

```

/*
 * symmetry_group_cusped.c
 *
 * This file provides the following functions:
 *
 * FuncResult  compute_cusped_symmetry_group(
 *                      Triangulation      *manifold,
 *                      SymmetryGroup      **symmetry_group_of_manifold,
 *                      SymmetryGroup      **symmetry_group_of_link);
 *
 * Computes the symmetry group of the manifold and also the symmetry
 * group of the corresponding link (defined below).
 *
 * Assumes *manifold is a cusped manifold, with all cusps complete.
 * To compute the symmetry group of a more general manifold, see symmetry_group.c.
 *
 * compute_cusped_symmetry_group() returns func_OK if its call to
 * compute_cusped_isometries() can get a canonical decomposition
 * for the manifold, and func_failed if it can't (e.g. because the manifold
 * lacks a hyperbolic structure).
 *
 * The "correspdoning link" lies in the space obtained by doing
 * meridional Dehn fillings on all cusps. The link itself consists
 * of the core curves of the filled-in solid tori and/or Klein bottles.
 * In the special case of a link complement in the 3-sphere, this
 * defintion leads to the usual notion of the symmetry group of a link.
 *
 * void recognize_group(SymmetryGroup *the_group);
 *
 * compute_cusped_symmetry_group() calls recognize_group() to attempt
 * to identify the SymmetryGroup. recognize_group() is,
 * in spirit, a local function, but it's externally visible
 * to allow communication with the recursive function
 * is_group_direct_product() in direct_product.c.
 *
 * I'd like to thank Pat Callahan for the many useful ideas
 * he contributed to SnapPea's symmetry group module.
 */

```

```

#include "kernel.h"

```

```

/*
 * The PrimaryPart data structure represents the
 * p-primary part of an abelian group G.
 */

typedef struct
{
    /*
     * What is the prime p?
     */
    int    p;

    /*
     * How many elements are in the p-primary part of G?
     */
    int    num_elements;

    /*
     * Here's an array giving the elements in the p-primary
     * part of G.
     */
    int    *element;

    /*
     * How many generators does this p-primary part have?
     */
    int    num_generators;

    /*
     * List the generators in order of decreasing order,
     * e.g. first a generator for Z/8, then Z/4, then another
     * Z/4, then Z/2.
     */

```

```

    */
    int      *generator;

} PrimaryPart;

static void      symmetry_list_to_group(SymmetryList **symmetry_list, SymmetryGroup **
    symmetry_group);
static void      put_identity_first(SymmetryGroup *the_group);
static int       find_index_of_identity(SymmetryList *the_symmetry_list);
static Boolean   is_identity(Symmetry *the_symmetry);
static void      compute_multiplication_table(SymmetryGroup *the_group);
static void      compose_symmetries(Symmetry *symmetry1, Symmetry *symmetry0, Symmetry *
    product);
static int       find_index(SymmetryList *the_symmetry_list, Symmetry *the_symmetry);
static Boolean   same_symmetry(Symmetry *symmetry0, Symmetry *symmetry1);
static Boolean   is_group_abelian(SymmetryGroup *the_group);
static Boolean   is_group_cyclic(SymmetryGroup *the_group);
static Boolean   is_group_dihedral(SymmetryGroup *the_group);
static Boolean   f_is_a_power_of_r(SymmetryGroup *the_group, int f, int r);
static void      set_cyclic_ordering(SymmetryGroup *the_group, int a_generator);
static void      attach_cyclic_description(SymmetryGroup *the_group);
static void      set_dihedral_ordering(SymmetryGroup *the_group, int f, int r);
static void      reorder_elements(SymmetryGroup *the_group, int *old_from_new);
static void      reorder_symmetries(SymmetryList *the_symmetry_list, int *old_from_new);
static void      reorder_product(SymmetryGroup *the_group, int *old_from_new, int *
    new_from_old);
static void      reorder_orders(SymmetryGroup *the_group, int *old_from_new);
static void      reorder_inverses(SymmetryGroup *the_group, int *old_from_new, int *
    new_from_old);
static void      describe_abelian_group(SymmetryGroup *the_group);
static void      find_basis(SymmetryGroup *the_group, int *num_generators, int **
    the_generators);
static Boolean   prime_power(int p, int n);
static void      primary_part_generators(SymmetryGroup *the_group, PrimaryPart *
    primary_part);
static void      find_lexicographic_ordering(SymmetryGroup *the_group, int num_generators,
    int the_generators[], int **desired_ordering);
static void      attach_abelian_description(SymmetryGroup *the_group, int num_generators,
    int the_generators[]);

FuncResult compute_cusped_symmetry_group(
    Triangulation      *manifold,
    SymmetryGroup      **symmetry_group_of_manifold,
    SymmetryGroup      **symmetry_group_of_link)
{
    SymmetryList      *symmetry_list_of_manifold,
        *symmetry_list_of_link;

    /*
     * There are two ways this function may be called:
     *
     * (1) compute_symmetry_group() may call it to compute the
     *     symmetry group of a cusped manifold.
     *
     * (2) compute_closed_symmetry_group() may call it as part
     *     of the algorithm to compute a symmetry group of a
     *     closed manifold.
     *
     * Either way, we ignore the Dehn filling coefficients,
     * and use only the complete hyperbolic structure.
     */

    /*
     * Make sure the variables used to pass back our results
     * are all initially empty.
     */
    if (*symmetry_group_of_manifold != NULL
        || *symmetry_group_of_link != NULL)
        uFatalError("compute_cusped_symmetry_group", "symmetry_group");

    /*
     * Symmetries are just Isometries from a manifold to itself.

```

```

    */
    if (compute_cusped_isometries( manifold,
                                   manifold,
                                   &symmetry_list_of_manifold,
                                   &symmetry_list_of_link) == func_failed)
    {
        *symmetry_group_of_manifold = NULL;
        *symmetry_group_of_link     = NULL;
        return func_failed;
    }

    /*
     * Convert the SymmetryLists to SymmetryGroups.
     */
    symmetry_list_to_group(&symmetry_list_of_manifold, symmetry_group_of_manifold);
    symmetry_list_to_group(&symmetry_list_of_link,     symmetry_group_of_link);

    return func_OK;
}

/*
 * symmetry_list_to_group() converts a SymmetryList to a SymmetryGroup.
 * The SymmetryGroup subsumes the SymmetryList, and adds additional
 * information, namely a multiplication table for the group, the orders
 * and inverses of the elements, and whether the group is abelian, cyclic,
 * dihedral and/or a spherical triangle group.
 *
 * IMPORTANT: Because the SymmetryGroup subsumes the SymmetryList,
 * symmetry_list_to_group() sets the pointer symmetry_list to NULL.
 * In particular, you don't need to free the SymmetryList, just the
 * SymmetryGroup.
 */

static void symmetry_list_to_group(
    SymmetryList    **symmetry_list,
    SymmetryGroup   **symmetry_group)
{
    SymmetryGroup   *the_group;

    /*
     * Allocate the SymmetryGroup.
     */
    (*symmetry_group) = NEW_STRUCT(SymmetryGroup);

    /*
     * Give it a local name to avoid the double indirection.
     */
    the_group = *symmetry_group;

    /*
     * Copy the pointer to the SymmetryList.
     */
    the_group->symmetry_list = *symmetry_list;

    /*
     * Clear the original pointer, since the_group has now
     * taken responsibility for the SymmetryList.
     */
    *symmetry_list = NULL;

    /*
     * Set the order of the group.
     */
    the_group->order = the_group->symmetry_list->num_isometries;

    /*
     * Make sure the group isn't empty.
     */
    if (the_group->order == 0)
        uFatalError("symmetry_list_to_group", "symmetry_group");

    /*
     * Make sure the identity is element 0.
     */

```

```

    */
    put_identity_first(the_group);

    /*
     * Compute the multiplication table for the group.
     */
    compute_multiplication_table(the_group);

    /*
     * Use the multiplication table to compute the order of each element.
     */
    compute_orders_of_elements(the_group);

    /*
     * Use the multiplication table to compute the inverse of each element.
     */
    compute_inverses(the_group);

    /*
     * Attempt to recognize the_group.
     */
    recognize_group(the_group);
}

void recognize_group(
    SymmetryGroup *the_group)
{
    /*
     * We assume the_group's order, symmetry_list, product[[[]],
     * order_of_element[] and inverse[] fields have been set.
     */
    /*
     * 96/11/30 It's OK to have symmetry_list == NULL.
     */

    /*
     * Set the abelian_description to NULL.
     * This will be overridden if the group is abelian.
     */
    the_group->abelian_description = NULL;

    /*
     * Check whether the group is abelian, cyclic, dihedral,
     * a spherical triangle group, or [eventually] a symmetric
     * or alternating group.
     */
    /*
     * If the group is cyclic or dihedral, the elements will be
     * reordered in a natural way.
     */
    the_group->is_abelian      = is_group_abelian(the_group);
    the_group->is_cyclic       = is_group_cyclic(the_group);
    the_group->is_dihedral     = is_group_dihedral(the_group);
    the_group->is_polyhedral   = is_group_polyhedral(the_group);
    the_group->is_S5           = is_group_S5(the_group);

    /*
     * If the group is abelian but not cyclic, we want to
     *
     * (1) figure out which group it is, and
     *
     * (2) order the elements in a natural way.
     *
     * (If the group is cyclic, the function is_group_cyclic() will have
     * already attached an abelian_description and reordered the elements.)
     */
    if (the_group->is_abelian == TRUE
        && the_group->is_cyclic == FALSE)
        describe_abelian_group(the_group);

    /*
     * If the_group hasn't yet been identified, check whether it's a
     * nontrivial direct product. is_group_direct_product() will set
     * the is_direct_product and factor[] field correctly, whether or
     * not the_group is a nontrivial direct product.
     */

```

```

    */
    if (the_group->is_cyclic      == FALSE
        && the_group->is_dihedral  == FALSE
        && the_group->is_polyhedral == FALSE
        && the_group->is_S5       == FALSE
        && the_group->is_abelian   == FALSE)

        the_group->is_direct_product = is_group_direct_product(the_group);

    else
    {
        the_group->is_direct_product = FALSE;
        the_group->factor[0] = NULL;
        the_group->factor[1] = NULL;
    }
}

static void put_identity_first(
    SymmetryGroup *the_group)
{
    int      index_of_identity;
    Symmetry **the_symmetries,
             *temp;

    index_of_identity = find_index_of_identity(the_group->symmetry_list);

    if (index_of_identity != 0)
    {
        the_symmetries = the_group->symmetry_list->isometry;

        temp
        the_symmetries[0]
        the_symmetries[index_of_identity]
            = the_symmetries[0];
            = the_symmetries[index_of_identity];
            = temp;
    }
}

static int find_index_of_identity(
    SymmetryList *the_symmetry_list)
{
    int i;

    for (i = 0; i < the_symmetry_list->num_isometries; i++)

        if (is_identity(the_symmetry_list->isometry[i]))

            return i;

    /*
     * The identity was not found on the list. Uh oh.
     */
    uFatalError("find_index_of_identity", "symmetry_group");

    /*
     * The C++ compiler would like a return value, even though
     * we never return from the uFatalError() call.
     */
    return 0;
}

static Boolean is_identity(
    Symmetry *the_symmetry)
{
    int i;

    for (i = 0; i < the_symmetry->num_tetrahedra; i++)

        if (the_symmetry->tet_image[i] != i
            || the_symmetry->tet_map[i] != IDENTITY_PERMUTATION)

            return FALSE;
}

```

```

    return TRUE;
}

static void compute_multiplication_table(
    SymmetryGroup *the_group)
{
    int i,
        j;
    Symmetry *the_product;
    int num_tetrahedra;

    /*
     * Allocate space for the multiplication table.
     */
    the_group->product = NEW_ARRAY(the_group->order, int *);
    for (i = 0; i < the_group->order; i++)
        the_group->product[i] = NEW_ARRAY(the_group->order, int);

    /*
     * Note how many Tetrahedra underlie each Symmetry.
     */
    num_tetrahedra = the_group->symmetry_list->isometry[0]->num_tetrahedra;

    /*
     * Allocate space to temporarily hold the product of two
     * symmetries.
     */
    the_product = NEW_STRUCT(Symmetry);
    the_product->tet_image = NEW_ARRAY(num_tetrahedra, int);
    the_product->tet_map = NEW_ARRAY(num_tetrahedra, Permutation);

    /*
     * For each pair of elements . . .
     */
    for (i = 0; i < the_group->order; i++)
        for (j = 0; j < the_group->order; j++)
        {
            /*
             * . . . compute their product . . .
             */
            compose_symmetries( the_group->symmetry_list->isometry[i],
                               the_group->symmetry_list->isometry[j],
                               the_product);

            /*
             * . . . and write its index into the multiplication table.
             */
            the_group->product[i][j] = find_index(the_group->symmetry_list, the_product);
        }

    /*
     * Free the temporary storage.
     */
    my_free(the_product->tet_image);
    my_free(the_product->tet_map);
    my_free(the_product);
}

static void compose_symmetries(
    Symmetry *symmetry1,
    Symmetry *symmetry0,
    Symmetry *product)
{
    int i;

    product->num_tetrahedra = symmetry0->num_tetrahedra;

    for (i = 0; i < product->num_tetrahedra; i++)
    {
        product->tet_image[i] = symmetry1->tet_image[symmetry0->tet_image[i]];

        product->tet_map[i] = compose_permutations(

```

```

        symmetry1->tet_map[symmetry0->tet_image[i]],
        symmetry0->tet_map[i]);
    }
}

/*
 * find_index() finds the position of the_symmetry on the_symmetry_list.
 * If the_symmetry does not occur on the_symmetry_list, it calls
 * uFatalError() to exit.
 */

static int find_index(
    SymmetryList    *the_symmetry_list,
    Symmetry        *the_symmetry)
{
    int i;

    for (i = 0; i < the_symmetry_list->num_isometries; i++)
        if (same_symmetry(the_symmetry, the_symmetry_list->isometry[i]))
            return i;

    /*
     * the_symmetry was not found on the_symmetry_list.
     */
    uFatalError("find_index", "symmetry_group");

    /*
     * The C++ compiler would like a return value, even though
     * we never return from the uFatalError() call.
     */
    return 0;
}

/*
 * same_symmetry() returns TRUE if symmetry0 and symmetry1 are the same,
 * FALSE otherwise.
 */

static Boolean same_symmetry(
    Symmetry    *symmetry0,
    Symmetry    *symmetry1)
{
    int i;

    for (i = 0; i < symmetry0->num_tetrahedra; i++)
        if (symmetry0->tet_image[i] != symmetry1->tet_image[i]
            || symmetry0->tet_map[i]  != symmetry1->tet_map[i])
            return FALSE;

    return TRUE;
}

void compute_orders_of_elements(
    SymmetryGroup    *the_group)
{
    int i;
    int running_product;

    /*
     * Allocate the array which will hold the orders of the elements.
     */
    the_group->order_of_element = NEW_ARRAY(the_group->order, int);

    /*
     * We'll use the fact that put_identity_first() has put the
     * identity in position 0.
     */

```

```

/*
 * Compute the order of each element.
 */
for (i = 0; i < the_group->order; i++)
{
    the_group->order_of_element[i] = 0;
    running_product = 0;
    do
    {
        running_product = the_group->product[i][running_product];
        the_group->order_of_element[i]++;
    }
    while (running_product != 0);
}

}

void compute_inverses(
SymmetryGroup *the_group)
{
    int i,
        j;

    /*
     * Allocate the array which will hold the inverses of the elements.
     */
    the_group->inverse = NEW_ARRAY(the_group->order, int);

    /*
     * Compute the inverse of each element.
     */
    for (i = 0; i < the_group->order; i++)
    {
        for (j = 0; j < the_group->order; j++)
            if (the_group->product[i][j] == 0)
            {
                the_group->inverse[i] = j;
                break;
            }
        if (j == the_group->order) /* no inverse was found */
            uFatalError("compute_inverses", "symmetry_group");
    }

    /*
     * Just for good measure, let's make sure the inverses are consistent.
     */
    for (i = 0; i < the_group->order; i++)
        if (the_group->inverse[the_group->inverse[i]] != i)
            uFatalError("compute_inverses", "symmetry_group");
}

static Boolean is_group_abelian(
SymmetryGroup *the_group)
{
    int i,
        j;

    for (i = 0; i < the_group->order; i++)
        for (j = i + 1; j < the_group->order; j++)
            if (the_group->product[i][j] != the_group->product[j][i])
                return FALSE;

    return TRUE;
}

static Boolean is_group_cyclic(

```



```

SymmetryGroup    *the_group)
{
    int i;

    for (i = 0; i < the_group->order; i++)

        if (the_group->order_of_element[i] == the_group->order)
        {
            set_cyclic_ordering(the_group, i);

            attach_cyclic_description(the_group);

            return TRUE;
        }

    return FALSE;
}

static Boolean is_group_dihedral(
    SymmetryGroup    *the_group)
{
    /*
     * Definition.  The dihedral group of order 2n, denoted Dn,
     * is the group of symmetries of a regular n-gon.
     *
     * Proposition 1.  The dihedral group Dn has the presentation
     *
     *      { F, R | F^2 = 1, R^n = 1, RF = FR^-1 }
     *
     * Notes:
     *
     * (1) Intuitively, F is a flip about some diagonal,
     *      and R is a 2pi/n rotation.  The relation F^2 = 1
     *      says that two consecutive flips take you back to
     *      where you started.  The relation R^n = 1 says that
     *      n rotations of 2pi/n take you back to where you
     *      started.  The relation RF = FR^-1 says that a
     *      flip followed by a counterclockwise rotation
     *      equals a clockwise rotation followed by a flip.
     *
     * (2) RF = FR^-1 iff F(RF)F = F(FR^-1)F iff FR = R^-1F,
     *      so it doesn't matter which 2pi/n rotation we call
     *      R, and which we call R^-1.
     *
     * Proof of Proposition 1.  Consider the map from the free
     * group on {F, R} to Dn defined by sending F to a flip
     * about some arbitrary but fixed diagonal, and R to a
     * 2pi/n rotation.  (If n is odd the diagonal will run from
     * the midpoint of an edge to the opposite vertex.  If n is
     * even it may run from vertex to vertex or edge to edge.)
     * Clearly this map is onto; we must show that the words
     * F^2, R^n, and RFRF generate the kernel.  It's trivial to
     * check that the three words lie in the kernel; we need
     * to prove that they generate it.  Imagine some arbitrary
     * word FRRFRFR^-1FRR...R^-3F in the kernel.  Use the relation
     * RF = FR^-1 to push all the F's to the left and all the R's
     * to the right, so the word ends up in the form (F^a)(R^b).
     * Use the other two relations to insure that 0 <= a < 2 and
     * 0 <= b < n.  The only way this can map to the trivial symmetry
     * of an n-gon is to have a = b = 0, which proves that the
     * relations F^2 = 1, R^n = 1, and RF = FR^-1 generate the
     * kernel.  Q.E.D.
     *
     * Notation: Let G denote the_group.  That is, what's called
     * the_group in the is_group_dihedral() function definition
     * will be called G in this documentation.
     *
     * We want an algorithm to check whether G is a dihedral group.
     *
     * Proposition 2.  A group G of order 2n is isomorphic to the
     * dihedral group Dn iff G contains elements F and R such that

```

```

*
* (1) F has order 2,
*
* (2) R has order n,
*
* (3)  $RF = FR^{-1}$ , and
*
* (4) F is not a power of R.
*
* Proof.
* (  $\Rightarrow$  ) Trivial.
* (  $\Leftarrow$  ) Assume G contains elements F and R satisfying the
* above conditions. We must construct an isomorphism from
*  $D_n = \{ f, r \mid f^2 = 1, r^n = 1, rf = fr^{-1} \}$  to G.
* Let phi be the map from  $D_n$  to G which sends f to F and r to R.
* By conditions (1)-(3), phi is a well defined homomorphism.
* Because  $D_n$  and G have the same order, phi will be bijective
* iff it is surjective. To see that it is surjective, note
* that the subgroup of G generated by R divides G into two
* cosets, and condition (4) implies that F does not lie in the
* coset containing the identity. It follows that both cosets
* are contained in the image of phi, hence phi is surjective.
* The above reasoning now implies that phi is an isomorphism.
* Q.E.D.
*/

/*
* Our algorithm is to check the conditions of Proposition 2.
*/

int n,
    f,
    r;

/*
* Does the_group have even order?
* If not, it can't possible be dihedral.
*/

if (the_group->order % 2 == 1)
    return FALSE;

/*
* Let the order of the group be 2n.
*/

n = the_group->order / 2;

/*
* Consider all candidates for r.
*/

for (r = 0; r < the_group->order; r++)
{
    /*
    * Proceed iff r has order n.
    */

    if (the_group->order_of_element[r] == n)
    {
        /*
        * Consider all candidates for f.
        */

        for (f = 0; f < the_group->order; f++)
        {
            /*
            * Proceed iff f has order 2.
            */

            if (the_group->order_of_element[f] == 2)
            {
                /*
                * If

```

```

        *      rf == fr^-1
        *      and
        *      f is not a power of r
        *      then
        *      we've satisfied the conditions
        *      of Proposition 2.
        */

    if (
        the_group->product[r][f]
        == the_group->product[f][the_group->inverse[r]]
        &&
        f_is_a_power_of_r(the_group, f, r) == FALSE
    )
    {
        set_dihedral_ordering(the_group, f, r);
        return TRUE;
    }

    /*
    *   There are no elements r and f satisfying the conditions
    *   of Proposition 2. Therefore the_group is not dihedral.
    */

    return FALSE;
}

static Boolean f_is_a_power_of_r(
    SymmetryGroup *the_group,
    int f,
    int r)
{
    int n,
        exponent,
        running_product;

    /*
    *   Let the order of the group be 2n.
    */
    n = the_group->order / 2;

    /*
    *   We use the fact that the identity Symmetry is element 0.
    */

    for (    exponent = 0, running_product = 0;
           exponent < n;
           exponent++, running_product = the_group->product[running_product][r])

        if (running_product == f)
            return TRUE;

    return FALSE;
}

/*
*   set_cyclic_ordering() reorders the elements of the_group
*   as consecutive powers of a_generator.
*/

static void set_cyclic_ordering(
    SymmetryGroup *the_group,
    int a_generator)
{
    int *desired_ordering,
        i,
        running_product;

    /*
    *   Allocate space for an array which will temporarily
    *   hold the desired ordering.
    */

```

```

    */
    desired_ordering = NEW_ARRAY(the_group->order, int);

    /*
     * Compute a running_product, which will equal the
     * zeroth, first, second, etc. power of a_generator.
     * Note that the identity symmetry is number 0.
     */
    for ( i = 0, running_product = 0;
          i < the_group->order;
          i++, running_product = the_group->product[running_product][a_generator])

        /*
         * Set desired_ordering[i] equal to the element which
         * is the i-th power of the generator.
         */
        desired_ordering[i] = running_product;

    /*
     * Reorder the elements of the_group according to the desired_ordering.
     */
    reorder_elements(the_group, desired_ordering);

    /*
     * Free the temporary storage.
     */
    my_free(desired_ordering);
}

static void attach_cyclic_description(
    SymmetryGroup *the_group)
{
    /*
     * The function which called attach_cyclic_description() just
     * discovered that the_group is cyclic. We must attach an
     * AbelianGroup structure to record this fact.
     */

    the_group->abelian_description = NEW_STRUCT(AbelianGroup);

    /*
     * Handle the trivial group separately.
     * JRW 2000/1/14
     */
    if (the_group->order > 1)
    {
        the_group->abelian_description->num_torsion_coefficients = 1;
        the_group->abelian_description->torsion_coefficients = NEW_ARRAY(1, long int);
        the_group->abelian_description->torsion_coefficients[0] = the_group->order;
    }
    else
    {
        the_group->abelian_description->num_torsion_coefficients = 0;
        the_group->abelian_description->torsion_coefficients = NULL;
    }
}

/*
 * set_dihedral_ordering() reorders the elements of the_group
 * as I, R, R^2, . . . , R^(n-1), F, FR, FR^2, . . . , FR^(n-1).
 */

static void set_dihedral_ordering(
    SymmetryGroup *the_group,
    int f,
    int r)
{
    int *desired_ordering,
        n,
        i,
        running_product;

```

```

/*
 * Let the order of the group be 2n.
 */
n = the_group->order / 2;

/*
 * Allocate space for an array which will temporarily
 * hold the desired ordering.
 */
desired_ordering = NEW_ARRAY(the_group->order, int);

/*
 * Compute successive powers of r.
 */
for ( i = 0, running_product = 0;
      i < n;
      i++, running_product = the_group->product[running_product][r])
{
    desired_ordering[i] = running_product; /* R^i */
    desired_ordering[i + n] = the_group->product[f][running_product]; /* FR^i */
}

/*
 * Reorder the elements of the_group according to the desired_ordering.
 */
reorder_elements(the_group, desired_ordering);

/*
 * Free the temporary storage.
 */
my_free(desired_ordering);
}

/*
 * reorder_elements() reorders the elements of the_group
 * according the prescribed ordering.
 */

static void reorder_elements(
    SymmetryGroup *the_group,
    int *old_from_new)
{
    int *new_from_old,
        i;

    /*
     * The array old_from_new[] (which the functions set_cyclic_ordering()
     * and set_dihedral_ordering() refer to as desired_ordering[])
     * gives a group element's old index in terms of its new (desired)
     * index. The array new_from_old[] does the opposite.
     */
    new_from_old = NEW_ARRAY(the_group->order, int);
    for (i = 0; i < the_group->order; i++)
        new_from_old[old_from_new[i]] = i;

    reorder_symmetries (the_group->symmetry_list, old_from_new);
    reorder_product (the_group, old_from_new, new_from_old);
    reorder_orders (the_group, old_from_new);
    reorder_inverses (the_group, old_from_new, new_from_old);

    my_free(new_from_old);
}

static void reorder_symmetries(
    SymmetryList *the_symmetry_list,
    int *old_from_new)
{
    Symmetry **old_symmetry_list;
    int i;

    /*
     * 96/11/30 If the group doesn't have a SymmetryList, do nothing.

```

```

    */
    if (the_symmetry_list == NULL)
        return;

    /*
     * Allocate space for a copy of the array of pointers to
     * the Symmetries . . .
     */
    old_symmetry_list = NEW_ARRAY(the_symmetry_list->num_isometries, Symmetry *);

    /*
     * . . . and copy in the pointers to the Symmetries
     * in their original order.
     */
    for (i = 0; i < the_symmetry_list->num_isometries; i++)
        old_symmetry_list[i] = the_symmetry_list->isometry[i];

    /*
     * Rewrite the_symmetry_list->isometry[] relative to the new
     * indexing system.
     */
    for (i = 0; i < the_symmetry_list->num_isometries; i++)
        the_symmetry_list->isometry[i] = old_symmetry_list[old_from_new[i]];

    /*
     * Free the old_symmetry_list[].
     */
    my_free(old_symmetry_list);
}

```

```

static void reorder_product(
    SymmetryGroup *the_group,
    int *old_from_new,
    int *new_from_old)
{
    int **old_product,
        i,
        j;

    /*
     * Allocate space for a copy of the group multiplication table . . .
     */
    old_product = NEW_ARRAY(the_group->order, int *);
    for (i = 0; i < the_group->order; i++)
        old_product[i] = NEW_ARRAY(the_group->order, int);

    /*
     * . . . and fill it in relative to the old numbering system.
     */
    for (i = 0; i < the_group->order; i++)
        for (j = 0; j < the_group->order; j++)
            old_product[i][j] = the_group->product[i][j];

    /*
     * Rewrite the_group->product[][] relative to the new numbering system.
     */
    for (i = 0; i < the_group->order; i++)
        for (j = 0; j < the_group->order; j++)
            the_group->product[i][j] = new_from_old[old_product[old_from_new[i]]
[old_from_new[j]]];

    /*
     * Free the copy of the multiplication table.
     */
    for (i = 0; i < the_group->order; i++)
        my_free(old_product[i]);
    my_free(old_product);
}

```

```

static void reorder_orders(
    SymmetryGroup *the_group,
    int *old_from_new)

```

```

{
    int *order_of_old_element,
        i;

    /*
     * Allocate the array order_of_old_element[] . . .
     */
    order_of_old_element = NEW_ARRAY(the_group->order, int);

    /*
     * . . . and copy in the orders of the elements indexed relative
     * to the old numbering system.
     */
    for (i = 0; i < the_group->order; i++)
        order_of_old_element[i] = the_group->order_of_element[i];

    /*
     * Rewrite the_group->order_of_element[] indexed relative to
     * the new numbering system.
     */
    for (i = 0; i < the_group->order; i++)
        the_group->order_of_element[i] = order_of_old_element[old_from_new[i]];

    /*
     * Free the order_of_old_element[] array.
     */
    my_free(order_of_old_element);
}

static void reorder_inverses(
    SymmetryGroup *the_group,
    int *old_from_new,
    int *new_from_old)
{
    int *old_inverse,
        i;

    /*
     * Allocate the array old_inverse . . .
     */
    old_inverse = NEW_ARRAY(the_group->order, int);

    /*
     * . . . and copy in the inverses as expressed
     * relative to the old numbering system.
     */
    for (i = 0; i < the_group->order; i++)
        old_inverse[i] = the_group->inverse[i];

    /*
     * Rewrite the array the_group->inverse[]
     * relative to the new numbering system.
     */
    for (i = 0; i < the_group->order; i++)
        the_group->inverse[i] = new_from_old[old_inverse[old_from_new[i]]];

    /*
     * Free the old_inverse[] array.
     */
    my_free(old_inverse);
}

static void describe_abelian_group(
    SymmetryGroup *the_group)
{
    int num_generators,
        *the_generators,
        *desired_ordering;

    /*
     * Throughout this documentation, we refer to the_group as G.
     */
}

```

```

    * By a standard theorem in finite group theory, G has the form
    *
    *       $Z/n_0 + Z/n_1 + \dots + Z/n_k$    where  $n_0 \mid n_1 \mid \dots \mid n_k$ .
    *
    * We first call a function which finds generators
    *
    *      a0 = (1, 0, 0, ..., 0, 0)
    *      a1 = (0, 1, 0, ..., 0, 0)
    *      ...
    *      ak = (0, 0, 0, ..., 0, 1)
    *
    * The generators will be reported in descending order
    * ak, ..., a0.
    */

find_basis(the_group, &num_generators, &the_generators);

/*
 * Attach an abelian_description.
 */

attach_abelian_description(the_group, num_generators, the_generators);

/*
 * We now call a function which computes a lexicographic
 * ordering for the elements. If, for example,  $n_0 = 2$ ,
 *  $n_1 = 2$  and  $n_2 = 6$ , the lexicographic ordering will be
 *
 *      (0, 0, 0)
 *      (0, 0, 1)
 *      (0, 0, 2)
 *      (0, 0, 3)
 *      (0, 0, 4)
 *      (0, 0, 5)
 *      (0, 1, 0)
 *      (0, 1, 1)
 *      (0, 1, 2)
 *      (0, 1, 3)
 *      (0, 1, 4)
 *      (0, 1, 5)
 *      (1, 0, 0)
 *      (1, 0, 1)
 *      ...
 *      (1, 1, 5)
 */

find_lexicographic_ordering(the_group, num_generators, the_generators, &
desired_ordering);

/*
 * Reorder the group elements according to the desired_ordering.
 */

reorder_elements(the_group, desired_ordering);

/*
 * Free the local arrays.
 */

my_free(the_generators);
my_free(desired_ordering);
}

static void find_basis(
SymmetryGroup *the_group,
int num_generators,
int **the_generators)
{
    int num_primary_parts;
    PrimaryPart *primary_part;
    int p,
        n;
    int i,

```



```

        j;

/*
 * Think of the_group, which we'll call G, in its most factored form.
 * For example, G might be
 *
 *       $Z/2 + Z/2 + Z/4 + Z/8 + Z/3 + Z/9 + Z/25$ 
 *
 * Definition.
 *  $Z/2 + Z/2 + Z/4 + Z/8$  is called the 2-primary part of G,
 *       $Z/3 + Z/9$  is called the 3-primary part of G, etc.
 *
 * Our plan is to find generators for each primary part separately,
 * and then combine them.
 */

/*
 * We don't know how many primary parts there'll be, but for
 * sure it'll be fewer than the number of element in the group.
 * So rather than fussing around, we'll allocate an array which
 * is much too big.
 */

primary_part = NEW_ARRAY(the_group->order, PrimaryPart);

/*
 * Find the prime divisors of |G|.
 */

num_primary_parts = 0;

for ( p = 2, n = the_group->order;
      n > 1;
      p++)

    if (n%p == 0)
    {
        /*
         * Allocate a primary part.
         */
        primary_part[num_primary_parts++].p = p;

        /*
         * Divide all powers of p out of n.
         */

        while (n%p == 0)
            n /= p;
    }

/*
 * Find each primary part.
 */

for (i = 0; i < num_primary_parts; i++)
{
    /*
     * Get set up.
     * We don't know how many elements we'll have, but
     * for sure not more than the order of the group.
     */

    primary_part[i].num_elements = 0;
    primary_part[i].element      = NEW_ARRAY(the_group->order, int);
    primary_part[i].generator    = NEW_ARRAY(the_group->order, int);

    /*
     * Find which group elements have order a power of p.
     */

    for (j = 0; j < the_group->order; j++)

        if (prime_power(primary_part[i].p, the_group->order_of_element[j]))

```

```

        primary_part[i].element[primary_part[i].num_elements++] = j;

    /*
     * Find the generators for this primary part.
     */

    primary_part_generators(the_group, &primary_part[i]);
}

/*
 * Combine the generators for the primary parts into
 * generators for the complete group. For example, if
 * the primary parts are
 *
 *      Z/8 + Z/4 + Z/2
 * and
 *      Z/3 + Z/3
 *
 * then the first generator will be the product (sum) of
 * the generators for the Z/8 and the first Z/3, the second
 * generator will be the product of the generators of the Z/4
 * and the second Z/3, and the third generator will be the
 * generator of the Z/2.
 */

/*
 * The number of generators for the group will be the maximum
 * number of generators for any primary part.
 */

*num_generators = 0;
for (i = 0; i < num_primary_parts; i++)
    if (primary_part[i].num_generators > *num_generators)
        *num_generators = primary_part[i].num_generators;

/*
 * Allocate memory for the generators for the group.
 */

*the_generators = NEW_ARRAY(*num_generators, int);

/*
 * The i-th generator will be the product (sum) of all
 * i-th generators of the primary parts.
 */

for (i = 0; i < *num_generators; i++)
{
    /*
     * Initialize the i-th generator to the identity.
     */

    (*the_generators)[i] = 0;

    /*
     * Add in the i-th generator of each primary part which
     * has an i-th generator.
     */

    for (j = 0; j < num_primary_parts; j++)
        if (i < primary_part[j].num_generators)
            (*the_generators)[i] = the_group->product
                [(*the_generators)[i]]
                [primary_part[j].generator[i]];
}

/*
 * Free local memory.
 */

```

```

    for (i = 0; i < num_primary_parts; i++)
    {
        my_free(primary_part[i].element);
        my_free(primary_part[i].generator);
    }
    my_free(primary_part);
}

/*
 * prime_power() returns TRUE if n is a power of p,
 * FALSE otherwise.
 */

static Boolean prime_power(
    int p,
    int n)
{
    while (TRUE)
    {
        if (n == 1)
            return TRUE;

        if (n%p)
            return FALSE;

        n /= p;
    }
}

/*
 * primary_part_generators() takes a PrimaryPart whose fields p,
 * num_elements, and element[] have been filled in (and whose generator[]
 * array has been allocated) and computes a set of standard generators.
 */

static void primary_part_generators(
    SymmetryGroup *the_group,
    PrimaryPart *primary_part)
{
    /*
     * The algorithm is more clearly explained in terms of an example,
     * without the burden of the general notation.
     *
     * We are working with a primary part which looks something like
     *
     * 
$$Z/8 + Z/4 + Z/4 + Z/2$$

     *
     * and we want to find generators
     *
     * 
$$\begin{aligned} a_3 &= (1, 0, 0, 0) \\ a_2 &= (0, 1, 0, 0) \\ a_1 &= (0, 0, 1, 0) \\ a_0 &= (0, 0, 0, 1) \end{aligned}$$

     *
     * We begin by finding an element of maximal order (order 8 in this
     * example). We may take the element of maximal order to be the
     * generator  $a_3$ . This follows from the following two lemmas.
     *
     * Lemma 1. An element of maximal order must have a first component
     * which generates the  $Z/8$  factor. That is, it must look like
     *  $(1, *, *, *)$ , relative to some choice of generator for the  $Z/8$  factor.
     *
     * Proof. If it looked like  $(0, *, *, *)$  or  $(2, *, *, *)$  it would have
     * order 4, not order 8. If the group had two  $Z/8$  factors, then the
     * element of maximal order would have to have a component which
     * generates at least one of the factors, and we'd list that factor first.
     *
     * Lemma 2. The map
     *
     * 
$$\begin{aligned} (1, 0, 0, 0) &\rightarrow (1, *, *, *) \\ (0, 1, 0, 0) &\rightarrow (0, 1, 0, 0) \\ (0, 0, 1, 0) &\rightarrow (0, 0, 1, 0) \end{aligned}$$

     */
}

```

```

*
*      ...
*      (0, 0, 0, 1) -> (0, 0, 0, 1)
*
* is an automorphism of the primary part.
*
* Proof. Each generator of order n goes to an element of order n,
* so the map is a homomorphism. It's onto, so it's an isomorphism.
*
* Lemmas 1 and 2 together imply that an arbitrary element
* of maximal order may be taken to be (1, 0, 0, 0).
*
*
* We'll extend this approach to find the remaining generators.
*
* Let P be the primary part we are working with, and
* let H be the subgroup of P generated by the generators we've
* found so far. (Initially H = {0}.)
*
* Define the "coset order" of an element to be the order of its
* coset in the quotient group P/H.
*
* Continuing with the above example, say we've found a3 and
* are looking for a2. We choose a highest order element
* whose coset order equals its regular order. The following
* two lemmas show that this element may be taken to be a2.
*
* Lemma 1'. A highest order element whose coset order equals
* its regular order is of the form (*, 1, *, *), relative to some
* choice of generator for the Z/4 factor.
*
* Proof. The element (0, 1, 0, 0) has coset order = regular order = 4,
* and no element has coset order 8, so clearly the given element
* must have coset order = regular order = 4. If it were of the
* form (*, 0or2, 0or2, *) it would have coset order 2, not four,
* so either the second or third component must be a 1 or a 3.
* After possibly interchanging the two Z/4 factors and/or choosing
* a new generator for a Z/4 factor, we may assume our element is
* of the form (*, 1, *, *).
*
* Lemma 2'. If the coset order of (*, 1, *, *) equals its
* regular order, then the map
*
*      (1, 0, 0, 0) -> (1, 0, 0, 0)
*      (0, 1, 0, 0) -> (*, 1, *, *)
*      (0, 0, 1, 0) -> (0, 0, 1, 0)
*      ...
*      (0, 0, 0, 1) -> (0, 0, 0, 1)
*
* is an automorphism of the primary part.
*
* Proof. Each generator of order n goes to an element of order n
* (this is where we use coset order == regular order),
* so the map is a homomorphism. It's onto, so it's an isomorphism.
* Note that this automorphism doesn't affect our previously
* chosen a3 = (1, 0, 0, 0).
*
* We continue in this fashion until all generators have been found.
*/

Boolean *belongs_to_H,
*old_belongs_to_H;
int i,
size_of_H,
*coset_order,
*regular_order,
*running_product,
*max_order,
*new_generator,
*power_of_new_generator;

/*
* Initially we have no generators.
*/
primary_part->num_generators = 0;

```

```

/*
 * The array belongs_to_H keeps track of which elements
 * are in the subgroup H. That is, belongs_to_H[i] is
 * true iff the element i belongs to H (the index i refers
 * to the elements index in the full group G, NOT its
 * array index in the PrimaryPart).
 */
belongs_to_H      = NEW_ARRAY(the_group->order, Boolean);
old_belongs_to_H  = NEW_ARRAY(the_group->order, Boolean);

/*
 * Initially only the identity belongs to H.
 */
belongs_to_H[0] = TRUE;
for (i = 1; i < the_group->order; i++)
    belongs_to_H[i] = FALSE;

/*
 * size_of_H keeps track of how many elements are in H.
 * Initially we've got only the identity.
 */
size_of_H = 1;

/*
 * We'll need to keep track of both the coset and
 * regular orders of each element of the primary part.
 * Unlike belongs_to_H[], coset_order[] and regular_order[]
 * are indexed relative to the primary part.
 */

coset_order      = NEW_ARRAY(primary_part->num_elements, int);
regular_order     = NEW_ARRAY(primary_part->num_elements, int);

/*
 * Record the regular_orders.
 */

for (i = 0; i < primary_part->num_elements; i++)
    regular_order[i] = the_group->order_of_element[primary_part->element[i]];

/*
 * We'll keep looking for new generators until H fills
 * up the whole primary part.
 */

while (size_of_H < primary_part->num_elements)
{
    /*
     * Compute the coset_orders.
     */

    for (i = 0; i < primary_part->num_elements; i++)
    {
        coset_order[i] = 1;
        running_product = primary_part->element[i];
        while (belongs_to_H[running_product] == FALSE)
        {
            running_product = the_group->product
                               [running_product]
                               [primary_part->element[i]];
            coset_order[i]++;
        }
    }

    /*
     * Find the highest order element whose order
     * equals its coset order.
     */

    max_order = 0;

    for (i = 0; i < primary_part->num_elements; i++)
    {

```

```

        if (coset_order[i] == regular_order[i]
            && coset_order[i] > max_order)
        {
            max_order = coset_order[i];
            new_generator = primary_part->element[i];
        }
    }

    /*
     * Just for an unnecessary error check . . .
     */

    if (max_order < 2)
        uFatalError("primary_part_generators", "symmetry_group");

    /*
     * Record the new generator.
     */

    primary_part->generator[primary_part->num_generators++] = new_generator;

    /*
     * Update H.
     */

    for (i = 0; i < the_group->order; i++)
        old_belongs_to_H[i] = belongs_to_H[i];

    power_of_new_generator = 0;
    do
    {
        for (i = 0; i < the_group->order; i++)
            if (old_belongs_to_H[i] == TRUE)
                belongs_to_H[the_group->product[i][power_of_new_generator]] = TRUE;
        power_of_new_generator = the_group->product[power_of_new_generator]
[new_generator];
    } while (power_of_new_generator != 0);

    size_of_H *= max_order;
}

my_free(belongs_to_H);
my_free(old_belongs_to_H);

my_free(coset_order);
my_free(regular_order);
}

static void find_lexicographic_ordering(
    SymmetryGroup *the_group,
    int num_generators,
    int the_generators[],
    int **desired_ordering)
{
    int i,
        j,
        count,
        old_count,
        power_of_generator;

    /*
     * Allocate space for the desired_ordering.
     */
    *desired_ordering = NEW_ARRAY(the_group->order, int);

    /*
     * The identity goes first.
     */
    (*desired_ordering)[0] = 0;

    /*
     * We've got one element in the array.
     */

```

```

count = 1;

/*
 * For each generator . . .
 */
for (i = 0; i < num_generators; i++)
{
    old_count = count;

    /*
     * . . . consider each of its nontrivial powers . . .
     */

    for (    power_of_generator = the_generators[i];
           power_of_generator != 0;
           power_of_generator = the_group->product
                               [power_of_generator]
                               [the_generators[i]]
        )

        /*
         * . . . and multiply all the old elements on the
         * desired_ordering array by each power_of_generator.
         */

        for (j = 0; j < old_count; j++)

            (*desired_ordering)[count++] = the_group->product
            [(*desired_ordering)[j]][power_of_generator];
}

static void attach_abelian_description(
    SymmetryGroup  *the_group,
    int            num_generators,
    int            the_generators[])
{
    int i;

    the_group->abelian_description = NEW_STRUCT(AbelianGroup);

    the_group->abelian_description->num_torsion_coefficients = num_generators;

    the_group->abelian_description->torsion_coefficients = NEW_ARRAY(num_generators, long int);

    /*
     * the_generators[] lists the generators in descending order,
     * so reverse that order here.
     */

    for (i = 0; i < num_generators; i++)

        the_group->abelian_description->torsion_coefficients[(num_generators - 1) - i]
            = the_group->order_of_element[the_generators[i]];
}

```