***       ATTENTION : les clauses de la base ne sont pas supprimees
***             physiquement
***/

base_strate * destruc_plus_base_strate(base_strate * b)
{

b = destruc_base_strate(b);
dispose((base_strate *)b) ;
return (NULL);
}




/**  fonction destruc_compl_base_strate
***       but : detruire completement une base_strate,
***             c'est-a-dire detruit la liste de
***             strates, remet le nb de strates a 0,
***             et detruit les clauses de la base
***       parametres : la base_strate,
***       retourne : la base_strate reinitialisee.
***
***       ATTENTION : les clauses de la base sont supprimees
***             physiquement
***/
```

```
base_strate * destruc_compl_base_strate(base_strate * b)
{

if (vide_base_strate(b) == faux)
        {
        b->les_strates = destruc_compl_liste_ens(b->les_strates);
        b->nb_strates = 0;
        }
return (b);
}



/**  fonction destruc_compl_plus_base_strate
***         but : detruire une base_strate + les clauses completement,
***              c'est-a-dire libere la place occupee par
***         cette base_strate,
***         parametres : la base_strate,
***         retourne : la base_strate a NULL.
***
***         ATTENTION : les clauses de la base sont supprimees
***                  physiquement
***/

base_strate * destruc_compl_plus_base_strate(base_strate * b)
{

b = destruc_compl_base_strate(b);
dispose((base_strate *)b) ;
return (NULL);
}



/**  fonction recopie
***         but : recopie la base stratifiee dans une structure
***                  base (Thierry Castell) pour
**                  pouvoir utiliser son prouveur,
***         parametres : la base stratifiee au format mcl, les
***                  informations permettant la creation
***                  de la base par Thierry Castell (nombres
***                  maximum de symboles, des clauses,
***                  de litteraux par clause), la table des poids
***         retourne : la base au format Thierry Castell.
***
***         ATTENTION : les clauses de la base ne sont pas dupliquees
***                  physiquement
***/

base * recopie (base_strate * base_mcl, int nb_max_symb, int nb_max_cl, int nb_max_lit_cl, elem_tab_poids * t_b)
{
base * base_th ;
clause * c ;
liste_clause * lc ;
strate * s;
liste_strate * ls ;
int cpts, cptc ;

base_th = base_creer(nb_max_symb, nb_max_cl, nb_max_lit_cl);
ls = base_mcl->les_strates ;
for (cpts = 1; cpts <= base_mcl->nb_strates; cpts++)
        {
        s = ls->un_ens ;
        lc = s->les_clauses ;
        for (cptc = 1; cptc <= s->nb_clauses; cptc++)
                {
```

```
            c = lc→une_clause ;
            base_pajouter(base_th,c,t_b[cpts - 1].poids);
            lc = lc→clause_suiv ;
            }
      ls = ls→ens_suiv ;
      }
return(base_th);
}


/**  fonction duplicat_base_strate
***        but : duplique la base stratifiee,
***        parametres : la base stratifiee a dupliquer,
***        retourne : la base dupliquee.
***
***        ATTENTION : les clauses de la base ne sont pas dupliquees
***              physiquement
***/

base_strate * duplicat_base_strate (base_strate * b)
{
base_strate * baux ;

baux = init_base_strate() ;
baux→nb_strates = b→nb_strates;
baux→les_strates = duplicat_liste_ens (b→les_strates);
return(baux);
}



/**  fonction develop_base_strate
***        but : cree une liste de bases stratifiees composees
***              a partir de la base stratifiee donnee en
***              dupliquant cette base et en supprimant une
***              des formules de la derniere strate (on
***              obtiendra donc N bases stratifiees dans la liste,
***              N etant le nombre de formules de
***              la derniere strate de la base initiale),
***        parametres : la base stratifiee a developper,
***        retourne : la liste des bases obtenues par developpement.
***
***      ATTENTION = la base donnee en parametre est detruite !
***
***        ATTENTION : les clauses de la base ne sont pas dupliquees
***              physiquement
***/

liste_base_strate * develop_base_strate (base_strate * b, int n)
{
int cpt ;
liste_clause * c1;
liste_clause * c2;
strate * s ;
liste_strate * ls ;
base_strate * baux ;
liste_base_strate * l ;

l = init_liste_base_strate() ;
if (vide_base_strate (b) == faux)
      {
      ls = b→les_strates ;
      s = ls→un_ens ;
      c1 = s→les_clauses ;
      s→nb_clauses = s→nb_clauses - 1;
        /* suppression de la premiere clause */
      s→les_clauses = c1→clause_suiv;
```

```
        baux = duplicat_base_strate(b);
        l = ajout_liste_base_strate(l,baux);
        s→les_clauses = c1 ;
          /* suppression des autres clauses */
        for (cpt = 2; cpt ≤ n; cpt++)
                {
                c2 = c1→clause_suiv ;
                if (vide_liste_clause(c2) == faux)
                        {
                        c1→clause_suiv = c2→clause_suiv ;
                        baux = duplicat_base_strate(b);
                        l = ajout_liste_base_strate(l,baux);
                        c1→clause_suiv = c2 ;
                        }
                c1 = c1→clause_suiv ;
                }
        b = destruc_plus_base_strate(b);
        }
return(l);
}
```

```
/**  fonction ajout_liste_clause_a_base
***         but : ajouter les clauses d'une liste de clauses a une base stratifiee
***         parametres : la base stratifiee, la liste de clauses a ajouter,
***         retourne : la base stratifiee mise a jour.
***
***         ATTENTION : La liste de clauses ajoutee doit deja exister physiquement.
***/

base_strate * ajout_liste_clause_a_base (base_strate * b, liste_clause * lc)
{
strate * s ;

s = init_ens();
while (vide_liste_clause (lc) == faux)
        {
        s = ajout_ens (s, lc→une_clause);
        lc = lc→clause_suiv ;
        }
b = ajout_base_strate(b,s);
return (b) ;
}
```

```
/**  fonction base_plus_clause_consistante
***         but : verifier si une base stratifiee a laquelle
***                 on rajoute une clause est consistante,
***                 ATTENTION quand la clause rajoutee est
***                 la base vide, la fonction permet de savoir
***                 si la base est consistante ou pas,
***         parametres : la base stratifiee, la clause a ajouter,
***                 le nombre maximum de symboles
***                  ds la base, le nombre maximum de clauses ds la base,
***                 le nombre maximum de litteraux par clause dans la base,
***                         la table des poids
***         retourne : un booleen a faux si base + clause est consistante,
***                 a vrai si inconsistante.
***/

bool base_plus_clause_consistante(base_strate * b, clause * Phi, int nb_max_symb,
                        int nb_max_cl, int nb_max_lit_cl, elem_tab_poids * t_b)
```

```
{
base_strate * b0 ;
base * base_th ;
clause * c ;
bool res ;
rapport * r ;
rapport * rp ;

if (clause_vide(Phi) == faux)
        {
          /* rajout de Phi a la strate la plus prioritaire */
        b0 = duplicat_base_strate(b) ;
        (b0->les_strates)->un_ens = ajout_ens ((b0->les_strates)->un_ens, Phi);
        base_th = recopie(b0,
                          nb_max_symb + clause_longueur(Phi),
                          nb_max_cl + 1,
                          MAX(nb_max_lit_cl,clause_longueur(Phi)),t_b);
        r = rapport_creer();
        rp = rapport_creer();
        res=!(prouveur(nom_prouveur,base_th,rp,r));
        base_free(base_th);
        rapport_free(r);
        rapport_free(rp);
        b0 = destruc_plus_base_strate(b0);
        }
else
        {
        base_th = recopie(b,nb_max_symb,nb_max_cl,nb_max_lit_cl,t_b);
        r = rapport_creer();
        rp = rapport_creer();
        res =!( prouveur (nom_prouveur,base_th, rp,r));
        base_free(base_th);
        rapport_free(r);
        rapport_free(rp);
        }
return(res);
}




/**  fonction egal_base_strate
***        but : compare deux bases stratifiees,
***        parametres : les deux bases a comparer,
***        retourne : le booleen resultat (vrai si les deux
***                  bases sont egales, faux sinon).
***/

bool egal_base_strate (base_strate * b1, base_strate * b2)
{
bool res ;
liste_ens * ls1;
liste_ens * ls2 ;

if (b1->nb_strates != b2->nb_strates)
        {
        res = faux ;
        }
else
        {
        res = vrai ;
        ls1 = b1->les_strates ;
        ls2 = b2->les_strates ;
        while ((vide_liste_ens(ls1) == faux) && (res== vrai))
                {
                if (egal_ens(ls1->un_ens, ls2->un_ens) == vrai)
                        {
```

```
                        ls1 = ls1→ens_suiv;
                        ls2= ls2→ens_suiv;
                        }
                else
                        {
                        res = faux ;
                        }
                }
        }
return(res);
}



/************************************************************/


/**   fonction init_liste_base_strate
***         but : initialiser une variable de type liste_base_strate avec la valeur NULL,
***         parametres : neant,
***         retourne : la liste_base_strate a NULL.
***/

liste_base_strate * init_liste_base_strate()
{
return(NULL);
}


/**   fonction ajout_liste_base_strate
***         but : ajouter une base_strate a une liste de base_strate,
***         parametres : la liste de base_strate, la base_strate a rajouter,
***         retourne : le pointeur sur la liste de base_strate modifiee.
***
***         ATTENTION : La base ajoutee doit deja exister physiquement.
***
***/

liste_base_strate * ajout_liste_base_strate(liste_base_strate * lb, base_strate * b)
{
liste_base_strate * lbaux ;

lbaux = lb ;
lb = new(liste_base_strate);
newerr(lb, "ajout_liste_base_strate : plus de memoire");

lb→une_base = b ;
lb→base_suiv = lbaux ;
return(lb);
}



/**   procedure affich_liste_base_strate
***         but : afficher une liste de base_strate,
***         parametres : la liste de base_strate.
***/

void affich_liste_base_strate(liste_base_strate * lb)
{
liste_base_strate * lbaux ;
int cpt ;

if (vide_liste_base_strate(lb) == faux)
        {
        lbaux = lb ;
```

```
        cpt = 1 ;
        while (lbaux ≠ NULL)
                {
                printf("base numero %d = ",cpt);
                affich_base_strate(lbaux→une_base);
                printf("\n");
                lbaux = lbaux→base_suiv;
                cpt = cpt + 1 ;
                }
        }
else
        {
        printf("La liste de bases stratifiees est vide \n");
        }
}



/**  fonction vide_liste_base_strate
***       but : verifie si la liste_base_strate est vide ou pas,
***       parametres : la liste de base_strate,
***       retourne : la valeur vrai si la liste_base_strate est vide, faux sinon.
***/

bool vide_liste_base_strate(liste_base_strate * lb)
{
if (lb == NULL)
        {
        return(1);
        }
else
        {
        return(0);
        }
}



/**  fonction suppr_liste_base_strate
***       but : supprimer la premiere base_strate
***               d'une liste de base_strate,
***       parametres : la liste de base_strate, l'adresse de
***               la base_strate qui a ete supprimee,
***       retourne : la nouvelle liste de base_strate modifiee
***               et la base_strate supprimee.
***
***       ATTENTION : la base n'est pas supprimee physiquement
***/

liste_base_strate * suppr_liste_base_strate(liste_base_strate * lb, base_strate ** pb)
{
liste_base_strate * lbaux ;

if (vide_liste_base_strate(lb) == faux)
        {
        *pb = lb→une_base ;
        lbaux = lb→base_suiv ;

        dispose((liste_base_strate *)lb) ;
        return(lbaux);
        }
else
    {
    return(lb);
    }
}
```

```
/**  fonction tete_liste_base_strate
***        but : renvoyer la premiere base_strate d'une liste de base_strate,
***        parametres : la liste de base_strate,
***        retourne : la premiere base_strate de la liste de base_strate.
***/

base_strate * tete_liste_base_strate(liste_base_strate * lb)
{

if (vide_liste_base_strate(lb) == faux)
        {
        return(lb->une_base);
        }
else
    {
    return(NULL);
    }
}



/**  fonction destruc_liste_base_strate
***        but : detruire une liste de base_strate,
***        parametres : la liste de base_strate,
***        retourne : la liste de base_strate a NULL.
***
***        ATTENTION : les clauses de la base ne sont pas supprimees
***                physiquement
***/

liste_base_strate * destruc_liste_base_strate(liste_base_strate * lb)
{
liste_base_strate * lbaux ;
base_strate * b ;

while (vide_liste_base_strate(lb) == faux)
        {
        lbaux = lb->base_suiv ;
        b = lb->une_base;
        b = destruc_plus_base_strate(b);
        dispose((liste_base_strate *)lb) ;
        lb = lbaux ;
        }
return (lb);
}




/**  fonction destruc_compl_liste_base_strate
***        but : detruire une liste de base_strate + clauses,
***        parametres : la liste de base_strate,
***        retourne : la liste de base_strate a NULL.
***
***        ATTENTION : les clauses de la base sont supprimees
***                physiquement
***/

liste_base_strate * destruc_compl_liste_base_strate(liste_base_strate * lb)
{
liste_base_strate * lbaux ;
base_strate * b ;
```

```
while (vide_liste_base_strate(lb) == faux)
          {
          lbaux = lb→base_suiv ;
          b = lb→une_base;
          b = destruc_compl_plus_base_strate(b);
          dispose((liste_base_strate *)lb) ;
          lb = lbaux ;
          }
return (lb);
}




/**  fonction concat_liste_base_strate
***          but : concatene deux listes de bases stratifiees,
***          parametres : les deux listes de bases stratifiees,
***          retourne : la liste resultat,
***          particularite : la liste resultat occupe physiquement
***                    l'emplacement des deux listes en entree.
***/

liste_base_strate * concat_liste_base_strate (liste_base_strate * l1, liste_base_strate * l2)
{
liste_base_strate * laux;
liste_base_strate * prec ;

if (vide_liste_base_strate(l1) == faux)
          {
          laux = l1 ;
          while (vide_liste_base_strate(l1) == faux)
                    {
                    prec = l1 ;
                    l1 = l1→base_suiv ;
                    }
          prec→base_suiv = l2 ;
          return(laux);
          }
else
          {
          return(l2);
          }
}




/**  fonction intro_strate_ds_liste
***          but : introduit une strate ds chacune des
***                    bases de la liste de bases stratifiees,
***          parametres : la liste de bases stratifiees, la strate a introduire,
***          retourne : la liste resultat.
***/

liste_base_strate * intro_strate_ds_liste (liste_base_strate * l, strate * s)
{
liste_base_strate *laux ;
base_strate * b ;
strate * saux ;

b = tete_liste_base_strate(l);
if (vide_base_strate(b)==faux)
          {
          laux = l ;
          while (vide_liste_base_strate (laux) == faux)
                    {
```

```
                    saux = duplicat_ens(s);
                    laux→une_base = ajout_base_strate(laux→une_base, saux);
                    laux = laux →base_suiv ;
                    }
          }
else
          {
          l = suppr_liste_base_strate(l,&b);
          saux = duplicat_ens(s);
          b = ajout_base_strate(b, saux);
          l = ajout_liste_base_strate(l,b);
          }
return(l) ;
}
```

```
/**  fonction base_appartient_a_liste
***         but : verifie si une base stratifiee appartient
***                   a une liste de bases stratifiees,
***         parametres : la base et la liste de bases,
***         retourne : le booleen resultat (vrai si la base
***                   appartient a la liste, faux sinon).
***/

bool base_appartient_a_liste (base_strate * b, liste_base_strate * lb)
{
bool res ;

res = faux ;
while ((vide_liste_base_strate(lb) == faux) && (res== faux))
          {
          if (egal_base_strate(b, lb→une_base) == vrai)
                    {
                    res = vrai ;
                    }
          else
                    {
                    lb = lb→base_suiv ;
                    }
          }
return(res);
}
```

Chapitre 2

Programmes pour Uni-Lex utilisant les BDD

Dans cette annexe, est donné le fichier source en langage C de l'implémentation utilisant les BDD. Il s'agit d'une liste de fonctions et procédures destinées à compléter le "package" de [BRB90].

```
/*****************************************************
  Nom du fichier = kbdd_extra.c

   But = completer le package cmubdd avec differentes procedures et fonctions
   destinees a prendre en compte les preferences dans une base de donnees.

   Contient les procedures et fonctions suivantes =
   - double delta_tps()
   - int get_formula_priority(string * aof_line, bool end_ok,string *nf)
   - int get_command(string *aof_line, bool end_ok)
   - my_own_bdd_print_fcn(FILE *bdd_fp2,bdd_formula f, int is_str)
   - void mybdd_print(bdd_formula f)
   - int  min(int x1,int x2)
   - ptr_sup_node my_scan_fc(int ind, ptr_sup_node*  thenp,
                             int  then_phase, ptr_sup_node* elsep,
                             int else_phase, bdd_manager bddm, int n)
   - unsigned int my_bdd_ptr_hash(char *p1,unsigned n)
   - bool my_bdd_ptr_equ(char *p1, char *p2)
   - static void my_scan_below(bdd_manager bddm,bdd_dagptr d,int (* fcn)() , int n)
   - static void my_unscan_fc(bdd_dagptr * ptr_d, ptr_sup_node pp, bdd_manager bddm)
   - void my_dispose_hash(hash_record_ptr h,VOID (*remove_data)(),bdd_manager bddm)
   - void my_bdd_dagptr_unscan(bdd_manager bddm)
   - void my_bdd_unscan(bdd_manager bddm)
   - void my_bdd_dagptr_scan(bdd_manager bddm, int (*fcn)(), buffer *d_buf,int n)
   - void my_bdd_scan(bdd_manager bddm, int (*fcn)(), int n)
   - int shortest(bdd_formula f,int n)
   - bdd_formula fatfree (bdd_formula f, int n, int com)

   Rajout des commandes kbbd suivantes (avec les fonctions associees) =
   - mybool => static bool domyboolean(string line)
   - mybdd => static bool domybdd(string line)
   - myinit => static bool doinit_tab_prior ()
   - mysearch => static bool dopluscourt(string line)
   - myff => static bool dofatfree(string line)
   - mysize => static bool domysize(string line)
   - mynet => static bool donettoyage (string line)

   Fait un include des fichiers suivants =
   - stdio.h
   - limits.h
   - bdd.h (issu du package cmubdd de Bryant)
   - sys/types.h
   - sys/times.h
   - time.h

   Les variables, constantes, macro, procedures ou fonctions suivantes sont
        redeclarees ici pour permettre la compilation et l'edition de lien =
   - #define true 1
   - #define false 0
   - bdd_formula get_formula();
   - bdd_formula get_formula_var();
   - static bool set_formula_name();
   - bdd_formula fetch_operand();
   - string get_formula_name();
   - void kbdd_set_name();
   - #define BDD_INDEX_INDEX(bddm,indexindex)
   - #define BDD_DAGPTR_INDEX_INDEX(bddm,indexindex)
   - hash_record print_table;
   - int print_id;
   - static int numvars;
   - hash_record kbdd_names;
   - bool storing_names;

   Les nouveaux types,  variables, constantes et macros sont les suivants =
   - #define max_prior 100
   - #define et &&
   - #define ou ||
```

```
  - #define new(x) (x *)malloc(sizeof(x))
  - #define newerr(p,x) if (p==NULL) {printf(x); printf("\n"); exit(1);}
  - #define dispose(x) free(x)
  - static int tab[max_prior+1] ;
  - typedef struct {
                    int ch_plcourt[2];
                   bdd_formula deg[2];
                  } sup_node ;
  - typedef sup_node * ptr_sup_node ;
  ************************************************************/

#include <stdio.h>
#include <limits.h>
#define NO_BDD_DAGPTR
#include "bdd.h"
#include <sys/types.h>
#include <sys/times.h>
#include <time.h>

extern bdd_manager bddm;


/******* Declarations of useful functions and variables ****/
#define true 1
#define false 0

#ifndef CLK_TCK
#define CLK_TCK 60
#endif

bdd_formula get_formula();
bdd_formula get_formula_var();
bdd_formula fetch_operand();
string get_formula_name();
void kbdd_set_name();

#if ERR_CHECK
#define BDD_INDEX_INDEX(bddm,indexindex) \
    (*(bdd_formula*)M_LOCATE_BUF(&((bddm)->varlist_buf), indexindex))
#define BDD_DAGPTR_INDEX_INDEX(bddm,indexindex) \
    (*(bdd_dagptr*)M_LOCATE_BUF(&((bddm)->varlist_dagptr_buf), indexindex))
#else
#define BDD_INDEX_INDEX(bddm,indexindex) \
    (*(bdd_formula*)FAST_LOC_BUF(&((bddm)->varlist_buf), indexindex))
#define BDD_DAGPTR_INDEX_INDEX(bddm,indexindex) \
    (*(bdd_dagptr*)FAST_LOC_BUF(&((bddm)->varlist_dagptr_buf), indexindex))
#endif


hash_record print_table;
int print_id;
static int numvars;
hash_record kbdd_names;
bool storing_names;
extern bool quiet_switch;

static bool set_formula_name(f, f_name)
      bdd_formula f;
      string f_name;
{
  bdd_formula tf;
  if (tf = fetch_operand(f_name)) {
    if (bdd_terminal(tf) == BDD_POSVAR) {
      fprintf(stderr, "Destination %s is an input variable\n", f_name);
      return(FALSE);
    }
    fprintf(stderr, "warning: freeing old formula %s\n", f_name);
```

```
    IGNORE bdd_free(tf);
  }
  kbdd_set_name(f, f_name);
  return(TRUE);
}


/******* Newly added types, constants, variables and macro ******/
#define max_prior 100                    /* 100 = nombre maximum de variables ds le BDD */
#define et &&
#define ou ||

#define new(x) (x *)malloc(sizeof(x))
#define newerr(p,x) if (p==NULL) {printf(x); printf("\n"); exit(1);}
#define dispose(x) free(x)

/* table des priorites : une priorite est
                             un nombre entier non signe,
                             l'indice dans le tableau correspond a
                             l'index des variables.
                             Attention !! l'index 0 n'est jamais atteint */
static unsigned int tab[max_prior+1] ;

/* structure d'aide pour la recherche du
plus court chemin et le degraissage */
typedef struct {
 /* table de 2 elements: la case 0 indique la valeur
                             du plus court chemin vers le noeud 0,
                             la case 1 indique la valeur du plus court
                             chemin vers le noeud 1 */
 int ch_plcourt[2];
/* table de 2 elements: la case 0 indique le bdd
                             degraisse vers le noeud 0,
                             la case 1 indique le bdd degraisse
                             vers le noeud 1 */ bdd_formula deg[2];
} sup_node ;

typedef sup_node * ptr_sup_node ;                    /* pointeur sur la structure d'aide */


/******** Newly added commands **********/
double delta_tps()
/* Fonction delta_tps :
   - but = renvoyer le delta de temps entre deux appels successifs;
   - pas de parametre ;
   - renvoie = un double representant le delta de temps en secondes;
   - est utilisee ds = fatfree*/
{
  static struct tms temps ;           /* variable temps pour les mesures de temps d'execution */
  static clock_t t1 ;                                          /* mesure du temps precedent */
  int err ;                             /* entier recuperant l'erreur lors de l'appel de times */
  clock_t t2 ;                                                   /* mesure du temps courant */
  double res ;

  err = times(&temps);
  t2 = temps.tms_utime;
  res = (double)(t2-t1)/CLK_TCK ;
  t1 = t2 ;
  return (res) ;
}


int get_formula_priority(string *aof_line, bool end_ok,string *nf)
/* Fonction get_formula_priority :
   - but = renvoyer la priorite tapee par l'utilisateur pour une variable donnee;
   - 3 parametres = la ligne tapee par l'utilisateur, un booleen de controle,
```

```
        le nom de la variable;
    - renvoie = un entier representant la priorite;
    - est utilisee ds = domyboolean*/
{
  string name;
  int num ;

  name = gettoken(aof_line, "");
  num = atoi(name) ;
  if (*name == NULLSTR) {
    if (!end_ok)
         fprintf(stderr, "expecting a priority ( number >= 0) \n");
   fprintf(stderr, "pas de priorite pour la variable %s, donc priorite forcee a zero \n",*nf);
   return(0);
   }
  if (num < 0) {
   fprintf(stderr, "priorite %d ignoree, donc forcee a zero \n", num);
   num = 0 ;
   }
  skipspace(aof_line);
  return(num);
}


int get_command(string *aof_line, bool end_ok)
/* Fonction get_command :
    - but = renvoyer la commande tapee par l'utilisateur sous la forme d'un entier;
    - 2 parametres = la ligne tapee par l'utilisateur, un booleen de controle;
    - renvoie = un entier representant la commande;
    - est utilisee ds = dofatfree*/
{
  string name;
  int num ;

  name = gettoken(aof_line, "");
  num = atoi(name) ;
  if (*name == NULLSTR) {
    if (!end_ok)
         fprintf(stderr, "expecting an integer ( number > 0) \n");
   fprintf(stderr, "pas de commande, donc commande forcee a 1 \n");
   return(1);
   }
  if (num <= 0) {
   fprintf(stderr, "commande %d ignoree, donc forcee a 1 \n", num);
   num = 1 ;
   }
  skipspace(aof_line);
  return(num);
}



void my_own_bdd_print_fcn(FILE *bdd_fp2, bdd_formula f, int is_str)
/* Procedure my_own_bdd_print_fcn :
    - but = changer le format de l'affichage pour faire apparaitre la
        prise en compte des priorites;
    - 3 parametres =le fichier dans lequel imprimer, la formule en
        cours d'impression, un entier non utilise;
    - est utilisee ds = mybdd_print, domybdd*/
{
    char *s;
    unsigned int ind ;
    int priority ;

   s = bdd_name_of(bdd_if(f));
    ind = f->bdd->indexindex;
    priority = tab[ind];
```

```
    (void) fprintf(bdd_fp2,"%s:%d (index=%d, priorite=%d)", s, BDD_ABSPTR(f→bdd), ind, priority);
}


void mybdd_print(bdd_formula f)
/* Procedure mybdd_print :
    - but = imprimer une formule;
    - 1 parametre = la formule a imprimer;
    - est utilisee ds =  les tests;
    - utilise = my_own_bdd_print_fcn*/
{
  void dag_fcn();
  unsigned int int_hash();
  bool int_equ();

  if (f ≠ NULL) {
    create_hash(&print_table, 100, int_hash, int_equ);
    print_id = 0;
    bdd_print(stdout,f, BDD_DAG, my_own_bdd_print_fcn, 1);
    dispose_hash(&print_table, BDD_NULL_FCN);
  }
  else
    printf(" BDD vide\n");
}



int min(int x1,int x2)
/* Fonction min :
    - but = calculer le min de deux valeurs;
    - 2 parametres = les deux valeurs;
    - renvoie = le min des deux valeurs;
    - est utilisee ds = my_scan_fc (pour le calcul du plus court chemin); */
{
  if (x1 ≤ x2)
    return(x1);
  else
    return(x2);
}


ptr_sup_node my_scan_fc(int ind, ptr_sup_node* thenp,int then_phase,
                         ptr_sup_node* elsep, int else_phase,
                         bdd_manager bddm, int n)
/* Fonction my_scan_fc :
    - but =  pour un noeud donne du bdd, calcule le plus court chemin
          passant par ce noeud et allant vers 0 ou vers 1, et le bdd
          degraisse partant de ce noeud vers 0 ou vers 1;
    - 7 parametres =l'index du noeud courant, le resultat de l'etude
          sur le fils then et sa phase, le resultat de l'etude sur le fils
          else et sa phase, le bdd manager de la formule en cours de
          traitement et l'entier indiquant vers quel noeud on se dirige;
    - renvoie = le resultat de l'etude du noeud courant (pointeur sur
           une structure de type sup_node);
    - est utilisee ds = my_scan_below, my_bdd_dagptr_scan, my_bdd_scan ; */
    {
      ptr_sup_node paux ;
      int n1, n2 ;
      bdd_formula ffg ;                              /* version degraissee du fils then */
      bdd_formula ffh ;                              /* version degraissee du fils else */
      bdd_formula f ;
      int spg ;                                 /* cout du plus court chemin du fils then */
      int sph ;                                 /* cout du plus court chemin du fils else */

      paux = new(sup_node);
      if ((elsep == NULL) et (thenp == NULL)) {                              /* noeud 1 */
        /* plus court chemin */
```

```
      paux→ch_plcourt[0] = INT_MAX ;
      paux→ch_plcourt[1] = 0 ;
      /* degraissage */
      paux→deg[0] = BDD_ONE(bddm) ;
      paux→deg[1] = BDD_ONE(bddm) ;
    }
    else {                                                        /* autre noeud */
      /* plus court chemin */
      n1 = (else_phase == 1) ? (1-n) : (n) ;
      n2 = (then_phase == 1) ? (1-n) : (n) ;
      spg = (*thenp)→ch_plcourt[n2];
      sph = tab[ind+1] + (*elsep)→ch_plcourt[n1];
      paux→ch_plcourt[n] = min (spg,sph) ;
      /* degraissage */
      ffg = (then_phase ==1) ? (bdd_not((*thenp)→deg[1-n])) : ((*thenp)→deg[n]) ;
      ffh = (else_phase ==1) ? (bdd_not((*elsep)→deg[1-n])) : ((*elsep)→deg[n]) ;
      if (paux→ch_plcourt[n] < spg)
        ffg = (n==0) ? BDD_ONE(bddm) : BDD_ZERO(bddm);
      if (paux→ch_plcourt[n] < sph)
        ffh = (n==0) ? BDD_ONE(bddm) : BDD_ZERO(bddm);
      f = BDD_INDEX_INDEX(bddm,ind+1);
      paux→deg[n] = bdd_ite(f, ffg, ffh);
    }
    return(paux);
  }


ptr_sup_node my_scan_fc2(int ind, ptr_sup_node* thenp,int then_phase,
                         ptr_sup_node* elsep, int else_phase,
                         bdd_manager bddm, int n)
/* Fonction my_scan_fc2 :
   - but =  pour un noeud donne du bdd, calcule le plus court chemin
        passant par ce noeud et allant vers 0 ou vers 1, et le bdd
        degraisse partant de ce noeud vers 0 ou vers 1, tout en
        faisant la quantification existentielle des variables
        hypotheses;
   - 7 parametres =l'index du noeud courant, le resultat de l'etude
        sur le fils then et sa phase, le resultat de l'etude sur le fils
        else et sa phase, le bdd manager de la formule en cours de
        traitement et l'entier indiquant vers quel noeud on se dirige;
   - renvoie = le resultat de l'etude du noeud courant (pointeur sur
        une structure de type sup_node);
   - est utilisee ds = my_scan_below, my_bdd_dagptr_scan, my_bdd_scan  ; */
  {
    ptr_sup_node paux ;
    int n1, n2 ;
    bdd_formula ffg ;                              /* version degraissee du fils then */
    bdd_formula ffh ;                              /* version degraissee du fils else */
    bdd_formula f ;
    int spg ;                                 /* cout du plus court chemin du fils then */
    int sph ;                                 /* cout du plus court chemin du fils else */

    paux = new(sup_node);
    if ((elsep == NULL) et (thenp == NULL)) {                          /* noeud 1 */
      /* plus court chemin */
      paux→ch_plcourt[0] = INT_MAX ;
      paux→ch_plcourt[1] = 0 ;
      /* degraissage */
      paux→deg[0] = BDD_ONE(bddm) ;
      paux→deg[1] = BDD_ONE(bddm) ;
    }
    else {                                                        /* autre noeud */
      /* plus court chemin */
      n1 = (else_phase == 1) ? (1-n) : (n) ;
      n2 = (then_phase == 1) ? (1-n) : (n) ;
      spg = (*thenp)→ch_plcourt[n2];
      sph = tab[ind+1] + (*elsep)→ch_plcourt[n1];
```

```
        paux→ch_plcourt[n] = min (spg,sph) ;
        /* degraissage + quantification existentielle des variables hypotheses */
        if (tab[ind+1] == 0) {                          /* ce n'est pas une variable hypothese */
          ffg = (then_phase ==1) ? (bdd_not((*thenp)→deg[1-n])) : ((*thenp)→deg[n]) ;
          ffh = (else_phase ==1) ? (bdd_not((*elsep)→deg[1-n])) : ((*elsep)→deg[n]) ;
          if (paux→ch_plcourt[n] < spg)
            ffg = (n==0) ? BDD_ONE(bddm) : BDD_ZERO(bddm);
          if (paux→ch_plcourt[n] < sph)
            ffh = (n==0) ? BDD_ONE(bddm) : BDD_ZERO(bddm);
          f = BDD_INDEX_INDEX(bddm,ind+1);
          paux→deg[n] = bdd_ite(f, ffg, ffh);
        }
        else {                                          /* c'est une variable hypothese */
          ffh = (else_phase ==1) ? (bdd_not((*elsep)→deg[1-n])) : ((*elsep)→deg[n]) ;
          if (paux→ch_plcourt[n] < sph)
            ffh = (n==0) ? BDD_ONE(bddm) : BDD_ZERO(bddm);
          paux→deg[n] = ffh;
        }
      }
      return(paux);
    }


unsigned int my_bdd_ptr_hash(char *p1,unsigned n)
/* Fonction my_bdd_ptr_hash :
   voir  la fonction bdd_ptr_hash dans bdd.c
   (redefinie ici a l'identique pour des besoins de compilation) */
{
    return (((unsigned)p1)%n);
}

bool my_bdd_ptr_equ(char *p1,char *p2)
/* Fonction my_bdd_ptr_equ :
   voir la fonction bdd_ptr_equ dans bdd.c
   (redefinie ici a l'identique pour des besoins de compilation) */
{
    return (p1==p2);
}


static void my_scan_below(bddm,d,fcn,n)
 bdd_manager bddm;
 bdd_dagptr d;
 int (*fcn)();
 int n ;
/* Procedure my_scan_below :
   voir dans bdd.c
   (redefinie ici
            - pour rajouter 1 parametre : int n et sa prise en compte dans le code,
            - pour passer  bddm lors de l'appel de la fonction my_scan_fc)
   utilise =  my_scan_fc */
{
    int *thenp, *elsep;
    int then_phase, else_phase, value;

    if (BDD_IS_COMPTR(d))                             /* code rajoute % a scan_below */
      n = 1 - n ;                      /* permet de mettre a jour l'entier a transmettre */

    d = BDD_ABSPTR(d);
    if (BDD_TESTMARK(d)) {
        return;
    }
    BDD_SETMARK(d);
    if (BDD_IS_CONST(d)) {
        /* rajout des par. bddm et n a l'appel de fcn */
        value = fcn(GETINDEX(bddm,d), 0, 0,0,0,bddm,n);
        insert_hash(&(bddm)→scan_table, (char*)d, (char*)value);
```

```
        return;
    }
    /* chgt nom fc + rajout de n a l'appel de my_scan_below */
    my_scan_below(bddm,d->Then,fcn,n);
    thenp = (int*)locate_hash(&(bddm)->scan_table, (char*)BDD_ABSPTR(d->Then));
    /* chgt nom fc + rajout de n a l'appel de my_scan_below */
    my_scan_below(bddm,d->Else,fcn,n);
    elsep = (int*)locate_hash(&(bddm)->scan_table, (char*)BDD_ABSPTR(d->Else));
    then_phase = BDD_IS_COMPTR(d->Then);
    else_phase = BDD_IS_COMPTR(d->Else);
                                        /* rajout de bddm et n a l'appel de fcn */
    value = (*fcn)(GETINDEX(bddm,d), thenp, then_phase, elsep, else_phase,bddm,n);
    insert_hash(&(bddm)->scan_table, (char*)d, (char*)value);
}


static void my_unscan_fc(bdd_dagptr * ptr_d, ptr_sup_node pp)
/* Procedure my_unscan_fc :
   - but =  pour un noeud donne du bdd, supprime les informations creees par le scan;
   - 2 parametres =le dagptr du noeud courant, la structure d'aide du
        scan du noeud courant;
   - est utilisee ds = my_bdd_dagptr_unscan; */
{
  dispose(pp);
}


void my_bdd_dagptr_unscan(bdd_manager bddm)
/* Procedure my_bdd_dagptr_unscan :
   voir dans bdd.c
   (redefinie ici  pour  l'appel de la fonction my_unscan_fc)
   utilise =  my_unscan_fc
   est utilisee ds = my_bdd_unscan, my_bdd_dagptr_scan */
{
    if (bddm->scan_on) {
        dispose_hash(&bddm->scan_table,my_unscan_fc);
        bddm->scan_on = FALSE;
    }
}


void my_bdd_unscan(bdd_manager bddm)
/* Procedure my_bdd_unscan :
   voir dans bdd.c
   (redefinie ici  pour  l'appel de la fonction my_bdd_dagptr_unscan)
   utilise =  my_bdd_dagptr_unscan
   est utilisee ds = donettoyage */
{
    my_bdd_dagptr_unscan(bddm);
}


void my_bdd_dagptr_scan(bddm, fcn, d_buf,n)
 bdd_manager bddm;
 int (*fcn)();
 buffer *d_buf;
 int n ;
/* Procedure my_bdd_dagptr_scan :
   voir dans bdd.c
   (redefinie ici
            - pour rajouter 1 parametre : int n et son passage lors de
                l'appel de la fonction my_scan_below,
            - changement du nom des procedure fonctions (rajout de my_))
   utilise =  my_scan_below ;
   est utilisee par = my_bdd_scan */
/*
 *  Apply the user function fcn to each node in the bdd.
```

```
 *  Meant to substitute for a recursive scan over a bdd, starting
 *  at the root nodes in d_buf.
 */
{
    int num_bdd_nodes,i;
    bdd_dagptr d;

    if (bddm->scan_on) {
        my_bdd_dagptr_unscan(bddm);                                /* chgt nom fc */
    }
    bddm->scan_on = TRUE;
    num_bdd_nodes = bddm->unique_table->num_entries;
    if (num_bdd_nodes<20) {
        num_bdd_nodes = 20;
    }
    create_hash(&bddm->scan_table, (unsigned) num_bdd_nodes/5,
                my_bdd_ptr_hash, my_bdd_ptr_equ);
   for (i=0; i<COUNT_BUF(d_buf); i++) {
        d = *(bdd_dagptr*)FAST_LOC_BUF(d_buf,i);
        my_scan_below(bddm,d,fcn,n);            /* chgt nom fc + passage de n en parametre */
    }
    for (i=0; i<COUNT_BUF(d_buf); i++) {
        d = *(bdd_dagptr*)FAST_LOC_BUF(d_buf,i);
        bdd_repairmark(d);
    }
}

void my_bdd_scan(bddm, fcn,n)
 bdd_manager bddm;
 int (*fcn)();
 int n ;
/* Procedure my_bdd_scan :
   voir dans bdd.c
   (redefinie ici
            - pour rajouter 1 parametre : int n et son passage lors de
                l'appel de la fonction my_bdd_dagptr_scan,
            - changement du nom des procedure fonctions (rajout de my_))
   utilise =  my_bdd_dagptr_scan ;
   est utilisee par = shortest, fatfree */
{
    buffer dbuf;
    bdd_formula f;
    bdd_dagptr d;
    int i;

    new_buf(&dbuf,0,sizeof(bdd_dagptr));
    resize_buf(&dbuf,0);
    for (i=0; i<COUNT_BUF(&bddm->po_buf); i++) {
        f = *(bdd_formula*)FAST_LOC_BUF(&bddm->po_buf,i);
        d = f->bdd;
        push_buf(&dbuf,(char*)&d);
    }
    /* chgt nom fc + passage de n en parametre */
    my_bdd_dagptr_scan(bddm, fcn, &dbuf,n);
    free_buf(&dbuf);
}


int shortest(bdd_formula f,int n)
/* Fonction shortest :
   - but =  calculer la valeur du plus court chemin vers 0 ou vers 1 dans une formule;
   - 2 parametres =  la formule, le noeud vers lequel on cherche a aller ;
   - renvoie = la longueur du plus court chemin ds la formule;
   - est utilisee ds = dopluscourt ;
   - utilise = my_bdd_scan ; */
{
    ptr_sup_node* ptr_res ;
```

```
    bdd_declare_po(f);
    my_bdd_scan(f→bdm, my_scan_fc,n);
    ptr_res = bdd_read(f);
    n = (BDD_IS_COMPTR(f→bdd))? (1-n) : n;
    return ((*ptr_res)→ch_plcourt[n]);
}


bdd_formula fatfree (bdd_formula f, int n, int com)
/* Fonction fatfree :
   - but =  calculer le bdd degraisse correspondant a la formule (vers 0 ou vers 1)
   et suivant la commande
   faire ou pas  la quantification existentielle des variables hypotheses ;
   - 3 parametres =  la formule, le noeud vers lequel on cherche a aller, la commande
   (1 pour degraissage simple, sinon  degraissage + quantification) ;
   - renvoie = le bdd degraisse de la formule;
   - est utilisee ds = dofatfree ;
   - utilise = my_bdd_scan ; */
{
    ptr_sup_node* ptr_res ;
    bdd_formula f_aux ;

    bdd_declare_po(f);
    if (com == 1)
      my_bdd_scan(f→bdm, my_scan_fc,n);
    else
      my_bdd_scan(f→bdm, my_scan_fc2,n);
    ptr_res = bdd_read(f);
    if (BDD_IS_COMPTR(f→bdd))
      f_aux = bdd_not((*ptr_res)→deg[1-n]) ;
    else
      f_aux =(*ptr_res)→deg[n];
    return(f_aux);
}



/****** Extending the KBDD command set ******/

static bool domyboolean(string line)
/* Fonction domyboolean :
   - but =  saisie de variables et des priorites associees a chaque variable;
   - 1 parametre =  la ligne tapee par l'utilisateur ;
   - renvoie = un booleen (faux si pb ds la saisie de la commande,
        a vrai si execution ok);
   - utilise = my_boolean ;
   - correspond a la commande = mybool  !!! */
{
  string name;
  int num ;

 num = 0 ;
   while (name = get_formula_name(&line, true)) {
        bdd_formula f;
        name = strsave(name);
        num = get_formula_priority(&line,true,&name) ;
        f = bdd_create_var_after(bddm, name, (bdd_formula) 0);
        if (!f) {
              fprintf(stderr, "ignoring already declared name %s\n", name);
        } else {
              if (storing_names) {
                      insert_check_hash(&kbdd_names, name, name);
                      }
              tab[f→bdd→indexindex] = num ;
```

```
            numvars++;
      }
   }
   return(true);
}


static bool domybdd(string line)
/* Fonction domybdd :
     - but = impression du bdd correspondant a la formule saisie;
     - 1 parametre = la ligne tapee par l'utilisateur ;
     - renvoie = un booleen (faux si pb ds la saisie de la commande,
            a vrai si execution ok);
     - utilise = my_bdd ;
     - correspond a la commande = mybdd !!! */
{
   bdd_formula f;
   void dag_fcn();
   unsigned int int_hash();
   bool int_equ();

   if (!(f = get_formula(&line, false)))
      return(false);
   create_hash(&print_table, 100, int_hash, int_equ);
   print_id = 0;
   bdd_print(stdout,f, BDD_DAG, my_own_bdd_print_fcn, 1);
   dispose_hash(&print_table, BDD_NULL_FCN);
   return(true);
}


static bool doinit_tab_prior ()
/* Fonction domyboolean :
     - but = initialise la table des priorites;
     - 1 parametre = la ligne tapee par l'utilisateur ;
     - renvoie = un booleen (faux si pb ds la saisie de la commande,
            a vrai si execution ok);
     - correspond a la commande = myinit !!! */
{
int i ;

 for (i = 1 ; i <= (max_prior); i++)
        {
         tab[i] = 0 ;
        }
 return(true);
}


static bool dopluscourt(string line)
/* Fonction dopluscourt :
     - but = calcule le plus court chemin dans le bdd correspondant a la formule saisie;
     - 1 parametre = la ligne tapee par l'utilisateur ;
     - renvoie = un booleen (faux si pb ds la saisie de la commande,
            a vrai si execution ok);
     - utilise = shortest ;
     - correspond a la commande = mysearch !!! */
{
   bdd_formula f;
   int lg_ch ;

   if (!(f = get_formula(&line, false)))
      return(false);
   lg_ch = shortest(f,1);
   fprintf(stdout,"La longueur du plus court chemin est %d \n",lg_ch);
   return(true);
}
```

```
static bool dofatfree(string line)
/* Fonction dofatfree :
    - but =  calcule le bdd degraisse correspondant a la formule
          saisie et suivant la commande de l'utilisateur fait ou pas
          la quantification des variables hypotheses;
    - 1 parametre =  la ligne tapee par l'utilisateur ;
    - renvoie = un booleen (faux si pb ds la saisie de la commande,
          a vrai si execution ok);
    - utilise = fatfree ;
    - correspond a la commande = myff  !!! */
{
  bdd_formula f, fdeg;
  string name, dest ;
  bdd_manager b_aux ;
  int com ;

  if (!(name = get_formula_name(&line, false)))
    return(false);
  dest = strsave(name);
  if (!(f = get_formula(&line, false)))
    return(false);
  com = get_command(&line,true) ;
  b_aux = f->bdm ;
  fdeg = fatfree(f,1,com) ;
  set_formula_name(fdeg, dest);
  return(true);
}


static bool donettoyage (string line)
/* Fonction domyboolean :
    - but =  supprime toutes les informations creees lors de l'execution des commandes
    mysearch ou myff et desormais inutiles;
    - 1 parametre =  la ligne tapee par l'utilisateur ;
    - renvoie = un booleen (faux si pb ds la saisie de la commande,
          a vrai si execution ok);
    - utilise = my_bdd_unscan ;
    - correspond a la commande = mynet  !!! */
{
  bdd_formula f;

  if (!(f = get_formula(&line, false)))
    return(false);
 my_bdd_unscan(f->bdm);
 return(true);
}


static bool dotime(string line)
/* Fonction dotime :
    - but =  faire le delta de temps depuis le dernier appel et
             l'afficher dans un fichier et a l'ecran si la commande 2 est donnee par l'utilsateur;
    - 1 parametre =  la ligne tapee par l'utilisateur ;
    - renvoie = un booleen (faux si pb ds la saisie de la commande,
          a vrai si execution ok);
    - utilise = delta_tps ;
    - correspond a la commande = time  !!! */
{
  int com ;
  double delta ;
  string name ;
  string ratio ;
  char temp [80];
```

```
  FILE * fich ;

  com = get_command(&line,true) ;
  delta = delta_tps();
  if (com==2)
    {
    printf("Le temps ecoule depuis le dernier appel est = %f \n", delta);
    ratio = get_formula_name(&line,true);
    strcpy(temp, ratio);
    name = get_formula_name(&line,true);
    fich = fopen(name,"a");
    fprintf(fich,"%s %f \n",temp, delta);
    fclose(fich);
    }
  else
    printf("Depart d'une mesure de temps \n");
  return(true);
}


static bool domysize(string line)
/* Fonction domysize :
   - but = calculer la taille du bdd en nombre de noeuds et
           l'afficher dans un fichier et a l'ecran;
   - 1 parametre = la ligne tapee par l'utilisateur ;
   - renvoie = un booleen (faux si pb ds la saisie de la commande,
         a vrai si execution ok);
   - utilise = bdd_size ;
   - correspond a la commande = mysize !!! */
{
  bdd_formula f;
  buffer f_buf;
  string name;
  string name_fich ;
  FILE * fich ;
  string ratio ;
  char temp [80];
  int res_size ;

  new_buf(&f_buf, 0, sizeof(bdd_formula));
  /* get names */
  if (!quiet_switch) {
    (void) printf("size [ ");
  }
  if (name = get_formula_name(&line, true)) {
    f = fetch_operand(name);
    if (!f)
      fprintf(stderr, "ignoring non formula name %s\n", name);
    else {
      if (!quiet_switch) {
        (void) printf("%s ", name);
      }
      push_buf(&f_buf, (pointer) &f);
    }
  }
  ratio = get_formula_name(&line,true);
  strcpy(temp, ratio);
  name_fich = get_formula_name (&line, true);
  if (!quiet_switch) {
    (void) printf("]\n");
  }
  res_size = bdd_size(&f_buf);
  (void) printf("%d\n", res_size);
  fich = fopen(name_fich,"a");
  fprintf(fich,"%s %d \n",temp,res_size);
  fclose(fich);
  free_buf(&f_buf);
```

```
  return(true);
}

void register_extra_commands()
/* Procedure register_extra_commands :
    - but =  rajouter de nouvelles commandes a kbdd;
    - pas de parametre ; */
{
  register_command("mybool",
                   "<v1>..<vn> \r\t\t\t\t-- declare boolean variables",
                   domyboolean, false);
  register_command("myinit",
                   "\r\t\t\t\t-- initialize priority table",
                   doinit_tab_prior, false);
  register_command("mybdd",
                   "<f> \r\t\t\t\t-- print out representation for formula",
                   domybdd, false);
  register_command("mysearch",
                   "<f> \r\t\t\t\t-- search the shortest path in the formula",
                   dopluscourt, false);
  register_command("myff",
                   "<f1> <f2> <com> \r\t\t\t\t-- f1 = fat free version of f2 (with existential
quantification if com = 2)",
                   dofatfree, false);
  register_command("mynet",
                   "<f> \r\t\t\t\t-- remove data which have been created during myff <f>",
                   donettoyage, false);
  register_command("time",
                   "<com> <ratio> <name_fich< \r\t\t\t\t-- mesure time delta (with print on
screen and in a fich if com = 2)",
                   dotime, false);
  register_command("mysize",
                   "<f1> <ratio> <name_fich> \r\t\t\t\t-- print (on screen and in fich) number
of bdd nodes under formulas",
                   domysize, false);
}
```

Chapitre 3

Programmes pour les tests aléatoires

Dans cette annexe, sont donnés les fichiers source en langage C utilisés pour l'implémentation des tests aléatoires.

```
/*******************
	Fichier : gen_aleatoire.h

	But : test aleatoire des procedures UNI-LEX et UNI-LEX2
	Algo :
		fixer le nb de variables;
		fixer le ratio min ;
		fixer le nb de strates ;
		initialiser le generateur aleatoire ;
		pour chaque ratio du min au max faire
			nb de clauses <- ratio * nb de variables ;
			definir une base de strates et sa table des
				poids avec nb de clauses,
				nb de strates, nb de variables ;
			definir une clause a tester avec nb de variables ;
			creer le fichier contenant les instructions BDD ;
			executer UNI-LEX2 avec la base de strates,
				sa table des poids et la clause a tester ;
			memoriser ds un fichier le temps et la reponse ;
			executer UNI-LEX avec la base de strates et la
				clause a tester ;
			memoriser ds un fichier le temps et la reponse ;
			supprimer les structures de donnees (base, table de poids)
		fin pour

	Contient les #define de :
		. ratio_min,
		. ratio_max,
		. precision_ratio,
		. nb_mesures,
		. nb_essai.

	Contient les declarations et le corps des fonctions
	et procedures :
		. tirer_aleatoire
		. definir base_strate
		. definir_clause
		. creer_fichier
		. memoriser
		. UNI_LEX_aleatoire

	Utilise les packages :
		. bdd.
********************/
#include <sys/times.h>
#include <sys/types.h>
#include <time.h>
#include "bdd.h"

#define ratio_min 2.0
#define ratio_max 8.0
#define precision_ratio 0.2
#define nb_mesures 10
#define nb_essai 30

int tirer_aleatoire (int);

void definir_base_strate (base_strate **, elem_tab_poids **, int, int, int);

bool definir_clause (clause **, int);

void creer_fichier (char *);

void memoriser (char *,float, double);

void UNI_LEX_aleatoire ();
```

```
/********************
        Fichier : gen_aleatoire.c

        But : test aleatoire des procedures UNI-LEX et UNI-LEX2
        Algo :
                fixer le nb de variables;
                fixer le ratio min ;
                fixer le nb de strates ;
                initialiser le generateur aleatoire ;
                pour chaque ratio du min au max faire
                        nb de clauses <- ratio * nb de variables ;
                        definir une base de strates et sa table des
                                poids avec nb de clauses,
                                nb de strates, nb de variables ;
                        definir une clause a tester avec nb de variables ;
                        creer le fichier contenant les instructions BDD ;
                        executer UNI-LEX2 avec la base de strates,
                                sa table des poids et la clause a tester ;
                        memoriser ds un fichier le temps et la reponse ;
                        executer UNI-LEX avec la base de strates et la
                                clause a tester ;
                        memoriser ds un fichier le temps et la reponse ;
                        supprimer les structures de donnees (base, table de poids)
                fin pour

        Contient les types de donnees :

        Contient les #define de :
                         MAX_TEMPS_REEL
                         MAX_TEMPS_VIRTUEL

        Contient les declarations et le corps des fonctions
        et procedures :
                . HandlerALRM
                . HandlerVTALRM
                . tirer_aleatoire
                . definir base_strate
                . definir_clause
                . creer_fichier
                . memoriser
                . UNI_LEX_aleatoire

        Utilise les packages :
                . gen_aleatoire.h.
********************/

#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <errno.h>
#include "gen_aleatoire.h"


#define MAX_TEMPS_VIRTUEL 700
#define MAX_TEMPS_REEL 750

/* Utilisation de setjmp */

jmp_buf continuation;

/** Handler pour le signal de depassement de temps reel
*** Ne doit pas retourner a l'applelant
**/
void HandlerALRM(int sig)
{
```

```
  longjmp(continuation,1);
}


/** Handler pour le signal de depassement de temps CPU
*** Ne doit pas retourner a l'applelant
**/
void HandlerVTALRM(int sig)
{
  longjmp(continuation,2);
}


/**   fonction tirer_aleatoire
***              but : tirer un nombre aleatoire dans un intervalle donne
***              parametres : la taille de l'intervalle
***              renvoie : le nb tire
**/
int tirer_aleatoire (int max)
{
int val ;
long tir1, tir2 ;

        val = 0 ;
        tir1 = lrand48() ;
        while (val == 0)
                {
                tir2 = lrand48();
                val = (int) (tir2 % (max + 1)) ;
                if ((tir1 % 2) == 0)                              /*  nb tire sera negatif */
                        val = - val ;
                }
        return(val);
}


/**   procedure definir_base_strate
***              but : definir une base de strates contenant des
***                     litteraux tires aleatoirement et definir la table
***                     des poids correspondante
***              parametres : la base a definir, la table des poids a maj,
***                     le nb de clauses, le nb de strates, le nb de
***                     litteraux
**/
void definir_base_strate (base_strate ** pb,
                         elem_tab_poids ** pt_b, int nb_cl_max ,
                         int nb_st_max, int nb_lit_max)
{
  int nb_cl, nb_par_st ;
  int i, k, lim ;
  strate * s ;
  clause * c ;
  int lit1, lit2, lit3 ;
  elem_tab_poids * t_baux ;
  bool res ;

  *pb = init_base_strate() ;
  *pt_b = creer_init_tab_poids() ;
  t_baux = creer_init_tab_poids() ;
  nb_cl = 0 ;
  nb_par_st = nb_cl_max / nb_st_max ;
  for (i=1; i <=nb_st_max; i++)
    {
      s = init_ens() ;
      if (i == nb_st_max)
        {
          lim = nb_cl_max - nb_cl ;
```

```
            nb_cl = nb_cl_max ;
          }
        else
          {
            lim = nb_par_st ;
            nb_cl = nb_cl + nb_par_st ;
          }
        for (k=1 ; k≤lim; k++)
          {
            res = definir_clause(&c,nb_lit_max);
            if (res == faux)
              exit(1) ;
            s = ajout_ens(s,c);
          }
        *pb = ajout_base_strate(*pb,s);
        ajout_debut_strate_tab_poids(t_baux,*pt_b,i-2,lim);
        *pt_b = t_baux ;
      }
}
```

```
/**   fonction definir_clause
***              but : definir une clause contenant des
***                      litteraux tires aleatoirement
***              parametres : la clause a definir, le nb de
***                      litteraux
***              renvoie : un booleen (vrai si definition est ok,
***                      faux sinon)
**/
bool definir_clause (clause ** pc, int nb_lit_max)
{
  int lit1, lit2, lit3 ;
  bool fin ;

  if (nb_lit_max ≥ nb_lit_par_cl)
    {
      lit1 = tirer_aleatoire(nb_lit_max) ;
      fin = faux ;
      while (fin == faux)
        {
          lit2 = tirer_aleatoire(nb_lit_max) ;
          if ((lit1 ≠ -lit2) et (lit1 ≠ lit2))
            {
              fin = vrai ;
            }
        }
      fin = faux ;
      while (fin == faux)
        {
          lit3 = tirer_aleatoire(nb_lit_max) ;
          if ((lit1 ≠ -lit3) et (lit1 ≠ lit3) et
                (lit2 ≠ -lit3) et (lit2 ≠ lit3))
            {
              fin = vrai ;
            }
        }
      *pc = clause_creer() ;
      clause_ajouter(*pc, lit_codage(lit1));
      clause_ajouter(*pc, lit_codage(lit2));
      clause_ajouter(*pc, lit_codage(lit3));
      return (vrai);
    }
  else
    {
      printf ("Impossible de generer une clause avec aussi peu de litteraux \n");
      return(faux);
```

```
    }
}

/**   procedure creer_fichier
***               but : creer le fichier des resultats des essais
***               parametres : le nom du fichier
**/
void creer_fichier (char * nom)
{
  FILE * fich ;

  READ_S ("Donnez le nom du fichier de resultat : ",nom);
  fich = fopen(nom,"w");
  fclose(fich);
}



/**   procedure memoriser
***               but : memoriser les resultats des essais
***               parametres : nom du fichier resultat, ratio, temps d'execution
**/
void memoriser (char * nom, float rat, double tps)
{
  FILE * fich ;

  fich = fopen(nom,"a");
  if (fich == NULL)
    printf("Pb ouverture fichier \n");
  else
    {
      fprintf(fich, "%1.1f %3.3f\n", rat, tps);
      fclose(fich);
    }
}



/**   procedure UNI_LEX_aleatoire
***               but : test de UNI-LEX sur une base aleatoire
***                     (appel de UNI-LEX, UNI-LEX2, et preparation
***                     du fichier d'instruction pour BDD)
***               parametres : le nom du fichier shell dans lequel mettre les
***                                    appels a mykbdd
**/
void UNI_LEX_aleatoire (void)
{
  base_strate * b ;
  elem_tab_poids * t_b ;
  elem_tab_poids * t_baux ;
  clause * c ;
  int nb_cl ;
  int nb_st ;
  int nb_lit ;
  int nb_mes;
  int nb_lit_reel ;
  int i,j, jmpcode ;
  bool res, res1 ;
  double ratio ;
  char nom [nb_car_max] ;          /* nom tape par l'utilisateur + nb lit */
  char nftcr[nb_car_max] ;         /* nom fichier resultat pour BDD cre*/
  char nftdg[nb_car_max] ;         /* nom fichier resultat pour BDD deg */
  char nftve[nb_car_max] ;         /* nom fichier resultat pour BDD ver */
  char nfsav[nb_car_max] ;         /* nom fichier resultat pour BDD taille av */
  char nfsap[nb_car_max] ;         /* nom fichier resultat pour BDD taille ap */
  char nfsat[nb_car_max] ;         /* nom fichier resultat pour UNI_LEX*/
  char nfmax[nb_car_max];          /* nom fichier resultat pour UNI_LEX2 */
```

```
struct tms temps ;                                   /* var temps : mesure temps exec */
int err ;                                            /* recuperer erreur appel de times */
clock_t t1, t2 ;                                     /* var temps : mesure temps execution */
FILE * fshell ;
bool maxhasfailed, sathasfailed, bddhasfailed;
bool testsat,testmax,testbdd;
char tempbuf[80];
int pidson;                                          /* PID fils */
struct stat buffer;
int tftcr,tftdg,tftve,tfsav,tfsap;

/* pour arreter les timers */
struct itimerval stop = {{0,0}, {0,0}};
/* pour initialiser timer temps reel */
struct itimerval reel = {{0,0}, {MAX_TEMPS_REEL,0}};
/* pour initialiser timer temps virtuel */
struct itimerval virtuel = {{0,0}, {MAX_TEMPS_VIRTUEL,0}};

/* on arme les Handler de signaux */
signal(SIGALRM, HandlerALRM);
signal(SIGVTALRM, HandlerVTALRM);

READ_I("\n Donner le nb de litteraux max : ",&nb_lit);
READ_D("\n Donner le ratio de depart : ", &ratio);
READ_I("\n Donner le nb de strates de la base : ",&nb_st);
READ_I("\n Donner le nb de mesures pour chaque ratio : ",&nb_mes);
READ_I("\n Desirez vous tester avec BDD (Oui = 1, Non = 0) : ",&testbdd);
READ_I("\n Desirez vous tester avec MaxSat (Oui = 1, Non = 0) : ",&testmax);
READ_I("\n Desirez vous tester avec Sat (Oui = 1, Non = 0) : ",&testsat);
srand48(0);
READ_S ("Donnez la racine du nom des fichiers de resultat : ",nom);
sprintf (nom, "%s_%d",nom,nb_lit);
sprintf (nfsat, "%s.sat", nom);
sprintf (nfmax, "%s.max", nom);
sprintf (nftcr, "%s.tcr", nom);
sprintf (nftdg, "%s.tdg", nom);
sprintf (nftve, "%s.tve", nom);
sprintf (nfsav, "%s.sav", nom);
sprintf (nfsap, "%s.sap", nom);

t_baux = creer_init_a_un_tab_poids();
i = 0;
while (ratio <= 8.001)
  {
    i++;
    nb_cl = (int) ((ratio+1e-4) * nb_lit) ;
    printf("\nEssai %d, %d variables, %d clauses, ratio %f\n\n",i,nb_lit, nb_cl,ratio);
    bddhasfailed = !testbdd;
    maxhasfailed = !testmax;
    sathasfailed = !testsat;
    for (j=1 ; j <= nb_mes; j++)
      {
        definir_base_strate(&b,&t_b,nb_cl, nb_st, nb_lit);
        res1 = definir_clause(&c,nb_lit);
        /****************** preparation test BDD */
        if (!bddhasfailed)
          {
            printf("Generation fichier BDD...");
            fflush(stdout);
            nb_lit_reel = creer_fich_BDD(b,t_b,c,i,j,nom, ratio);
            printf("terminee !\n");

            setitimer(ITIMER_REAL, &reel, NULL);
            setitimer(ITIMER_VIRTUAL, &virtuel, NULL);

            lstat(nftcr,&buffer);
            tftcr = buffer.st_size;
```

```
lstat(nftdg,&buffer);
tftdg = buffer.st_size;
lstat(nftve,&buffer);
tftve = buffer.st_size;
lstat(nfsav,&buffer);
tfsav = buffer.st_size;
lstat(nfsap,&buffer);
tfsap = buffer.st_size;

switch(jmpcode = setjmp(continuation))
  {
  case 0:
    if ((pidson = fork()) == 0)
      {
        sprintf(tempbuf,"%s_%d_%d.bdd",nom,i,j);
        errno = 0 ;
        if (execl(path_bdd,"mykbdd",tempbuf," > ", "/dev/null",NULL) != 0)
          perror("\n\n---> Erreur ds execl <---");
        exit(0);
      }
    waitpid(pidson,NULL,0) ;
    break;

  case 1:
    errno = 0 ;
    if (kill(pidson,9) != 0)
      perror("\n\n---> Erreur ds kill <---");
    printf("SetJmp code 1 : depassement du temps reel maximum\n");
    waitpid(pidson,NULL,0);
    bddhasfailed = 1;
    signal(SIGALRM, HandlerALRM);
    signal(SIGVTALRM, HandlerVTALRM);
    break;

  case 2:
    if (kill(pidson,9) != 0)
      perror("\n\n---> Erreur ds kill <---");
    printf("SetJmp code 2 : depassement du temps virtuel maximum\n");
    waitpid(pidson,NULL,0);
    bddhasfailed = 1;
    signal(SIGVTALRM, HandlerVTALRM);
    signal(SIGALRM, HandlerALRM);
    break;

  default:
    printf("SetJmp code inconnu !\n");
    break;
  };
setitimer(ITIMER_REAL,&stop,NULL);
setitimer(ITIMER_VIRTUAL,&stop,NULL);
lstat(nftcr,&buffer);
if (tftcr == buffer.st_size)
  memoriser (nftcr, ratio,-1.0) ;
lstat(nftdg,&buffer);
if (tftdg == buffer.st_size)
  memoriser (nftdg, ratio,-1.0) ;
lstat(nftve,&buffer);
if (tftve == buffer.st_size)
  memoriser (nftve, ratio,-1.0) ;
lstat(nfsav,&buffer);
if (tfsav == buffer.st_size)
  memoriser (nfsav, ratio,-1.0) ;
lstat(nfsap,&buffer);
if (tfsap == buffer.st_size)
  memoriser (nfsap, ratio,-1.0) ;
sprintf(tempbuf,"\\rm %s_%d_%d.bdd", nom,i,j);
system(tempbuf);
```

```
   }
else
   {
      memoriser (nftcr, ratio,-1.0) ;
      memoriser (nftdg, ratio,-1.0) ;
      memoriser (nftve, ratio,-1.0) ;
      memoriser (nfsav, ratio,-1.0) ;
      memoriser (nfsap, ratio,-1.0) ;
   }

/***************** test de UNI_LEX2 */
if (!maxhasfailed)
   {
      setitimer(ITIMER_REAL, &reel, NULL);
      setitimer(ITIMER_VIRTUAL, &virtuel, NULL);

      printf("Resolution par MaxSat...");
      fflush(stdout);

      switch(jmpcode = setjmp(continuation))
         {
         case 0:
            if ((pidson = fork()) == 0)
               {
                  err = times(&temps);
                  ERROR(err==-1, "Erreur Times");
                  t1 = temps.tms_utime;
                  res = UNI_LEX2 (b, c, nb_lit, nb_cl, nb_lit_par_cl, t_b);
                  err = times(&temps);
                  printf("terminee !\n");
                  ERROR(err==-1, "Erreur Times");
                  t2 = temps.tms_utime;
                  memoriser (nfmax, ratio, (double)(t2-t1)/CLK_TCK) ;
                  exit(0);
               }
            waitpid(pidson,NULL,0);
            break;

         case 1:
            kill(pidson,9);
            printf("SetJmp code 1 : depassement du temps reel maximum\n");
            waitpid(pidson,NULL,0);
            memoriser (nfmax, ratio,-1.0) ;
            maxhasfailed = 1;
            signal(SIGALRM, HandlerALRM);
            signal(SIGVTALRM, HandlerVTALRM);
            break;

         case 2:
            kill(pidson,9);
            printf("SetJmp code 2 : depassement du temps virtuel maximum\n");
            waitpid(pidson,NULL,0);
            memoriser (nfmax, ratio,-1.0) ;
            maxhasfailed = 1;
            signal(SIGVTALRM, HandlerVTALRM);
            signal(SIGALRM, HandlerALRM);
            break;

         default:
            printf("SetJmp code inconnu !\n");
            memoriser (nfmax, ratio,-1.0) ;
            break;
         };
      setitimer(ITIMER_REAL,&stop,NULL);
      setitimer(ITIMER_VIRTUAL,&stop,NULL);
   }
else
```

```
  memoriser (nfmax, ratio,-1.0) ;

/******************* test de UNI_LEX */
if (!sathasfailed)
  {
    setitimer(ITIMER_REAL, &reel, NULL);
    setitimer(ITIMER_VIRTUAL, &virtuel, NULL);

    printf("Resolution par Sat...");
    fflush(stdout);

    switch(jmpcode = setjmp(continuation))
      {
      case 0:
        if ((pidson = fork()) == 0)
          {
            err = times(&temps);
            ERROR(err==-1, "Erreur Times");
            t1 = temps.tms_utime;
            res = UNI_LEX (b, c, nb_lit, nb_cl, nb_lit_par_cl, t_baux,0);
            err = times(&temps);
            printf("terminee !\n");
            ERROR(err==-1, "Erreur Times");
            t2 = temps.tms_utime;
            memoriser (nfsat, ratio, (double)(t2-t1)/CLK_TCK) ;
            exit(0);
          }
        waitpid(pidson,NULL,0);
        /***************** menage */
        destruc_compl_plus_base_strate(b);
        clause_free(c);
        t_b = destruc_tab_poids(t_b);
        break;

      case 1:
        kill(pidson,9);
        printf("SetJmp code 1 : depassement du temps reel maximum\n");
        waitpid(pidson,NULL,0);
        memoriser (nfsat, ratio,-1.0) ;
        destruc_compl_plus_base_strate(b);
        clause_free(c);
        t_b = destruc_tab_poids(t_b);
        sathasfailed = 1;
        signal(SIGALRM, HandlerALRM);
        signal(SIGVTALRM, HandlerVTALRM);
        break;

      case 2:
        kill(pidson,9);
        printf("SetJmp code 2 : depassement du temps virtuel maximum\n");
        waitpid(pidson,NULL,0);
        memoriser (nfsat, ratio,-1.0) ;
        destruc_compl_plus_base_strate(b);
        clause_free(c);
        t_b = destruc_tab_poids(t_b);
        sathasfailed = 1;
        signal(SIGVTALRM, HandlerVTALRM);
        signal(SIGALRM, HandlerALRM);
        break;

      default:
        printf("SetJmp code inconnu !\n");
        memoriser (nfsat, ratio,-1.0) ;
        break;
      }
    setitimer(ITIMER_REAL,&stop,NULL);
    setitimer(ITIMER_VIRTUAL,&stop,NULL);
```

```
            }
          else
            memoriser (nfsat, ratio,-1.0) ;
        }
      ratio = ratio + precision_ratio ;
    }
  printf("Les resultats sont ds les fichiers %s, %s \n", nfsat, nfmax);
  sprintf(tempbuf,"%sanalyse %s \n",path_anal,nfsat);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s \n",path_anal,nfmax);
  system(tempbuf);
  sprintf(tempbuf,"%ssomme %s \n",path_anal,nom);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s.tcr \n",path_anal,nom);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s.tdg \n",path_anal,nom);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s.tve \n",path_anal,nom);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s.tot \n",path_anal,nom);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s.tcp \n",path_anal,nom);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s.sav \n",path_anal,nom);
  system(tempbuf);
  sprintf(tempbuf,"%sanalyse %s.sap \n",path_anal,nom);
  system(tempbuf);

  t_baux = destruc_tab_poids(t_baux);
}
```

```
/*******************
        Fichier : bdd.h

        Contient les types de donnees :
                . elem_tab_symb,
                . elem_tab_hyp.

        Contient les #define de :

        Contient les declarations en externe des fonctions et procedures :
                . affich_fichier,
                . maj_tab_symb,
                . maj_tab_hyp,
                . creer_fichier_bdd,
                . creer_fichier_eval,
                . creer_fichier_mybool,
                . concat_fichiers,
                . maj_fichier_mybool,
                . maj_fichier_eval,
                . appartient_tab_symb,
                . creer_tab_symb,
                . creer_tab_hyp,
                . destruc_tab_symb,
                . destruc_tab_hyp,
                . affich_tab_symb,
                . affich_tab_hyp.

        Utilise les packages :
                . lexico.
********************/

#include "lexico.h"


/***** les types  *******/
       /* la table des symboles contiendra chaque symbole
          de la base dans l'ordre de declaration pour
          le BDD - le champ flag sert a savoir si apres
          le symbole il faut introduire dans la declaration
         une hypothese (flag a 1, sinon a 0),
         les champs pre_hyp et der_hyp pointent sur les
         hypotheses situees apres le symbole ds l'ordre */
typedef struct elem_tab_symb {
  int symb ;
  int flag;
  int pre_hyp;
  int der_hyp;} elem_tab_symb;

    /* la table des hypotheses contiendra chaque hypothese
            (egale a l'indice de la table)
         de la base dans l'ordre de declaration pour
           le BDD - le champ flag sert a savoir si apres
           l'hypothese il faut continuer a introduire
         une hypothese dans la declaration (flag a 1, sinon a 0)
         le champ hyp_suiv pointe sur l'hypothese suivante */
typedef struct elem_tab_hyp {
  unsigned int poids ;
  int flag;
  int hyp_suiv ;} elem_tab_hyp;



/****** les procedures et fonctions ******/
void affich_fichier (char *);

int maj_tab_symb (base_strate *, clause * c, elem_tab_symb *);
```

```
int maj_tab_hyp (base_strate *, elem_tab_poids *, elem_tab_symb * , int, elem_tab_hyp *, char *);

int creer_fich_BDD (base_strate *, elem_tab_poids *,
                    clause *, int, int, char *, float);

void creer_fichier_eval (char *, int, int, char *);

void creer_fichier_mybool (char *, int, int, char *);

void concat_fichiers (char *, float, int, int, char *, int, char *, char *);

void maj_fichier_mybool (char *, elem_tab_symb *, int, elem_tab_hyp *, int);

void maj_fichier_eval (char *, clause *, int);

bool appartient_tab_symb (int, elem_tab_symb *, int);

elem_tab_symb * creer_tab_symb ();

elem_tab_hyp * creer_tab_hyp ();

elem_tab_symb * destruc_tab_symb (elem_tab_symb *);

elem_tab_hyp * destruc_tab_hyp (elem_tab_hyp *);

void affich_tab_symb (elem_tab_symb *);

void affich_tab_hyp (elem_tab_hyp *);
```

```
/********************
        Fichier : bdd.c

        Contient les types de donnees :

        Contient les #define de :

        Contient le corps des fonctions et procedures :
                . affich_fichier,,
                . maj_tab_symb,
                . maj_tab_hyp,
                . creer_fichier_bdd,
                . creer_fichier_eval,
                . creer_fichier_mybool,
                . concat_fichiers,
                . maj_fichier_mybool,
                . maj_fichier_eval,
                . appartient_tab_symb,
                . creer_tab_symb,
                . creer_tab_hyp,
                . destruc_tab_symb,
                . destruc_tab_hyp,
                . affich_tab_symb,
                . affich_tab_hyp.

        Utilise les packages :
                . bdd.
*********************/

#include "bdd.h"


/**   procedure affich_fichier
***               but : affiche le fichier des resultats des essais
***               parametres : le nom du fichier
**/
void affich_fichier (char * nom)
{
FILE * fich ;
char * res;
char chaine [nb_car_max] ;

fich = fopen(nom,"r");
if (fich == NULL)
  printf("Fichier n'existe pas \n");
else
  {
    res = fgets(chaine, nb_car_max,fich);
    if (res == NULL)
      printf("Fichier vide \n");
    while (res ≠ NULL)
        {
        printf("%s",chaine);
        res = fgets(chaine, nb_car_max,fich);
        }
    fclose(fich);
  }
}



/**   fonction maj_tab_symb
***               but : mettre a jour la table des symboles
***               parametres : la base, la clause, la table des symboles,
***               renvoie : l'indice du dernier element significatif de la table
**/
int maj_tab_symb (base_strate * b, clause * c, elem_tab_symb * ts)
```

```
{
  int der_ts ;
  int i , j, k ;
  strate * s ;
  liste_strate * ls ;
  liste_clause * lc ;
  int symbol ;

  der_ts = 0 ;
  ls = b->les_strates ;
  /* traitement de la base */
  for (i=1; i <= b->nb_strates; i ++)
    {                                                        /* pour chaque strate */
      s = ls->un_ens ;
      lc = s->les_clauses ;
      for (j=1; j <= s->nb_clauses; j++)
        {                                                    /* pour chaque clause */
          for (k = 1 ; k <= clause_longueur(lc->une_clause); k++)
            {                                 /* pour chaque litteral de la clause */
              symbol = lit_decodage(clause_element(lc->une_clause));
              if (symbol <0)
                symbol = - symbol ;
              /* mise a jour de ts */
              if (appartient_tab_symb(symbol,ts,der_ts - 1) == faux)
                {
                  ts[der_ts].symb = symbol;
                  der_ts ++ ;
                }
            }
          lc = lc->clause_suiv ;
        }
      ls = ls->ens_suiv ;
    }
  /* traitement de la clause c */
    for (k = 1 ; k <= clause_longueur(c); k++)
            {                                 /* pour chaque litteral de la clause */
              symbol = lit_decodage(clause_element(c));
              if (symbol <0)
                symbol = - symbol ;
              /* mise a jour de ts */
              if (appartient_tab_symb(symbol,ts,der_ts - 1) == faux)
                {
                  ts[der_ts].symb = symbol;
                  der_ts ++ ;
                }
            }
  der_ts -- ;
  return(der_ts) ;
}


/**   fonction maj_tab_hyp
***             but : mettre a jour la table des hypotheses et le
***                   fichier des instructions eval pour le BDD
***             parametres : la base, la table des poids,  la table des
***                   symboles, l'indice du dernier element significatif
***                   de la table des symboles, la table des hypotheses,
***                le nom du fichier d'instructions eval pour le BDD
***             renvoie : l'indice du dernier element significatif de la table
**/
int maj_tab_hyp (base_strate * b, elem_tab_poids * t_b,
                elem_tab_symb * ts,
                                int der_ts, elem_tab_hyp * th, char * nfeval)
{
  int der_th ;
  int i , j ;
  strate * s ;
```

```
  liste_strate * ls ;
  liste_clause * lc ;
  int flag_hyp ;
  int ind ;

  der_th = 0 ;
  ls = b→les_strates ;
  for (i=1; i ≤ b→nb_strates; i ++)
    {                                                        /* pour chaque strate */
      s = ls→un_ens ;
      lc = s→les_clauses ;
      for (j=1; j ≤ s→nb_clauses; j++)
        {                                                    /* pour chaque clause */
          ind = ordre_der_lit(lc→une_clause,ts,der_ts);
/* trouver indice du lit. de c le plus loin du debut de ts */
          /* mise a jour de th et de ts */
          if (ts[ind].flag == 0)
                {
                 ts[ind].flag = 1 ;
                ts[ind].pre_hyp = der_th ;
                }
          else
                {
                th[ts[ind].der_hyp].flag = 1 ;
                th[ts[ind].der_hyp].hyp_suiv = der_th ;
                }
          ts[ind].der_hyp = der_th ;
          th[der_th].poids = t_b[i-1].poids ;
          /* mise a jour du fichier eval */
          maj_fichier_eval (nfeval, lc→une_clause, der_th);
          /* passage a la clause suivante */
          lc = lc→clause_suiv ;
          der_th ++ ;
        }
      ls = ls→ens_suiv ;
    }
  return(der_th - 1) ;
}


/**   fonction ordre_der_lit
***            but : trouver l'indice du litteral de la clause situe
***                     le plus loin du debut de la table des symboles
***            parametres : la clause, la table des symboles,
***                  l'indice du dernier element significatif de la
***                     table des symboles,
***            renvoie :  l'indice du litteral de la clause situe le
***                     plus loin du debut de la table des symboles
**/
int ordre_der_lit (clause * c, elem_tab_symb * ts, int der_ts)
{
  int i ;
  int nb_lit ;

  i = 0 ;
  nb_lit = 0;
  while ((i≤der_ts) et (nb_lit ≠ clause_longueur(c)))
    {
      if ((clause_appartient(c,lit_codage(ts[i].symb))) ou (clause_appartient(c,lit_codage(-ts[i].symb))))
          nb_lit ++ ;
      i++;
    }
  ERROR((nb_lit ≠ clause_longueur(c)), "ordre_der_lit : Pb dans la table des symboles !\n");
  return(i-1);
}
```

```
/**   fonction creer_fich_BDD
***               but : creer et mettre a jour le fichier d'instructions pour BDD
***               parametres : la base, la table des poids, la clause,
***                       le numero de l'essai,  le numero de la mesure et
***                       le nom du fichier d'essai
***               renvoie : le nombre reel de symboles (litteraux) utilises
**/
int creer_fich_BDD (base_strate * b, elem_tab_poids * t_b,
              clause * c, int num_ess, int num_mes, char * nom, float ratio)
{
  elem_tab_symb * ts ;
  elem_tab_hyp * th ;
  int der_ts ;
  int der_th ;
  char nfeval[nb_car_max] ;
  char nfbool[nb_car_max] ;
  char nfconcat[nb_car_max] ;
  int num_hyp ;
  int i , j, k ;
  strate * s ;
  liste_strate * ls ;
  clause * caux ;
  liste_clause * lc ;
  int symbol, hypoth ;

  /* les init */
  creer_fichier_eval (nom, num_ess,num_mes,nfeval);
  creer_fichier_mybool(nom,num_ess,num_mes,nfbool);
  ts = creer_tab_symb();
  th = creer_tab_hyp();
  /* maj de la table des symboles en parcourant la base */
  der_ts = maj_tab_symb (b, c, ts) ;
  /* maj de la table des hypotheses en parcourant la base et la table des symboles
   et en meme temps maj du fichier eval */
  der_th = maj_tab_hyp(b,t_b, ts, der_ts, th, nfeval) ;
  /* rajout de la clause a prouver dans le bdd */
  maj_fichier_eval(nfeval,c,-1);
  /* maj du fichier des instructions mybool */
  maj_fichier_mybool(nfbool, ts, der_ts, th, der_th);
  /* concatenation des deux fichiers */
  concat_fichiers (nom, ratio, num_ess, num_mes, nfeval, der_th, nfbool, nfconcat);

  destruc_tab_symb(ts);
  destruc_tab_hyp(th);
  return(der_ts + 1);
}


/**  procedure creer_fichier_eval
***               but : creer le fichier qui contiendra les instructions
***               d'evaluation destinees au BDD (fichier eval)
***               parametres : le nom du fichier d'essai et le numero de l'essai,
***               le numero de la mesure, le nom du fichier cree.
***/
void creer_fichier_eval (char * nom, int num_ess, int num_mes, char * nfeval)
{
  FILE * fich ;

  sprintf(nfeval, "%s_%d_%d.eval",nom, num_ess, num_mes);
  fich = fopen(nfeval,"w");
  fprintf(fich,"time 1\n");
  fclose(fich);
}
```

```
/**  procedure creer_fichier_mybool
***            but : creer le fichier qui contiendra les instructions de
***                 declaration destinees au BDD (fichier mybool)
***            parametres : le nom du fichier d'essai et le numero de l'essai,
***            le numero de la mesure, le nom du fichier cree.
***/
void creer_fichier_mybool (char * nom, int num_ess, int num_mes, char * nfbool)
{
  FILE * fich ;

  sprintf(nfbool, "%s_%d_%d.bool",nom, num_ess,num_mes);
  fich = fopen(nfbool,"w");
  fprintf(fich,"myinit\n");
  fclose(fich);
}



/** procedure concat_fichiers
***            but : concatene deux fichiers (mybool et eval),
***        ATTENTION : les deux fichiers source seront detruits !!
***            parametres : le nom du fichier d'essai, le numero de l'essai,
***                    le numero de la mesure,
***                le nom des deux fichiers a concatener,
***                le nom du fichier resultat de la concatenation.
***/
void concat_fichiers (char * nom, float ratio, int num_ess, int num_mes, char * nfeval, int der_th, char *
nfbool, char * nfresul)
{
  FILE * feval;
  FILE * fbool ;
  FILE * fresul;
  char * res ;
  char chaine[nb_car_max];

  /* les init avec maj du nom du fichier resultat */
  sprintf(nfresul, "%s_%d_%d.bdd",nom, num_ess,num_mes);
  fresul = fopen(nfresul,"w");
  fbool = fopen(nfbool,"r");
  feval = fopen(nfeval,"r");
  /* parcours du fichier mybool */
  res = fgets(chaine, nb_car_max,fbool);
  while (res ≠ NULL)
    {
      fprintf(fresul, "%s", chaine);
      res = fgets(chaine, nb_car_max,fbool);
    }
  fclose(fbool);
  /* parcours du fichier eval */
  res = fgets(chaine, nb_car_max,feval);
  while (res ≠ NULL)
    {
      fprintf(fresul, "%s", chaine);
      res = fgets(chaine, nb_car_max,feval);
    }
  fclose(feval);
  /* les dernieres instructions */
  fprintf(fresul,"time 2 %1.1f %s.tcr \n", ratio, nom);
  fprintf(fresul,"garbage\n");
  fprintf(fresul,"mysize F%d %1.1f %s.sav\n", der_th, ratio, nom);
  fprintf(fresul,"time 1 \n");
  fprintf(fresul,"myff F%d F%d 2\n", der_th + 1, der_th);
  fprintf(fresul,"time 2 %1.1f %s.tdg \n", ratio, nom);
  fprintf(fresul,"mysize F%d %1.1f %s.sap\n", der_th + 1, ratio, nom);
  fprintf(fresul,"time 1 \n");
  fprintf(fresul,"implies F%d F_A_INFERER \n",der_th + 1);
  fprintf(fresul,"time 2 %1.1f %s.tve \n", ratio, nom);
```

```
  fprintf(fresul,"quit \n");
  fclose(fresul);
  /* nettoyage */
  remove(nfeval);
  remove(nfbool);
}



/**  procedure maj_fichier_mybool
***                 but : met a jour le fichier mybool avec le contenu des
***                 tables de symboles et d'hypothese,
***                 parametres : le nom du fichier mybool, les tables de
***                 symboles et d'hypotheses,
***/
void maj_fichier_mybool (char * nom, elem_tab_symb * ts, int der_ts, elem_tab_hyp * th, int der_th)
{
  int ind_ds_th ;
  int i ;
  FILE * fich ;

  fich = fopen(nom,"a");
  for (i=0 ; i<= der_ts; i++)
    {                                               /* pour chaque symbole propositionnel */
      fprintf(fich,"mybool S%d 0\n",ts[i].symb);
      if (ts[i].flag == 1)
        {                                           /* il faut intercaler des hypotheses */
          ind_ds_th = ts[i].pre_hyp;
          fprintf(fich,"mybool C%d %d\n", ind_ds_th, th[ind_ds_th].poids);
          while (th[ind_ds_th].flag == 1)
            {                                       /* il y a encore des hypotheses a intercaler */
              ind_ds_th = th[ind_ds_th].hyp_suiv;
              fprintf(fich,"mybool C%d %d\n", ind_ds_th, th[ind_ds_th].poids);
            }
        }
    }
  fclose(fich);
}



/**  procedure maj_fichier_eval
***                 but : met a jour le fichier eval avec une clause,
***                 parametres : le nom du fichier eval, la clause, le numero
***                         de l'hypothese
***/
void maj_fichier_eval (char * nom, clause * c, int num_hyp)
{
  FILE * fich ;
  int i ;
  char buf[nb_car_max];
  int lit ;
  char str_lit[nb_car_max];

  /* les init */
  fich = fopen(nom,"a");
  sprintf(buf, "");
  for (i=1; i<=clause_longueur(c); i++)
    {                                               /* parcours de la clause */
      lit = lit_decodage(clause_element(c));
      if (lit < 0)                                  /* cas d'un litteral negatif */
        sprintf(str_lit,"!S%d",abs(lit));
      else                                          /* cas d'un litteral positif */
        sprintf(str_lit,"S%d",lit);
      sprintf(buf, "%s%s",buf,str_lit);
      if (i < clause_longueur(c))
        sprintf(buf,"%s + ",buf);
```

```
    }
  /* ecriture de l'instruction eval dans le fichier */
  if (num_hyp == 0)                                          /* premiere instruction eval */
      fprintf(fich,"eval F%d (!C%d + %s)\n",num_hyp,num_hyp,buf);
  else
    {
    if (num_hyp == -1)                    /* ecriture de l'instruction eval pour la clause a inferer */
      fprintf(fich,"eval F_A_INFERER (%s)\n",buf);
    else
      {                                                       /* autres instructions evals */
        fprintf(fich,"eval F%d F%d & (!C%d + %s)\n",num_hyp, num_hyp - 1,num_hyp,buf);
        fprintf(fich,"free F%d\n",num_hyp - 1);
      }
    }
  fclose(fich);
}



/**  fonction appartient_tab_symb
***              but : teste si un symbole donne apparait deja ds la table
***                      des symboles,
***              parametres : le symbole, la table des symboles,
***                      l'indice du dernier element significatif de la table.
***              retourne : vrai si le symbole est deja ds la table, faux sinon.
***/
bool appartient_tab_symb (int symb, elem_tab_symb * ts, int der_ts)
{
 int i ;
 bool trouve ;

 i = 0 ;
 trouve = faux ;
 while ((i≤der_ts) et (trouve == faux))
   {
     if (symb == ts[i].symb)
       trouve = vrai ;
     else
       i++;
   }
 return(trouve);
}



/**  fonction creer_tab_symb
***              but : cree et initialise une table des symboles
***              parametres : neant,
***              retourne : la table des symboles creee.
***/
elem_tab_symb * creer_tab_symb ()
{
  elem_tab_symb * t ;
  int i ;

  t = newtab(elem_tab_symb, nbmaxlit);
  newerr(t, "creation table des symboles : plus de memoire \n");

  for (i = 0 ; i < nbmaxlit ; i++)
    {
      t[i].symb = 0 ;
      t[i].flag = 0 ;
      t[i].pre_hyp = -1 ;
      t[i].der_hyp = -1 ;
    }
  return (t) ;
}
```

```
/**  fonction creer_tab_hyp
***              but : cree et initialise une table des hypotheses
***              parametres : neant,
***              retourne : la table des hypotheses creee.
***/
elem_tab_hyp * creer_tab_hyp ()
{
  elem_tab_hyp * t ;
  int i ;

  t = newtab(elem_tab_hyp, nbmaxclause);
  newerr(t, "creation table des hypothese : plus de memoire \n");

  for (i = 0 ; i < nbmaxclause ; i++)
    {
      t[i].flag = 0 ;
      t[i].poids = 0 ;
      t[i].hyp_suiv = -1 ;
    }
  return (t) ;
}



/**  fonction destruc_tab_symb
***              but : supprime une table des symboles,
***              parametres :  la table des symboles,
***              renvoie : le pointeur NULL.
***/
elem_tab_symb * destruc_tab_symb (elem_tab_symb * t)
{
  if (t != NULL)
    dispose(t) ;
  return(NULL);
}



/**  fonction destruc_tab_hyp
***              but : supprime une table des hypotheses,
***              parametres : la table des hypotheses.
***              renvoie : le pointeur NULL.
***/
elem_tab_hyp * destruc_tab_hyp (elem_tab_hyp * t)
{
  if (t != NULL)
    dispose(t) ;
  return(NULL);
}



/**  procedure affich_tab_symb
***              but : afficher une table de symboles,
***              parametres : la table de symboles.
***/


void affich_tab_symb (elem_tab_symb * t)
{
  int i ;

  if (t == NULL)
```

```
    printf("\n table des symboles n'existe pas \n");
  else
    {
      printf("\n Table des symboles : \n");
      i = 0 ;
      while ((i < nbmaxlit) et (t[i].symb ≠ 0))
        {
          printf(" Symbole %d, flag = %d, pre_hyp = % d, der_hyp = %d \n", t[i].symb, t[i].flag,
t[i].pre_hyp, t[i].der_hyp) ;
          i ++;
        }
    }
}



/**  procedure affich_tab_hyp
***               but : afficher une table d'hypotheses,
***               parametres : la table d'hypothese.
***/


void affich_tab_hyp (elem_tab_hyp * t)
{
  int i ;

  if (t == NULL)
    printf("\n table des hypotheses n'existe pas \n");
  else
    {
      printf("\n Table des hypotheses : \n");
      i = 0 ;
      while ((i < nbmaxclause) et (t[i].poids ≠ 0))
        {
          printf(" Hypothese %d : poids = %d, flag = %d, hyp_suiv = %d\n", i, t[i].poids, t[i].flag,
t[i].hyp_suiv) ;
          i ++;
        }
    }
}
```

Bibliographie

[BRB90] Brace (Karl S.), Rudell (Richard R.) et Bryant (Randal E.). – Efficient implementation of a BDD package. *In : 27th ACM/IEEE Design Automation Conference*, pp. 40–45.

[LS95] Lagasquie-Schiex (Marie-Christine). – *Contribution à l'étude des relations d'inférence non-monotone combinant inférence classique et préférences.* – Thèse, Université Paul Sabatier, IRIT, 1995.