

```

/*
 * fundamental_group.c
 *
 * This file exports the following functions to the UI:
 *
 *     GroupPresentation *fundamental_group(
 *         Triangulation *manifold,
 *         Boolean        simplify_presentation,
 *         Boolean        fillings_may_affect_generators,
 *         Boolean        minimize_number_of_generators);
 *
 *     int    fg_get_num_generators    (GroupPresentation *group);
 *     Boolean fg_integer_fillings     (GroupPresentation *group);
 *     int     fg_get_num_relations    (GroupPresentation *group);
 *     int     *fg_get_relation        (GroupPresentation *group,
 *                                     int                which_relation);
 *     int     fg_get_num_cusps        (GroupPresentation *group);
 *     int     *fg_get_meridian        (GroupPresentation *group,
 *                                     int                which_cusp);
 *     int     *fg_get_longitude       (GroupPresentation *group,
 *                                     int                which_cusp);
 *     void     fg_free_relation        (int                *relation);
 *
 *     void     free_group_presentation(GroupPresentation *group);
 *
 * The UI will call fundamental_group() to compute a GroupPresentation,
 * then make calls as needed to the fg_get.../fg_free... functions,
 * and finally call free_group_presentation() to release the memory.
 * The actual structure of a GroupPresentation is private to this file;
 * SnapPea.h contains an "opaque typedef" which lets the UI pass pointers
 * to GroupPresentations without knowing their internal structure.
 *
 * fundamental_group() computes the fundamental group of the manifold,
 * and returns a pointer to it. It takes into account Dehn fillings
 * with relatively prime integer coefficients, but ignores all
 * other Dehn fillings. The Boolean arguments correspond
 * to the fields in the GroupPresentation typedef below.
 *
 * fg_get_num_generators() returns the number of generators in the
 * GroupPresentation.
 *
 * fg_integer_fillings() says whether the space is a manifold or orbifold,
 * as opposed to some other generalized Dehn filling.
 *
 * fg_get_num_relations() returns the number of relations in the
 * GroupPresentation.
 *
 * fg_get_relation() returns the specified relation. Its allocate
 * the memory for it, so you should pass the pointer back to
 * fg_free_relation() when you're done with it.
 * Each relation is a string of integers. The integer 1 means the
 * first generator, 2 means the second, etc., while -1 is the inverse
 * of the first generator, -2 is the inverse of the second, etc.
 * The integer 0 indicates the end of the string.
 *
 * fg_get_num_cusps() returns the number of cusps of the underlying
 * manifold. This *includes* the filled cusps. So, for example,
 * if you do (5,1) Dehn filling on the figure eight knot complement,
 * you can see the words in the fundamental group corresponding to
 * the (former!) cusp's meridian and longitude.
 *
 * fg_get_meridian() and fg_get_longitude() return the word corresponding
 * to a meridian or longitude, in the same format used by
 * fg_get_relation() above. They allocate the memory for the string
 * of integers, so you should pass the pointer back to
 * fg_free_relation() when you're done with it. Meridians and
 * longitudes are available whether the cusps are filled or not as
 * explained for fg_get_num_cusps() above.
 *
 * fg_free_relation() frees relations allocated by fg_get_relation().
 *
 * free_group_presentation() frees the memory occupied by a GroupPresentation.
 */

```

```

/*
 * 96/9/13 There are two sets of generators kicking around:
 *
 *      (1) the "geometric generators" defined in choose_generators.c, and
 *
 *      (2) the "simplified generators" used by fundamental_group()
 *          in its final simplified presentation.
 *
 * The algorithm in representations.c needs to express the former as words
 * in the latter, so we now keep track of that information. As with
 * the expressions for meridians and longitudes, we use words with well
 * defined basepoints, and don't do any cyclic cancellations.
 *
 * 96/9/29 fundamental_group() nows records the basepoint of each Cusp
 * in the Cusp's basepoint_tet, basepoint_vertex and basepoint_orientation
 * fields. The basepoint is the point where the meridian and longitude
 * meet; the meridional and longitudinal words are computed relative to
 * this point to guarantee that they compute.
 */

/*
 * Visualizing the fundamental group.
 *
 * Sometimes we'll think of a presentation of a manifold's
 * fundamental group not just as an abstract presentation, but as a
 * sort of Heegaard diagram. To be precise, we'll think of it as a
 * handlebody with a collection of disjoint simple closed curves showing
 * where (thickened) disks are to be attached. In a true Heegaard
 * diagram the boundary of the handlebody-with-disks-attached is
 * a single 3-sphere, but in our case the boundary will consist of
 * a 2-torus or Klein bottle for each unfilled cusp, and a 2-sphere
 * for each filled cusp. The number of relations may be less than,
 * equal to, or greater than the genus of the handlebody.
 *
 * Constructing the pseudo-Heegaard diagram.
 *
 * The manifold's Triangulation provides a pseudo-Heegaard diagram,
 * which we use to obtain the initial, unsimplified presentation
 * of the fundamental group.
 *
 * The handlebody is the thickened 1-skeleton of the ideal
 * triangulation's dual complex. (Please see choose_generators.c for an
 * explanation of how a set of generators is chosen for the handlebody,
 * and how the generators are represented internally.) There are two
 * types of relations. Each thickened edge (in the original ideal
 * triangulation, not the dual) is a relation (note that a thickened edge
 * is topologically the same as a thickened disk), and each Dehn filling
 * curve specifies a relation.
 *
 * Visualizing the pseudo-Heegaard diagram.
 *
 * Visualize the pseudo-Heegaard diagram as follows. Start with the
 * pseudo-Heegaard diagram drawn as an actual handlebody, then make
 * a meridional cut through each handle, so that the handlebody opens
 * into a 3-ball. Label the cut-disks A+ and A-, B+ and B-, etc.,
 * so that identifying A+ to A-, B+ to B-, etc. restores the original
 * handlebody. The relation aBc would then be a curve which goes in A+
 * and comes out A-, goes in B- and comes out B+, then goes in C+ and
 * comes out C- to its starting point.
 *
 * Operations which do or do not respect the Heegaard diagram.
 *
 * My original hope in developing this code was to provide an option
 * whereby the algorithm uses only simplifications which respect the
 * Heegaard diagram. This would have let us conclude, for example,
 * that a manifold with a genus zero presentation is a topological
 * 3-sphere, a manifold with a genus one presentation is a lens space,
 * etc. Some simplifications (e.g. handle slides) can rigorously
 * be interpreted as operations on the pseudo-Heegaard diagram.
 * Others, unfortunately, are slipperier. For example, cancellation
 * of inverses (e.g. "abBcAd" -> "acAd") is almost always a valid
 * operation on the Heegaard diagram -- just isotope the little loop
 * "bB" across the B- disk -- but one has to worry about whether the
 * loop encloses other disks. I had worked out an algorithm to

```

```

* removed any such other disks (by isotoping them across the B-
* disk) but then the proof that the algorithm terminates was no
* longer so clear. At that point I finally decided to abandon the
* Heegaard interpretation of the simplifications. I was feeling too
* uncomfortable working with a data structure (the GroupPresentation)
* which contained only a subset of the information needed for the
* calculations and for the proofs that the calculations are correct.
* Solid, reliable code requires a data structure which models the
* underlying mathematics as directly as possible.
*
* If you want a topological description of the space, you can compute
* an unsimplified presentation (which *does* correspond to a pseudo-
* Heegaard diagram, as explained above) and pass it to John Berge's
* program "Heegaard". "Heegaard" does an excellent job of recognizing
* lens space, for example. If you give it a sufficiently complicated
* presentation, it can even distinguish, say,  $L(5,1)$  from  $L(5,2)$ !
* (Note: The unsimplified presentation uses the standard generators
* defined in choose_generators.c. The code in representations.c
* relies on this fact.)
*
*
* Conventions.
*
* (1) The generators of an abstract group presentation are represented
* by lowercase letters. Their inverses are respresented by the
* corresponding uppercase letters. For example, "A" is the inverse
* of "a".
*
* (2) Words in an abstract group presentation are read left to right.
* For example, "abC" means do "a", "b" and "c inverse", in that order.
* However,  $O(3,1)$  matrices act on column vectors (matrix times column
* vector equals column vector), so products of such matrices are read
* right to left. For example, (M2)(M1) means do matrix M1, then do
* matrix M2.
*/

#include "kernel.h"
#include <limits.h>

typedef struct Letter
{
    /*
     * itsValue contains the index of a generator.
     * The generators are implicitly numbered 1, 2 ..., n, and their
     * inverses are -1, -2, ..., -n, where n is the number of generators.
     */
    int itsValue;

    /*
     * Letters are kept on circular doubly-linked lists.
     */
    struct Letter *prev,
                 *next;
} Letter;

typedef struct CyclicWord
{
    /*
     * itsLength gives the number of Letters in the CyclicWord.
     */
    int itsLength;

    /*
     * itsLetters points to an arbitrary Letter in the CyclicWord.
     * The Letters are kept on a circular doubly-linked list.
     *
     * If a Cyclic Word is empty, itsLength is set to 0 and
     * itsLetters is set to NULL.
     */
    Letter *itsLetters;

    /*
     * is_Dehn_relation says whether this relation comes from a Dehn

```

```

    * filling. When group->fillings_may_affect_generators is FALSE,
    * such relations may not influence the choice of generators.
    */
    Boolean          is_Deohn_relation;

    /*
    * The "next" field points to the next CyclicWord
    * in the GroupPresentation.
    */
    struct CyclicWord *next;
} CyclicWord;

struct GroupPresentation
{
    /*
    * How many generators does the GroupPresentation have?
    * (Geometrically, what is the genus of the handlebody in
    * the pseudo-Heegaard diagram?)
    */
    int          itsNumGenerators;

    /*
    * We maintain an array of matrices, one for each generator,
    * which defines the representation of the fundamental group
    * into Isom(H^3). The matrices could be given in either O(3,1)
    * or PSL(2,C); I chose the former because it handles orientation-
    * reversing isometries more naturally.
    */
    O3lMatrix    *itsMatrices;

    /*
    * How many relations does the presentation have?
    * (Geometrically, how many thickened disks are glued to the
    * boundary of the handlebody? See "Constructing the pseudo-Heegaard
    * diagram" above for more details.)
    */
    int          itsNumRelations;

    /*
    * itsRelations points to a NULL-terminated, singly-linked list
    * of relations. Typically the relations are interpreted as curves
    * on a handlebody, as explained above.
    */
    CyclicWord  *itsRelations;

    /*
    * Is this space a manifold or orbifold (e.g. FigureEight(5,1)
    * or FigureEight(6,3)) as opposed to some other generalized
    * Dehn filling (e.g. FigureEight(5.01, 1.0))? We compute
    * Dehn relations only for cusps with integer coefficients,
    * and ignore other (generalized) Dehn fillings. The UI should
    * display the relations only for integer fillings, but should
    * display the matrix representation for all generalized
    * Dehn fillings.
    */
    Boolean      integer_fillings;

    /*
    * We keep track of the words corresponding to the meridians and
    * longitudes. Note that
    *
    * (1) we keep track of the meridian and longitude even
    *     on filled cusps,
    *
    * (2) the two longitudes on a Klein bottle are not homotopic to one
    *     another, so we report the lift to the orientation double cover,
    *
    * (3) the words have well defined basepoints -- cyclic cancellations
    *     are not allowed.
    */
    int          itsNumCusps;
    CyclicWord  *itsMeridians,

```

```

        *itsLongitudes;

/*
 * We keep track of words which express each of the original generators
 * (the ones defined in choose_generators.c) as products of the
 * current generators. The words have well defined basepoints --
 * cyclic cancellations are not allowed.
 */
int      itsNumOriginalGenerators;
CyclicWord *itsOriginalGenerators;

/*
 * Should we simplify the presentation?
 *
 * For most purposes simplify_presentation should be TRUE, but
 * occasionally the user may want access to the unsimplified
 * presentation. For example, you can pass an unsimplified
 * presentation of a lens space to John Berge's program Heegaard,
 * and it will most likely be able to recognize the exact lens space.
 * Yes, it distinguishes L(5,1) from L(5,2), but only if it
 * begins with a sufficiently complicated presentation.
 * Passing in the presentation < a | a^5 = 1 > isn't good enough!
 *
 * 96/9/11 The code in representations.c relies on the fact that
 * the unsimplified presentation uses the standard generators
 * defined in choose_generators.c.
 */
Boolean      simplify_presentation;

/*
 * Is it OK for the choice of generators to depend on the Dehn fillings?
 * Sometimes the user may want this flag to be FALSE, for example
 * when he or she is studying how the matrix generators vary across
 * a Dehn filling plane, and wants a consistent choice of generators.
 * Other times the user may want this flag to be TRUE, for example
 * when he or she wants to see which Dehn fillings give lens spaces.
 */
Boolean      fillings_may_affect_generators;

/*
 * If minimize_number_of_generators is TRUE, simplify_presentation()
 * will try to reduce the number of generators at the expense of
 * increasing the total length of the relations. If it's FALSE,
 * it does the opposite.
 */
Boolean      minimize_number_of_generators;
};

static GroupPresentation      *compute_unsimplified_presentation(Triangulation *manifold);
static void                  compute_matrix_generators(Triangulation *manifold,
GroupPresentation *group);
static void                  compute_relations(Triangulation *manifold, GroupPresentation *
group);
static void                  compute_edge_relations(Triangulation *manifold,
GroupPresentation *group);
static void                  compute_one_edge_relation(EdgeClass *edge, GroupPresentation *
group);
static void                  compute_Dehn_relations(Triangulation *manifold,
GroupPresentation *group);
static void                  compute_peripheral_word(Cusp *cusp, PeripheralCurve which_curve
, CyclicWord **word_list);
static void                  find_standard_basepoint(Triangulation *manifold, Cusp *cusp);
static void                  find_curve_start(Cusp *cusp, PeripheralCurve which_curve,
PositionedTet *ptet);
static void                  compute_Dehn_word(CyclicWord *meridian, CyclicWord *longitude,
int m, int l, CyclicWord **word_list);
static void                  append_copies(CyclicWord *source, int n, CyclicWord *dest);
static void                  append_word(CyclicWord *source, CyclicWord *dest);
static void                  append_inverse(CyclicWord *source, CyclicWord *dest);
static void                  initialize_original_generators(GroupPresentation *group, int
num_generators);

static void                  simplify(GroupPresentation *group);

```

```

static void                insert_basepoints(GroupPresentation *group);
static void                insert_basepoints_on_list(CyclicWord *list);
static void                insert_basepoint_in_word(CyclicWord *word);
static void                remove_basepoints(GroupPresentation *group);
static void                remove_basepoints_on_list(CyclicWord *list);
static void                remove_basepoint_in_word(CyclicWord *word);
static Boolean             word_length_one(GroupPresentation *group);
static Boolean             word_length_two(GroupPresentation *group);
static Boolean             try_handle_slides(GroupPresentation *group);
static Boolean             substring_occurs_in_group(GroupPresentation *group, int a, int b);
static Boolean             substring_occurs_in_word(CyclicWord *word, int a, int b);
static Boolean             handle_slide_improves_presentation(GroupPresentation *group,
    int a, int b);
static void                evaluate_handle_slide_in_group(GroupPresentation *group, int a,
    int b, int *shortest_nonempty_relation_before, int *shortest_nonempty_relation_after,
    int *change_in_total_length, int *change_in_num_runs);
static void                evaluate_handle_slide_on_word(CyclicWord *word, int a, int b,
    int *shortest_nonempty_relation_before, int *shortest_nonempty_relation_after, int *
    change_in_total_length, int *change_in_num_runs);
static int                compute_delta_length(CyclicWord *word, int a, int b);
static int                compute_delta_runs(CyclicWord *word, int a, int b);
static Boolean             two_singletons_in_group(GroupPresentation *group);
static Boolean             generator_occurs_as_two_singletons_in_group(GroupPresentation *
    group, int value, CyclicWord **word_containing_singletons);
static Boolean             generator_occurs_as_two_singletons_in_word(CyclicWord *word,
    int value);
static Boolean             generator_occurs_in_no_other_word_in_group(GroupPresentation *
    group, int value, CyclicWord *word_containing_singletons);
static Boolean             generator_occurs_in_word(CyclicWord *word, int value);
static void                make_singletons_adjacent(GroupPresentation *group, int value,
    CyclicWord *word);
static Boolean             eliminate_word_in_group(GroupPresentation *group);
static CyclicWord          *shortest_word_in_which_generator_occurs_precisely_once
    (GroupPresentation *group, int generator);
static Boolean             generator_occurs_precisely_once_in_word(CyclicWord *word, int
    generator);
static int                occurrences_in_group(GroupPresentation *group, int generator);
static int                occurrences_in_word(CyclicWord *word, int generator);
static void                eliminate_word(GroupPresentation *group, CyclicWord *word, int
    generator);
static Boolean             remove_empty_relations(GroupPresentation *group);
static Boolean             insert_word_from_group(GroupPresentation *group);
static Boolean             insert_word_into_group(GroupPresentation *group, CyclicWord *
    word);
static Boolean             insert_word_into_list(CyclicWord *list, CyclicWord *word);
static Boolean             insert_word_into_word(CyclicWord *word, CyclicWord *target);
static Boolean             insert_word_forwards(CyclicWord *word, CyclicWord *target);
static Boolean             insert_word_backwards(CyclicWord *word, CyclicWord *target);
static Boolean             simplify_one_word_presentations(GroupPresentation *group);
static Boolean             word_contains_pattern(Letter *unmatched_letter, int period, int
    repetitions);
static CyclicWord          *introduce_generator(GroupPresentation *group, Letter *
    substring, int length);
static void                lens_space_recognition(GroupPresentation *group);
static int                count_runs(CyclicWord *word);
static Boolean             lens_space_recognition_using_generator(GroupPresentation *group,
    int generator0);
static Boolean             invert_generators_where_necessary(GroupPresentation *group);
static void                count_signed_occurrences_in_group(GroupPresentation *group, int
    a, int *positive_occurrences, int *negative_occurrences);
static void                increment_signed_occurrences_in_group(GroupPresentation *group,
    int a, int *positive_occurrences, int *negative_occurrences);
static void                increment_signed_occurrences_in_word(CyclicWord *word, int a,
    int *positive_occurrences, int *negative_occurrences);
static int                count_signed_occurrences_in_word(CyclicWord *word, int a);
static void                invert_generator_in_group(GroupPresentation *group, int a);
static void                invert_generator_on_list(CyclicWord *list, int a);
static void                invert_generator_in_word(CyclicWord *word, int a);
static Boolean             invert_words_where_necessary(GroupPresentation *group);
static Boolean             invert_word_if_necessary(CyclicWord *word);
static int                sum_of_powers(CyclicWord *word);
static void                invert_word(CyclicWord *word);
static void                choose_word_starts(GroupPresentation *group);

```

```

static void                choose_word_start(CyclicWord *word);
static void                conjugate_peripheral_words(GroupPresentation *group);
static Boolean             conjugate_peripheral_pair(CyclicWord *word0, CyclicWord *word1) ✓
;
static void                conjugate_word(CyclicWord *word, int value);

static void                cancel_inverses(GroupPresentation *group);
static void                cancel_inverses_word_list(CyclicWord *list);
static void                cancel_inverses_word(CyclicWord *word);
static void                handle_slide(GroupPresentation *group, int a, int b);
static void                handle_slide_word_list(CyclicWord *list, int a, int b);
static void                handle_slide_word(CyclicWord *word, int a, int b);
static void                handle_slide_matrices(GroupPresentation *group, int a, int b);
static void                cancel_handles(GroupPresentation *group, CyclicWord *word);
static void                remove_word(GroupPresentation *group, CyclicWord *word);
static void                remove_generator(GroupPresentation *group, int dead_generator);
static void                remove_generator_from_list(CyclicWord *list, int ✓
    dead_generator);
static void                remove_generator_from_word(CyclicWord *word, int ✓
    dead_generator);
static void                renumber_generator(GroupPresentation *group, int old_index, int ✓
    new_index);
static void                renumber_generator_on_word_list(CyclicWord *list, int old_index ✓
    , int new_index);
static void                renumber_generator_in_word(CyclicWord *word, int old_index, int ✓
    new_index);

static int                *fg_get_cyclic_word(CyclicWord *list, int which_relation);
static void                free_word_list(CyclicWord *aWordList);
static void                free_cyclic_word(CyclicWord *aCyclicWord);

```

```

GroupPresentation *fundamental_group(
    Triangulation    *manifold,
    Boolean          simplify_presentation,
    Boolean          fillings_may_affect_generators,
    Boolean          minimize_number_of_generators)
{
    GroupPresentation    *group;

    /*
     * Read a group presentation from the manifold, without worrying
     * about simplifying it. This group presentation will be that
     * of a pseudo-Heegaard diagram, as discussed above.
     */
    group = compute_unsimplified_presentation(manifold);

    /*
     * Note the user's preferences.
     *
     * (Please see the GroupPresentation typedef above for
     * an explanation of these flags.)
     */
    group->simplify_presentation      = simplify_presentation;
    group->fillings_may_affect_generators = fillings_may_affect_generators;
    group->minimize_number_of_generators = minimize_number_of_generators;

    /*
     * Simplify the group presentation if requested to do so.
     */
    if (group->simplify_presentation == TRUE)
        simplify(group);

    return group;
}

static GroupPresentation *compute_unsimplified_presentation(
    Triangulation    *manifold)
{
    GroupPresentation    *group;

    group = NEW_STRUCT(GroupPresentation);

```



```

    choose_generators(manifold, FALSE, FALSE);

    group->itsNumGenerators = manifold->num_generators;

    compute_matrix_generators(manifold, group);

    compute_relations(manifold, group);

    initialize_original_generators(group, group->itsNumGenerators);

    group->integer_fillings = all_Deahn_coefficients_are_integers(manifold);

    return group;
}

static void compute_matrix_generators(
    Triangulation      *manifold,
    GroupPresentation   *group)
{
    /*
     * Pass centroid_at_origin = FALSE to matrix_generators()
     * so the initial Tetrahedron will be positioned with vertices
     * at {0, 1, infinity, z}. This brings out nice number theoretic
     * properties in the matrix generators, and also forces Triangulations
     * with all flat tetrahedra to lie in a coordinate plane.
     */

    group->itsMatrices = NEW_ARRAY(manifold->num_generators, O3lMatrix);

    if (get_filled_solution_type(manifold) != not_attempted
        && get_filled_solution_type(manifold) != no_solution)
    {
        MoebiusTransformation *moebius_generators;

        moebius_generators = NEW_ARRAY(manifold->num_generators, MoebiusTransformation);

        matrix_generators(manifold, moebius_generators, FALSE);

        Moebius_array_to_O3l_array( moebius_generators,
                                    group->itsMatrices,
                                    manifold->num_generators);

        my_free(moebius_generators);
    }
    else
    {
        int i;

        for (i = 0; i < manifold->num_generators; i++)
            o3l_copy(group->itsMatrices[i], O3l_identity);
    }
}

static void compute_relations(
    Triangulation      *manifold,
    GroupPresentation   *group)
{
    group->itsNumRelations = 0;
    group->itsRelations = NULL;

    /*
     * Compute the Deahn relations first, so they appear
     * on the linked list *after* the edge relations.
     */
    compute_Deahn_relations(manifold, group);
    compute_edge_relations(manifold, group);
}

static void compute_edge_relations(
    Triangulation      *manifold,
    GroupPresentation   *group)

```



```

{
    EdgeClass    *edge;

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)

        compute_one_edge_relation(edge, group);
}

static void compute_one_edge_relation(
    EdgeClass    *edge,
    GroupPresentation *group)
{
    CyclicWord    *new_word;
    PositionedTet  ptet0,
                  ptet;
    Letter        dummy_letter,
                  *new_letter;
    int           index;

    /*
     * Ignore EdgeClasses which choose_generators() has already
     * eliminated via handle_cancellations or handle_merging.
     * (choose_generators() doesn't eliminate them from the Triangulation;
     * it just eliminates them from its picture of the pseudo-Heegaard
     * diagram.)
     */
    if (edge->active_relation == FALSE)
        return;

    /*
     * choose_generator()'s algorithm ensures that each active relation
     * has at least two letters. (They may cancel, but there are
     * nominally at least two of them.)
     */
    if (edge->num_incident_generators < 2)
        uFatalError("compute_one_edge_relation", "fundamental_group");

    /*
     * Initialize the new_word, and install it on the linked list.
     */
    new_word = NEW_STRUCT(CyclicWord);
    new_word->itsLength      = 0;
    new_word->is_Dehn_relation = FALSE;
    new_word->next           = group->itsRelations;
    group->itsRelations      = new_word;
    group->itsNumRelations++;

    /*
     * We'll use a temporary dummy_letter to initialize
     * the circular doubly linked list.
     */
    dummy_letter.next = &dummy_letter;
    dummy_letter.prev = &dummy_letter;

    /*
     * Traverse the EdgeClass, recording each generator we find.
     */
    set_left_edge(edge, &ptet0);
    ptet = ptet0;
    do
    {
        /*
         * Are we passing a generator?
         * If so, convert from the generator_index's 0-based numbering
         * to the GroupPresentation's 1-based numbering (with negative
         * numbers for inverses).  inbound_generators are considered
         * positively oriented (for later consistency with the conventions
         * for storing peripheral curves).
         */
        switch (ptet.tet->generator_status[ptet.near_face])
        {

```

```

        case inbound_generator:
            index = ptet.tet->generator_index[ptet.near_face] + 1;
            break;

        case outbound_generator:
            index = -(ptet.tet->generator_index[ptet.near_face] + 1);
            break;

        case not_a_generator:
            index = 0;
            break;

        default:
            uFatalError("compute_one_edge_relation", "fundamental_group");
    }

    if (index != 0)
    {
        new_letter = NEW_STRUCT(Letter);
        new_letter->itsValue = index;
        INSERT_BEFORE(new_letter, &dummy_letter);

        new_word->itsLength++;
    }

    veer_left(&ptet);

} while (same_positioned_tet(&ptet, &ptet0) == FALSE);

/*
 * Did we find the right number of generators?
 */
if (new_word->itsLength != edge->num_incident_generators)
    uFatalError("compute_one_edge_relation", "fundamental_group");

/*
 * Give new_word a valid pointer to the circular doubly linked list
 * of Letters, and then remove the temporary dummy_letter.
 */
new_word->itsLetters = dummy_letter.next;
REMOVE_NODE(&dummy_letter);
}

static void compute_Deahn_relations(
    Triangulation      *manifold,
    GroupPresentation  *group)
{
    Cusp      *cusp;
    int       i;

    group->itsNumCusps      = manifold->num_cusps;

    group->itsMeridians      = NULL;
    group->itsLongitudes     = NULL;

    /*
     * Examine the cusps in reverse order, so the Dehn filling relations
     * get pushed onto the relation list in the correct order.
     */
    for (i = manifold->num_cusps; --i >= 0; )
    {
        cusp = find_cusp(manifold, i);

        /*
         * First compute the meridian and longitude...
         */
        find_standard_basepoint(manifold, cusp);
        compute_peripheral_word(cusp, M, &group->itsMeridians);
        compute_peripheral_word(cusp, L, &group->itsLongitudes);

        /*
         * ...and then, if the Dehn coefficients are integers,
         * compute the Dehn relation by concatenating copies

```

```

    * of the meridian and longitude.
    */
    if (cusp->is_complete == FALSE
        && Dehn_coefficients_are_integers(cusp) == TRUE)
    {
        compute_Dein_word( group->itsMeridians,
                           group->itsLongitudes,
                           (int) cusp->m,
                           (int) cusp->l,
                           &group->itsRelations);
        group->itsNumRelations++;
    }
}

static void compute_peripheral_word(
    Cusp *cusp,
    PeripheralCurve which_curve,
    CyclicWord **word_list)
{
    /*
     * Note that the triangulation.h data structure works with the
     * orientation double cover of each cusp, so both torus and Klein
     * bottle cusps appear as tori, and can be handled the same.
     * The only difference is that the "longitude" of a Klein bottle
     * cusp is actually the double cover of a longitude. Please
     * see peripheral_curves.c for a careful and complete discussion
     * of these issues.
     */

    PositionedTet ptet0,
    int strand0,
    Strand strand,
    near_strands,
    left_strands;
    CyclicWord *new_word;
    Letter dummy_letter,
    *new_letter;
    int index;

    /*
     * Initialize the new_word, and install it on the linked list.
     *
     * Use a temporary dummy_letter to initialize
     * the circular doubly linked list.
     */
    new_word = NEW_STRUCT(CyclicWord);
    new_word->itsLength = 0;
    new_word->itsLetters = &dummy_letter;
    dummy_letter.next = &dummy_letter;
    dummy_letter.prev = &dummy_letter;
    new_word->is_Dein_relation = TRUE;
    new_word->next = *word_list;
    *word_list = new_word;

    /*
     * Start where the meridian and longitude intersect.
     * This insures that
     *
     * (1) the meridian and longitude commute, and
     *
     * (2) we can form linear combinations of meridians and
     *     longitudes by concatenation.
     */
    find_curve_start(cusp, which_curve, &ptet0);

    /*
     * Here's how we keep track of where we are. At each step, we are
     * always at the near edge of the top vertex (i.e. the truncated vertex
     * opposite the bottom face) of the PositionedTet ptet (please see
     * positioned_tet.h if necessary). The curve may cross that edge
     * several times. The variable "strand" keeps track of which

```

```

    * intersection we are at; 0 means we're at the strand on the
    * far left, 1 means we're at the next strand, etc.
    */

/*
 * Start at the leftmost strand.
 */
strand0 = 0;

ptet    = ptet0;
strand  = strand0;
do
{
    /*
     * Record the generator (if any) corresponding to the near_face.
     */

    switch (ptet.tet->generator_status[ptet.near_face])
    {
        case inbound_generator:
            index = ptet.tet->generator_index[ptet.near_face] + 1;
            break;

        case outbound_generator:
            index = -(ptet.tet->generator_index[ptet.near_face] + 1);
            break;

        case not_a_generator:
            index = 0;
            break;

        default:
            uFatalError("compute_peripheral_word", "fundamental_group");
    }

    if (index != 0)
    {
        new_letter = NEW_STRUCT(Letter);
        new_letter->itsValue = index;
        INSERT_BEFORE(new_letter, &dummy_letter);

        new_word->itsLength++;
    }

    /*
     * Decide whether to veer_left() or veer_right().
     */

    /*
     * Note the curve's intersection numbers
     * with the near side and the left side.
     */
    near_strands = ptet.tet->curve [which_curve]
                                     [ptet.orientation]
                                     [ptet.bottom_face]
                                     [ptet.near_face];
    left_strands = ptet.tet->curve [which_curve]
                                     [ptet.orientation]
                                     [ptet.bottom_face]
                                     [ptet.left_face];

    /*
     * Does the current strand bend to the left or to the right?
     */
    if (strand < FLOW(near_strands, left_strands))
    {
        /*
         * The current strand bends to the left.
         */

        /*
         * Some of the near strands may branch off towards the
         * right, or some strands may come in from the right and
         * join the near strands as they head out to the left.

```

```

        * But either way, the variable "strand" remains unchanged.
        */

    /*
     * Move the PositionedTet onward, following the curve.
     */
    veer_left(&ptet);
}
else
{
    /*
     * The current strand bends to the right.
     */

    /*
     * Some strands from the near edge may have gone off
     * to the left edge, in which case the variable "strand"
     * should be decreased by that amount.
     * Alternatively, some strands may have come in from the
     * left edge, joining us at the right edge, in which case
     * the variable "strand" should be increased by that amount.
     * The code "strand += left_strands" works for both cases,
     * because left_strands will be negative in the former case
     * and positive in the latter.
     */
    strand += left_strands;

    /*
     * Move the PositionedTet onward, following the curve.
     */
    veer_right(&ptet);
}

} while (    same_positioned_tet(&ptet, &ptet0) == FALSE
           || strand != strand0);

/*
 * Give new_word a valid pointer to the circular doubly linked list
 * of Letters, and then remove the temporary dummy_letter.
 *
 * Note that for meridians and longitudes, new_word->itsLetters
 * is set to the beginning of the based word, so the basepoints
 * for the meridian and longitude are the same, and the words will
 * commute.
 */
new_word->itsLetters = dummy_letter.next;
REMOVE_NODE(&dummy_letter);
}

static void find_standard_basepoint(
    Triangulation *manifold,
    Cusp *cusp)
{
    /*
     * Find an ideal vertex where both the meridian and longitude
     * both pass through, and let an arbitrary point in its interior
     * be the basepoint for the cusp.
     */

    FaceIndex face;

    for (cusp->basepoint_tet = manifold->tet_list_begin.next;
         cusp->basepoint_tet != &manifold->tet_list_end;
         cusp->basepoint_tet = cusp->basepoint_tet->next)

        for (cusp->basepoint_vertex = 0;
             cusp->basepoint_vertex < 4;
             cusp->basepoint_vertex++)
        {
            if (cusp->basepoint_tet->cusp[cusp->basepoint_vertex] != cusp)
                continue;

            for (face = 0; face < 4; face++)

```

```

    {
        if (face == cusp->basepoint_vertex)
            continue;

        for (cusp->basepoint_orientation = 0;
             cusp->basepoint_orientation < 2;
             cusp->basepoint_orientation++)

            if (cusp->basepoint_tet->curve
                [M]
                [cusp->basepoint_orientation]
                [cusp->basepoint_vertex]
                [face] != 0
                && cusp->basepoint_tet->curve
                [L]
                [cusp->basepoint_orientation]
                [cusp->basepoint_vertex]
                [face] != 0)

                /*
                 * We found the basepoint!
                 */
                return;
    }
}

/*
 * If we get to this point, it means that no intersection
 * was found, which is impossible.
 */
uFatalError("find_standard_basepoint", "fundamental_group");
}

static void find_curve_start(
    Cusp *cusp,
    PeripheralCurve which_curve,
    PositionedTet *ptet)
{
    /*
     * We assume the standard basepoint has already been found by a
     * previous call to find_standard_basepoint(), and use it to
     * find the PositionedTet where the requested curve leaves
     * the standard basepoint. (So ptet->tet typically will NOT be
     * cusp->basepoint_tet.)
     */

    /*
     * Temporarily set ptet to be the standard basepoint.
     * We'll change it in a moment.
     */
    ptet->tet = cusp->basepoint_tet;
    ptet->bottom_face = cusp->basepoint_vertex;
    ptet->orientation = cusp->basepoint_orientation;

    /*
     * Let near_face be where the requested curve leaves the triangle.
     */
    for (ptet->near_face = 0; ptet->near_face < 4; ptet->near_face++)
    {
        if (ptet->near_face == ptet->bottom_face)
            continue;

        if (0 > ptet->tet->curve[which_curve][ptet->orientation][ptet->bottom_face][ptet->
near_face])
        {
            /*
             * We've found where the curve leaves the triangle.
             * Our starting ptet will be just on the other side.
             *
             * First fill in the remaining details for this ptet...
             */
            if (ptet->orientation == right_handed)
            {

```

```

        ptet->left_face      = remaining_face[ptet->bottom_face][ptet->near_face];
        ptet->right_face     = remaining_face[ptet->near_face][ptet->bottom_face];
    }
    else /* ptet->orientation == left_handed */
    {
        ptet->left_face      = remaining_face[ptet->near_face][ptet->bottom_face];
        ptet->right_face     = remaining_face[ptet->bottom_face][ptet->near_face];
    }

    /*
     * ...then turn around.
     */
    veer_backwards(ptet);

    /*
     * Make sure it worked out like we planned.
     */
    if (0 >= ptet->tet->curve[which_curve][ptet->orientation][ptet->bottom_face]
        [ptet->near_face])
        uFatalError("find_curve_start", "fundamental_group");

    return;
}

/*
 * We should find a negative intersection number somewhere within
 * the above loop, so we should never get to this point.
 */
uFatalError("find_curve_start", "fundamental_group");
}

static void compute_Dehn_word(
    CyclicWord *meridian,
    CyclicWord *longitude,
    int m,
    int l,
    CyclicWord **word_list)
{
    CyclicWord *new_word;
    Letter dummy_letter;

    /*
     * We should never be passed (m,l) = (0,0) when
     * cusp->is_complete is FALSE, but we check anyhow.
     */
    if (m == 0 && l == 0)
        uFatalError("compute_Dehn_word", "fundamental_group");

    /*
     * Initialize the new_word, and install it on the linked list.
     *
     * Use a temporary dummy_letter to initialize
     * the circular doubly linked list.
     */
    new_word = NEW_STRUCT(CyclicWord);
    new_word->itsLength      = 0;
    new_word->itsLetters     = &dummy_letter;
    dummy_letter.next       = &dummy_letter;
    dummy_letter.prev       = &dummy_letter;
    new_word->is_Dehn_relation = TRUE;
    new_word->next           = *word_list;
    *word_list              = new_word;

    /*
     * Append m meridians and l longitudes to new_word,
     * taking into account the signs of m and l.
     */
    append_copies(meridian, m, new_word);
    append_copies(longitude, l, new_word);

    /*
     * Give new_word a valid pointer to the circular doubly linked list

```



```

    * of Letters, and then remove the temporary dummy_letter.
    *
    * Note that for meridians and longitudes, new_word->itsLetters
    * is set to the beginning of the based word, so the basepoints
    * for the meridian and longitude are the same, and the words will
    * commute.
    */
    new_word->itsLetters = dummy_letter.next;
    REMOVE_NODE(&dummy_letter);
}

static void append_copies(
    CyclicWord *source,
    int n,
    CyclicWord *dest)
{
    int i;

    for (i = 0; i < ABS(n); i++)

        if (n > 0)
            append_word(source, dest);
        else
            append_inverse(source, dest);
}

static void append_word(
    CyclicWord *source,
    CyclicWord *dest)
{
    int i;
    Letter *letter,
            *letter_copy;

    for (letter = source->itsLetters, i = 0;
         i < source->itsLength;
         letter = letter->next, i++)
    {
        letter_copy = NEW_STRUCT(Letter);
        letter_copy->itsValue = letter->itsValue;
        INSERT_BEFORE(letter_copy, dest->itsLetters);
        dest->itsLength++;
    }
}

static void append_inverse(
    CyclicWord *source,
    CyclicWord *dest)
{
    int i;
    Letter *letter,
            *letter_copy;

    for (letter = source->itsLetters->prev, i = 0;
         i < source->itsLength;
         letter = letter->prev, i++)
    {
        letter_copy = NEW_STRUCT(Letter);
        letter_copy->itsValue = - letter->itsValue;
        INSERT_BEFORE(letter_copy, dest->itsLetters);
        dest->itsLength++;
    }
}

static void initialize_original_generators(
    GroupPresentation *group,
    int num_generators)
{
    int index;
    Letter *new_letter;

```

```

CyclicWord  *new_word;

group->itsNumOriginalGenerators = num_generators;

/*
 * Initially the original generators are the current generators.
 * Put the highest numbered generator on the linked list first,
 * and work backwards, so that they will appear in the correct order.
 */

group->itsOriginalGenerators = NULL;

for (index = num_generators; index >= 1; --index)
{
    new_letter = NEW_STRUCT(Letter);
    new_letter->itsValue      = index;
    new_letter->prev          = new_letter;
    new_letter->next          = new_letter;

    new_word = NEW_STRUCT(CyclicWord);
    new_word->itsLength       = 1;
    new_word->itsLetters      = new_letter;
    new_word->is_Dehn_relation = FALSE;
    new_word->next            = group->itsOriginalGenerators;
    group->itsOriginalGenerators = new_word;
}

static void simplify(
    GroupPresentation  *group)
{
    /*
     * The Induction Variable
     *
     * If group->minimize_number_of_generators is TRUE, then
     * each operation in the simplification algorithm decreases
     * the value of the ordered quintuple
     *
     * (number of generators,
     *  length of shortest nonempty relation,
     *  total length of all relations,
     *  total number of runs in all relations,
     *  total length of all meridians and longitudes)
     *
     * relative to the lexicographic ordering. In other words, each
     * operation either decreases the number of generators, or leaves
     * the number of generators constant while decreasing the length
     * of the shortest nonempty relation, or leaves both the number of
     * generators and the length of the shortest relation constant while
     * decreasing the total length of all relations, etc. This provides
     * a simple proof that the algorithm terminates in a finite number
     * of steps.
     *
     * If group->minimize_number_of_generators is FALSE, then we ignore
     * the number of generators and try to minimize the ordered quadruple
     *
     * (total length of all relations,
     *  total number of runs in all relations,
     *  length of shortest nonempty relation,
     *  total length of all meridians and longitudes).
     *
     * By the "total number of runs" I mean that "aaabb" is simpler than
     * "aabAb" because the former has two runs ("aaa" and "bb") while
     * the latter has four ("aa", "b", "A" and "b"). The two words are
     * equivalent via a handle slide. Actually, for technical simplicity
     * we count the number of transitions from one run to another.
     * For words with at least two runs, the number of runs equals the
     * number of transitions. But a word with only one run has no
     * transitions.
     *
     * "length of shortest nonempty relation" may be defined as zero
     * if there are no nonempty relations.
     */
}

```

```

/*
 * Comment: eliminate_word_in_group() is called after
 * try_handle_slides() on the assumption that the former is more
 * likely to make a mess of the presentation than the latter, but I
 * don't have any hard evidence to support this assumption.
 */

/*
 * Insert a dummy basepoint Letter into each meridian and longitude,
 * and also into the expressions for the original generators, to make
 * sure they remain based at the same point. The basepoint has
 * itsValue == 0 so that it can't possibly cancel with anything.
 */
insert_basepoints(group);

/*
 * Cancel obvious inverses in each CyclicWord,
 * e.g. "abCcBefA" -> "ef". Hereafter, each low-level function
 * which changes the GroupPresentation (e.g. handle_slide() etc.)
 * will call cancel_inverses() before returning. cancel_inverses()
 * decreases the total length of all relations without increasing
 * any other component of The Induction Variable.
 */
cancel_inverses(group);

/*
 * The following while() loop call various mid-level functions
 * in the preferred order. As soon as some mid-level function
 * returns TRUE, the while() loop begins again at the start of
 * the list. The idea is that we want to do the more basic
 * simplifications before considering the fancier ones, and when we
 * do make some progress with the fancier ones, we want to try the
 * basic ones again.
 */
while
(
    remove_empty_relations(group)

    /*
     * If there is a relation of length one, e.g. "a", do a handle
     * slide to cancel the relation (which is topologically a
     * thickened disk) with the generator (which is topologically
     * a handle of the handlebody). word_length_one() decreases both
     * the number of generators and the total length of all relations.
     */
    || word_length_one(group)

    /*
     * If there is a relation of the form "ab", do the handle slide
     * and then a handle cancellation. word_length_two() decreases both
     * the number of generators and the total length of all relations.
     */
    || word_length_two(group)

    /*
     * Consider all possible handle slides. If we find one which
     * reduces The Induction Variable (cf. above), do it.
     */
    || try_handle_slides(group)

    /*
     * If a generator occurs in precisely one word, and occurs in that
     * word precisely twice, both times with the same sign, then we may
     * do handle slides to make the two occurrences of the generator
     * adjacent to one another. This decreases the number of runs
     * without increasing any other component of The Induction Variable.
     * For example, we could simplify the word "aabAAb" to "aaaabb".
     */
    || two_singletons_in_group(group)

    /*
     * Look for a word in which a generator occurs precisely once,
     * and use that word to eliminate the generator. (Say the word

```

```

    * is "bcacb". First do handle slides to reduce it to "a", and
    * then do a handle cancellation.)
    * If group->minimize_number_of_generators is FALSE, a word will
    * be eliminated only if it does not increase the total length
    * of all relations.
    */
|| eliminate_word_in_group(group)

/*
 * Try to insert a copy of one word into another so that
 * after cancellations the second word is shorter than it
 * used to be. This reduces the total length of all
 * relations without increasing either the number of
 * generators or the length of the shortest nonempty
 * relation (unless a relation is eliminated entirely,
 * but that's OK).
 */
|| insert_word_from_group(group)

/*
 * If we have a GroupPresentation with exactly one word,
 * we can look for patterns in that word, and introduce
 * a new generator which simplifies the presentation. E.g.
 * {(bbaa)b(bbbaa)(bbbaa)} -> {Cbbaa, (bbaa)b(bbbaa)(bbbaa)}
 * -> {Cbbaa, bccc} -> {CCCCCCCaa}. This approach is
 * particular useful for recognizing fundamental groups
 * of torus knots as  $a^n = b^m$ .
 */
|| simplify_one_word_presentations(group)
)
;

/*
 * Try to simplify presentations of finite cyclic groups.
 */
lens_space_recognition(group);

/*
 * If a generator appears more often as an inverse than as a
 * positive power, replace it with its inverse.
 * E.g. {AAbbc, abCCC, cB} -> {aabbC, Abccc, CB}.
 *
 * If a word contains more inverses than positive powers,
 * invert it. E.g. {aabbC, Abccc, CB} -> {aabbC, Abccc, bc}.
 *
 * Repeat as necessary.
 */
while ( invert_generators_where_necessary(group) == TRUE
|| invert_words_where_necessary(group) == TRUE)
;

/*
 * The starting point of a cyclic word is arbitrary.
 * Choose it to meet the following aesthetic criteria,
 * in descending order of importance:
 *
 * (1) Don't start a word in the middle of a run.
 * (2) Start with a positive power of a generator.
 * (3) Start with the lowest-numbered generator.
 *
 * These criteria are applied only to the relations,
 * not to the peripheral curves, because we want the
 * latter to commute.
 */
choose_word_starts(group);

/*
 * Can we conjugate any (meridian, longitude) pairs
 * to shorten their length? E.g. (CBcbcc, Cac) -> (BCbc, a)
 * or (cbbc, Caac) -> (ccbb, aa).
 */
conjugate_peripheral_words(group);

/*

```

```

    * Remove the dummy basepoint Letters from the meridians and longitudes,
    * and from the expressions for the original generators.
    */
    remove_basepoints(group);
}

static void insert_basepoints(
    GroupPresentation *group)
{
    insert_basepoints_on_list(group->itsMeridians);
    insert_basepoints_on_list(group->itsLongitudes);
    insert_basepoints_on_list(group->itsOriginalGenerators);
}

static void insert_basepoints_on_list(
    CyclicWord *list)
{
    CyclicWord *word;

    for (word = list; word != NULL; word = word->next)
        insert_basepoint_in_word(word);
}

static void insert_basepoint_in_word(
    CyclicWord *word)
{
    Letter *basepoint;

    basepoint = NEW_STRUCT(Letter);
    basepoint->itsValue = 0;
    if (word->itsLength > 0)
        INSERT_BEFORE(basepoint, word->itsLetters)
    else
    {
        basepoint->prev = basepoint;
        basepoint->next = basepoint;
    }
    word->itsLetters = basepoint;
    word->itsLength++;
}

static void remove_basepoints(
    GroupPresentation *group)
{
    remove_basepoints_on_list(group->itsMeridians);
    remove_basepoints_on_list(group->itsLongitudes);
    remove_basepoints_on_list(group->itsOriginalGenerators);
}

static void remove_basepoints_on_list(
    CyclicWord *list)
{
    CyclicWord *word;

    for (word = list; word != NULL; word = word->next)
        remove_basepoint_in_word(word);
}

static void remove_basepoint_in_word(
    CyclicWord *word)
{
    Letter *letter,
           *basepoint;
    int i;

    /*

```

```

    * Find the basepoint.
    * There should be precisely one.
    */

    basepoint = NULL;

    for (    letter = word->itsLetters, i = 0;
           i < word->itsLength;
           letter = letter->next, i++)

        if (letter->itsValue == 0)
        {
            /*
             * Report an error if we've already found a basepoint before this one.
             */
            if (basepoint != NULL)
                uFatalError("remove_basepoint_in_word", "fundamental_group");

            basepoint = letter;
        }

    /*
     * Report an error if we found no basepoint.
     */
    if (basepoint == NULL)
        uFatalError("remove_basepoint_in_word", "fundamental_group");

    if (word->itsLength > 1)
    {
        word->itsLetters = basepoint->next;
        REMOVE_NODE(basepoint);
    }
    else
        word->itsLetters = NULL;

    my_free(basepoint);
    word->itsLength--;
}

static Boolean word_length_one(
    GroupPresentation *group)
{
    CyclicWord *word;

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Deohn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            if (word->itsLength == 1)
            {
                cancel_handles(group, word);
                return TRUE;
            }

    return FALSE;
}

static Boolean word_length_two(
    GroupPresentation *group)
{
    CyclicWord *word;
    int a,
        b;

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Deohn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            if (word->itsLength == 2)

```

```

    {
        a = word->itsLetters->itsValue;
        b = word->itsLetters->next->itsValue;

        if (a != b && a != -b)
        {
            handle_slide(group, a, b);
            cancel_handles(group, word);
            return TRUE;
        }
    }

    return FALSE;
}

static Boolean try_handle_slides(
    GroupPresentation *group)
{
    /*
     * We want to consider all possible handles slides.
     * As explained in handle_slide(), a handle slide is determined by
     * two generators. For example, if the generators are {a, b},
     * the potential handle slides are
     *
     *      (BB)      BA      Ba      (Bb)
     *      AB      (AA)      (Aa)      Ab
     *      aB      (aA)      (aa)      ab
     *      (bB)      bA      ba      (bb)
     *
     * Geometrically impossible combinations are shown in parentheses.
     * In a GroupPresentation with n generators, the above chart would have
     * (2n)*(2n) entries, (2n)2 of which are impossible, leaving 4n(n-1)
     * potential handle slides to consider.
     */

    int a, b;

    /*
     * Abuse notation and let "ab" be a generic entry in the above table.
     * That is, "ab" will range over all possible handle slides.
     */

    for (a = - group->itsNumGenerators; a <= group->itsNumGenerators; a++)
    {
        if (a == 0) /* There is no generator 0. */
            continue;

        for (b = - group->itsNumGenerators; b <= group->itsNumGenerators; b++)
        {
            if (b == 0) /* There is no generator 0. */
                continue;

            if (b == a || b == -a) /* Geometrically meaningless, cf. above. */
                continue;

            if (substring_occurs_in_group(group, a, b) == TRUE
                && handle_slide_improves_presentation(group, a, b) == TRUE)
            {
                handle_slide(group, a, b);
                return TRUE;
            }
        }
    }

    return FALSE;
}

static Boolean substring_occurs_in_group(
    GroupPresentation *group,
    int a,
    int b)

```



```

{
    CyclicWord *word;

    /*
     * a and b should be distinct generators.
     */

    if (a == b || a == -b)
        uFatalError("substring_occurs_in_group", "fundamental_group");

    /*
     * Does the substring "ab" occur somewhere in the group?
     */

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Dehn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            if (substring_occurs_in_word(word, a, b) == TRUE)

                return TRUE;

    return FALSE;
}

static Boolean substring_occurs_in_word(
    CyclicWord *word,
    int a,
    int b)
{
    Letter *letter;
    int i;

    for (letter = word->itsLetters, i = 0;
         i < word->itsLength;
         letter = letter->next, i++)

        if ((letter->itsValue == a && letter->next->itsValue == b)
            || (letter->itsValue == -a && letter->prev->itsValue == -b))

            return TRUE;

    return FALSE;
}

static Boolean handle_slide_improves_presentation(
    GroupPresentation *group,
    int a,
    int b)
{
    /*
     * We want to evaluate the effect of the handle slide "ab".
     * As explained in handle_slide(), the handle slide "ab" acts by
     *
     *
     *          "a" -> "aB"
     *          "A" -> "bA"
     *
     * If group->minimize_number_of_generators is TRUE, we want to see
     * whether the handle slide would decrease The Induction Variable
     * defined in simplify() as
     *
     * (number of generators,
     *  length of shortest nonempty relation,
     *  total length of all relations,
     *  total number of runs in all relations,
     *  total length of all meridians and longitudes).
     *
     * If group->minimize_number_of_generators is FALSE, we want to see
     * whether the handle slide would decrease The Induction Variable
     *
     * (total length of all relations,

```

```

    *      total number of runs in all relations,
    *      length of shortest nonempty relation,
    *      total length of all meridians and longitudes).
    *
    * Handle slides never change the number of generators,
    * so we may ignore that component of The Induction Variable.
    * We check how the handle slide would change the other components
    * of The Induction Variable, and return TRUE if it would be improved,
    * or FALSE if it would be the same or worse.
    */

int shortest_nonempty_relation_before,
    shortest_nonempty_relation_after,
    change_in_total_length,
    change_in_num_runs;

shortest_nonempty_relation_before = INT_MAX;
shortest_nonempty_relation_after  = INT_MAX;
change_in_total_length = 0;
change_in_num_runs      = 0;

evaluate_handle_slide_in_group( group,
                                a,
                                b,
                                &shortest_nonempty_relation_before,
                                &shortest_nonempty_relation_after,
                                &change_in_total_length,
                                &change_in_num_runs);

if (group->minimize_number_of_generators == TRUE)
{
    if (shortest_nonempty_relation_after
        < shortest_nonempty_relation_before)
        return TRUE;

    if (shortest_nonempty_relation_after
        > shortest_nonempty_relation_before)
        return FALSE;
}

if (change_in_total_length < 0)
    return TRUE;
if (change_in_total_length > 0)
    return FALSE;

if (change_in_num_runs < 0)
    return TRUE;
if (change_in_num_runs > 0)
    return FALSE;

if (group->minimize_number_of_generators == FALSE)
{
    if (shortest_nonempty_relation_after
        < shortest_nonempty_relation_before)
        return TRUE;

    if (shortest_nonempty_relation_after
        > shortest_nonempty_relation_before)
        return FALSE;
}

/*
 * The value of The Induction Variable wouldn't change,
 * so return FALSE.
 */
return FALSE;
}

static void evaluate_handle_slide_in_group(
    GroupPresentation *group,
    int a,
    int b,
    int *shortest_nonempty_relation_before,

```

```

    int             *shortest_nonempty_relation_after,
    int             *change_in_total_length,
    int             *change_in_num_runs)
{
    CyclicWord  *word;

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Deohn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            evaluate_handle_slide_on_word(
                word,
                a,
                b,
                shortest_nonempty_relation_before,
                shortest_nonempty_relation_after,
                change_in_total_length,
                change_in_num_runs);
}

static void evaluate_handle_slide_on_word(
    CyclicWord  *word,
    int         a,
    int         b,
    int         *shortest_nonempty_relation_before,
    int         *shortest_nonempty_relation_after,
    int         *change_in_total_length,
    int         *change_in_num_runs)
{
    int delta_length,
        delta_runs,
        old_length,
        new_length;

    delta_length = compute_delta_length(word, a, b);
    delta_runs   = compute_delta_runs (word, a, b);

    old_length = word->itsLength;
    new_length = old_length + delta_length;

    if (old_length < *shortest_nonempty_relation_before)
        *shortest_nonempty_relation_before = old_length;

    if (new_length < *shortest_nonempty_relation_after)
        *shortest_nonempty_relation_after = new_length;

    *change_in_total_length += delta_length;

    *change_in_num_runs += delta_runs;
}

static int compute_delta_length(
    CyclicWord  *word,
    int         a,
    int         b)
{
    /*
     * Lemma. Handle slides may cause cancellations, but they don't
     * cause "secondary cancellations". That is, when you do "a" -> "aB"
     * you may find that the following letter is a 'b', so you get
     * "ab" -> "aBb" -> "a" (indeed, this is the whole purpose of
     * handle slides). However, the "Bb" cancellation won't allow
     * any additional cancellations to occur. If you had "abA",
     * you'd just get "abA" -> "aBbbA" -> "abA". This last example
     * is an important special case in the following code.
     */

    int delta_length,
        i;
    Letter *letter;

```

```

delta_length = 0;

for (    letter = word->itsLetters, i = 0;
      i < word->itsLength;
      letter = letter->next, i++)
{
    /*
     * When we see 'a' or 'A' we increment the length of the word
     * to account for the handle slide. (Cancellations will be
     * dealt with momentarily.)
     */
    if (letter->itsValue == a || letter->itsValue == -a)
        delta_length++;

    /*
     * When we see a 'b' immediately preceded by an 'a' we decrement
     * the length of the word by two to account for the cancellation,
     * and similarly for a 'B' immediately followed by an 'A'.
     * As explained in the above lemma, these are the only
     * cancellations which may occur.
     */
    if ((letter->itsValue == b && letter->prev->itsValue == a)
        || (letter->itsValue == -b && letter->next->itsValue == -a))
        delta_length -= 2;
}

return delta_length;
}

static int compute_delta_runs(
    CyclicWord *word,
    int a,
    int b)
{
    /*
     * Counting the change in the number of runs is trickier than
     * counting the change in the length.
     *
     * Recall that the number of runs is actually the number of
     * transitions from one run to another; i.e. a word with one
     * run has no transitions.
     *
     * The number of transitions can change just after an 'a' or just
     * before an 'A', but nowhere else. Typically a handle slide
     * introduces one new transition after an 'a': "ac" -> "aBc".
     * The exceptional cases appear in the table below, along with
     * the number of new transitions each introduces.
     *
     *      "aa"  -> "aBa"      +2
     *      "aA"                (shouldn't occur)
     *      "aba" -> "aa"       -2
     *      "abA" -> "abA"      0
     *      "abb" -> "ab"       0
     *      "abB"                (shouldn't occur)
     *      "abc" -> "ac"       -1
     *      "aB"  -> "aBB"      0
     *      "ac"  -> "aBc"      +1
     *
     * A similar table lists the possible transition changes
     * preceding an 'A'.
     *
     *      "AA"  -> "AbA"      +2
     *      "aA"                (shouldn't occur)
     *      "ABA" -> "AA"       -2
     *      "aBA" -> "aBA"      0
     *      "BBA" -> "BA"       0
     *      "bBA"                (shouldn't occur)
     *      "CBA" -> "CA"       -1
     *      "bA"  -> "bbA"      0
     *      "CA"  -> "CbA"      +1
     *
     * The only case where we have to worry about "double counting"
     * is "aBA" -> "aBA". It gets considered once when we examine

```

```

    * the 'a' and again when we examine the 'A'. Fortunately the
    * net change in transitions is zero, so this "double counting"
    * is harmless.
    */

    int    delta_runs,
           i;
    Letter *letter;

    delta_runs = 0;

    for (    letter = word->itsLetters, i = 0;
          i < word->itsLength;
          letter = letter->next, i++)
    {
        if (letter->itsValue == a)
        {
            if (letter->next->itsValue ==  a)
                /*
                 * "aa" -> "aBa"
                 */
                delta_runs += 2;
            else
            if (letter->next->itsValue == -a)
                /*
                 * "aA" should not occur.
                 */
                uFatalError("compute_delta_runs", "fundamental_group");
            else
            if (letter->next->itsValue ==  b)
                /*
                 * "ab" -> "aBb" -> "a"
                 *
                 * Break into cases according to the value
                 * of the next letter.
                 */
                {
                    if (letter->next->next->itsValue ==  a)
                        /*
                         * "aba" -> "aBba" -> "aa"
                         */
                        delta_runs -= 2;
                    else
                    if (letter->next->next->itsValue == -a)
                        /*
                         * "abA" -> "aBbbA" -> "abA"
                         */
                        delta_runs += 0;
                    else
                    if (letter->next->next->itsValue ==  b)
                        /*
                         * "abb" -> "aBbb" -> "ab"
                         */
                        delta_runs += 0;
                    else
                    if (letter->next->next->itsValue == -b)
                        /*
                         * "abB" should not occur.
                         */
                        uFatalError("compute_delta_runs", "fundamental_group");
                    else
                        /*
                         * "abc" -> "aBbc" -> "ac"
                         */
                        delta_runs -= 1;
                }
            else
            if (letter->next->itsValue == -b)
                /*
                 * "aB" -> "aBB"
                 */
                delta_runs += 0;
            else
                /*

```

```

        * "ac" -> "aBc"
        */
        delta_runs += 1;
    }

    if (letter->itsValue == -a)
    {
        if (letter->prev->itsValue == -a)
            /*
             * "AA" -> "AbA"
             */
            delta_runs += 2;
        else
            if (letter->prev->itsValue == a)
                /*
                 * "aA" should not occur.
                 */
                uFatalError("compute_delta_runs", "fundamental_group");
            else
                if (letter->prev->itsValue == -b)
                    /*
                     * "BA" -> "BbA" -> "A"
                     *
                     * Break into cases according to the value
                     * of the preceding letter.
                     */
                    {
                        if (letter->prev->prev->itsValue == -a)
                            /*
                             * "ABA" -> "ABbA" -> "AA"
                             */
                            delta_runs -= 2;
                        else
                            if (letter->prev->prev->itsValue == a)
                                /*
                                 * "aBA" -> "aBBbA" -> "aBA"
                                 */
                                delta_runs += 0;
                            else
                                if (letter->prev->prev->itsValue == -b)
                                    /*
                                     * "BBA" -> "BBbA" -> "BA"
                                     */
                                    delta_runs += 0;
                                else
                                    if (letter->prev->prev->itsValue == b)
                                        /*
                                         * "bBA" should not occur.
                                         */
                                        uFatalError("compute_delta_runs", "fundamental_group");
                                    else
                                        /*
                                         * "CBA" -> "CBbA" -> "CA"
                                         */
                                        delta_runs -= 1;
                                }
                            else
                                if (letter->prev->itsValue == b)
                                    /*
                                     * "bA" -> "bbA"
                                     */
                                    delta_runs += 0;
                                else
                                    /*
                                     * "CA" -> "CbA"
                                     */
                                    delta_runs += 1;
                                }
                    }
            }
    }

    return delta_runs;
}

```

```

static Boolean two_singletons_in_group(
    GroupPresentation *group)
{
    /*
     * If a generator occurs in precisely one word, and occurs in that
     * word precisely twice, both times with the same sign,
     * then we may do handle slides to make the two occurrences of the
     * generator adjacent to one another, without lengthening the word.
     * For example, we could simplify the word "aabAAb" to "aaaabb".
     *
     * This is a geometric operation. It preserves the pseudo-Heegaard
     * diagram discussed at the top of this file.
     */

    int i;
    CyclicWord *word_containing_singletons;

    for (i = 1; i <= group->itsNumGenerators; i++)

        if (generator_occurs_as_two_singletons_in_group(group, i, &
word_containing_singletons)
            && generator_occurs_in_no_other_word_in_group(group, i,
word_containing_singletons))
        {
            make_singletons_adjacent(group, i, word_containing_singletons);
            return TRUE;
        }

    return FALSE;
}

static Boolean generator_occurs_as_two_singletons_in_group(
    GroupPresentation *group,
    int value,
    CyclicWord **word_containing_singletons)
{
    CyclicWord *word;

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Dehn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            if (generator_occurs_as_two_singletons_in_word(word, value) == TRUE)
            {
                *word_containing_singletons = word;
                return TRUE;
            }

    *word_containing_singletons = NULL;

    return FALSE;
}

static Boolean generator_occurs_as_two_singletons_in_word(
    CyclicWord *word,
    int value)
{
    int num_plus,
        num_minus,
        i;
    Letter *letter;

    num_plus = 0;
    num_minus = 0;

    for (letter = word->itsLetters, i = 0;
         i < word->itsLength;
         letter = letter->next, i++)
    {
        /*
         * Count positive occurrences.

```



```

        */
        if (letter->itsValue == value)
            num_plus++;

        /*
         * Count negative occurrences.
         */
        if (letter->itsValue == - value)
            num_minus++;

        /*
         * Reject consecutive occurrences.
         */
        if
        (
            ( letter->itsValue == value
              || letter->itsValue == - value)
            &&
            ( letter->next->itsValue == value
              || letter->next->itsValue == - value)
        )
            return FALSE;
    }

    return ((num_plus == 2 && num_minus == 0)
           || (num_plus == 0 && num_minus == 2));
}

static Boolean generator_occurs_in_no_other_word_in_group(
    GroupPresentation *group,
    int value,
    CyclicWord *word_containing_singletons)
{
    CyclicWord *word;

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Dehn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            if (word != word_containing_singletons
                && generator_occurs_in_word(word, value) == TRUE)

                return FALSE;

    return TRUE;
}

static Boolean generator_occurs_in_word(
    CyclicWord *word,
    int value)
{
    Letter *letter;
    int i;

    for ( letter = word->itsLetters, i = 0;
          i < word->itsLength;
          letter = letter->next, i++)

        if (letter->itsValue == value
            || letter->itsValue == - value)

            return TRUE;

    return FALSE;
}

static void make_singletons_adjacent(
    GroupPresentation *group,
    int value,
    CyclicWord *word)

```

```

{
    /*
     * Other functions have already verified that value occurs exactly
     * twice in word, both times with the same sign, and occurs nowhere
     * else in the GroupPresentation.
     */

    /*
     * Advance word->itsLetters to point at an occurrence of value
     * or its inverse.
     */

    while ( word->itsLetters->itsValue != value
            && word->itsLetters->itsValue != - value)

        word->itsLetters = word->itsLetters->next;

    /*
     * Do handle slides until the two occurrence of value are adjacent.
     */

    while (word->itsLetters->itsValue != word->itsLetters->next->itsValue)

        handle_slide(    group,
                        word->itsLetters->itsValue,
                        word->itsLetters->next->itsValue);
}

static Boolean eliminate_word_in_group(
    GroupPresentation *group)
{
    /*
     * Look for a generator which occurs precisely once in some word,
     * and use the word to eliminate the generator.  For example,
     * say the word is "bcacb".  First do handle slides to reduce it
     * to "a", and then do a handle cancellation.
     *
     * Try to choose a generator and a word which increase the total
     * length of all relations as little as possible.
     * If group->minimize_number_of_generators is FALSE, insist that
     * the total length of all relations must decrease.
     *
     * Return TRUE if successful, FALSE if no such word exists.
     */

    int          delta_length,
                best_delta,
                best_generator,
                m,
                n,
                generator;
    CyclicWord   *word_with_singleton,
                *best_word;

    best_delta    = INT_MAX;
    best_generator = 0;
    best_word     = NULL;

    for (generator = 1; generator <= group->itsNumGenerators; generator++)
    {
        word_with_singleton
            = shortest_word_in_which_generator_occurs_precisely_once
              (group, generator);

        if (word_with_singleton != NULL)
        {
            /*
             * By how much would the total length of all relations increase
             * if we eliminated this generator via this word?
             */

            m = word_with_singleton->itsLength;
            n = occurrences_in_group(group, generator);

```

```

        delta_length      = n*(m-1)    /* effect of handle slides          */
                          - 2*(m-1)    /* effect of cancellations in this word */
                          - n;         /* effect of eliminating generator      */

        if (delta_length < best_delta)
        {
            best_delta      = delta_length;
            best_generator    = generator;
            best_word        = word_with_singleton;
        }
    }
}

if
(
    best_word != NULL
    && (
        group->minimize_number_of_generators == TRUE
        || best_delta < 0
    )
)
{
    eliminate_word(group, best_word, best_generator);
    return TRUE;
}
else
    return FALSE;
}

static CyclicWord *shortest_word_in_which_generator_occurs_precisely_once(
    GroupPresentation *group,
    int generator)
{
    /*
     * Find the shortest word in which the generator occurs precisely once.
     */

    CyclicWord *word,
               *best_word;

    best_word = NULL;

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Dehn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            if (generator_occurs_precisely_once_in_word(word, generator) == TRUE)

                if (best_word == NULL
                    || word->itsLength < best_word->itsLength)

                    best_word = word;

    return best_word;
}

static Boolean generator_occurs_precisely_once_in_word(
    CyclicWord *word,
    int generator)
{
    return (occurrences_in_word(word, generator) == 1);
}

static int occurrences_in_group(
    GroupPresentation *group,
    int generator)
{
    int occurrences;
    CyclicWord *word;

```

```

    occurrences = 0;

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Dehn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            occurrences += occurrences_in_word(word, generator);

    return occurrences;
}

static int occurrences_in_word(
    CyclicWord *word,
    int generator)
{
    int i,
        num_occurrences;
    Letter *letter;

    num_occurrences = 0;

    for (letter = word->itsLetters, i = 0;
         i < word->itsLength;
         letter = letter->next, i++)

        if (letter->itsValue == generator
            || letter->itsValue == - generator)

            num_occurrences++;

    return num_occurrences;
}

static void eliminate_word(
    GroupPresentation *group,
    CyclicWord *word,
    int generator)
{
    Letter *letter;

    /*
     * eliminate_word_in_group() should have already checked
     * that generator occurs precisely once in word.
     */

    if (generator_occurs_precisely_once_in_word(word, generator) == FALSE)
        uFatalError("eliminate_word", "fundamental_group");

    /*
     * Find the Letter containing the unique occurrence of the generator.
     */

    for (letter = word->itsLetters;
         letter->itsValue != generator && letter->itsValue != - generator;
         letter = letter->next
        )
        ;

    /*
     * Do handle slides until only "letter" is left, then
     * do a handle cancellation.
     */

    while (word->itsLength > 1)
        handle_slide(group, letter->itsValue, letter->next->itsValue);

    cancel_handles(group, word);
}

```

```

static Boolean remove_empty_relations(
    GroupPresentation *group)
{
    Boolean words_were_removed;
    CyclicWord **list,
                *dead_word;

    words_were_removed = FALSE;

    list = &group->itsRelations;

    while (*list != NULL)
    {
        if ((*list)->itsLength == 0)
        {
            dead_word = *list;
            *list = (*list)->next;
            free_cyclic_word(dead_word);
            group->itsNumRelations--;
            words_were_removed = TRUE;
        }
        else
            list = &(*list)->next;
    }

    return words_were_removed;
}

static Boolean insert_word_from_group(
    GroupPresentation *group)
{
    /*
     * Try to insert a copy of one word into another so that after
     * cancellations the second word is shorter than it used to be.
     * For example, in the presentation
     *
     *      BaBABABaB
     *      BaBAA
     *
     * we can substitute the inverse of the second word into the
     * first word to obtain
     *
     *      BaBA(abAba)BABaB
     *      BaBAA
     *
     * which simplifies to
     *
     *      aBABaB
     *      BaBAA
     *
     * Do it again to obtain
     *
     *      aBABaB(bAbaa)
     *      BaBAA
     *
     * which simplifies to
     *
     *      aBa
     *      BaBAA
     *
     * The usual geometric simplifications now reduce this presentation to
     *
     *      AAAAA
     */
    CyclicWord *word;

    /*
     * Consider all possible words which we might want to insert
     * into another word.
     */
    for (word = group->itsRelations; word != NULL; word = word->next)

```

```

    if (word->is_Deohn_relation == FALSE
        || group->fillings_may_affect_generators == TRUE)

        if (insert_word_into_group(group, word) == TRUE)

            return TRUE;

    return FALSE;
}

static Boolean insert_word_into_group(
    GroupPresentation *group,
    CyclicWord *word)
{
    /*
     * The "word" shouldn't be a Dehn relation if
     * group->fillings_may_affect_generators is FALSE,
     * but the "target" can always be any kinds of relation.
     * Simplifying the Dehn relations won't have any effect
     * on either the choice of generators or the edge relations.
     */

    return
    (    insert_word_into_list(group->itsRelations, word) == TRUE
      || insert_word_into_list(group->itsMeridians, word) == TRUE
      || insert_word_into_list(group->itsLongitudes, word) == TRUE
      || insert_word_into_list(group->itsOriginalGenerators, word) == TRUE);
}

static Boolean insert_word_into_list(
    CyclicWord *list,
    CyclicWord *word)
{
    CyclicWord *target;

    for (target = list; target != NULL; target = target->next)

        if (insert_word_into_word(word, target) == TRUE)

            return TRUE;

    return FALSE;
}

static Boolean insert_word_into_word(
    CyclicWord *word,
    CyclicWord *target)
{
    int i,
        j;

    /*
     * Don't insert a word into itself.
     */
    if (word == target)
        return FALSE;

    /*
     * One CyclicWord may be inserted into another in many different
     * ways. For example, if word = "abc" and target = "defg", there
     * are 24 ways to insert word into target:
     *
     *      (abc)defg    d(abc)efg    de(abc)fg    def(abc)g
     *      (bca)defg    d(bca)efg    de(bca)fg    def(bca)g
     *      (cab)defg    d(cab)efg    de(cab)fg    def(cab)g
     *
     *      (CBA)defg    d(CBA)efg    de(CBA)fg    def(CBA)g
     *      (ACB)defg    d(ACB)efg    de(ACB)fg    def(ACB)g
     *      (BAC)defg    d(BAC)efg    de(BAC)fg    def(BAC)g
     *
     * Comment: The algorithm considers cancellations at only one end

```

```

    * of the inserted word. For example, when it inserts "abcd" into
    * "ADCef" to obtain "A(abcd)DCef", the program will consider the
    * cancellations "cdDC", but will ignore the "Aa". This is OK.
    * At some other point in its nested loops it will consider
    * "(bcda)ADCef", thereby recognizing the fullest cancellation.
    *
    * We use word->itsLetters and target->itsLetters to mark
    * the possible insertion points. Note too that the following
    * code automatically ignores relations of length zero.
    */

for (i = 0; i < word->itsLength; i++)
{
    for (j = 0; j < target->itsLength; j++)
    {
        if (insert_word_forwards(word, target) == TRUE
            || insert_word_backwards(word, target) == TRUE)

            return TRUE;

        target->itsLetters = target->itsLetters->next;
    }

    word->itsLetters = word->itsLetters->next;
}

return FALSE;
}

static Boolean insert_word_forwards(
    CyclicWord *word,
    CyclicWord *target)
{
    /*
    * If, say, word = "abc" and target = "defg",
    * would replacing target with "abcdefg" reduce the length
    * of target after cancelling inverses?
    */

    int remaining_cancellations,
        i;
    Letter *next_letter_in_word,
        *next_letter_in_target,
        *letter,
        *letter_copy;

    /*
    * insert_word_into_word guarantees that
    * both word and target are nonempty.
    */
    if (word->itsLength == 0
        || target->itsLength == 0)
        uFatalError("insert_word_forwards", "fundamental_group");

    /*
    * More than half the Letters in word must cancel with letters in target.
    */
    remaining_cancellations = (word->itsLength + 2) / 2;

    /*
    * If target isn't long enough, let's give up now so we can ignore
    * the cyclic nature of the target in the code below. (Otherwise
    * a target "ab" might give the illusion of completely cancelling
    * a word "BABABABABA".)
    */
    if (target->itsLength < remaining_cancellations)
        return FALSE;

    /*
    * Check whether the last remaining_cancellations Letters in word
    * cancel with the first remaining_cancellations Letters in target.
    * (See the comment in insert_word_into_word() for an explanation
    * of why it's OK to ignore possible cancellations of letters

```

```

    * at the beginning of word with letters at the end of target.)
    */

    next_letter_in_word    = word->itsLetters->prev;
    next_letter_in_target  = target->itsLetters;

    while (remaining_cancellations > 0)
    {
        if (next_letter_in_word->itsValue + next_letter_in_target->itsValue == 0)
        {
            remaining_cancellations--;
            next_letter_in_word    = next_letter_in_word    ->prev;
            next_letter_in_target  = next_letter_in_target->next;
        }
        else
            return FALSE;
    }

    /*
     * Great! They cancel!
     * Let's insert a copy of word into target, and do the cancellations.
     * If we're lucky, we may even get more cancellations than just the
     * minimum.
     */

    /*
     * Insert the copy.
     */
    for (    letter = word->itsLetters, i = 0;
          i < word->itsLength;
          letter = letter->next, i++)
    {
        letter_copy = NEW_STRUCT(Letter);
        letter_copy->itsValue = letter->itsValue;
        INSERT_BEFORE(letter_copy, target->itsLetters);
        target->itsLength++;
    }

    /*
     * Do the cancellations.
     */
    cancel_inverses_word(target);

    return TRUE;
}

static Boolean insert_word_backwards(
    CyclicWord *word,
    CyclicWord *target)
{
    /*
     * If, say, word = "abc" and target = "defg",
     * would replacing target with "CBAdefg" reduce the length
     * of target after cancelling inverses?
     */

    int    remaining_cancellations,
           i;
    Letter *next_letter_in_word,
           *next_letter_in_target,
           *letter,
           *letter_copy;

    /*
     * insert_word_into_word guarantees that
     * both word and target are nonempty.
     */
    if (word->itsLength == 0
        || target->itsLength == 0)
        uFatalError("insert_word_backwards", "fundamental_group");

    /*
     * More than half the Letters in word must cancel with letters in target.

```



```

    */
    remaining_cancellations = (word->itsLength + 2) / 2;

    /*
    * If target isn't long enough, let's give up now so we can ignore
    * the cyclic nature of the target in the code below. (Otherwise
    * a target "ab" might gives the illusion of completely cancelling
    * a word "ababababab".)
    */
    if (target->itsLength < remaining_cancellations)
        return FALSE;

    /*
    * Check whether the first remaining_cancellations Letters in word
    * match the first remaining_cancellations Letters in target.
    * (See the comment in insert_word_into_word() for an explanation
    * of why it's OK to ignore possible matches of letters at the ends
    * of word and target.)
    */

    next_letter_in_word      = word->itsLetters;
    next_letter_in_target    = target->itsLetters;

    while (remaining_cancellations > 0)
    {
        if (next_letter_in_word->itsValue == next_letter_in_target->itsValue)
        {
            remaining_cancellations--;
            next_letter_in_word = next_letter_in_word ->next;
            next_letter_in_target = next_letter_in_target->next;
        }
        else
            return FALSE;
    }

    /*
    * Great! They cancel!
    * Let's insert a copy of word's inverse into target, and do the
    * cancellations. If we're lucky, we may even get more cancellations
    * than just the minimum.
    */

    /*
    * Insert the copy.
    */
    for (    letter = word->itsLetters->prev, i = 0;
          i < word->itsLength;
          letter = letter->prev, i++)
    {
        letter_copy = NEW_STRUCT(Letter);
        letter_copy->itsValue = - letter->itsValue;
        INSERT_BEFORE(letter_copy, target->itsLetters);
        target->itsLength++;
    }

    /*
    * Do the cancellations.
    */
    cancel_inverses_word(target);

    return TRUE;
}

static Boolean simplify_one_word_presentations(
    GroupPresentation *group)
{
    /*
    * In general we'd like one-word presentations to be as simple
    * as possible. In particular, we'd like to display the fundamental
    * group of a torus knot as  $a^n = b^m$ , so the user can recognize
    * it easily.
    *
    * Often the presentation for a torus knot group is something like
    */

```



```

    {
        /*
         * Create the new word,
         * e.g. {bbaabbbbaabbbaa} -> {bbaabbbbaabbbaa, Cbbaa}.
         */
        new_word = introduce_generator( group,
                                       unmatched_letter->next,
                                       period);

        /*
         * Make sure the new_word is inserted
         * at the correct position.
         * JRW 28 Feb 2002
         */
        /* Cbbaa -> bbbaaC */
        new_word->itsLetters = new_word->itsLetters->next;
        /* bbaabbbbaabbbaa -> bbaabbbbaabbbaaC */
        word->itsLetters = unmatched_letter->next;

        /*
         * Insert the new word into the old,
         * e.g. {bbaabbbbaabbbaaC, bbbaaC}
         * -> {cbbaabbbbaaC, bbbaaC}
         * -> {bbaabbbbaaC, bbbaaC}
         * -> {cbbaabbbbaaC, bbbaaC}
         * -> {bbaabbbbaaC, bbbaaC}
         * -> {cbbaabbbbaaC, bbbaaC}
         * -> {bbaabbbbaaC, bbbaaC}
         */
        for (j = 0; j < repetitions; j++)
        {
            if (insert_word_backwards(new_word, word) == FALSE)
                uFatalError("simplify_one_word_presentations",
                           "fundamental_group");
            word->itsLetters = word->itsLetters->next;
        }

        /*
         * Eliminate the original word,
         * e.g. {bbaabbbbaaC, bbbaaC} -> {bbaabbbbaaC}.
         */
        eliminate_word(group, word, unmatched_letter->itsValue);

        /*
         * All done.
         */
        return TRUE;
    }

    return FALSE;
}

static Boolean word_contains_pattern(
    Letter    *unmatched_letter,
    int       period,
    int       repetitions)
{
    int       i,
             j,
             k;
    Letter    *letter,
             *image;

    for (i = 0, letter = unmatched_letter->next;
         i < period;
         i++, letter = letter->next)
    {
        image = letter;

        for (j = 0;
             j < repetitions;
             j++)
    }

```

```

    {
        if (image->itsValue != letter->itsValue)
            return FALSE;

        for (k = 0; k < period; k++)
            image = image->next;
    }

    return TRUE;
}

static CyclicWord *introduce_generator(
    GroupPresentation *group,
    Letter *substring,
    int length)
{
    /*
     * Introduce a new generator into the group presentation.
     * It will be of the form c = baaba, where baaba is a substring of
     * some other word; the function argument "substring" points to the
     * first letter in the substring, and "length" gives its length.
     */

    O3lMatrix *new_array,
               the_inverse;
    int i;
    Letter *letter,
           *new_generator_letter,
           *letter_copy;
    CyclicWord *new_word;

    /*
     * Should the new relation be an edge relation or a Dehn relation?
     * Really, of course, it is neither.
     * But to have gotten this far, either
     * group->fillings_may_affect_generators is TRUE, or it is FALSE and
     * the only relation is an edge relation. So it is safe to call
     * the new relation an edge relation.
     */
    if
    (
        group->fillings_may_affect_generators == FALSE
        &&
        (
            group->itsNumRelations != 1
            || group->itsRelations->is_Deohn_relation == TRUE
        )
    )
        uFatalError("introduce_generator", "fundamental_group");

    /*
     * Create the new generator.
     */

    /*
     * Allocate a bigger array for the matrix generators,
     * copy in the existing values, and free the old array.
     */
    new_array = NEW_ARRAY(group->itsNumGenerators + 1, O3lMatrix);
    for (i = 0; i < group->itsNumGenerators; i++)
        o3l_copy(new_array[i], group->itsMatrices[i]);
    my_free(group->itsMatrices);
    group->itsMatrices = new_array;

    /*
     * Create the new matrix.
     */

    o3l_copy(group->itsMatrices[group->itsNumGenerators], O3l_identity);

    for (i = 0, letter = substring;
         i < length;

```

```

        i++, letter = letter->next)

    if (letter->itsValue > 0)

        o31_product(group->itsMatrices[group->itsNumGenerators],
                    group->itsMatrices[letter->itsValue - 1],
                    group->itsMatrices[group->itsNumGenerators]);

    else
    {
        o31_invert( group->itsMatrices[(-letter->itsValue) - 1],
                    the_inverse);
        o31_product(group->itsMatrices[group->itsNumGenerators],
                    the_inverse,
                    group->itsMatrices[group->itsNumGenerators]);
    }

/*
 * Increment group->itsNumGenerators.
 */
group->itsNumGenerators++;

/*
 * Create the new relation.
 *
 * If the new word is, say, c = babba, we'll construct it as Cbabba.
 * Note that it begins with the inverse of the new generator.
 */

new_generator_letter = NEW_STRUCT(Letter);
new_generator_letter->itsValue = - group->itsNumGenerators;
new_generator_letter->next = new_generator_letter;
new_generator_letter->prev = new_generator_letter;

new_word = NEW_STRUCT(CyclicWord);
new_word->itsLength = length + 1;
new_word->itsLetters = new_generator_letter;
for ( i = 0, letter = substring;
      i < length;
      i++, letter = letter->next)
{
    letter_copy = NEW_STRUCT(Letter);
    letter_copy->itsValue = letter->itsValue;
    INSERT_BEFORE(letter_copy, new_generator_letter);
}

/*
 * The new_word may be considered an edge relation, as explained above.
 */
new_word->is_Deohn_relation = FALSE;

new_word->next = group->itsRelations;
group->itsRelations = new_word;
group->itsNumRelations++;

return new_word;
}

static void lens_space_recognition(
    GroupPresentation *group)
{
/*
 * We want to be able to recognize a presentation like
 *
 *      bbbaaaa
 *      bbbbbbbbaaa
 *
 * as a lens space. We use the Euclidean algorithm.
 * (In this example it would be more efficient to start with the a's,
 * but I'll start with the b's instead so I can show two iterations
 * of the basic process.)
 * Interpret the first relation as bbb = AAAA, and substitute it
 * into the second relation.
 */

```

```

*
*      bbbaaaa
*      AAAAAAAbbaaa
*
* Cancel AAA with aaa.
*
*      bbbaaaa
*      AAAAAbb
*
* Now interpret the second relation as bb = aaaaa and substitute
* it back into the first relation.
*
*      baaaaaaaaa
*      AAAAAbb
*
* Interpret the first relation as b = AAAAAAAA and substitute it
* into the second relation.
*
*      baaaaaaaaa
*      AAAAAAAAAAAAAAAAAAAAAA
*
* Eliminate generator b.
*
*      AAAAAAAAAAAAAAAAAAAAAA
*
* This shows that the group is Z/23.
*
* (The actual code in lens_space_recognition_using_generator() uses
* a slightly different implementation, but this is the general idea.)
*/

int generator;

/*
* Do we have permission to reduce the number of generators?
*/
if (group->fillings_may_affect_generators == FALSE)
    return;

/*
* A more general version of this code would apply to free
* products of finite cyclic groups with other groups.
* For now we insist that the group as a whole is finite cyclic.
*/

/*
* Are there exactly two generators?
*/
if (group->itsNumGenerators != 2)
    return;

/*
* Are there exactly two relations?
* (Note: By this point empty relations will have been eliminated.)
*/
if (group->itsNumRelations != 2)
    return;

/*
* Do a quick error check.
*/
if (group->itsRelations == NULL
    || group->itsRelations->next == NULL
    || group->itsRelations->next->next != NULL)
    uFatalError("lens_space_recognition", "fundamental_group");

/*
* Are both words of the form a^m = b^n ?
*/
if (count_runs( group->itsRelations ) > 2
    || count_runs( group->itsRelations->next ) > 2)
    return;

/*

```

```

    * Try the Euclidean algorithm on each generator in turn.
    * Break if successful.
    */
for (generator = 1; generator <= 2; generator++)
    if (lens_space_recognition_using_generator(group, generator) == TRUE)
        break;
}

static int count_runs(
    CyclicWord *word)
{
    int    num_runs,
           i;
    Letter *letter;

    num_runs = 0;

    for (    letter = word->itsLetters, i = 0;
           i < word->itsLength;
           letter = letter->next, i++)

        if (letter->itsValue != letter->next->itsValue)

            num_runs++;

    return num_runs;
}

static Boolean lens_space_recognition_using_generator(
    GroupPresentation *group,
    int generator0)
{
    int    i,
           j,
           value_of_b,
           power_of_b,
           occurrences[2],
           generator[2],
           n[2][2];
    long int p,
            q;
    CyclicWord *word[2],
               *new_word;
    Letter *letter_a,
           *letter_b;

    /*
     * Note the two words.
     */
    word[0] = group->itsRelations;
    word[1] = group->itsRelations->next;

    /*
     * How many times does the generator occur in each word?
     */
    for (i = 0; i < 2; i++)
        occurrences[i] = occurrences_in_word(word[i], generator0);

    /*
     * If this generator occurs in only one word, give up.
     */
    for (i = 0; i < 2; i++)
        if (occurrences[i] == 0)
            return FALSE;

    /*
     * If the number of occurrences are not relatively prime,
     * give up.
     */
    if (gcd(occurrences[0], occurrences[1]) > 1)
        return FALSE;
}

```

```

/*
 * Think of the given generator as 'a' and the
 * other generator as 'b'.
 */
generator[0] = generator0;
generator[1] = 3 - generator0;

/*
 * Count the number of signed occurrences
 * of each generator in each word.
 */
for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
        n[i][j] = count_signed_occurrences_in_word(word[i], generator[j]);

/*
 * The relations are
 *
 *       $a^{n[0][0]} * b^{n[0][1]} = 1$ 
 *       $a^{n[1][0]} * b^{n[1][1]} = 1$ 
 * or
 *       $a^{n[0][0]} = b^{-n[0][1]}$ 
 *       $a^{n[1][0]} = b^{-n[1][1]}$ 
 *
 * Find p and q such that  $p*n[0][0] + q*n[1][0] = \gcd(n[0][0], n[1][0]) = 1$ .
 */
(void) euclidean_algorithm(n[0][0], n[1][0], &p, &q);

/*
 *  $a = a^1$ 
 *  $= a^{(p*n[0][0] + q*n[1][0])}$ 
 *  $= a^{(p*n[0][0])} * a^{(q*n[1][0])}$ 
 *  $= (a^{n[0][0]})^p * (a^{n[1][0]})^q$ 
 *  $= (b^{-n[0][1]})^p * (b^{-n[1][1]})^q$ 
 *  $= b^{(-p*n[0][1])} * b^{(-q*n[1][1])}$ 
 *  $= b^{-(p*n[0][1] + q*n[1][1])}$ 
 *
 * Add the redundant relation
 *
 *       $a * b^{(p*n[0][1] + q*n[1][1])} = 1$ 
 *
 * (Remember -- nongeometric operations are OK.)
 */

value_of_b = (p*n[0][1] + q*n[1][1] >= 0) ? generator[1] : -generator[1];
power_of_b = ABS(p*n[0][1] + q*n[1][1]);

new_word = NEW_STRUCT(CyclicWord);
new_word->itsLength = 1 + power_of_b;
new_word->is_Dehn_relation = FALSE;
new_word->next = group->itsRelations;
group->itsRelations = new_word;
group->itsNumRelations++;

letter_a = NEW_STRUCT(Letter);
letter_a->itsValue = generator[0];
letter_a->prev = letter_a;
letter_a->next = letter_a;
new_word->itsLetters = letter_a;

for (i = 0; i < power_of_b; i++)
{
    letter_b = NEW_STRUCT(Letter);
    letter_b->itsValue = value_of_b;
    INSERT_AFTER(letter_b, letter_a);
}

/*
 * Use the new_word to eliminate generator 'a'.
 */
eliminate_word(group, new_word, generator[0]);

/*
 * We're left with two relations of the form

```



```

    *
    *      b^r = 1
    *      b^s = 1
    *
    * Keep substituting one into the other until one
    * becomes trivial and can be removed.
    */
while (remove_empty_relations(group) == FALSE)

    if (insert_word_from_group(group) == FALSE)

        uFatalError("lens_space_recognition_using_generator", "fundamental_group");

return TRUE;
}

static Boolean invert_generators_where_necessary(
GroupPresentation *group)
{
    /*
    * If a generator appears more often as an inverse than as a
    * positive power, replace it with its inverse.
    * E.g. {AAbbc, abCCC} -> {aabbC, Abccc}.
    */

    int a,
        positive_occurrences,
        negative_occurrences;
    Boolean progress;

    progress = FALSE;

    for (a = 1; a <= group->itsNumGenerators; a++)
    {
        count_signed_occurrences_in_group( group,
                                            a,
                                            &positive_occurrences,
                                            &negative_occurrences);

        if (negative_occurrences > positive_occurrences)
        {
            invert_generator_in_group(group, a);
            progress = TRUE;
        }
    }

    return progress;
}

static void count_signed_occurrences_in_group(
GroupPresentation *group,
int a,
int *positive_occurrences,
int *negative_occurrences)
{
    *positive_occurrences = 0;
    *negative_occurrences = 0;

    increment_signed_occurrences_in_group( group,
                                            a,
                                            positive_occurrences,
                                            negative_occurrences);
}

static void increment_signed_occurrences_in_group(
GroupPresentation *group,
int a,
int *positive_occurrences,
int *negative_occurrences)
{
    CyclicWord *word;

```

```

    for (word = group->itsRelations; word != NULL; word = word->next)

        if (word->is_Dehn_relation == FALSE
            || group->fillings_may_affect_generators == TRUE)

            increment_signed_occurrences_in_word(    word,
                                                    a,
                                                    positive_occurrences,
                                                    negative_occurrences);
}

static void increment_signed_occurrences_in_word(
    CyclicWord *word,
    int a,
    int *positive_occurrences,
    int *negative_occurrences)
{
    Letter *letter;
    int i;

    for (    letter = word->itsLetters, i = 0;
          i < word->itsLength;
          letter = letter->next, i++)
    {
        if (letter->itsValue == a)
            (*positive_occurrences)++;

        if (letter->itsValue == -a)
            (*negative_occurrences)++;
    }
}

static int count_signed_occurrences_in_word(
    CyclicWord *word,
    int a)
{
    Letter *letter;
    int i,
        num_occurrences;

    num_occurrences = 0;

    for (    letter = word->itsLetters, i = 0;
          i < word->itsLength;
          letter = letter->next, i++)
    {
        if (letter->itsValue == a)
            num_occurrences++;

        if (letter->itsValue == -a)
            num_occurrences--;
    }

    return num_occurrences;
}

static void invert_generator_in_group(
    GroupPresentation *group,
    int a)
{
    if (a < 1 || a > group->itsNumGenerators)
        uFatalError("invert_generator_in_group", "fundamental_group");

    o3l_invert(group->itsMatrices[a - 1], group->itsMatrices[a - 1]);

    invert_generator_on_list(group->itsRelations, a);
    invert_generator_on_list(group->itsMeridians, a);
    invert_generator_on_list(group->itsLongitudes, a);
    invert_generator_on_list(group->itsOriginalGenerators, a);
}

```

```

static void invert_generator_on_list(
    CyclicWord *list,
    int a)
{
    CyclicWord *word;

    for (word = list; word != NULL; word = word->next)

        invert_generator_in_word(word, a);
}

static void invert_generator_in_word(
    CyclicWord *word,
    int a)
{
    Letter *letter;
    int i;

    for (letter = word->itsLetters, i = 0;
         i < word->itsLength;
         letter = letter->next, i++)
    {
        if (letter->itsValue == a)
            letter->itsValue = -a;
        else
            if (letter->itsValue == -a)
                letter->itsValue = a;
    }
}

static Boolean invert_words_where_necessary(
    GroupPresentation *group)
{
    /*
     * Design decision: I considered inverting peripheral words where
     * necessary as well, but decided against it. Inverting a peripheral
     * word corresponds to reorienting the curve it represents. We
     * would still get the same length and torsion, so the reorientation
     * may be harmless, but it seems like a less than robust approach.
     * Who knows what somebody may someday do with this code. I'd hate
     * to introduce a bizarre bug. Better to just live with peripheral
     * curves which may contain more inverses than absolutely necessary.
     */

    CyclicWord *word;
    Boolean progress;

    progress = FALSE;

    for (word = group->itsRelations; word != NULL; word = word->next)
    {
        /*
         * Fix the word no matter what, but ...
         */
        if (invert_word_if_necessary(word) == TRUE)
        {
            /*
             * ...set progress = TRUE iff this inversion might allow
             * some generator to be profitably inverted as well.
             */
            if (word->is_Dehn_relation == FALSE
                || group->fillings_may_affect_generators == TRUE)
                progress = TRUE;
        }

        return progress;
    }
}

static Boolean invert_word_if_necessary(

```

```
CyclicWord *word)
{
    if (sum_of_powers(word) < 0)
    {
        invert_word(word);
        return TRUE;
    }
    else
        return FALSE;
}

static int sum_of_powers(
    CyclicWord *word)
{
    Letter *letter;
    int i,
        sum;

    sum = 0;

    for ( letter = word->itsLetters, i = 0;
          i < word->itsLength;
          letter = letter->next, i++)
    {
        if (letter->itsValue > 0)
            sum++;

        if (letter->itsValue < 0)
            sum--;
    }

    return sum;
}

static void invert_word(
    CyclicWord *word)
{
    /*
     * For each Letter we must
     *
     * (1) negate itsValue, and
     *
     * (2) switch prev and next.
     *
     * The tricky part is keeping track of the Letters
     * while their prev and next fields are in flux.
     */

    Letter *letter,
        *temp;

    if (word->itsLength == 0)
        return;

    letter = word->itsLetters;
    do
    {
        letter->itsValue = - letter->itsValue;

        temp          = letter->prev;
        letter->prev    = letter->next;
        letter->next    = temp;

        /*
         * This is the delicate step.
         * We've swapped prev and next, so to move on to what used
         * to be the next Letter, we follow the prev pointer.
         */
        letter = letter->prev;
    } while (letter != word->itsLetters);
}
```

```

static void choose_word_starts(
    GroupPresentation *group)
{
    CyclicWord *word;

    for (word = group->itsRelations; word != NULL; word = word->next)
        choose_word_start(word);
}

static void choose_word_start(
    CyclicWord *word)
{
    /*
     * The starting point of a cyclic word is arbitrary.
     * Choose it to meet the following criteria, in descending
     * order of importance:
     *
     * (1) Don't start a word in the middle of a run.
     * (2) Start with a positive power of a generator.
     * (3) Start with the lowest-numbered generator.
     *
     * Note the following code considers the letters of the CyclicWord
     * in order, so once it switches word->itsLetters to the beginning
     * of a run, it will always be at the beginning of a run.
     * Similarly, once it switches to a positive value, it will always
     * be at a positive value.
     */

    Letter *letter;
    int i;

    for (letter = word->itsLetters, i = 0;
         i < word->itsLength;
         letter = letter->next, i++)

        if
        (
            (word->itsLetters->itsValue == word->itsLetters->prev->itsValue
             && letter->itsValue != letter->prev->itsValue)
            ||
            (word->itsLetters->itsValue < 0
             && letter->itsValue > 0)
            ||
            (letter->itsValue > 0
             && letter->itsValue < word->itsLetters->itsValue)
            ||
            (letter->itsValue < 0
             && letter->itsValue > word->itsLetters->itsValue)
        )
            word->itsLetters = letter;
}

static void conjugate_peripheral_words(
    GroupPresentation *group)
{
    /*
     * Can we conjugate any (meridian, longitude) pairs
     * to shorten their length? E.g. (CBChcc, Cac) -> (BCbc, a)
     * or (cbbbc, Caac) -> (ccbb, aa).
     */

    int i;
    CyclicWord *theMeridian,
                *theLongitude;

    for
    (
        i = 0,
        theMeridian = group->itsMeridians,

```

```

        theLongitude = group->itsLongitudes;
    i < group->itsNumCusps;
    i++,
        theMeridian = theMeridian ->next,
        theLongitude = theLongitude->next
    )
    {
        if (theMeridian == NULL
            || theLongitude == NULL)
            uFatalError("conjugate_peripheral_words", "fundamental_group");

        /*
         * Conjugate as necessary.
         */
        while ( conjugate_peripheral_pair(theMeridian, theLongitude) == TRUE
                || conjugate_peripheral_pair(theLongitude, theMeridian) == TRUE)
            ;
    }
}

static Boolean conjugate_peripheral_pair(
    CyclicWord *word0,
    CyclicWord *word1)
{
    int value;

    /*
     * For each peripheral word, set itsLetters
     * to point to the dummy basepoint Letter.
     */
    while (word0->itsLetters->itsValue != 0)
        word0->itsLetters = word0->itsLetters->next;
    while (word1->itsLetters->itsValue != 0)
        word1->itsLetters = word1->itsLetters->next;

    if /* If... */
    (
        /* ...word0 contain more than just the dummy */
        /* basepoint letter... */
        word0->itsLength > 1
        &&
        /* ...the ends of word0 are inverses of one another... */
        word0->itsLetters->next->itsValue + word0->itsLetters->prev->itsValue == 0
        &&
        /* ...and at least one end matches the corresponding */
        /* end of word1, or word1 contains only the dummy */
        /* basepoint Letter, ... */
        (
            word0->itsLetters->next->itsValue == word1->itsLetters->next->itsValue
            ||
            word0->itsLetters->prev->itsValue == word1->itsLetters->prev->itsValue
            ||
            word1->itsLength == 1
        )
    )
    {
        /*
         * ...do the conjugation.
         */

        value = -word0->itsLetters->next->itsValue;

        conjugate_word(word0, value);
        conjugate_word(word1, value);

        return TRUE;
    }
    else
        return FALSE;
}

static void conjugate_word(

```

```

    CyclicWord *word,
    int value)
{
    Letter *new_letter;

    /*
     * This function assumes word->itsLetters is already
     * pointing to the dummy basepoint Letter.
     */
    if (word->itsLetters->itsValue != 0)
        uFatalError("conjugate_word", "fundamental_group");

    new_letter = NEW_STRUCT(Letter);
    new_letter->itsValue = value;
    INSERT_AFTER(new_letter, word->itsLetters);

    new_letter = NEW_STRUCT(Letter);
    new_letter->itsValue = -value;
    INSERT_BEFORE(new_letter, word->itsLetters);

    word->itsLength += 2;

    cancel_inverses_word(word);
}

static void cancel_inverses(
    GroupPresentation *group)
{
    /*
     * This routine cancels subwords of the form "aA".
     * It works "recursively", so that "bAaBc" gets cancelled properly.
     */

    cancel_inverses_word_list(group->itsRelations);

    /*
     * The meridians and longitudes aren't part of the pseudo-Heegaard
     * diagram, but we want to simplify them anyhow, and similarly
     * for the expressions for the original generators.
     */

    cancel_inverses_word_list(group->itsMeridians);
    cancel_inverses_word_list(group->itsLongitudes);
    cancel_inverses_word_list(group->itsOriginalGenerators);
}

static void cancel_inverses_word_list(
    CyclicWord *list)
{
    CyclicWord *word;

    for (word = list; word != NULL; word = word->next)

        cancel_inverses_word(word);
}

static void cancel_inverses_word(
    CyclicWord *word)
{
    /*
     * Attempt to verify that no cancellations are possible,
     * by checking that each letter is not followed by its inverse.
     *
     * If some letter is followed by its inverse, cancel them
     * and reset the loop counter.
     */

    int i;
    Letter *letter,
        *dead_letter;

```

```

/*
 * Look for adjacent Letters which cancel.
 * Continue cyclically until we've checked itsLength
 * consecutive Letters which don't cancel.
 *
 * Meridians and longitudes use a temporary "basepoint"
 * Letter with itsValue == 0. Don't let it cancel with itself
 * if the word becomes trivial!
 */
for ( i = 0, letter = word->itsLetters;
      i < word->itsLength;
      i++, letter = letter->next)

    if (letter->itsValue + letter->next->itsValue == 0
        && letter->itsValue != 0)
    {
        if (word->itsLength == 2)
        {
            my_free(letter->next);
            my_free(letter);
            word->itsLetters = NULL;
            word->itsLength = 0;
            break;
        }
        else
        {
            /*
             * Remove the letter following the current one.
             */
            dead_letter = letter->next;
            REMOVE_NODE(dead_letter);
            my_free(dead_letter);

            /*
             * Back up one space.
             * (We hit some letter other than the one we're cancelling,
             * because word->itsLength > 2.)
             */
            letter = letter->prev;

            /*
             * Remove what used to be the current letter.
             */
            dead_letter = letter->next;
            REMOVE_NODE(dead_letter);
            my_free(dead_letter);

            /*
             * Make sure word->itsLetters isn't left dangling.
             */
            word->itsLetters = letter;

            /*
             * The word is now two letters shorter.
             */
            word->itsLength -= 2;

            /*
             * We want to resume the for(;;) loop at i == 0.
             * If we set i to -1, the i++ in the for(;;) statement
             * will immediately increment i to 0.
             */
            i = -1;
        }
    }
}

static void handle_slide(
    GroupPresentation *group,
    int a,
    int b)
{
    /*

```



```

    * Visualize the pseudo-Heegaard diagram as in "Visualizing
    * the pseudo-Heegaard diagram" at the top of this file.
    * If some word contains the substring "ab", this means that
    * a curve runs from the disk A- to the disk B+. At this point
    * it may be helpful to draw yourself an illustration showing
    * a curve running into A+, out of A-, into B+ and out of B-.
    * For future reference it will also be helpful to draw a few
    * other random curves going in or out of A-. Slide the disk A-
    * into B+ so that it comes out at B-, and redraw your sketch to
    * show this. Note that the existence of the substring "ab"
    * guarantees that we can do this "handle slide" without crossing
    * any curves. Now look what's happened to the random curves
    * coming in and out of A-. Each curve which comes out of A- now
    * goes into B- and out of B+ before proceeding on its way.
    * Algebraically, this means each occurrence of the letter "a"
    * is replaced by "aB". Each curve going into A- now goes first
    * into B+ and out of B-. Algebraically, this means that "A"
    * is replaced by "bA". Symbolically,
    *
    *           "a" -> "aB"
    *           "A" -> "bA"
    *
    * Note that the original substring "ab" which got us started
    * is replaced by "aBb", which simplifies to "a".
    */

/*
 * We assume that the calling function has already checked
 * that "ab" occurs as a substring of some word.
 */

/*
 * a and b should be distinct generators.
 */
if (a == b || a == -b)
    uFatalError("handle_slide", "fundamental_group");

/*
 * Fix up the relations.
 */
handle_slide_word_list(group->itsRelations, a, b);

/*
 * The meridians and longitudes aren't part of the pseudo-Heegaard
 * diagram, but we want to keep track of them anyhow, and similarly
 * for the expressions for the original generators.
 */
handle_slide_word_list(group->itsMeridians, a, b);
handle_slide_word_list(group->itsLongitudes, a, b);
handle_slide_word_list(group->itsOriginalGenerators, a, b);

/*
 * Fix up the matrices.
 */
handle_slide_matrices(group, a, b);

/*
 * Cancel any pairs of inverses we may have created.
 */
cancel_inverses(group);
}

static void handle_slide_word_list(
    CyclicWord *list,
    int a,
    int b)
{
    CyclicWord *word;

    for (word = list; word != NULL; word = word->next)
        handle_slide_word(word, a, b);
}

```

```

static void handle_slide_word(
    CyclicWord *word,
    int a,
    int b)
{
    Letter *letter,
           *new_letter;

    if (word->itsLength > 0)
    {
        letter = word->itsLetters;

        do
        {
            if (letter->itsValue == a)
            {
                new_letter = NEW_STRUCT(Letter);
                new_letter->itsValue = -b;
                INSERT_AFTER(new_letter, letter);
                word->itsLength++;
            }

            if (letter->itsValue == -a)
            {
                new_letter = NEW_STRUCT(Letter);
                new_letter->itsValue = b;
                INSERT_BEFORE(new_letter, letter);
                word->itsLength++;
            }

            letter = letter->next;
        } while (letter != word->itsLetters);
    }
}

static void handle_slide_matrices(
    GroupPresentation *group,
    int a,
    int b)
{
    /*
     * Initially, generators a, b, etc. may be visualized as curves
     * in the interior of the handlebody which pass once around their
     * respective handles. Now assume we are doing a handle slide
     * as described in handle_slide() above. Generator b is not affected,
     * but the curve (in the interior of the handle body) corresponding
     * to generator a gets dragged across handle b. In otherwords, it
     * takes the trip a'B, where a' is the loop which passes once
     * around handle a in its new location, avoiding all other handles.
     * Symbolically,  $a = a'B$ . This corresponds to the matrix equation
     *  $M(a) = M(a') M(B)$ . (As explained in fg_word_to_matrix(), the
     * order of the factors is reversed for two different reasons, so
     * it doesn't get reversed at all.) Solve for  $M(a') = M(a) M(B)^{-1}$ 
     * or  $M(a') = M(a) M(b)$ .
     */

    O31Matrix temp;

    /*
     * Split into four cases, according to whether a and b
     * are positive or negative.
     */

    if (a > 0)
    {
        if (b > 0)
        {
            /*
             * Use  $M(a') = M(a) M(b)$ .
             */

```

```

        o31_product(      group->itsMatrices[a-1],
                          group->itsMatrices[b-1],
                          group->itsMatrices[a-1]);
    }
    else /* b < 0 */
    {
        /*
         * Use  $M(a') = M(a) [M(B)^{-1}]$ .
         */
        o31_invert(      group->itsMatrices[(-b)-1],
                        temp);
        o31_product(      group->itsMatrices[a-1],
                        temp,
                        group->itsMatrices[a-1]);
    }
}
else /* a < 0 */
{
    if (b > 0)
    {
        /*
         * Use  $M(A') = [M(b)^{-1}] M(A)$ 
         */
        o31_invert(      group->itsMatrices[b-1],
                        temp);
        o31_product(      temp,
                        group->itsMatrices[(-a)-1],
                        group->itsMatrices[(-a)-1]);
    }
    else /* b < 0 */
    {
        /*
         * Use  $M(A') = M(B) M(A)$ 
         */
        o31_product(      group->itsMatrices[(-b)-1],
                        group->itsMatrices[(-a)-1],
                        group->itsMatrices[(-a)-1]);
    }
}
}

static void cancel_handles(
    GroupPresentation *group,
    CyclicWord *word)
{
    /*
     * cancel_handles() cancels a relation of length one (a 2-handle)
     * with its corresponding generator (a 1-handle). This is a
     * geometric operation, in the sense that it corresponds to a
     * simplification of the pseudo-Heegaard diagram discussed in
     * "Visualizing the fundamental group" at the top of this file.
     * The proof is trivial when you visualize the pseudo-Heegaard diagram
     * as in "Visualizing the pseudo-Heegaard diagram" at the top of
     * this file (but oddly, the proof is less obvious when you visualize
     * the pseudo-Heegaard diagram as a handlebody).
     *
     * Comment: The 1-handle must be orientable, even in a nonorientable
     * manifold. Proof #1: The boundary of the thickened disk is a
     * cylinder, so the handle's "A+" and "A-" disks must be identified
     * in an orientation-preserving way. Proof #2: The handle's core
     * curve is homotopically trivial, so it must be orientation-preserving.
     */

    int dead_generator;

    /*
     * Double check that the word has length one.
     */
    if (word->itsLength != 1)
        uFatalError("cancel_handles", "fundamental_group");

    /*
     * Which generator is being cancelled?
     */

```

```

    */
    dead_generator = ABS(word->itsLetters->itsValue);

    /*
     * Remove the word from the GroupPresentation, and decrement
     * group->itsNumRelations.
     */
    remove_word(group, word);

    /*
     * Remove all occurrences of the generator from all other words.
     * Note that even if a word becomes empty, it is *not* deleted,
     * because empty words have geometrical significance in the
     * pseudo-Heegaard diagram. For example,  $S^2 \times S^1$  has a presentation
     * with one generator and one empty word.
     */
    remove_generator(group, dead_generator);

    /*
     * The highest numbered generator should assume the index of the
     * dead_generator, to keep the indexing contiguous.
     */

    renumber_generator(group, group->itsNumGenerators, dead_generator);
    o3l_copy( group->itsMatrices[dead_generator - 1],
              group->itsMatrices[group->itsNumGenerators - 1]);

    group->itsNumGenerators--;

    /*
     * Cancel any adjacent inverses which may have been created.
     */
    cancel_inverses(group);
}

static void remove_word(
    GroupPresentation *group,
    CyclicWord *word)
{
    CyclicWord **list;

    list = &group->itsRelations;

    while (*list != NULL)
    {
        if (*list == word)
        {
            *list = word->next;
            free_cyclic_word(word);
            group->itsNumRelations--;

            return;
        }

        list = &(*list)->next;
    }

    uFatalError("remove_word", "fundamental_group");
}

static void remove_generator(
    GroupPresentation *group,
    int dead_generator)
{
    remove_generator_from_list( group->itsRelations,
                                dead_generator);

    /*
     * Strictly speaking, the peripheral curves and the original
     * generator expressions are not part of the pseudo-Heegaard
     * diagram, but we want to keep them up to date.
     */
}

```

```

    remove_generator_from_list( group->itsMeridians,
                               dead_generator);
    remove_generator_from_list( group->itsLongitudes,
                               dead_generator);
    remove_generator_from_list( group->itsOriginalGenerators,
                               dead_generator);
}

static void remove_generator_from_list(
    CyclicWord  *list,
    int         dead_generator)
{
    CyclicWord  *word;

    for (word = list; word != NULL; word = word->next)
        remove_generator_from_word(word, dead_generator);
}

static void remove_generator_from_word(
    CyclicWord  *word,
    int         dead_generator)
{
    /*
     * We want to keep looking at Letters until the number of non-removed
     * Letters equals the length of the word.  A "non-removed Letter" is
     * one which we have examined and found to be something other than
     * dead_generator.
     */

    Letter      *letter;
    int         nonremoved;

    for (    letter = word->itsLetters, nonremoved = 0;
           nonremoved < word->itsLength;
           )
    {
        if (letter->itsValue == dead_generator
            || letter->itsValue == -dead_generator)
        {
            if (word->itsLength > 1)
            {
                word->itsLetters = letter->next;
                REMOVE_NODE(letter);
                my_free(letter);
                letter = word->itsLetters;
            }
            else
            {
                word->itsLetters = NULL;
                my_free(letter);
            }

            word->itsLength--;
        }
        else
        {
            nonremoved++;
            letter = letter->next;
        }
    }
}

static void renumber_generator(
    GroupPresentation *group,
    int               old_index,
    int               new_index)
{
    /*
     * Each occurrence of the old_index should be replaced

```

```

    * with the new_index.
    */

    renumber_generator_on_word_list(group->itsRelations, old_index, new_index);

    /*
    * Strictly speaking, the peripheral curves and the original
    * generator expressions are not part of the pseudo-Heegaard
    * diagram, but we want to keep them up to date.
    */

    renumber_generator_on_word_list(group->itsMeridians, old_index, new_index);
    renumber_generator_on_word_list(group->itsLongitudes, old_index, new_index);
    renumber_generator_on_word_list(group->itsOriginalGenerators, old_index, new_index);
}

static void renumber_generator_on_word_list(
    CyclicWord *list,
    int old_index,
    int new_index)
{
    CyclicWord *word;

    for (word = list; word != NULL; word = word->next)

        renumber_generator_in_word(word, old_index, new_index);
}

static void renumber_generator_in_word(
    CyclicWord *word,
    int old_index,
    int new_index)
{
    Letter *letter;
    int i;

    for (letter = word->itsLetters, i = 0;
         i < word->itsLength;
         letter = letter->next, i++)
    {
        if (letter->itsValue == old_index)
            letter->itsValue = new_index;

        if (letter->itsValue == - old_index)
            letter->itsValue = - new_index;
    }
}

int fg_get_num_generators(
    GroupPresentation *group)
{
    return group->itsNumGenerators;
}

Boolean fg_integer_fillings(
    GroupPresentation *group)
{
    return group->integer_fillings;
}

FuncResult fg_word_to_matrix(
    GroupPresentation *group,
    int *word,
    O31Matrix result_O31,
    MoebiusTransformation *result_Moebius)
{
    /*
    * In an abstract presentation of the fundamental group,
    * the word "ab" means "do 'a', then do 'b'". However,

```

```

* when we map elements of the abstract group to elements
* of the group of covering transformations, we find that
* the isometry corresponding to "ab" is obtained by first
* doing 'b', then 'a'. (Try it out in the case where
* 'a' and 'b' are generators of the fundamental group
* of an octagon-with-opposite-sides-glued, and all will
* be clear.) Matrix products are read right-to-left,
* so the isometry "'b' followed by 'a'" is M(a)M(b),
* where M(a) and M(b) are the matrices representing
* 'a' and 'b'. In summary, the order of the factors
* is reversed for two different reasons, so therefore
* it remains the same: "ab" maps to M(a)M(b).
*/

/*
* Alan Reid and Craig Hodgson have pointed out that sometimes one
* wants to look at the trace of a product of generators in SL(2,C)
* (not just PSL(2,C)). To accomodate this, fg_word_to_matrix() now
* computes the value of a word not only as an O3lMatrix, but also
* as a MoebiusTransformation, taking care to do the inversions and
* matrix multiplications in SL(2,C).
*
* A more ambitious project would be to provide a consistent
* representation into SL(2,C) whenever one is possible, but this
* has *not* been implemented in the current code.
*
* JRW 96/1/6
*/

/*
* When input word is not valid, returns func_bad_input instead
* of posting a fatal error. JRW 99/10/30
*/

MoebiusTransformation  *theMoebiusGenerators,
                        theMoebiusFactor;
O3lMatrix               theO3lFactor;

theMoebiusGenerators = NEW_ARRAY(group->itsNumGenerators, MoebiusTransformation);
O3l_array_to_Moebius_array(group->itsMatrices, theMoebiusGenerators, group->
itsNumGenerators);

o3l_copy      (result_O3l,      O3l_identity      );
Moebius_copy(result_Moebius, &Moebius_identity);

for ( ; *word != 0; word++)
{
    /*
    * Find the matrix corresponding to this letter in the word...
    */
    if (*word > 0
        && *word <= group->itsNumGenerators)
    {
        o3l_copy      ( theO3lFactor,      group->itsMatrices [*word - 1]);
        Moebius_copy(&theMoebiusFactor, &theMoebiusGenerators[*word - 1]);
    }
    else
    if (*word < 0
        && *word >= - group->itsNumGenerators)
    {
        o3l_invert      ( group->itsMatrices [-( *word) - 1], theO3lFactor      );
        Moebius_invert(&theMoebiusGenerators[-( *word) - 1], &theMoebiusFactor);
    }
    else
    {
        my_free(theMoebiusGenerators);
        return func_bad_input;
    }

    /*
    * ...and right-multiply it onto the matrix_generator.
    */
    o3l_product      (result_O3l,      theO3lFactor,      result_O3l);
    Moebius_product(result_Moebius, &theMoebiusFactor, result_Moebius);
}

```

```
    }

    my_free(theMoebiusGenerators);

    return func_OK;
}

int fg_get_num_relations(
    GroupPresentation *group)
{
    return group->itsNumRelations;
}

int fg_get_num_cusps(
    GroupPresentation *group)
{
    return group->itsNumCusps;
}

int *fg_get_relation(
    GroupPresentation *group,
    int which_relation)
{
    if (which_relation < 0 || which_relation >= group->itsNumRelations)
        uFatalError("fg_get_relation", "fundamental_group");

    return fg_get_cyclic_word(group->itsRelations, which_relation);
}

int *fg_get_meridian(
    GroupPresentation *group,
    int which_cusp)
{
    if (which_cusp < 0 || which_cusp >= group->itsNumCusps)
        uFatalError("fg_get_meridian", "fundamental_group");

    return fg_get_cyclic_word(group->itsMeridians, which_cusp);
}

int *fg_get_longitude(
    GroupPresentation *group,
    int which_cusp)
{
    if (which_cusp < 0 || which_cusp >= group->itsNumCusps)
        uFatalError("fg_get_longitude", "fundamental_group");

    return fg_get_cyclic_word(group->itsLongitudes, which_cusp);
}

int *fg_get_original_generator(
    GroupPresentation *group,
    int which_generator)
{
    if (which_generator < 0 || which_generator >= group->itsNumOriginalGenerators)
        uFatalError("fg_get_original_generator", "fundamental_group");

    return fg_get_cyclic_word(group->itsOriginalGenerators, which_generator);
}

static int *fg_get_cyclic_word(
    CyclicWord *list,
    int which_relation)
{
    int i;
    CyclicWord *word;
    Letter *letter;
    int *result;
```



```

    word = list;
    for (i = 0; i < which_relation; i++)
        if (word != NULL)
            word = word->next;
    if (word == NULL)
        uFatalError("fg_get_cyclic_word", "fundamental_group");

    result = NEW_ARRAY(word->itsLength + 1, int);
    for (i = 0, letter = word->itsLetters;
         i < word->itsLength;
         i++, letter = letter->next)
    {
        result[i] = letter->itsValue;
    }
    result[word->itsLength] = 0;

    return result;
}

void fg_free_relation(
    int *relation)
{
    my_free(relation);
}

void free_group_presentation(
    GroupPresentation *group)
{
    if (group != NULL)
    {
        if (group->itsMatrices != NULL)
            my_free(group->itsMatrices);

        free_word_list(group->itsRelations);
        free_word_list(group->itsMeridians);
        free_word_list(group->itsLongitudes);
        free_word_list(group->itsOriginalGenerators);

        my_free(group);
    }
}

static void free_word_list(
    CyclicWord *aWordList)
{
    CyclicWord *theDeadWord;

    while (aWordList != NULL)
    {
        theDeadWord = aWordList;
        aWordList = aWordList->next;
        free_cyclic_word(theDeadWord);
    }
}

static void free_cyclic_word(
    CyclicWord *aCyclicWord)
{
    Letter *theDeadLetter;

    while (aCyclicWord->itsLength > 1)
    {
        theDeadLetter = aCyclicWord->itsLetters;
        aCyclicWord->itsLetters = aCyclicWord->itsLetters->next;
        REMOVE_NODE(theDeadLetter);
        my_free(theDeadLetter);

        aCyclicWord->itsLength--;
    }
}

```

```
/*
 * The preceding code would typically work fine right down to
 * aCyclicWord->itsLength == 1, but I don't want to write code which
 * depends on the implementation of the REMOVE_NODE() macro, so I'll
 * go ahead and make a special case for aCyclicWord->itsLength == 1.
 */

if (aCyclicWord->itsLength == 1)
    my_free(aCyclicWord->itsLetters);

my_free(aCyclicWord);
}
```