Apéndice: Implementación del código de una curva algebraica

Presentamos aquí nuestros añadidos al programa Xtopo, para hacerlo capaz de reconocer si dos curvas algebraicas planas afines son topológicamente equivalentes (cf. la Sección 1.4, en particular el párrafo 1.4.2). Consisten esencialmente en los dos fragmentos que se dan a continuación, más las funciones que ellos utilizan. El primero de los fragmentos calcula un código asociado a la curva, a partir de la información de su descomposición algebraica cilíndrica (C.A.D.) y traduce dicho código a un código canónico. El segundo, compara el código canónico con los obtenidas para las curvas que hayan sido procesadas con anterioridad. Consúltense las páginas 29 y siguientes para una somera descripción de las principales funciones.

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

 computation of the canonical code from the C.A.D. information

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

  timeTaken = ACLOCK();
  if (isempty)
    S = MK_COMPONENT(NIL,LIST1(INFTY_PT));
  else {
    S = CODE(HHCODE(T)); }
  C = SEP_PROJ(S);
  F = FACES(C);
  B = TREE(F);
  B = SIMPLIFY(B);
  B = CANONIZE(B);
  timeTaken = ACLOCK() - timeTaken;
  SWRITE("\n             THIS IS THE CANONICAL TREE: ");
  TWRITE(B,0);    SWRITE("\n");
  printf("   %d milli-seconds took computing the canonical code\n",timeTaken);


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

 compares B with the codes so far obtained and prompts it coincides
 with one of them

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

  Codes1 = NIL;
  i = 0;
  timeTaken = ACLOCK();
  while (Codes != NIL) {
    ADV(Codes,&B1,&Codes);
    i++;
    s = TREECOMP(B,B1);
    if ( s==0 ) {
      SWRITE("\nTHIS CURVE HAS THE SAME TOPOLOGY AS CURVE NUMBER ");
      OWRITE(i);SWRITE("\n");     }
    Codes1 = COMP(B1,Codes1); }
  timeTaken = ACLOCK() - timeTaken;
  printf("   %d milli-seconds took comparing \n",timeTaken);
  Codes1 = COMP(B,Codes1);
  Codes = INV(Codes1);
```

Las macros y funciones utilizadas se incluyen a continuación. Cada función comienza con un párrafo comentado (acotado por `/*` y `*/`) que describe su finalidad, su INPUT y su OUTPUT. Las macros definen las estructuras de datos que se utilizan. Todas las estructuras son listas de listas y utilizan las primitivas de SACLIB para su manejo, como FIRST, SECOND, etc. (que obtienen elementos de una lista), LISTn (que construye una lista de n elementos) o ADV, CONC e INV (las cuales, respectivamente, obtienen el primer elemento de una lista y lo eliminan de ella, contanenan dos listas e invierten una lista).

Definición de las estructuras de datos

```
/* macros --------------------------------------------------------------- */

#define MK_COMPONENT(E,P)    LIST2(E,P)
#define MK_COMP_FACE(E,P,f)  LIST3(E,P,f)
#define EDGES(C)             FIRST(C)
#define POINTS(C)            SECOND(C)
#define face(C)              THIRD(C)

#define MK_POINT(i,j)        LIST2(i,j)

#define INFTY_PT             LIST2(0,1)

#define IS_UNBOUNDED(C)      (EQUAL(FIRST(POINTS(C)),INFTY_PT))

/*======================================================================
Make half edge data structure.

Inputs
  p : a point.
  b : a nonnegative BETA-digit.
  m : a nonnegative BETA-digit.

Output
  A data structure representing the half edge with number b adjacent to
  the point p which has multipilicity m.
======================================================================*/
#define MK_HALF_EDGE(p,b,m)  LIST3(p,b,m)
```

```
#define POINT(H)         FIRST(H)
#define BRANCH(H)        SECOND(H)
#define MULTIPLICITY(H)  THIRD(H)

#define MK_EDGE(h1,h2)   LIST2(h1,h2)
#define HALF_EDGE1(E)    FIRST(E)
#define HALF_EDGE2(E)    SECOND(E)

#define POINT1(E)        POINT(HALF_EDGE1(E))
#define POINT2(E)        POINT(HALF_EDGE2(E))


/*===========================================================================
Make tree node.

Inputs
  C : a component (E,P,f).
  g : the face containing C.
  S : the list of the sons of C.

Output
  N : a tree node structure.
===========================================================================*/
#define MK_TREE_NODE(C,g,S)  LIST3(C,g,S)
#define MK_TREE_SYM(C,g,S,sym)  LIST4(C,g,S,sym)

#define COMPONENT(N)         FIRST(N)
#define CONTAINING_FACE(N)   SECOND(N)
#define SONS(N)              THIRD(N)
#define SYMMETRY(N)          FOURTH(N)


/*===========================================================================
Make dictionary data structure.

Inputs
  h = (p,b,m) : a halfedge, in old code (point, branch, multiplicity).
  n : new name for point p.

Output
  An item for the dictionary used to translate from old code to new code
and vice-versa.
===========================================================================*/
#define DICT(C)        SECOND(C)

#define MK_ITEM(n,h)  LIST2(n,h)

#define LABEL(item)      FIRST(item)
#define NEW(item)        LIST3(FIRST(item),1,MULTIPLICITY(SECOND(item)))
#define OLD(item)        SECOND(item)
```

Funciones utilizadas

ADD_TO_TREE

```
/*=========================================================================
       Tp <- ADD_TO_TREE(C,F,f,T)

Add to tree.

Inputs
  C : a component.
  F : the topological father of C.
  f : the face of F containing C.
  T : a topological tree.

Output
  Tp : a topological tree or NIL.  If T contains F then Tp is the tree
       obtained by adding C to T as a son of F.  Otherwise Tp = NIL.
======================================================================*/
#include "code.h"

Word ADD_TO_TREE(C,F,f,T)
       Word C,F,f,T;
{
       Word D,N,S,Sp,S1,Tp,Tpp,g;

Step1: /* Initialize. */
       D = COMPONENT(T);
       g = CONTAINING_FACE(T);
       S = SONS(T);

Step2: /* The father is the root of the T. */
       if (EQUAL(F,D)) {
  N = MK_TREE_NODE(C,f,NIL);
  S = COMP(N,S);
  Tp = MK_TREE_NODE(D,g,S);
  goto Return; }

Step3: /* Check the sons. */
       Sp = NIL;
       while (S != NIL) {
  ADV(S,&S1,&S);
  Tpp = ADD_TO_TREE(C,F,f,S1);
  if (Tpp != NIL) {
     Sp = COMP(Tpp,Sp);
     Sp = INV(Sp);
     Sp = CONC(Sp,S);
     Tp = MK_TREE_NODE(D,g,Sp);
     goto Return; } }
       Tp = NIL;

Return: /* Prepare for return. */
       return(Tp);
}
```

CANONIZE

```
/*======================================================================
   Tp <- CANONIZE(T)

Canonical tree.
```

```
Input
  T : a topological tree.

Output
  Tp : the canonical tree with the same topology as T.
======================================================================*/
#include "code.h"

Word CANONIZE(T)
       Word T;
{
       Word S,S1,S2,f,ft,g,e,NewComp,NewSons,NewTree,
                                NewDict,BestTree,BestDict,C,s,sym;


Step1: /* Initialize */
       C = COMPONENT(T);
       g = CONTAINING_FACE(T);
       S = SONS(T);
       sym = 0;
       BestTree = NIL;
       f = face(C);

Step2: /* Recursively make canonical each son */
       NewSons = NIL;
       while (S != NIL) {
 ADV(S,&S1,&S);
 S2 = CANONIZE(S1);
 NewSons = COMP(S2,NewSons);  }
       S = INV(NewSons);
       /* CHECK for ERROR */

Step3: /* Test for empty component */

       if (f == NIL) {
         if (POINTS(C) == NIL) {
           NewDict = NIL; }
         else {
           NewDict = LIST1(MK_ITEM(1,LIST3(FIRST(POINTS(C)),0,0))); }
         NewComp = MK_COMP_FACE(NIL,NewDict,NIL);
 NewSons = TREEORDER(S);
         BestTree = MK_TREE_NODE(NewComp,g,NewSons);

         sym = 0; }


Step4: /* For each edge in FACE(COMPONENT(T)), compute canonical tree
  assuming that this edge is the good one */
       ft = f;
       while (ft != NIL) {
 ADV(ft,&e,&ft);
 NewComp = MAKE_DICT(C,e);
 NewDict = DICT(NewComp);
 NewSons = TRANSLATE_SONS(S,NewDict);
 NewSons = TREEORDER(NewSons);
 NewTree = MK_TREE_NODE(NewComp,g,NewSons);

         if (BestTree == NIL) {
   BestTree = NewTree;
   BestDict = NewDict;
   sym = 1; }
         else {
           s = TREECOMP(NewTree,BestTree);
             sym = 1;
     BestTree = NewTree;
     BestDict = NewDict; }
   else {
             if (s==0) {
               sym++;
     } } } }


Step5: /* Insert symmetry information */

       NewComp = COMPONENT(BestTree);
       g = CONTAINING_FACE(BestTree);
       S = SONS(BestTree);
       BestTree = MK_TREE_SYM(NewComp,g,S,sym);


Return: /* Prepare for return. */
       return(BestTree);
}
```

CFACE

```
/*===========================================================================
                           f <- CFACE(C,D)

Containing face.

Inputs
  C, D : components such that D contains C.

Outputs
  f : a face, the smallest face of D containing C.
===========================================================================*/
#include "code.h"

Word CFACE(C,D)
       Word C,D;
{
       Word E,Ep,E0,E1,H0,H1,H2;
       Word b0,b1,b2,f,m1,p,pi,pj,p0,p1,p2,q,s,si,sj;

Step1: /* Initialize. */
       p = FIRST(POINTS(C));
       pi = FIRST(p);
       pj = SECOND(p);
       E = EDGES(D);

Step2: /* D is empty and unbounded. */
       if (E == NIL) {
  f = NIL;
  goto Return; }
```

```
Step3: /* Look for the starting edge. */
       Ep = E;
       E0 = NIL;
       q = NIL;
       while (Ep != NIL) {
          ADV(Ep,&E1,&Ep);
  if (BRANCH(HALF_EDGE2(E1)) == 1) {
     s = POINT2(E1);
     si = FIRST(s);
     sj = SECOND(s);
     if (pi == si && pj < sj)
if (q == NIL) {
   E0 = CINV(E1);
   q = s; }
else
   if (sj < SECOND(q)  ) {
      E0 = CINV(E1);
      q = s; } } }

       if (E0 == NIL) {
  /* D is the unbounded component.
     Look for the topmost left branch to infinity. */
  Ep = E;
  while (Ep != NIL) {
     ADV(Ep,&E1,&Ep);
     if (EQUAL(POINT(HALF_EDGE1(E1)),INFTY_PT))
if ((E0 == NIL) ||
    (BRANCH(HALF_EDGE1(E0)) > BRANCH(HALF_EDGE1(E1)))) {
   E0 = E1; } } }

       if (E0 == NIL) {
  /* No left branches to infinity.
     Look for the first (bottom) right branch to infinity. */
  Ep = E;
  while (Ep != NIL) {
     ADV(Ep,&E1,&Ep);
     H1 = HALF_EDGE2(E1);
     if (EQUAL(POINT(H1),INFTY_PT) && BRANCH(H1) == 1) {
E0 = CINV(E1);
break; } } }

       H0 = HALF_EDGE1(E0);
       H1 = HALF_EDGE2(E0);
       p0 = POINT(H0);
       b0 = BRANCH(H0);
       p1 = POINT(H1);
       b1 = BRANCH(H1);
       m1 = MULTIPLICITY(H1);
       b1 = (b1 % m1) + 1;
       f = LIST1(H0);

Step4: /* Loop through edges. */
       Ep = E;
       while (!EQUAL(p0,p1) || !EQUAL(b0,b1)) {
  Ep = E;
  while (1) {
     ADV(Ep,&E1,&Ep);

     H2 = HALF_EDGE1(E1);
     p2 = POINT(H2);
     b2 = BRANCH(H2);
     if (EQUAL(p1,p2) && EQUAL(b1,b2)) {
f = COMP(H2,f);
H1 = HALF_EDGE2(E1);
p1 = POINT(H1);
b1 = BRANCH(H1);
m1 = MULTIPLICITY(H1);
b1 = (b1 % m1) + 1;
break; }

     H2 = HALF_EDGE2(E1);
     p2 = POINT(H2);
     b2 = BRANCH(H2);
     if (EQUAL(p1,p2) && EQUAL(b1,b2)) {
f = COMP(H2,f);
H1 = HALF_EDGE1(E1);
p1 = POINT(H1);
b1 = BRANCH(H1);
m1 = MULTIPLICITY(H1);
b1 = (b1 % m1) + 1;
break; } } }
       f = INV(f);

Return: /* Prepare for return, */
       return(f);
}
```

CODE

```
/*======================================================================
     C <- CODE(T)

Input
  T : a list of the left-and-right branch iformation.

Output
  C : a pair (E,P), where E is the list of edges with the branches
      numbered and P is the list of points.
======================================================================*/
#include "code.h"

Word CODE(T)
       Word T;
{
       Word C,E,H,L,Lp,Lt,P,R,Rp,Rt,T1,Tp,a,b,h,h1,h2,i,j,k,p,s,t;

Step1: /* Initialize. */
       H = NIL;
       P = NIL;
       Tp = T;

Step2: /* Count number of half edges off to the left. */
       T1 = FIRST(Tp);
       L = FIRST(T1);
       s = 0;
       while (L != NIL) {
```

```
  ADV(L,&a,&L);
  s += a; }

Step3: /* Get the half edges not at infinity. */
       i = 0;
       while (Tp != NIL) {
  ADV(Tp,&T1,&Tp);
  ADV2(T1,&L,&R,&T1);
  i++;
  L = Lt = RED(L);
  R = Rt = RED(R);
  j = 0;
  Lp = NIL;
  while (RED(L) != NIL) {
     ADV(L,&a,&L);
     ADV(R,&b,&R);
     j++;
     p = MK_POINT(i,j);
     P = COMP(p,P);
     if (a >= 1)
       for (k = 1; k <= a; k++) {
  h = MK_HALF_EDGE(p,k,a+b);
  Lp = COMP(h,Lp); }
     else {
h = MK_HALF_EDGE(p,0,a+b);
Lp = COMP(h,Lp); } }
  Lp = INV(Lp);
  L = Lt;
  R = Rt;
  j = 0;
  Rp = NIL;
  while (RED(R) != NIL) {
     ADV(L,&a,&L);
     ADV(R,&b,&R);
     j++;
     p = MK_POINT(i,j);
     if (b >= 1)
for (k = a+b; k >= a+1; k--) {
   h = MK_HALF_EDGE(p,k,a+b);
   Rp = COMP(h,Rp); }
     else {
h = MK_HALF_EDGE(p,0,a+b);
Rp = COMP(h,Rp); } }
  Rp = INV(Rp);
  H = COMP2(Rp,Lp,H); }

Step4: /* Count number of half-edges off to the right. */
       t = 0;
       while (Rt != NIL) {
  ADV(Rt,&b,&Rt);
  t += b; }

Step5: /* Append the half-edges at infinity. */
       p = MK_POINT(0,1);
       Rp = NIL;
       if (t >= 1)
  for (k = t; k >= 1; k--) {
     h = MK_HALF_EDGE(p,k,s+t);
     Rp = COMP(h,Rp); }
```

```
       else {
  h = MK_HALF_EDGE(p,0,s+t);
  Rp = COMP(h,Rp); }
       H = COMP(Rp,H);
       H = INV(H);
       Lp = NIL;
       if (s >= 1)
  for (k = t+1; k <= t+s; k++) {
     h = MK_HALF_EDGE(p,k,s+t);
     Lp = COMP(h,Lp); }
       else {
  h = MK_HALF_EDGE(p,0,s+t);
  Lp = COMP(h,Lp); }
       H = COMP(Lp,H);
       P = INV(P);
       P = COMP(p,P);

Step6: /* Combine half-edges into edges. */
       E = NIL;
       while (H != NIL) {
  ADV2(H,&L,&R,&H);
  while (L != NIL) {
     do
if (L != NIL)
   ADV(L,&h1,&L);
     while (BRANCH(h1) == 0 && L != NIL);
     do
if (R != NIL)
   ADV(R,&h2,&R);
     while (BRANCH(h2) == 0 && R != NIL);
     if (SECOND(h1) != 0)
E = COMP(MK_EDGE(h1,h2),E); } }
       E = INV(E);
       C = LIST2(E,P);

Return: /* Prepare for return. */
       return(C);
}
```

CONTAINS

```
/*======================================================================
  t <- CONTAINS(F,C)

Inputs
  F, C : components.

Output
  t : 0 or 1.  If F contains C then t = 1, otherwise t = 0.
======================================================================*/
#include "code.h"

Word CONTAINS(F,C)
       Word F,C;
{
       Word e,f,f1,f2,i,j,i2,j2,p,pi,pj,t,E;
Step1: /* Is F the unbounded component? */
       if (IS_UNBOUNDED(F)) {
  t = 1;
  goto Return; }

Step2: /* Initialize. */
       f = face(F);
       E = EDGES(F);
       p = FIRST(POINTS(C));
       pi = FIRST(p);
       pj = SECOND(p);
       e = 0;

Step3: /* Check if f surrounds p. */
       while (f != NIL) {
  ADV(f,&f1,&f);
  i = FIRST(POINT(f1));
  j = SECOND(POINT(f1));
  if (i == pi && j > pj) {
    TAKE_EDGE(f1,E,&f2,&E);
    if (FIRST(POINT(f2)) < pi) {
      e++; } }
  if (i == pi - 1) {
    TAKE_EDGE(f1,E,&f2,&E);
    if ((FIRST(POINT(f2)) == pi) && (SECOND(POINT(f2)) > pj) ) {
      e++; } }
       }
       if ((e % 2) == 1)
  t = 1;
       else
  t = 0;

Return: /* Prepare for return. */
       return(t);
}
```

FACE

```
/*======================================================================
     F <- FACE(C)

Face.

Input
  C : a component (E,P), where E is a list of edges and P is
      the corresponding list of points.

Output
  F : the component (E,P,f), where f is the outer face.
======================================================================*/
#include "code.h"

Word FACE(C)
       Word C;
{
       Word E,Ep,E0,E1,H0,H1,H2,P,b0,b1,b2,f,p0,m1,p1,p2;
```

```
Step1: /* Initialize. */
       E = EDGES(C);
       P = POINTS(C);
       p0 = FIRST(P);
       b0 = 1;

Step2: /* The component is an isolated point. */
       if (E == NIL) {
  f = NIL;
  goto Return; }

Step3: /* Find the first edge. */
       Ep = E;
       while (1) {
  ADV(Ep,&E1,&Ep);
  p1 = POINT1(E1);
  b1 = BRANCH(HALF_EDGE1(E1));
  if (EQUAL(p0,p1) && EQUAL(b0,b1)) {
     E0 = E1;
     H0 = HALF_EDGE1(E0);
     b0 = BRANCH(H0);
     H1 = HALF_EDGE2(E0);
     p1 = POINT(H1);
     b1 = BRANCH(H1);
     m1 = MULTIPLICITY(H1);
     b1 = (b1 % m1) + 1;
     f = LIST1(H0);
     break; } }

Step4: /* Loop through the edges. */
       while (!EQUAL(p0,p1) || !EQUAL(b0,b1)) {
  Ep = E;
  while (1) {
     ADV(Ep,&E1,&Ep);
     H2 = HALF_EDGE1(E1);
     p2 = POINT(H2);
     b2 = BRANCH(H2);
     if (EQUAL(p1,p2) && EQUAL(b1,b2)) {
f = COMP(H2,f);
H1 = HALF_EDGE2(E1);
p1 = POINT(H1);
b1 = BRANCH(H1);
m1 = MULTIPLICITY(H1);
b1 = (b1 % m1) + 1;
break; }

     H2 = HALF_EDGE2(E1);
     p2 = POINT(H2);
     b2 = BRANCH(H2);
     if (EQUAL(p1,p2) && EQUAL(b1,b2)) {
f = COMP(H2,f);
H1 = HALF_EDGE1(E1);
p1 = POINT(H1);
b1 = BRANCH(H1);
m1 = MULTIPLICITY(H1);
b1 = (b1 % m1) + 1;
break; } } }
       f = INV(f);
```

```
Return: /* Prepare for return. */
       return(f);
}
```

FACES

```
/*==========================================================================
    F <- FACES(C)

Compute the outer faces of the components.

Input
  C : a list ((E_1,P_1),(E_2,P_2)...,(E_n,P_n)) of components,
      where (E_1,P_1) is the infinity component.

Output
  F : the list ((E_1,P_1,f_1),(E_2,P_2,f_2),..,(E_n,P_n.f_n)) of components
      with outer face information. Each f_i is a list of half-edges in the
      outer face, ordered cyclically and only the first half-edge of each
      edge. For the infinity component f_1 = ( ((0,1),1,m) , ... ,
      ((0,1),m,m) where m is the multiplicity of the infinity point.
==========================================================================*/
#include "code.h"

Word FACES(C)
       Word C;
{
       Word C1,Cp,F,F1,f,E,e,t,h1,h2;

Step1: /* For the infinity component, we will take the (ordered) list of
          halfedges coming from infinity as "outer face". */
       Cp = C;
       ADV(Cp,&C1,&Cp);
       E = EDGES(C1);
       f = NIL;
       while (E != NIL) {
 ADV(E,&e,&E);
 h1 = HALF_EDGE1(e);
 h2 = HALF_EDGE2(e);
 if (EQUAL(POINT(h1),INFTY_PT)) {
   PINSERT(h1,f,&f,&t); }
 if (EQUAL(POINT(h2),INFTY_PT)) {
   PINSERT(h2,f,&f,&t); }
       }
       F1 = MK_COMP_FACE(EDGES(C1),POINTS(C1),f);
       F = LIST1(F1);

Step2: /* Loop. */
       while (Cp != NIL) {
 ADV(Cp,&C1,&Cp);
 f = FACE(C1);
 F1 = MK_COMP_FACE(EDGES(C1),POINTS(C1),f);
 F = COMP(F1,F); }
       F = INV(F);

Return: /* Prepare for return. */
       return(F);
}
```

FATHER

```
/*======================================================================
   F <- FATHER(C,T)

Father.

Inputs
  C : a component (E,P,f).
  T : a tree of components, none of which is a topological descendant of C.

Output
  F : a component, the father of C, or NIL if the father of C is not in T.
======================================================================*/
#include "code.h"

Word FATHER(C,T)
       Word C,T;
{
       Word F,F1,S,S1;

Step1: /* Initialize. */
       F = COMPONENT(T);
       S = SONS(T);

Step2: /* Look for father. */
       if (CONTAINS(F,C)) {
  F1 = NIL;
  while (S != NIL && F1 == NIL) {
     ADV(S,&S1,&S);
     F1 = FATHER(C,S1); }
  if (F1 != NIL)
     F = F1; }
       else
  F = NIL;

Return: /* Prepare for return. */
       return(F);
}
```

GETCOMPONENT

```
/*======================================================================
c <- GETCOMPONENT(p,C)

Get component.

Inputs
  p : a point.
  C : a list (C_1,...,C_k) of components.

Output
  c : a BETA-digit.  If p belongs to C_i then c = i.
```

```
      If p belongs to no C_i, then c = 0.
=========================================================================*/
#include "code.h"

Word GETCOMPONENT(p,C)
       Word p,C;
{
       Word Cp,C1,P,c,i;

Step1: /* Initialize. */
       Cp = C;
       c = 0;
       i = 0;

Step2: /* Loop through the components. */
       while (Cp != NIL) {
  ADV(Cp,&C1,&Cp);
  i++;
  P = POINTS(C1);
  if (MEMBER(p,P)) {
     c = i;
     goto Return; } }

Return: /* Prepare for return. */
#if 0
       SWRITE("The component of point ");
       OWRITE(p);
       SWRITE(" is ");
       IWRITE(c);
       SWRITE(".\n");
#endif
       return(c);
}
```

HHCODE

```
#include "code.h"

Word HHCODE(T)
       Word T;
{
       Word C,L1,R1,T1,Tp;

Step1: /* Initialize. */
       Tp = T;
       C = NIL;

Step2: /* Get branch information. */
       while (Tp != NIL) {
 ADV(Tp,&T1,&Tp);
 ADV2(T1,&L1,&R1,&T1);
 C = COMP(LIST2(L1,R1),C); }
       C = INV(C);

Return: /* Prepare for return. */
```

```
       return(C);
}
```

LEXCOMP

```
/*======================================================================
  s <- LEXCOMP(A,B)

Lexicographic comparison.

Inputs
  A,B : words.

Output
  s : -1, 0, or 1.  If A = B then s = 0.  If A comes before B in
      lexicographic order then s = -1.  Otherwise, s = 1.
      Uses recursion in the case that the elements a of A and b of B
      to be compared are both lists.
======================================================================*/
#include "code.h"

Word LEXCOMP(A,B)
       Word A,B;
{
       Word Ap,Bp,a,b,s;

Step1: /* Initialize. */
       Ap = A;
       Bp = B;
       s = 0;

Step2: /* Compare. */
       while (Ap != NIL) {
 if (Bp == NIL) {
   s = 1;
   goto Return; }
 ADV(Ap,&a,&Ap);
 ADV(Bp,&b,&Bp);
 if ((a < BETA) || (b < BETA)) {
   if (a < b)
     s = -1;
   if (a > b)
     s = 1; }
 else
           s = LEXCOMP(a,b);
 if (s != 0)
   goto Return; }
       if (Bp != NIL)
 s = -1;

Return: /* Prepare for return. */

       return(s);
}
```

LEX_ORDER

```
/*======================================================================
  Lp <- LEX_ORDER(L)

Inputs
  L: an object (supposed list).

Output
  Lp: same list, with elements ordered lexicographically.
======================================================================*/
#include "code.h"

Word LEX_ORDER(L)
       Word L;
{
       Word Lp, Lt, p,q,s;

Step1: /* Initialize and do nothing if L has one only element */

if ((LENGTH(L) < 2) || (L < NIL) ) {
Lp = L;
goto Return; }
Lp = NIL;

Step2:  /* Take first element, order the rest and insert */

        ADV(L,&p,&Lt);
Lt = LEX_ORDER(Lt);
       while (Lt != NIL) {
 ADV(Lt,&q,&Lt);
 s = LEXCOMP(p,q);
 if (s <= 0) {
   Lp = COMP2(q,p,Lp);
   Lp = INV(Lp);
   Lp = CONC(Lp,Lt);
   goto Return; }
 Lp = COMP(q,Lp); }
       Lp = COMP(p,Lp);
       Lp = INV(Lp);

Return: /* Prepare for return. */
       return(Lp);
}
```

MAKE_DICT

```
/*======================================================================
      NewC <- MAKE_DICT(C,e)

Partner edge.

Inputs
  C : a component (E,P,f).
  e : a starting half-edge for the Dictionary.

Output
```

```
  NewC : New code (NewE,Dict,NewF) obtained translating C
to be canonical, assuming that e is the right starting edge.

Note: In the output Dict takes the place of the list of points. Actually,
each element in Dict is of the form (n,h) where n is the name of a point
and h is the old code for the starting branch of point n.
======================================================================*/
#include "code.h"

Word MAKE_DICT(C,e)
       Word C,e;
{
       Word NewC,Dict,E,Points,f,hOld1,hOld2,hNew1,hNew2,item,NewE,
                                                NewF,n,b,m,p,bb,HighPoint;



Step1: /* Initialize */

       E = EDGES(C);
       Points = POINTS(C);
       f = face(C);
       if ((e == NIL) && (Points != NIL)) {
 Dict = LIST1(MK_ITEM(1,MK_HALF_EDGE(FIRST(Points),0,0)));
 NewE = NIL; }
       hOld1 = e;
       hNew1 = MK_HALF_EDGE(1,1,MULTIPLICITY(hOld1));
       HighPoint = 1;
       item = MK_ITEM(1,hOld1);
       Dict = LIST1(item);
       NewE = NIL;

Step2: /* Loop. Constructs the Dictionary and the list of NewEdges  */

       while (E != NIL) {
 hOld2 = NIL;
 hNew2 = NIL;
 TAKE_EDGE(hOld1,E,&hOld2,&E);
 if (hOld2 != NIL) {
           hNew2 = TRANSLATE_ITEM(hOld2,Dict);
   if (hNew2 == NIL) {
     HighPoint = HighPoint + 1;
     hNew2 = MK_HALF_EDGE(HighPoint,1,MULTIPLICITY(hOld2));
     Dict = COMP(MK_ITEM(HighPoint,hOld2),Dict); }
   NewE = COMP(MK_EDGE(hNew1,hNew2),NewE); }
 n = POINT(hNew1);
 b = BRANCH(hNew1);
 m = MULTIPLICITY(hNew1);
 if (b < m) {
   hNew1 = MK_HALF_EDGE(n,b+1,m);
   p = POINT(hOld1);
   bb = BRANCH(hOld1);
   hOld1 = MK_HALF_EDGE(p,( bb % m) +1,m); }
 else {
   if (n < HighPoint) {
     item = LELTI(Dict,HighPoint-n);
     hOld1 = OLD(item);
     hNew1 = MK_HALF_EDGE(n+1,1,MULTIPLICITY(hOld1)); }
   else
```

```
     SWRITE("\n ERROR in function MAKE_DICT\n"); }
       }
       Dict = INV(Dict);
       NewE = INV(NewE);

Step3: /* Compute New outer face and glue things together */

       NewF = TRANSLATE_FACE(f, Dict);
       NewC = MK_COMP_FACE(NewE,Dict,NewF);


Return: /* Prepare for return, */

       return(NewC);
}
```

PARTNER_EDGE

```
/*======================================================================
      PARTNER_EDGE(H1,E; H2,Ep)

Partner edge.

Inputs
  H1 : a half-edge whose point has multiplicity 1.
  E : a list of edges that contains an edge the point of one of
      whose half-edges is the point of H1.

Outputs
  H2 : the half-edge in E whose other half-edge has point the same as
       that of H1.
  Ep : a list of edges, the list obtained from E by removing the edge
       of which H2 is a half-edge.  The order in E is maintained.
======================================================================*/
#include "code.h"

void PARTNER_EDGE(H1,E, H2_,Ep_)
       Word H1,E, *H2_,*Ep_;
{
       Word Ep,Et,E1,H2,p;

Step1: /* Initialize. */
       H2 = NIL;
       Ep = NIL;
       Et = E;
       p = POINT(H1);

Step2: /* Look for H2. */
       while (Et != NIL) {
  ADV(Et,&E1,&Et);
  if (EQUAL(p,POINT1(E1))) {
     H2 = HALF_EDGE2(E1);
     break; }
  else if (EQUAL(p,POINT2(E1))) {
     H2 = HALF_EDGE1(E1);
     break; }
  else
```

```
     Ep = COMP(E1,Ep); }
       Ep = INV(Ep);
       Ep = CONC(Ep,Et);

Return: /* Prepare for return, */
       *H2_ = H2;
       *Ep_ = Ep;
       return;
}
```

PDELETE

```
/*======================================================================
   Q = PDELETE(p,P)

Point delete.

Inputs
  p : a point (i.e., a pair of nonnegative BETA-digits).
  P : a list of points in lexicographic order.

Outputs
  Q : a list of points in lexicographic order.  Q is the list obtained
      by deleting the point p from the list P.  If p is not in P,
      then Q = P.
======================================================================*/
#include "code.h"

Word PDELETE(p,P)
       Word p,P;
{
       Word Pp,Q,q,s;

#if 0
       SWRITE("\np = "); OWRITE(p); SWRITE("\n");
       SWRITE("P = "); OWRITE(P); SWRITE("\n");
#endif

Step1: /* Initialize. */
       Pp = P;
       Q = NIL;

Step2: /* Loop. */
       while (Pp != NIL) {
 ADV(Pp,&q,&Pp);
 s = LEXCOMP(p,q);
 if (s < 0) {
   Q = P;
   goto Return; }
 if (s == 0) {
   Q = INV(Q);
   Q = CONC(Q,Pp);
   goto Return; }
 Q = COMP(q,Q); }
       Q = P;

Return: /* Prepare for return. */
#if 0
       SWRITE("Q = "); OWRITE(Q); SWRITE("\n");
#endif
       return(Q);
}
```

PINSERT

```
/*======================================================================
  PINSERT(p,L; Lp,t)

Point insert.

Inputs
  p : a point (i.e., a pair of nonnegative BETA-digits).
  L : a list of points in lexicographic order.

Outputs
  Lp : a list of points in lexicographic order.  If p is in L,
       Lp = L.  Otherwise, Lp is the list obtained by inserting
       p in L.
  t : 0 or 1.  If p is in L, then t = 1, otherwise t = 0.
======================================================================*/
#include "code.h"

void PINSERT(p,L, Lp_,t_)
       Word p,L, *Lp_,*t_;
{
       Word Lp,Lt,q,s,t;

Step1: /* Initialize. */
       Lp = NIL;
       Lt = L;
       t = 0;

Step2: /* Insert in proper place. */
       while (Lt != NIL) {
 ADV(Lt,&q,&Lt);
 s = LEXCOMP(p,q);
 if (s == 0) {
   t = 1;
   Lp = L;
   goto Return; }
 if (s < 0) {
   Lp = COMP2(q,p,Lp);
   Lp = INV(Lp);
   Lp = CONC(Lp,Lt);
   goto Return; }
 Lp = COMP(q,Lp); }
       Lp = COMP(p,Lp);
       Lp = INV(Lp);

Return: /* Prepare for return. */

       *Lp_ = Lp;
       *t_ = t;
       return;
}
```

REM_REG

```
/*======================================================================
       REM_REG(E,P; Ep,Pp)

Remove regular points.

Input
  E : a list of edges.
  P : a list of points.

Output
  Ep : a list of edges, the list obtained from E by removing the
       regular points.
  Pp : a list of points, the list obtained from P by removing the
       regular points.
======================================================================*/
#include "code.h"

void REM_REG(E,P, Ep_,Pp_)
       Word E,P, *Ep_,*Pp_;
{
       Word Ep,Et,E1,E2,H1,H2,H3,Pp,p1,p2;

Step1: /* Initialize. */
       Et = E;
       Ep = NIL;
       Pp = P;

Step2: /* Loop. */
       while (Et != NIL) {
  ADV(Et,&E1,&Et);
  H1 = HALF_EDGE1(E1);
  p1 = POINT(H1);
  H2 = HALF_EDGE2(E1);
  p2 = POINT(H2);
  if (MULTIPLICITY(H2) == 2 && !EQUAL(p2,INFTY_PT)) {
     if (!EQUAL(p1,p2)) {
PARTNER_EDGE(H2,Et,&H3,&Et);
E2 = MK_EDGE(H1,H3);
Et = COMP(E2,Et); }
     Pp = PDELETE(p2,Pp); }
  else if (MULTIPLICITY(H1) == 2 && !EQUAL(p1,INFTY_PT)) {
     if (!EQUAL(p1,p2)) {
PARTNER_EDGE(H1,Et,&H3,&Et);
E2 = MK_EDGE(H2,H3);
Et = COMP(E2,Et); }
     Pp = PDELETE(p1,Pp); }
  else
     Ep = COMP(E1,Ep); }
       Ep = INV(Ep);

Return: /* Prepare for return. */
       *Ep_ = Ep;
       *Pp_ = Pp;
       return;
}
```

REM_REG_OR

```
/*========================================================================
       Hp <- REM_REG_OR(H)

Remove regular points from an oriented edge list.

Input
  H : a list of edges that is oriented.

Output
  Hp : a list of edges, the list obtained from H by removing the
       regular points.
========================================================================*/
#include "code.h"

Word REM_REG_OR(H)
       Word H;
{
       Word Hp,Ht,H1,p;

Step1: /* Initialize. */
       Ht = H;
       Hp = NIL;

Step2: /* Look for a singular point.
       while (Ht != NIL) {
  ADV(Ht,&H1,&Ht);
  if (EQUAL(POINT(H1),INFTY_PT) || MULTIPLICITY(H1) != 2) {
     Ht = COMP(H1,Ht);
     Ht = CONC(Ht,INV(Hp));
     break; }
  else
     Hp = COMP(H1,Hp); }
       Hp = NIL;   */

Step3: /* Loop. */
       while (Ht != NIL) {
 ADV(Ht,&H1,&Ht);
 p = POINT(H1);
 if (MULTIPLICITY(H1) != 2 || EQUAL(p,INFTY_PT)) {
   Hp = COMP(H1,Hp); } }
       Hp = INV(Hp);

Return: /* Prepare for return. */
       return(Hp);
}
```

SEP_PROJ

```
/*========================================================================
      SEP_PROJ(D)
```

```
Separate connected components in the sphere.

Input
  D : the output of CODE().

Output
  C : a list ((E_1,P_1),...,(E_k,P_k)) of the edges and points of
      the connected components on the sphere.
====================================================================*/
#include "code.h"

Word SEP_PROJ(D)
       Word D;
{
       Word C,E,Ep,Et,E1,P,P1,Q1,e,p,p1,p2,t;

Step1: /* Initialize. */
       E = EDGES(D);
       P = POINTS(D);
       C = NIL;

Step2: /* Get a point. */
       if (P != NIL)
  ADV(P,&p,&P);
       else {
  C = INV(C);
  goto Return; }

Step3: /* Initialize connected component of p. */
       P1 = LIST1(p);
       Q1 = LIST1(p);
       E1 = NIL;

Step4: /* Have we looked at all the new points? */
       if (Q1 == NIL) {
  E1 = INV(E1);
  C = COMP(MK_COMPONENT(E1,P1),C);
  goto Step2; }
       else
 ADV(Q1,&p1,&Q1);

Step5: /* Look for points adjacent to p1. */
       Ep = E;
       Et = NIL;
       while (Ep != NIL) {
  ADV(Ep,&e,&Ep);
  if (!EQUAL(p1,POINT1(e)) && !EQUAL(p1,POINT2(e)))
     Et = COMP(e,Et);
 else {
    if (EQUAL(p1,POINT1(e)))
       p2 = POINT2(e);
    else
       p2 = POINT1(e);
    PINSERT(p2,P1,&P1,&t);
    if (t == 0) {
       Q1 = COMP(p2,Q1);
       P = PDELETE(p2,P); }
    E1 = COMP(e,E1); } }
```

```
       E = INV(Et);
       goto Step4;

Return: /* Prepare for return. */
       return(C);
}
```

SIMPLIFY

```
/*======================================================================
  Tp <- SIMPLIFY(T)

Simplify.

Input
  T : a topological tree.

Output
  Tp : a topological tree, the tree obtained from T by deleting
       the regular points.
======================================================================*/
#include "code.h"

Word SIMPLIFY(T)
       Word T;
{
       Word C,Cp,E,Ep,P,Pp,S,Sp,S1,S1p,Tp,f,fp,g,gp;

Step1: /* Decompose T. */
       C = COMPONENT(T);
       g = CONTAINING_FACE(T);
       S = SONS(T);

Step2: /* Remove regular points from the component. */
       E = EDGES(C);
       P = POINTS(C);
       REM_REG(E,P,&Ep,&Pp);
       f = face(C);
       if (IS_UNBOUNDED(C)) {
         fp = f; }
       else {
         fp = REM_REG_OR(f); }
       Cp = MK_COMP_FACE(Ep,Pp,fp);

Step3: /* Remove regular points from the containing face. */
       gp = REM_REG_OR(g);

Step4: /* Simplify the sons. */
       Sp = NIL;
       while (S != NIL) {
  ADV(S,&S1,&S);
  S1p = SIMPLIFY(S1);
  Sp = COMP(S1p,Sp); }
       Sp = INV(Sp);

Step5: /* Reconstruct the tree node. */
       Tp = MK_TREE_NODE(Cp,gp,Sp);
Return: /* Prepare for return. */
       return(Tp);
}
```

TAKE_EDGE

```
/*======================================================================
TAKE_EDGE(h,E;&ht,&Et)


Find an edge in a list edge E containing h and remove it.

Inputs
  E : a list of edges.
  h : a halfedge.

Outputs
  ht : the other half-edge of the edge containing h (NIL if none).
  Et : E with the edge (h,ht) or (ht,h) removed. (E if ht is NIL).
======================================================================*/
#include "code.h"

void TAKE_EDGE(h,E, ht_,Et_)
       Word h, E, *Et_,*ht_;
{
       Word h1,h2,ht,Ep,Et,e;

Step1: /* Initialize */

       Ep = E;
       Et = NIL;
       ht = NIL;

Step2: /* Find edge containing h and proceed */
       while ((Ep != NIL) && (ht == NIL)) {
 ADV(Ep,&e,&Ep);
 h1 = HALF_EDGE1(e); h2 = HALF_EDGE2(e);
 if ( EQUAL(h,h1) ) {
   ht = h2; }
 else {
   if ( EQUAL(h,h2) ) {
     ht = h1; }
   else {
     Et = COMP(e,Et); } } }
       Et = INV(Et);
       Et = CONC(Et,Ep);


Step3: /* Check for ERROR

if ( ht == NIL ) {
  SWRITE("\n ERROR in function TAKE_EDGE:");
  SWRITE("\nList:");OWRITE(E);
  SWRITE("\nHalf-edge:");OWRITE(h);SWRITE("\n");
  *Et_ = Et;
  *ht_ = ht;
  return; }  */



Return: /* Prepare for return. */
       *Et_ = Et;
       *ht_ = ht;
       return;
}
```

TRANSLATE_FACE

```
/*===========================================================================
  gt <- TRANSLATE_FACE(g,Dict)

Inputs
  g: a list of half-edges for a certain component C.
  Dict: A dictionary, saying how to translate half-edges of Edges to a
       new code.


Output
  gt: New list of edges, cyclically reordered to be minimal.
===========================================================================*/
#include "code.h"

Word TRANSLATE_FACE(g,Dict)
       Word g, Dict;
{

Word e,gbeg,gend,gt,gp,e1,h1,h2;


Step1: /* Translate each half-edge, put in a list and keep */

gt = NIL;
gp = g;
while (gp!= NIL) {
  ADV(gp,&e,&gp);
  e1 = TRANSLATE_ITEM(e,Dict);
  gt = COMP(e1,gt); }
gt = INV(gt);


Step2: /* Cyclically permute, to be minimal */

e1 = NIL;
gend = NIL;
gbeg = NIL;
if (gt != NIL) {
  ADV(gt,&e,&gt);
  e1=e;
  gbeg=LIST1(e); }
while (gt != NIL) {
  ADV(gt,&e,&gt);
  if (LEXCOMP(e1,e) != 1) {
    gbeg = COMP(e,gbeg); }
  else {
    gend = CCONC(gbeg,gend);
    gbeg = LIST1(e);
    e1 = e;
  }
}
gt = INV(CCONC(gend,gbeg));



Return: /* Prepare for return */
return(gt);

}
```

TRANSLATE_ITEM

```
/*===========================================================================
  hNew <- TRANSLATE_ITEM(h,Dict)

Inputs
  h: a halfedge, in old code.
  Dict: A dictionary, saying how to translate half-edges to a
       new code. A Dictionary is a list of items, each item having the
structure (n,h) with:
h: a certain half-edge of the old code, which is
supposed to heve index 1 in the new code.
n: the label of the point at half-edge h, in new code.


Output
  hNew: Name of h, in the new code.
===========================================================================*/
#include "code.h"

Word TRANSLATE_ITEM(h,Dict)
       Word h, Dict;
{

Word hNew,D,item,p,h0,m,n,b;


Step1: /* Look for the item containing the point of h, in the dictionary */
D = Dict;
        item = NIL;
h0 = NIL;
p = POINT(h);
while ((D != NIL) && (!EQUAL(p,POINT(h0))) ) {
  ADV(D,&item,&D);
          h0 = OLD(item); }

Step2: /* Check for ERROR */

if (!EQUAL(p,POINT(h0))) {
  hNew = NIL;
  return(NIL); }
Step3: /* Compute branch and make half-edge */

m = MULTIPLICITY(h);
n = LABEL(item);
b =((BRANCH(h) + m - BRANCH(h0)) % m ) + 1;
hNew = MK_HALF_EDGE(n,b,m);


Return: /* Prepare for return */
return(hNew);

}
```

TRANSLATE_SONS

```
/*======================================================================
  NewSons <- TRANSLATE_SONS(Sons,Dict)

Inputs
  Sons: list of sons for a certain component C.
  Dict: A dictionary, saying how to translate half-edges of Edges to a
       new code.


Output
  NewSons: New list of sons (only containing faces of sons change).
======================================================================*/
#include "code.h"

Word TRANSLATE_SONS(Sons,Dict)
       Word Sons, Dict;
{

Word St, S1, NewSons, g, C, S;


Step1: /* Translate containing faces of each of the sons,
                                              and make new list */

St = Sons;
NewSons = NIL;
while (St != NIL) {
ADV(St,&S1,&St);
g = CONTAINING_FACE(S1);
C = COMPONENT(S1);
S = SONS(S1);
g = TRANSLATE_FACE(g,Dict);
                S1 = MK_TREE_NODE(C,g,S);
NewSons = COMP(S1,NewSons); }
NewSons = INV(NewSons);


Step2:  /* CHECK for ERROR */
if (LENGTH(Sons) != LENGTH(NewSons) ) {
SWRITE("\n ERROR in FUNCTION TRANSLATE_SONS \n");
OWRITE(Sons);SWRITE("\n");
OWRITE(NewSons);SWRITE("\n"); }
Return: /* Prepare for return */
return(NewSons);

}
```

TREE

```
/*======================================================================
     T <- TREE(F)

Compute the topological tree.

Input
  F : a list of components ((E_1,P_1,f_1),(E_2,P_2,f_2),...,(E_n,P_n,f_n)),
      where (E_1,P_1,f_1) is the unbounded component, and for i > 1,
      E_i is a list of edges, P_i is the corresponding list
      of points, and f_i is the outermost face of the component
      (for the infinity component, f_1 is the list of edges coming
      from infinity).

Output
  T : a tree of the components.
======================================================================*/
#include "code.h"

Word TREE(F)
       Word F;
{
       Word C,Fp,T,f;

Step1: /* Initialize the tree. */
       T = MK_TREE_NODE(FIRST(F),NIL,NIL);
       Fp = RED(F);

Step2: /* Compute the tree. */
       while (Fp != NIL) {
  ADV(Fp,&C,&Fp);
  F = FATHER(C,T);           /* Compute father of component. */
  f = CFACE(C,F);            /* Compute the containing face. */
  T = ADD_TO_TREE(C,F,f,T);  /* Add son to the tree. */
       }

Return: /* Prepare for return. */
       return(T);
}
```

TREECOMP

```
/*======================================================================
  s <- TREECOMP(A,B)

Lexicographic comparison of Trees. The list points in
each component is not compared. It contains extra information.
```

```
Inputs
  A,B : Topological Trees.

Output
  s : -1, 0, or 1.  If A = B then s = 0.  If A comes before B in
      lexicographic order then s = -1.  Otherwise, s = 1.
      Uses recursion in the case that the elements a of A and b of B
      to be compared are both lists.
=========================================================================*/
#include "code.h"

Word TREECOMP(A,B)
       Word A,B;
{
       Word Ap,Bp,a,b,s;

Step1: /* Initialize and check if A or B are ovals. */
       s = 0;


Step2: /* Compare components (only edges, outer face and
                                                     number of points). */
       Ap = COMPONENT(A);
       Bp = COMPONENT(B);
       a = EDGES(Ap);
       b = EDGES(Bp);
       s = LEXCOMP(a,b);
       if (s != 0) {
 goto Return; }
       a = face(Ap);
       b = face(Bp);
       s = LEXCOMP(a,b);
       if (s != 0) { goto Return; }
       a = LENGTH(POINTS(Ap));
       b = LENGTH(POINTS(Bp));
       if (a < b) {
         s = -1;
 goto Return; }
       if (a > b) {
         s = 1;
 goto Return; }


Step3: /* Compare containing faces. */
       a = CONTAINING_FACE(A);
       b = CONTAINING_FACE(B);
       s = LEXCOMP(a,b);
       if (s != 0) { goto Return; }

Step4: /* Compare sons. */
       Ap = SONS(A);
       Bp = SONS(B);
       while (Ap != NIL) {
 if (Bp == NIL) {
   s = 1;
   goto Return; }
 ADV(Ap,&a,&Ap);
 ADV(Bp,&b,&Bp);
 if ((a < BETA) || (b < BETA)) {
   if (a < b)
     s = -1;
   if (a > b)
     s = 1; }
 else
           s = TREECOMP(a,b);
 if (s != 0)
   goto Return; }
       if (Bp != NIL)
 s = -1;

Return: /* Prepare for return. */

       return(s);
}
```

TREEORDER

```
/*======================================================================
  Lp <- TREEORDER(L)

Inputs
  L: a list of topological trees.

Output
  Lp: same list, with elements ordered tree-lexicographically (i.e.
taking only account of edges and containing faces at each node).
======================================================================*/
#include "code.h"

Word TREEORDER(L)
       Word L;
{
       Word Lp, Lt, p,q,s;

Step1: /* Initialize and do nothing if L has one only element */

if ((LENGTH(L) < 2) || (L < NIL) ) {
Lp = L;
goto Return; }
Lp = NIL;

Step2:  /* Take first element, order the rest and insert */

        ADV(L,&p,&Lt);
Lt = TREEORDER(Lt);
        while (Lt != NIL) {
 ADV(Lt,&q,&Lt);
 s = TREECOMP(p,q);
 if (s <= 0) {
   Lp = COMP2(q,p,Lp);
   Lp = INV(Lp);
   Lp = CONC(Lp,Lt);
   goto Return; }
 Lp = COMP(q,Lp); }
       Lp = COMP(p,Lp);
       Lp = INV(Lp);
Return: /* Prepare for return. */
       return(Lp);
}
```

TWRITE

```
/*======================================================================
      TWRITE(T,i)

Topological tree write.
input:
T: the tree to be written.
i: a counter of the depth we are in the big tree. Used to indent and
   increased by recursion.
======================================================================*/
#include "code.h"

void BLANKS(i)
       int i;
{
       SWRITE("\n");
       while (i!=0) {
 SWRITE(" ");
 i--; }
       return;
}



void TWRITE(T,i)
       Word T;
       int i;
{
       Word C,g,S,S1,P,sym;

Step1: /* Initialize. */
       C = COMPONENT(T);
       g = CONTAINING_FACE(T);
       S = SONS(T);
       sym = SYMMETRY(T);

Step2: /* Write the containing face. */
       if (i != 0) {
         BLANKS(6*i-3);
         SWRITE("Containing face: ");
         OWRITE(g); }


Step3: /* Write the component. */

       BLANKS(6*i);
       SWRITE("Cyclic symmetry: ");
       OWRITE(sym);

       BLANKS(6*i);
       SWRITE("Points: ");
       OWRITE(POINTS(C));

       BLANKS(6*i);
       SWRITE("Edges: ");
       OWRITE(EDGES(C));

       BLANKS(6*i);
       SWRITE("Outer Face: ");
       OWRITE(face(C));


Step4: /* Write the sons. */
       BLANKS(6*i);
       if (S == NIL)
 SWRITE("NO SONS");
       else
 SWRITE("Sons: ");
       while (S != NIL) {
 ADV(S,&S1,&S);
 TWRITE(S1,i+1); }

Return: /* Prepare for return. */
       SWRITE("\n");
       return;
}
```