

```

/*
 * symmetry_group_info.c
 *
 * The SymmetryGroup data structure is private to the kernel.
 * The UI accesses its fields via the following function calls.
 *
 * Boolean symmetry_group_is_abelian( SymmetryGroup *symmetry_group,
 *                                   AbelianGroup **abelian_description);
 *
 * Says whether the SymmetryGroup is abelian. If it is, it sets
 * abelian_description to point to the SymmetryGroup's abelian_description.
 * It points to the original, not a copy, so please don't modify it.
 *
 * Boolean symmetry_group_is_dihedral(SymmetryGroup *symmetry_group);
 *
 * Says whether the SymmetryGroup is dihedral.
 *
 * Boolean symmetry_group_is_polyhedral( SymmetryGroup *symmetry_group
 *                                     Boolean *is_binary_group,
 *                                     int *p,
 *                                     int *q,
 *                                     int *r);
 *
 * Says whether the SymmetryGroup is polyhedral. If it is, reports
 * whether it's the binary group, and reports the values for (p,q,r).
 * The pointers for is_binary_group, p, q and r may be NULL if this
 * information is not desired.
 *
 * Boolean symmetry_group_is_direct_product(SymmetryGroup *symmetry_group);
 *
 * Says whether the SymmetryGroup is a nontrivial, nonabelian
 * direct product.
 *
 * SymmetryGroup *get_symmetry_group_factor( SymmetryGroup *symmetry_group,
 *                                           int factor_number);
 *
 * If the SymmetryGroup is a nontrivial, nonabelian direct product,
 * returns a pointer to factor "factor_number" (factor_number = 0 or 1).
 * Otherwise returns NULL.
 *
 * Boolean symmetry_group_is_amphicheiral(SymmetryGroup *symmetry_group);
 *
 * Says whether the SymmetryGroup contains orientation-reversing
 * elements. Assumes the underlying manifold is oriented.
 *
 * Boolean symmetry_group_invertible_knot(SymmetryGroup *symmetry_group);
 *
 * Assumes the underlying manifold is oriented and has exactly
 * one Cusp. Returns TRUE if some Symmetry acts on the Cusp
 * via the matrix (-1, 0; 0, -1); returns FALSE otherwise.
 *
 * int symmetry_group_order(SymmetryGroup *symmetry_group);
 *
 * Returns the order of the SymmetryGroup.
 *
 * int symmetry_group_product(SymmetryGroup *symmetry_group, int i, int j);
 *
 * Returns the product of group elements i and j. We use the
 * convention that products of symmetries read right to left.
 * That is, the composition symmetry[i] o symmetry[j] acts by
 * first doing symmetry[j], then symmetry[i].
 *
 * IsometryList *get_symmetry_list(SymmetryGroup *symmetry_group);
 *
 * Returns the list of "raw" Isometries comprising a SymmetryGroup.
 *
 * SymmetryGroup *get_commutator_subgroup(SymmetryGroup *symmetry_group);
 * SymmetryGroup *get_abelianization(SymmetryGroup *symmetry_group);
 *
 * Compute the commutator subgroup [G,G] and the abelianization
 * G/[G,G]. The UI should eventually use free_symmetry_group()
 * to free them.
 *
 * SymmetryGroup *get_center(SymmetryGroup *symmetry_group);

```

```

*
*   Computes the center of G, which is the subgroup consisting of
*   elements which commute with all elements in G. The UI should
*   eventually use free_symmetry_group() to free it.
*
* SymmetryGroupPresentation *get_symmetry_group_presentation(
*                               SymmetryGroup                *symmetry_group);
* int sg_get_num_generators(   SymmetryGroupPresentation *group);
* int sg_get_num_relations(    SymmetryGroupPresentation *group);
* int sg_get_num_factors(      SymmetryGroupPresentation *group,
*                               int                        which_relation);
* void sg_get_factor(          SymmetryGroupPresentation *group,
*                               int                        which_relation,
*                               int                        which_factor,
*                               int                        *generator,
*                               int                        *power);
* void free_symmetry_group_presentation(SymmetryGroupPresentation *group);
*
*   get_symmetry_group_presentation() computes a presentation for
*   the SymmetryGroup, the various sg_ functions let you get
*   about it, and free_symmetry_group_presentation() frees it when
*   you're done. All these functions are documented in SnapPea.h,
*   so I won't repeat the details here.
*/

/*
* Symmetry Group Presentations
*
* This comment provides the theoretical underpinnings for computing
* presentations of symmetry groups. The main proposition shows how
* to compute an inefficient -- but correct! -- presentation. Various
* improvements follow. [The writing in this comment isn't very polished,
* but I wanted to get my thoughts down so if I come back to this
* in the future I'll have some record of what I was thinking. Also I
* wanted to make sure that my rough intuitions about presentations
* are rigorously correct.]
*
* The idea of a presentation for a group G is formalized as a map from
* a free group F onto G. The "generators" of the presentation are a
* set of generators for the free group F, and the "relations" are a set
* of words whose normal closure in F is the kernel of the map  $F \rightarrow G$ .
*
* The inefficient method is, roughly speaking, to make every element
* a generator, and every entry in the multiplication table a relation.
* More formally, let G be a finite group with elements  $\{0, 1, 2, \dots\}$ ,
* and let F be the free group generated by  $\{a, b, c, \dots\}$ , where the
* number of generators of F equals the number of elements of G.
* The map  $F \rightarrow G$  is the obvious one:  $a \rightarrow 0, b \rightarrow 1, \dots$ .
* (Warning: Somehow I ended up using right-to-left composition
* in the multiplication table, even though I use left-to-right
* composition in the free group. So the map  $F \rightarrow G$  is an antihomeomorphism.
* To further confuse matters, the Mac UI now displays the transposed
* multiplication table, so the user sees left-to-right composition
* even though right-to-left composition is still used internally.
* Sorry about that.) The relations correspond to the entries in G's
* multiplication table; for example, if  $1*3 = 4$  in G, then  $db(e^{-1})$ 
* is a relation in the kernel. Clearly each relation is in the kernel;
* it remains to prove that they generate the kernel. The proof is easy.
* Imagine some word  $e(k^{-1})cb(a^{-2})d$  which maps to 0. We want to show
* that we can build it up by multiplying and conjugating relations.
* First note that our set contains the relation  $aa(a^{-1}) = a = 1$ .
* So if the sample word contains a power of "a", we may conjugate it
* to put the power of "a" on the outside,  $de(k^{-1})cb(a^{-2})$ , and we've
* reduce the problem to showing that our relations generate  $de(k^{-1})cb$ 
* (if there were more factors of "a", we'd eliminate them, too).
* Now let's get rid of the inverses. Our sample word  $de(k^{-1})cb$  contains
* one negative power, namely  $k^{-1}$ . The generator k maps to the
* element 10 in G, and 10 has some inverse, say 7, in G. This means
* there's a relation  $hk(a^{-1})$  in our set, hence our relation set also
* produces  $kh = 1$ . The sample word is conjugate to  $cbde(k^{-1})$ , so if
* our set produces  $cbdeh$ , it must produce  $cbde(k^{-1})$  as well.
* If there were more negative powers we eliminate them too, until we
* arrive at the case of all positive factors. Look at the first two
* factors in  $cbdeh$ , namely  $cb$ . They map to the group elements

```

```

* 2 and 1, respectively. In G,  $1*2$  has some value, say 5, so there's
* a relation  $cb(f^{-1})$  in our set. Rewrite the above word as
*  $(cb(f^{-1}))fdeh$ . It's clear that if our set of relations produces
*  $fdeh$ , then it produces  $(cb(f^{-1}))fdeh$  and hence  $cbdeh$  as well.
* Continue in this fashion until we're down to a single element.
* By assumption the sample word is in the kernel, and "a" is the only
* generator which maps to 0, therefore that last letter must be "a",
* which we already know is produced by our set. Therefore our set
* generates the entire kernel, and our presentation really is a
* presentation for G. Q.E.D.

```

```

* Improvement #1. Reduce the number of generators.

```

```

* Any time a presentation has a relation expressing one of the
* generators in terms of the others, say  $c = abdbab$ , that generator
* may be eliminated. Here's how to do it. First eliminate the
* redundant generator (in this case "c") from all other relations
* by substituting the equivalent expression (in this case "abdbab").
* (To be completely rigorous, one takes the other relation, say
*  $bcdca$ , conjugates it to bring "c" to the outside,  $cddcab$ , and
* then multiplies to get  $(abdbab(c^{-1}))(cddcab) = abdbabddcab$ .
* One continues in this way until all occurrences of "c" have been
* eliminated from all relations except the original  $c = abdbab$ .)
* Then define a new presentation which is just like the old one
* except that (1) the generator "c" has been eliminated, and
* (2) the relation  $c = abdbab$  has been eliminated. To prove the
* old and new presentations are equivalent, define a map from
* one to the other by  $a \rightarrow a$ ,  $b \rightarrow b$ ,  $c \rightarrow abdbab$ ,  $d \rightarrow d$ , ...
* Clearly this map is onto, and clearly it maps the normal closure
* of the old relation set to the normal closure of the new one.
* So the quotients are the same. Q.E.D.

```

```

* In practice one doesn't want to explicitly create n generators,
* and then eliminate all but a handful of them. Instead, one
* explicitly creates a first generator, preferably one which maps
* to an element of high order in G. Then one eliminates all the
* generators which correspond to powers of the first one (one
* eliminates them mentally in this proof, not explicitly in the
* computer program, because the computer program never created them
* to begin with). One then explicitly creates another generator,
* and eliminates other generators (not explicitly created) which
* can be expressed in terms of it and first explicitly created
* generator. One continues in this fashion until all generators
* have either been explicitly created or eliminated.

```

```

* Improvement #2. Reduce the number of relations.

```

```

* After eliminating most of the generators, we'll find that many
* of the relations have become trivial (e.g.  $ba = ba$ ) or equivalent
* to other relations. So we should reduce the set of relations to
* the smallest set which generates the same normal closure.

```

```

#include "kernel.h"

```

```

/*
* I chose to define a SymmetryGroupPresentation data structure
* independent of fundamental_groups.c's GroupPresentation structure,
* for the following two reasons.
*
* (1) The GroupPresentation structure carries a lot of information
* about peripheral curves, representations into  $\text{Isom}(H^3)$ , etc.
* These fields obviously don't apply to symmetry groups,
* so fundamental_group.c would require a lot of modifications
* to make them optional.
*
* (2) The algorithms for simplifying group presentations are
* different for symmetry groups than for fundamental groups.
* For fundamental groups, a major goal is reducing the number
* of generators. For symmetry groups we have a reasonable
* set of generators at the beginning, and our main goal is
* reducing the vast number of relations in some efficient way.
*/

```

```

typedef struct Factor
{
    /*
     * A Factor is a generator to a power.
     * For example, a^3 would be {0,3}, b^-2 would be {1,-2} etc.
     */
    int generator,
        power;

    /*
     * The linear words assigned to group elements are stored as
     * NULL-terminated singly-linked list of Factors.
     * Relations, on the other hand, are stored as CyclicWords.
     */
    struct Factor *next;
} Factor;

typedef struct CyclicWord
{
    /*
     * itsFactors points to a circular singly-linked list of Factors.
     * The UI may decide whether to print a CyclicWord left-to-right or
     * right-to-left. Either way, the order of composition is in the
     * sense of the linked list: the symmetry corresponding to itsFactors
     * is done first, then the symmetry corresponding to itsFactors->next,
     * and so on.
     */
    Factor *itsFactors;

    /*
     * The size of a CyclicWord is the sum of the absolute values of the
     * powers in its Factors. It serves two purposes in create_relations().
     * First, it makes checking for duplicates quicker. Second, and
     * more importantly, it serves as the criterion of "simplicity" used
     * in simplifying the relations. That is, a "helper" relation is
     * inserted into a "target" relation iff the target's size decreases.
     */
    int size;

    /*
     * The (signed) sum of the powers and the number of Factors
     * are also used to speed up duplicate checking.
     */
    int sum,
        num_factors;

    /*
     * A SymmetryGroupPresentation keeps its relations on
     * a NULL-terminated, singly-linked list of CyclicWords.
     */
    struct CyclicWord *next;
} CyclicWord;

struct SymmetryGroupPresentation
{
    int itsNumGenerators,
        itsNumRelations;
    CyclicWord *itsRelations;
};

#define NOT_ASSIGNED ((Factor *) -1)

static Boolean is_inverting_matrix(MatrixInt22 a_matrix);
static Boolean *compute_commutator_subset(SymmetryGroup *symmetry_group);
static SymmetryGroup *create_subgroup(SymmetryGroup *symmetry_group, Boolean *subset);
static SymmetryGroup *create_quotient(SymmetryGroup *symmetry_group, Boolean *subset);
static Boolean *compute_center(SymmetryGroup *symmetry_group);
static void assign_generators(SymmetryGroup *symmetry_group, Factor ***elements,
    int *num_generators);
static Factor *compose_right_to_left(Factor *word1, Factor *word0);
static Factor *invert_word(Factor *word);

```

```

static void      simplify_linearly(Factor **word);
static void      free_elements_array(Factor **elements, int order);
static void      free_factor_list(Factor *factor_list);
static void      create_relations(SymmetryGroup *symmetry_group, Factor **elements,
    SymmetryGroupPresentation *group);
static Boolean   same_word(Factor *word0, Factor *word1);
static void      combine_like_factors(CyclicWord *word);
static void      normalize_powers(CyclicWord *word, int *powers);
static void      normalize_power(int *power, int modulus);
static Boolean   remove_zero_factors(CyclicWord *word);
static CyclicWord *invert_cyclic_word(CyclicWord *word);
static Boolean   cyclic_word_is_on_list(CyclicWord *word, CyclicWord *list);
static Boolean   same_cyclic_word(CyclicWord *word0, CyclicWord *word1);
static Boolean   same_based_cyclic_word(Factor *word0, Factor *word1);
static void      compute_word_info(CyclicWord *word);
static Boolean   substitute_to_simplify(CyclicWord *helper, CyclicWord *target, int *
    *powers);
static Boolean   substitute_word_to_simplify(CyclicWord *helper, CyclicWord *target,
    int *powers);
static int      cancellation_size(CyclicWord *word0, CyclicWord *word1, int *
    powers);
static void      insert_word(CyclicWord *helper, CyclicWord *target, int *powers);
static void      invert_relations_as_necessary(CyclicWord **relation_list);
static void      free_cyclic_word(CyclicWord *word);

```

```

Boolean symmetry_group_is_abelian(
    SymmetryGroup *symmetry_group,
    AbelianGroup **abelian_description)
{
    if (abelian_description != NULL)
        *abelian_description = symmetry_group->abelian_description;

    return symmetry_group->is_abelian;
}

```

```

Boolean symmetry_group_is_dihedral(
    SymmetryGroup *symmetry_group)
{
    return symmetry_group->is_dihedral;
}

```

```

Boolean symmetry_group_is_polyhedral(
    SymmetryGroup *symmetry_group,
    Boolean *is_binary_group,
    int *p,
    int *q,
    int *r)
{
    if (symmetry_group->is_polyhedral == TRUE)
    {
        if (is_binary_group != NULL)
            *is_binary_group = symmetry_group->is_binary_group;

        if (p != NULL)
            *p = symmetry_group->p;
        if (q != NULL)
            *q = symmetry_group->q;
        if (r != NULL)
            *r = symmetry_group->r;

        return TRUE;
    }
    else
    {
        if (is_binary_group != NULL)
            *is_binary_group = FALSE;

        if (p != NULL)
            *p = 0;
        if (q != NULL)
            *q = 0;
    }
}

```

```

        if (r != NULL)
            *r = 0;

        return FALSE;
    }
}

Boolean symmetry_group_is_S5(
    SymmetryGroup *symmetry_group)
{
    return symmetry_group->is_S5;
}

Boolean symmetry_group_is_direct_product(
    SymmetryGroup *symmetry_group)
{
    return symmetry_group->is_direct_product;
}

SymmetryGroup *get_symmetry_group_factor(
    SymmetryGroup *symmetry_group,
    int factor_number)
{
    if (factor_number != 0
        && factor_number != 1)
        uFatalError("get_symmetry_group_factor", "symmetry_group");

    if (symmetry_group->is_direct_product == TRUE)
        return symmetry_group->factor[factor_number];
    else
        return NULL;
}

Boolean symmetry_group_is_amphicheiral(
    SymmetryGroup *symmetry_group)
{
    /*
     * We assume the underlying manifold is oriented.
     */

    int i;

    for (i = 0; i < symmetry_group->order; i++)
        if (parity[symmetry_group->symmetry_list->isometry[i]->tet_map[0]] == 1)
            return TRUE;

    return FALSE;
}

Boolean symmetry_group_invertible_knot(
    SymmetryGroup *symmetry_group)
{
    /*
     * We assume the underlying manifold is oriented and has
     * exactly one Cusp.
     */

    int i;

    for (i = 0; i < symmetry_group->order; i++)
        if (is_inverting_matrix(symmetry_group->symmetry_list->isometry[i]->cusp_map[0]))
            return TRUE;

    return FALSE;
}

static Boolean is_inverting_matrix(
    MatrixInt22 a_matrix)
{

```

```

    int i,
        j;
    const static MatrixInt22    inverting_matrix = {{-1, 0}, {0, -1}};

    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            if (a_matrix[i][j] != inverting_matrix[i][j])
                return FALSE;

    return TRUE;
}

int symmetry_group_order(
    SymmetryGroup *symmetry_group)
{
    return symmetry_group->order;
}

int symmetry_group_product(
    SymmetryGroup *symmetry_group,
    int i,
    int j)
{
    return symmetry_group->product[i][j];
}

int symmetry_group_order_of_element(
    SymmetryGroup *symmetry_group,
    int i)
{
    return symmetry_group->order_of_element[i];
}

IsometryList *get_symmetry_list(
    SymmetryGroup *symmetry_group)
{
    return symmetry_group->symmetry_list;
}

SymmetryGroup *get_commutator_subgroup(
    SymmetryGroup *symmetry_group)
{
    Boolean *subset;
    SymmetryGroup *subgroup;

    if (symmetry_group != NULL)
    {
        subset = compute_commutator_subset(symmetry_group);
        subgroup = create_subgroup(symmetry_group, subset);
        my_free(subset);
        return subgroup;
    }
    else
        return NULL;
}

SymmetryGroup *get_abelianization(
    SymmetryGroup *symmetry_group)
{
    Boolean *subset;
    SymmetryGroup *quotient;

    if (symmetry_group != NULL)
    {
        subset = compute_commutator_subset(symmetry_group);
        quotient = create_quotient(symmetry_group, subset);
        my_free(subset);
        return quotient;
    }

```

```

    }
    else
        return NULL;
}

static Boolean *compute_commutator_subset(
    SymmetryGroup *symmetry_group)
{
    Boolean *subset,
        progress;
    int i,
        j;

    /*
     * Allocate an array of Booleans, to keep track of which
     * elements are in the commutator subgroup.
     */
    subset = NEW_ARRAY(symmetry_group->order, Boolean);

    /*
     * Initialize the subset to be empty.
     */
    for (i = 0; i < symmetry_group->order; i++)
        subset[i] = FALSE;

    /*
     * For each pair of elements i and j, add the commutator ijIj
     * to the subset.
     */
    for (i = 0; i < symmetry_group->order; i++)
        for (j = 0; j < symmetry_group->order; j++)
            subset
                [symmetry_group->product
                 [symmetry_group->product[i][j]]
                 [symmetry_group->inverse[symmetry_group->product[j][i]]]]
                = TRUE;

    /*
     * At this point the subset is closed under inverses, and
     * contains the identity, but may or may not be closed under
     * multiplication. So keep adding products of elements until
     * it is closed under multiplication.
     */
    do
    {
        progress = FALSE;

        for (i = 0; i < symmetry_group->order; i++)
            for (j = 0; j < symmetry_group->order; j++)
                if (subset[i] && subset[j])
                {
                    if ( ! subset[symmetry_group->product[i][j]])
                    {
                        subset[symmetry_group->product[i][j]] = TRUE;
                        progress = TRUE;
                    }
                }

    } while (progress == TRUE);

    /*
     * All done!
     */
    return subset;
}

static SymmetryGroup *create_subgroup(
    SymmetryGroup *symmetry_group,
    Boolean *subset)
{
    SymmetryGroup *subgroup;
    int subgroup_element,

```



```

        i,
        j;

/*
 * Allocate the SymmetryGroup data structure.
 */
subgroup = NEW_STRUCT(SymmetryGroup);

/*
 * The array subgroup_element[] will translate indices in the
 * full symmetry_group to indices in the subgroup. As we set
 * it up, we count how many elements belong to the subgroup.
 */
subgroup_element = NEW_ARRAY(symmetry_group->order, int);
subgroup->order = 0;
for (i = 0; i < symmetry_group->order; i++)
    if (subset[i])
        subgroup_element[i] = subgroup->order++;
    else
        subgroup_element[i] = -1;

/*
 * The subgroup won't have a SymmetryList,
 * so we'd better not ever pass it to a function which requires one!
 * (If desired we could write code to copy the symmetries.
 * In fact direct_product.c contains such code.)
 */
subgroup->symmetry_list = NULL;

/*
 * The subgroup's multiplication table is essentially
 * a subset of the full SymmetryGroup's multiplication table.
 */

subgroup->product = NEW_ARRAY(subgroup->order, int *);
for (i = 0; i < subgroup->order; i++)
    subgroup->product[i] = NEW_ARRAY(subgroup->order, int);

for (i = 0; i < symmetry_group->order; i++)
    for (j = 0; j < symmetry_group->order; j++)
        if (subset[i] && subset[j])
            subgroup->product[subgroup_element[i]][subgroup_element[j]] =
                subgroup_element[symmetry_group->product[i][j]];

/*
 * Copy the orders of the elements.
 */
subgroup->order_of_element = NEW_ARRAY(subgroup->order, int);
for (i = 0; i < symmetry_group->order; i++)
    if (subset[i])
        subgroup->order_of_element[subgroup_element[i]] =
            symmetry_group->order_of_element[i];

/*
 * Copy the inverses.
 */
subgroup->inverse = NEW_ARRAY(subgroup->order, int);
for (i = 0; i < symmetry_group->order; i++)
    if (subset[i])
        subgroup->inverse[subgroup_element[i]] =
            subgroup_element[symmetry_group->inverse[i]];

/*
 * Free the temporary array.
 */
my_free(subgroup_element);

/*
 * Try to find a humanly comprehensible description of the subgroup.
 */
recognize_group(subgroup);

return subgroup;
}

```

```

static SymmetryGroup *create_quotient(
    SymmetryGroup *symmetry_group,
    Boolean *subset)
{
    SymmetryGroup *quotient;
    int *coset,
        i,
        j;

    /*
     * Allocate the SymmetryGroup data structure.
     */
    quotient = NEW_STRUCT(SymmetryGroup);

    /*
     * We'll assign each element of symmetry_group to a coset
     * of the given subset. We assume the subset is a normal subgroup,
     * which is certainly the case when it's the commutator subgroup.
     */
    coset = NEW_ARRAY(symmetry_group->order, int);

    /*
     * First assign the elements of the subset to the identity coset.
     * Temporarily assign all other elements to the dummy coset -1.
     */
    for (i = 0; i < symmetry_group->order; i++)
        if (subset[i])
            coset[i] = 0;
        else
            coset[i] = -1;

    /*
     * We now go down the list of group elements, and whenever we
     * encounter an element not assigned to a coset, we create a
     * new coset, and locate all elements which belong to it.
     * We count the cosets as we go along.
     */
    quotient->order = 1;
    for (i = 0; i < symmetry_group->order; i++)
        if (coset[i] == -1)
        {
            for (j = 0; j < symmetry_group->order; j++)
                if (subset[j])
                    coset[symmetry_group->product[i][j]] = quotient->order;
            quotient->order++;
        }

    /*
     * The quotient won't have a SymmetryList,
     * so we'd better not ever pass it to a function which requires one!
     */
    quotient->symmetry_list = NULL;

    /*
     * Compute a multiplication table for the quotient.
     * (This isn't the most efficient way to do it, but
     * I don't think it really matters.)
     */

    quotient->product = NEW_ARRAY(quotient->order, int *);
    for (i = 0; i < quotient->order; i++)
        quotient->product[i] = NEW_ARRAY(quotient->order, int);

    for (i = 0; i < symmetry_group->order; i++)
        for (j = 0; j < symmetry_group->order; j++)
            quotient->product[coset[i]][coset[j]] =
                coset[symmetry_group->product[i][j]];

    /*
     * Free the temporary array.
     */
    my_free(coset);
}

```

```

/*
 * Use existing code to compute orders of elements.
 */
compute_orders_of_elements(quotient);

/*
 * Use existing code to compute inverses.
 */
compute_inverses(quotient);

/*
 * Try to find a humanly comprehensible description of the quotient.
 */
recognize_group(quotient);

return quotient;
}

SymmetryGroup *get_center(
    SymmetryGroup *symmetry_group)
{
    Boolean *subset;
    SymmetryGroup *center;

    if (symmetry_group != NULL)
    {
        subset = compute_center(symmetry_group);
        center = create_subgroup(symmetry_group, subset);
        my_free(subset);
    }
    else
        center = NULL;

    return center;
}

static Boolean *compute_center(
    SymmetryGroup *symmetry_group)
{
    Boolean *subset;
    int i,
        j;

    /*
     * Allocate an array of Booleans to keep track of which
     * elements are in the center.
     */
    subset = NEW_ARRAY(symmetry_group->order, Boolean);

    /*
     * An element is in the center iff it commutes with all group elements.
     */
    for (i = 0; i < symmetry_group->order; i++)
    {
        subset[i] = TRUE;

        for (j = 0; j < symmetry_group->order; j++)
            if (symmetry_group->product[i][j] != symmetry_group->product[j][i])
            {
                subset[i] = FALSE;
                break;
            }
    }

    /*
     * All done!
     */
    return subset;
}

```

```

SymmetryGroupPresentation *get_symmetry_group_presentation(
    SymmetryGroup    *symmetry_group)
{
    SymmetryGroupPresentation    *group;
    Factor                        **elements;

    group = NEW_STRUCT(SymmetryGroupPresentation);

    /*
     * Choose a set of generators. The "elements" array reports the
     * word assigned to each element in the group. (In terms of the
     * theoretical discussion at the top of this file, it reports
     * the name of each explicitly constructed generator, or the word
     * in the free group which has replaced each eliminated generator.)
     */
    assign_generators(symmetry_group, &elements, &group->itsNumGenerators);

    /*
     * Assemble a set of relations, eliminating the redundancies
     * as much as possible.
     */
    create_relations(symmetry_group, elements, group);

    free_elements_array(elements, symmetry_group->order);

    return group;
}

static void assign_generators(
    SymmetryGroup    *symmetry_group,
    Factor            ***elements,
    int               *num_generators)
{
    int    i,
           j,
           elements_remaining,
           max_element,
           max_power,
           product,
           power;
    Boolean progress;

    /*
     * Try to choose a fairly small set of generators.
     * Let the first generator "a" be an element of maximal order;
     * that is, let it be an element which maximizes the size of the
     * subgroup [a] which it generates. Let the second generator "b"
     * maximize the size of the subgroup [a,b], etc. I haven't implemented
     * this in a rigorous way -- all we really need is a heuristic to
     * get a fairly small set of generators.
     */

    /*
     * Initialize the number of generators to zero.
     */
    *num_generators = 0;

    /*
     * Allocate the array which will keep track of the linked list
     * of Factors assigned to each group element.
     */
    *elements = NEW_ARRAY(symmetry_group->order, Factor *);

    /*
     * Initialize *elements to indicate that no words
     * have yet been assigned.
     */
    for (i = 0; i < symmetry_group->order; i++)
        (*elements)[i] = NOT_ASSIGNED;

    /*
     * Keep track of how many elements still require words.
     */

```

```

elements_remaining = symmetry_group->order;

/*
 * Assign the identity the empty word.
 */
(*elements)[0] = NULL;
elements_remaining--;

/*
 * Add generators until all elements have been assigned one.
 */
while (elements_remaining > 0)
{
    /*
     * Choose each new generator so as to maximize the power of it
     * required to get an element which has already been assigned
     * a generator. The hope is that this will more or less
     * maximize the number of new assignments provided by this
     * generator. (It will certainly be quicker than explicitly
     * computing the number of new assignments provided by each
     * potential new generator).
     */
    max_element = -1;
    max_power = 0;
    for (i = 0; i < symmetry_group->order; i++)
    {
        product = i;
        power = 1;
        while ((*elements)[product] == NOT_ASSIGNED)
        {
            product = symmetry_group->product[i][product];
            power++;
        }
        if (power > max_power)
        {
            max_power = power;
            max_element = i;
        }
    }
    if (max_power < 2)
        uFatalError("assign_generators", "symmetry_group_info");

    /*
     * Assign a new generator to max_element.
     */
    (*elements)[max_element] = NEW_STRUCT(Factor);
    (*elements)[max_element]->generator = (*num_generators)++;
    (*elements)[max_element]->power = 1;
    (*elements)[max_element]->next = NULL;
    elements_remaining--;

    /*
     * Use the symmetry group's multiplication table to deduce the
     * assignments of words to as many other group elements as possible.
     * The symmetry group's multiplication table composes symmetries
     * right-to-left: product[i][j] is obtained by doing symmetry j,
     * followed by symmetry i.
     */
    do
    {
        progress = FALSE;

        for (i = 0; i < symmetry_group->order; i++)
            for (j = 0; j < symmetry_group->order; j++)
                if ((*elements)[i] != NOT_ASSIGNED
                    && (*elements)[j] != NOT_ASSIGNED
                    && (*elements)[symmetry_group->product[i][j]] == NOT_ASSIGNED)
                {
                    (*elements)[symmetry_group->product[i][j]] =
                        compose_right_to_left((*elements)[i], (*elements)[j]);
                    simplify_linearly(&(*elements)[symmetry_group->product[i][j]]);
                    elements_remaining--;
                    progress = TRUE;
                }
    }
}

```

```

    } while (progress == TRUE);
}

static Factor *compose_right_to_left(
    Factor *word1,
    Factor *word0)
{
    Factor *product,
        **p,
        *factor;

    product = NULL;
    p = &product;

    for (factor = word0; factor != NULL; factor = factor->next)
    {
        *p = NEW_STRUCT(Factor);
        (*p)->generator = factor->generator;
        (*p)->power = factor->power;
        (*p)->next = NULL;
        p = &(*p)->next;
    }

    for (factor = word1; factor != NULL; factor = factor->next)
    {
        *p = NEW_STRUCT(Factor);
        (*p)->generator = factor->generator;
        (*p)->power = factor->power;
        (*p)->next = NULL;
        p = &(*p)->next;
    }

    return product;
}

static Factor *invert_word(
    Factor *word)
{
    Factor *inverse,
        *factor,
        *new_factor;

    inverse = NULL;

    for (factor = word; factor != NULL; factor = factor->next)
    {
        new_factor = NEW_STRUCT(Factor);
        new_factor->generator = factor->generator;
        new_factor->power = - factor->power;
        new_factor->next = inverse;
        inverse = new_factor;
    }

    return inverse;
}

static void simplify_linearly(
    Factor **word)
{
    Boolean progress;
    Factor **factor,
        *dead_factor;

    do
    {
        progress = FALSE;

        for (factor = word; *factor != NULL; factor = &(*factor)->next)
            if ((*factor)->next != NULL && (*factor)->generator == (*factor)->next->

```

```

generator)
{
    dead_factor = (*factor)->next;
    (*factor)->power += dead_factor->power;
    (*factor)->next = dead_factor->next;
    my_free(dead_factor);
    if ((*factor)->power == 0)
    {
        dead_factor = *factor;
        *factor = (*factor)->next;
        my_free(dead_factor);
    }
    progress = TRUE;
    break;
}

} while (progress == TRUE);
}

```

```

static void free_elements_array(
    Factor **elements,
    int order)
{
    int i;

    for (i = 0; i < order; i++)
        free_factor_list(elements[i]);

    my_free(elements);
}

```

```

static void free_factor_list(
    Factor *factor_list)
{
    Factor *dead_factor;

    while (factor_list != NULL)
    {
        dead_factor = factor_list;
        factor_list = factor_list->next;
        my_free(dead_factor);
    }
}

```

```

static void create_relations(
    SymmetryGroup          *symmetry_group,
    Factor                 **elements,
    SymmetryGroupPresentation *group)
{
    int      *powers,
            i,
            j,
            k;
    Factor    *temp1,
            *temp2,
            *relation,
            *last;
    CyclicWord **end_of_relation_list,
            **first_generic_relation,
            *word,
            *inverse_word,
            *helper,
            *helper_inverse,
            **target,
            *dead_relation;
    Boolean   progress;

    /*
     * The list of relations is initially empty.
     */
    group->itsNumRelations = 0;
}

```

```

group->itsRelations      = NULL;
end_of_relation_list    = &group->itsRelations;

/*
 * Begin by creating the relations of the form a^n.
 * All other relations will be simplified modulo a^n,
 * it'll be useful to store the powers in an array.
 */
powers = NEW_ARRAY(group->itsNumGenerators, int);
for (i = 0; i < symmetry_group->order; i++)
    if (elements[i] != NULL
        && elements[i]->power == 1
        && elements[i]->next == NULL)
    {
        powers[elements[i]->generator] = symmetry_group->order_of_element[i];

        word = NEW_STRUCT(CyclicWord);
        word->itsFactors = NEW_STRUCT(Factor);
        word->itsFactors->generator = elements[i]->generator;
        word->itsFactors->power = symmetry_group->order_of_element[i];
        word->itsFactors->next = word->itsFactors;
        word->size = symmetry_group->order_of_element[i];
        word->sum = symmetry_group->order_of_element[i];
        word->num_factors = 1;
        word->next = NULL;
        *end_of_relation_list = word;
        end_of_relation_list = &word->next;
        group->itsNumRelations++;
    }

/*
 * During the simplification phase (below) we'll need to know where
 * the basic a^n relations end, and where the other relations begin.
 */
first_generic_relation = end_of_relation_list;

/*
 * Each entry in the multiplication table gives a relation
 *
 *     elements[j] elements[i] = elements[i*j]
 *
 * (Recall that pesky right-to-left composition.)
 *
 * Eliminate the obvious duplications as we go along, to minimize
 * the size of the list which must then be simplified.
 */
for (i = 0; i < symmetry_group->order; i++)
    for (j = 0; j < symmetry_group->order; j++)
    {
        /*
         * The plan is to compute the left and right hand sides
         * of the relation (cf. above). Often they will be equal,
         * and we know right away the relation is trivial.
         * If they're not equal, go ahead and create the relation.
         */

        k = symmetry_group->product[i][j];
        temp1 = compose_right_to_left(elements[i], elements[j]);
        simplify_linearly(&temp1);
        if (same_word(temp1, elements[k]))
            relation = NULL;
        else
        {
            temp2 = invert_word(elements[k]);
            relation = compose_right_to_left(temp1, temp2);
            free_factor_list(temp2);
        }
        free_factor_list(temp1);

        /*
         * If the relation is nontrivial, consider adding
         * it to itsRelations list.
         */
        if (relation != NULL)

```



```

{
    /*
     * Make the relation circular.
     */
    for (last = relation; last->next != NULL; last = last->next)
        ;
    last->next = relation;

    /*
     * Create a new CyclicWord, and install the
     * (now circular) relation.
     */
    word = NEW_STRUCT(CyclicWord);
    word->itsFactors = relation;
    word->next = NULL;

    /*
     * Simplify the CyclicWord by combining adjacent Factors
     * with the same generator. While we're at it, normalize
     * all powers to make it easier to spot duplicates.
     */
    do
    {
        combine_like_factors(word);
        normalize_powers(word, powers);
    }
    while (remove_zero_factors(word) == TRUE);

    /*
     * Compute the size of the CyclicWord, the sum of its
     * powers, and the number of factors.
     * This will speed up duplicate checking, since we know
     * two words of different sizes can't possibly be equal.
     */
    compute_word_info(word);

    /*
     * Compute the inverse.
     */
    inverse_word = invert_cyclic_word(word);
    normalize_powers(inverse_word, powers);

    /*
     * Install the new CyclicWord on itsRelations list
     * iff it's nontrivial and neither it nor its inverse
     * is already already there.
     */
    if (word->itsFactors != NULL
        && cyclic_word_is_on_list(word, group->itsRelations) == FALSE
        && cyclic_word_is_on_list(inverse_word, group->itsRelations) == FALSE)
    {
        *end_of_relation_list = word;
        end_of_relation_list = &word->next;
        group->itsNumRelations++;
    }
    else
        free_cyclic_word(word);

    free_cyclic_word(inverse_word);
}

/*
 * Simplify the generic relations by substituting other relations
 * into them so as to reduce the sum of the absolute values
 * of the powers. Simply modulo the a^n relations. Remove empty
 * relations as they occur.
 *
 * "target" is the relation being simplified.
 * "helper" is the relation which will be substituted into target,
 * if doing so reduces target's size.
 */
do
{

```

```

    progress = FALSE;

    for (helper = group->itsRelations; helper != NULL; helper = helper->next)
    {
        helper_inverse = invert_cyclic_word(helper);

        target = first_generic_relation;

        while (*target != NULL)
        {
            if (*target == helper)
            {
                target = &(*target)->next;
                continue;
            }

            if (substitute_to_simplify(helper, *target, powers) == TRUE
                || substitute_to_simplify(helper_inverse, *target, powers) == TRUE)
                progress = TRUE;

            if ((*target)->size == 0)
            {
                dead_relation = *target;
                *target = dead_relation->next;
                free_cyclic_word(dead_relation);
                group->itsNumRelations--;
            }
            else
                target = &(*target)->next;
        }

        free_cyclic_word(helper_inverse);
    }

    while (progress == TRUE);

    /*
     * If a relation has more negative than positive factors,
     * replace it with its inverse.
     */
    invert_relations_as_necessary(&group->itsRelations);

    /*
     * Free the local array of powers.
     */
    my_free(powers);
}

static Boolean same_word(
    Factor *word0,
    Factor *word1)
{
    while (TRUE)
    {
        if (word0 == NULL && word1 == NULL)
            return TRUE;

        if (word0 == NULL || word1 == NULL)
            return FALSE;

        if (word0->generator != word1->generator
            || word0->power != word1->power)
            return FALSE;

        word0 = word0->next;
        word1 = word1->next;
    }
}

static void combine_like_factors(
    CyclicWord *word)
{

```

```

    Factor    *factor,
              *dead_factor;

    /*
     * Combine adjacent factors with the same generator.
     */

    if (word->itsFactors != NULL)
    {
        factor = word->itsFactors;
        do
        {
            while (factor != factor->next
                    && factor->generator == factor->next->generator)
            {
                dead_factor = factor->next;
                if (dead_factor == word->itsFactors)
                    word->itsFactors = dead_factor->next;
                factor->power += dead_factor->power;
                factor->next = dead_factor->next;
                my_free(dead_factor);
            }

            factor = factor->next;
        } while (factor != word->itsFactors);
    }
}

static void normalize_powers(
    CyclicWord *word,
    int *powers)
{
    Factor *factor;

    if (word->itsFactors != NULL)
    {
        factor = word->itsFactors;
        do
        {
            normalize_power(&factor->power, powers[factor->generator]);

            factor = factor->next;
        } while (factor != word->itsFactors);
    }
}

static void normalize_power(
    int *power,
    int modulus)
{
    while (*power <= -((modulus + 1)/2))
        *power += modulus;

    while (*power > modulus/2)
        *power -= modulus;
}

static Boolean remove_zero_factors(
    CyclicWord *word)
{
    Factor *factor,
          *dead_factor;
    Boolean zero_factors_were_removed;

    /*
     * Eliminate Factors with power zero.
     */

    zero_factors_were_removed = FALSE;

```

```

if (word->itsFactors != NULL)
{
    factor = word->itsFactors;
    do
    {
        while (factor->next->power == 0)
        {
            /*
             * If this Factor is the only one on the ciruclar linked
             * list, eliminite it and leave the CyclicWord empty.
             */
            if (factor->next == factor)
            {
                my_free(factor);
                word->itsFactors = NULL;
                return TRUE;
            }

            /*
             * Eliminate factor->next.
             */
            dead_factor = factor->next;
            if (dead_factor == word->itsFactors)
                word->itsFactors = dead_factor->next;
            factor->next = dead_factor->next;
            my_free(dead_factor);
            zero_factors_were_removed = TRUE;
        }

        factor = factor->next;
    } while (factor != word->itsFactors);
}

return zero_factors_were_removed;
}

static CyclicWord *invert_cyclic_word(
CyclicWord *word)
{
    CyclicWord *inverse_word;
    Factor *factor,
           *inverse_factor;

    inverse_word = NEW_STRUCT(CyclicWord);
    inverse_word->itsFactors = NULL;
    inverse_word->size = word->size;
    inverse_word->sum = - word->sum;
    inverse_word->num_factors = word->num_factors;
    inverse_word->next = NULL;

    if (word->itsFactors != NULL)
    {
        factor = word->itsFactors;
        do
        {
            inverse_factor = NEW_STRUCT(Factor);
            inverse_factor->generator = factor->generator;
            inverse_factor->power = - factor->power;
            if (inverse_word->itsFactors == NULL)
            {
                inverse_word->itsFactors = inverse_factor;
                inverse_factor->next = inverse_factor;
            }
            else
            {
                inverse_factor->next = inverse_word->itsFactors->next;
                inverse_word->itsFactors->next = inverse_factor;
            }

            factor = factor->next;
        }
    }
}

```

```

        } while (factor != word->itsFactors);
    }

    return inverse_word;
}

static Boolean cyclic_word_is_on_list(
    CyclicWord *word,
    CyclicWord *list)
{
    CyclicWord *word1;

    for (word1 = list; word1 != NULL; word1 = word1->next)
        if (word->size == word1->size /* compare size, sum and */
            && word->sum == word1->sum /* num_factors first, */
            && word->num_factors == word1->num_factors /* for greater speed */
            && same_cyclic_word(word, word1))
            return TRUE;

    return FALSE;
}

static Boolean same_cyclic_word(
    CyclicWord *word0,
    CyclicWord *word1)
{
    Factor *start;

    if (word0->itsFactors == NULL && word1->itsFactors == NULL)
        return TRUE;

    if (word0->itsFactors == NULL || word1->itsFactors == NULL)
        return FALSE;

    start = word0->itsFactors;
    do
    {
        if (same_based_cyclic_word(start, word1->itsFactors))
            return TRUE;

        start = start->next;
    } while (start != word0->itsFactors);

    return FALSE;
}

static Boolean same_based_cyclic_word(
    Factor *word0,
    Factor *word1)
{
    Factor *factor0,
           *factor1;

    factor0 = word0;
    factor1 = word1;

    while (TRUE)
    {
        if (factor0->generator != factor1->generator
            || factor0->power != factor1->power)
            return FALSE;

        factor0 = factor0->next;
        factor1 = factor1->next;

        if (factor0 == word0 && factor1 == word1)
            return TRUE;

        if (factor0 == word0 || factor1 == word1)
            return FALSE;
    }
}

```

```

    }
}

static void compute_word_info(
    CyclicWord *word)
{
    Factor *factor;

    word->size      = 0;
    word->sum        = 0;
    word->num_factors = 0;

    if (word->itsFactors != NULL)
    {
        factor = word->itsFactors;
        do
        {
            word->size += ABS(factor->power);
            word->sum   += factor->power;
            word->num_factors++;

            factor = factor->next;
        } while (factor != word->itsFactors);
    }
}

static Boolean substitute_to_simplify(
    CyclicWord *helper,
    CyclicWord *target,
    int *powers)
{
    Factor *original_helper_base,
           *original_target_base;

    /*
     * There shouldn't be any empty words on the relation list,
     * but in case of error let's not crash the whole system.
     */
    if (helper->itsFactors == NULL || target->itsFactors == NULL)
        return FALSE;

    /*
     * If the target isn't more than half the size of the helper,
     * then the helper can't possibly simplify it.
     */
    if (target->size <= helper->size/2)
        return FALSE;

    /*
     * Don't worry about substituting in helper's inverse.
     * create_relations() passes us helper's inverse in a separate call.
     */

    /*
     * A priori one might want to consider substituting helper into
     * target in the middle of a factor, e.g.
     *
     *      aaabbbbcc -> aaab(BAAdB)bbbcc = adbbcc.
     *
     * But it's easy to prove that the same cancellations may be obtained
     * by substituting at "factor boundaries", e.g.
     *
     *      aaa(AAdBB)bbbcc -> adbbcc.
     */

    /*
     * Consider all possible "basepoints" for both helper and target.
     */

    original_helper_base = helper->itsFactors;
    original_target_base = target->itsFactors;

```

```

do
{
    do
    {
        if (substitute_word_to_simplify(helper, target, powers) == TRUE)
        {
            helper->itsFactors = original_helper_base;
            return TRUE;
        }

        target->itsFactors = target->itsFactors->next;

    } while (target->itsFactors != original_target_base);

    helper->itsFactors = helper->itsFactors->next;

} while (helper->itsFactors != original_helper_base);

return FALSE;
}

static Boolean substitute_word_to_simplify(
CyclicWord *helper,
CyclicWord *target,
int *powers)
{
    /*
    * The helper has factors
    *
    *          h0 h1 h2 ... hm
    *
    * and the target has factors
    *
    *          t0 t1 t2 ... tn.
    *
    * We want to replace the target with
    *
    *          h0 h1 h2 ... hm t0 t1 t2 ... tn
    *
    * iff enough cancellations will occur to decrease target's size.
    * For this to happen, the size of the cancellations must exceed
    * half the size of the helper.
    *
    * We compute the "tail end" cancellations (hm with t0,
    * h(m-1) with t1, etc.) separately from the "head end" cancellations
    * (h0 with tn, h1 with t(n-1), etc.) and add their sizes.
    * It's possible that the head end and tail end cancellations
    * will overlap, and the sum of their sizes will be greater than
    * the true cancellation, but this doesn't matter. If they overlap
    * in either of the two words (helper or target), then they are
    * sure to overlap in the small of the two words. If the smaller
    * word is the helper, then the entire helper word is cancelling
    * with a substring of the target, and we are happy. If the smaller
    * of the two words is the target, then the entire target is
    * cancelling with a substring of the helper, and
    * substitute_to_simplify() has already checked that
    * target->size > helper->size/2, so again we are happy, because
    * the replacement has shortened the target.
    * (Hmmm . . . does the fact that, say, a^3 can "cancel" with
    * a^2 to give a^-2 when a^7 == 1 affect the validity of this
    * proof? I doubt it, but . . .)
    *
    * We "double count" cancellations. That is, if a^3 cancels with
    * a^(-3), we add 6 to the cancellation count. This makes it
    * easier to count cancellations like a^3 "cancelling" with
    * a^2 to give a^-2, which counts for a saving of 3 in a group
    * in which a^7 == 1.
    */

    if (cancellation_size(helper, target, powers)
        + cancellation_size(target, helper, powers)
        > helper->size)

```

```

{
    /*
     * Given the doubt in the above proof,
     * let's check explicitly that size has truly been reduced.
     */
    int old_size;

    old_size = target->size;

    insert_word(helper, target, powers);

    if (target->size >= old_size)
        uFatalError("substitute_word_to_simplify", "symmetry_group_info");

    return TRUE;
}
else
    return FALSE;
}

static int cancellation_size(
    CyclicWord *word0,
    CyclicWord *word1,
    int *powers)
{
    Factor *h_first,
           *t_last,
           *h,
           *t,
           *old_t;
    int cancellation_size,
        sum;

    /*
     * Count the size of the potential cancellations
     * between the head of word0 and the tail of word1.
     */

    /*
     * Let h_first be the first factor at the head of word0,
     * and t_last be the last factor at the tail of word1.
     * substitute_to_simplify() has checked that neither word is empty.
     */
    h_first = word0->itsFactors;
    for ( t_last = word1->itsFactors;
          t_last->next != word1->itsFactors;
          t_last = t_last->next)
        ;

    /*
     * Count the size of the potential cancellation.
     */
    cancellation_size = 0;
    h = h_first;
    t = t_last;
    do
    {
        if (h->generator == t->generator)
        {
            sum = h->power + t->power;
            normalize_power(&sum, powers[h->generator]);

            cancellation_size += ABS(h->power);
            cancellation_size += ABS(t->power);
            cancellation_size -= ABS(sum);

            if (sum != 0)
                break;
        }
        else
            break;

        /*

```



```

    * Move h to the next Factor in helper.
    * Move t to the previous Factor in target.
    */
    h = h->next;
    old_t = t;
    while (t->next != old_t)
        t = t->next;

} while (h != h_first && t != t_last);

return cancellation_size;
}

```

```

static void insert_word(
    CyclicWord *helper,
    CyclicWord *target,
    int *powers)
{
    /*
     * The helper has factors
     *
     *          h0 h1 h2 ... hm
     *
     * and the target has factors
     *
     *          t0 t1 t2 ... tn.
     *
     * Replace the target with
     *
     *          h0 h1 h2 ... hm t0 t1 t2 ... tn
     *
     * and simplify.
     */

    Factor **last,
            *h,
            *h_copy;

    /*
     * Find tn's next field.
     */
    for (    last = &target->itsFactors->next;
          *last != target->itsFactors;
          last = &(*last)->next)
        ;

    /*
     * Insert copies of the h0...hm.
     */
    h = helper->itsFactors;
    do
    {
        h_copy = NEW_STRUCT(Factor);
        h_copy->generator = h->generator;
        h_copy->power = h->power;
        h_copy->next = target->itsFactors;
        *last = h_copy;
        last = &h_copy->next;

        h = h->next;
    } while (h != helper->itsFactors);

    /*
     * Simplify the result.
     */
    do
    {
        combine_like_factors(target);
        normalize_powers(target, powers);
    }
    while (remove_zero_factors(target) == TRUE);
}

```

```

    /*
     *   Recompute the size, sum and num_factors.
     */
    compute_word_info(target);
}

static void invert_relations_as_necessary(
    CyclicWord **relation_list)
{
    /*
     *   If a relation has more negative than positive factors,
     *   replace it with its inverse.
     */

    CyclicWord **word,
                *dead_word,
                *inverse_word;
    Factor      *factor;
    int         num_positive_factors,
                num_negative_factors;

    for (word = relation_list; *word != NULL; word = &(*word)->next)
    {
        num_positive_factors = 0;
        num_negative_factors = 0;

        if ((*word)->itsFactors != NULL)
        {
            factor = (*word)->itsFactors;
            do
            {
                if (factor->power > 0)
                    num_positive_factors++;
                else
                    num_negative_factors++;

                factor = factor->next;
            } while (factor != (*word)->itsFactors);
        }

        if (num_negative_factors > num_positive_factors)
        {
            dead_word      = *word;
            inverse_word    = invert_cyclic_word(dead_word);
            inverse_word->next = dead_word->next;
            *word           = inverse_word;
            free_cyclic_word(dead_word);
        }
    }
}

int sg_get_num_generators(
    SymmetryGroupPresentation *group)
{
    return group->itsNumGenerators;
}

int sg_get_num_relations(
    SymmetryGroupPresentation *group)
{
    return group->itsNumRelations;
}

int sg_get_num_factors(
    SymmetryGroupPresentation *group,
    int which_relation)
{
    CyclicWord *relation;
    Factor      *factor;

```

```

    int            num_factors;

    if (which_relation < 0 || which_relation >= group->itsNumRelations)
        uFatalError("sg_get_relation", "symmetry_group_info");

    relation = group->itsRelations;
    while (--which_relation >= 0)
        relation = relation->next;

    num_factors = 0;
    if (relation->itsFactors != NULL)
    {
        factor = relation->itsFactors;
        do
        {
            num_factors++;

            factor = factor->next;

        } while (factor != relation->itsFactors);
    }

    return num_factors;
}

void sg_get_factor(
    SymmetryGroupPresentation *group,
    int which_relation,
    int which_factor,
    int *generator,
    int *power)
{
    CyclicWord *relation;
    Factor *factor;

    if (which_relation < 0 || which_relation >= group->itsNumRelations)
        uFatalError("sg_get_relation", "symmetry_group_info");

    relation = group->itsRelations;
    while (--which_relation >= 0)
        relation = relation->next;
    if (relation->itsFactors == NULL)
        uFatalError("sg_get_relation", "symmetry_group_info");

    factor = relation->itsFactors;
    while (--which_factor >= 0)
        factor = factor->next;

    *generator = factor->generator;
    *power = factor->power;
}

void free_symmetry_group_presentation(
    SymmetryGroupPresentation *group)
{
    CyclicWord *dead_word;

    if (group != NULL)
    {
        while (group->itsRelations != NULL)
        {
            dead_word = group->itsRelations;
            group->itsRelations = group->itsRelations->next;
            free_cyclic_word(dead_word);
        }

        my_free(group);
    }
}

static void free_cyclic_word(

```

```
CyclicWord *word)
{
    Factor *list;

    if (word->itsFactors != NULL)
    {
        /*
         * Convert the circular factor list to a NULL-terminated
         * linear list, and let free_factor_list() dispose of it.
         */
        list = word->itsFactors->next;
        word->itsFactors->next = NULL;
        free_factor_list(list);
    }

    my_free(word);
}
```