

```

/*
 * abelian_group.c
 *
 * This file contains the following functions which the kernel
 * provides for the UI:
 *
 *     void expand_abelian_group(AbelianGroup *g);
 *     void compress_abelian_group(AbelianGroup *g);
 *     void free_abelian_group(AbelianGroup *g);
 *
 * expand_abelian_group() expands an AbelianGroup into its most
 * factored form, e.g.  $Z/2 + Z/2 + Z/4 + Z/3 + Z/9 + Z$ .
 * Each nonzero torsion coefficient is a power of a prime.
 * These are the "primary invariants" of the group
 * (cf. Hartley & Hawkes' Rings, Modules and Linear Algebra,
 * Chapman & Hall 1970, page 154).
 *
 * compress_abelian_group compresses an AbelianGroup into its least
 * factored form, e.g.  $Z/2 + Z/6 + Z/36 + Z$ .
 * Each torsion coefficient divides all subsequent torsion coefficients.
 * These are the "torsion invariants" of the group
 * (cf. Hartley & Hawkes' Rings, Modules and Linear Algebra,
 * Chapman & Hall 1970, page 153).
 *
 * free_abelian_group() frees the memory used to hold the AbelianGroup *g.
 * As explained in the documentation preceding the definition of an
 * AbelianGroup in SnapPea.h, only the kernel may allocate or deallocate
 * AbelianGroups.
 */

#include "kernel.h"
#include <stdlib.h>      /* needed for qsort() */

typedef struct prime_power
{
    int                prime,
                    power;
    struct prime_power *next;
} PrimePower;

static int CDECL compare_prime_powers(const void *pp0, const void *pp1);

void expand_abelian_group(
    AbelianGroup *g)
{
    PrimePower *prime_power_list,
                *new_prime_power,
                **array_of_pointers,
                *this_prime_power;
    int         num_prime_powers,
                torsion_free_rank;
    long int    m,
                p,
                this_coefficient;
    int         i,
                count;

    /*
     * The documentation at the top of this file specifies the behavior
     * of expand_abelian_group().
     */

    /*
     * Ignore nonexistent groups.
     */
    if (g == NULL)
        return;

    /*
     * The algorithm is to factor each torsion coefficient into prime
     * powers, combine the prime powers for all the torsion coefficients

```

```

    * into a single list, sort the list, and write the results back
    * to the torsion coefficients array (after allocating more memory
    * for the array, of course).
    */

prime_power_list      = NULL;
num_prime_powers      = 0;
torsion_free_rank     = 0;

for (i = 0; i < g->num_torsion_coefficients; i++)
{
    /*
     * For notational convenience, let m be the torsion coefficient
     * under consideration.
     */

    m = g->torsion_coefficients[i];

    /*
     * If m is zero (indicating an infinite cyclic factor), make a
     * note of it and move on to the next torsion coefficient.
     */

    if (m == 0L)
    {
        torsion_free_rank++;
        continue;
    }

    /*
     * Factor m.
     * (Much more efficient algorithms could be used to factor m, but at the
     * moment I'm assuming the numbers involved will be small, so the
     * simplicity of the code is more important than its efficiency.)
     */

    /*
     * Consider each potential prime divisor p of m.
     * (We "accidentally" consider composite divisors as well,
     * but the wasted effort shouldn't be too significant.)
     */
    for (p = 2; m > 1L; p++)
    {
        /*
         * Does p divide m?
         * If so, then p must be prime, since otherwise some previous value
         * of p would have divided m, and would have been factored out.
         * Find the largest power of p which divides m, and record it on
         * the prime_power_list.
         */

        if (m%p == 0L)
        {
            new_prime_power = NEW_STRUCT(PrimePower);
            new_prime_power->prime = p;
            new_prime_power->power = 0;
            new_prime_power->next = prime_power_list;
            prime_power_list = new_prime_power;
            num_prime_powers++;

            while (m%p == 0L)
            {
                m /= p;
                new_prime_power->power++;
            }
        }

        /*
         * If m is less than p^2, then m must be prime or one.
         *
         * if m is prime, we set p = m - 1, so that on the next
         * pass through the loop (after the "p++"), p will
         * equal m, and we'll finish up.
         */
    }
}

```

```

        *   If m is one, then the loop will terminate no matter
        *       what p is, so there's no harm in setting p = m - 1.
        */

        if (m < p * p)
            p = m - 1;
    }
}

/*
 *   Set num_torsion_coefficients, and reallocate space
 *   for the (presumably larger) array.
 */

g->num_torsion_coefficients = num_prime_powers + torsion_free_rank;

if (g->torsion_coefficients != NULL)
    my_free(g->torsion_coefficients);

if (g->num_torsion_coefficients > 0)
    g->torsion_coefficients = NEW_ARRAY(g->num_torsion_coefficients, long int);
else
    g->torsion_coefficients = NULL;

/*
 *   If the list of PrimePowers is nonempty, sort it and read it
 *   into the torsion_coefficients array.
 */

if (num_prime_powers > 0)
{
    /*
     *   Create an array of pointers to the PrimePowers.
     */

    array_of_pointers = NEW_ARRAY(num_prime_powers, PrimePower *);

    for (    i = 0, this_prime_power = prime_power_list;
           i < num_prime_powers;
           i++, this_prime_power = this_prime_power->next)

        array_of_pointers[i] = this_prime_power;

    if (this_prime_power != NULL)
        uFatalError("expand_abelian_group", "abelian_group");

    qsort(array_of_pointers, num_prime_powers, sizeof(PrimePower *),
compare_prime_powers);

    for (i = 0; i < num_prime_powers; i++)
    {
        /*
         *   Multiply out the current torsion coefficient . . .
         */

        this_coefficient = 1L;

        for (count = 0; count < array_of_pointers[i]->power; count++)
            this_coefficient *= array_of_pointers[i]->prime;

        /*
         *   . . . write it into the array . . .
         */

        g->torsion_coefficients[i] = this_coefficient;

        /*
         *   . . . and free the PrimePower.
         */

        my_free(array_of_pointers[i]);
    }
}

```

```

    my_free(array_of_pointers);
}

/*
 * Write a torsion coefficient of zero for each infinite cyclic factor.
 */

for (i = num_prime_powers; i < g->num_torsion_coefficients; i++)
    g->torsion_coefficients[i] = 0L;
}

static int CDECL compare_prime_powers(
    const void *pp0,
    const void *pp1)
{
    if ((*((PrimePower **)pp0)->prime < *((PrimePower **)pp1)->prime)
        return -1;

    if ((*((PrimePower **)pp0)->prime > *((PrimePower **)pp1)->prime)
        return +1;

    if ((*((PrimePower **)pp0)->power < *((PrimePower **)pp1)->power)
        return -1;

    if ((*((PrimePower **)pp0)->power > *((PrimePower **)pp1)->power)
        return +1;

    return 0;
}

void compress_abelian_group(
    AbelianGroup *g)
{
    int        i,
              ii,
              j;
    long int    m,
              n,
              d;

    /*
     * The documentation at the top of this file specifies the behavior
     * of compress_abelian_group().
     */

    /*
     * Ignore nonexistent groups.
     */
    if (g == NULL)
        return;

    /*
     * Beginning at the start of the list, adjust each torsion coefficient
     * so that it divides all subsequent torsion coefficients.
     */

    for (i = 0; i < g->num_torsion_coefficients; i++)
        for (j = i + 1; j < g->num_torsion_coefficients; j++)
        {
            /*
             * For clarity, let the torsion coefficients under
             * consideration be called m and n.
             */

            m = g->torsion_coefficients[i];
            n = g->torsion_coefficients[j];

            /*
             * If both m and n are zero, go on to the next
             * iteration of the loop.

```

```

    */

    if (m == 0L && n == 0L)
        continue;

    /*
     * Compute the greatest common divisor of m and n.
     * Note that the greatest common divisor, which is
     * defined iff m and n are not both zero, will always
     * be nonzero.
     */

    d = gcd(m, n);

    /*
     * Define m' = m/d. Replace m with d, and n with n * m'.
     * To convince yourself that this is valid, consider
     * the factorizations of m and n into powers of primes.
     * In effect, we are swapping the higher power of a given
     * prime from m to n, e.g. {m = 8, n = 4} -> {m = 4, n = 8}.
     */

    n *= m / d;
    m = d;

    /*
     * Write the values of m and n back into the torsion_coefficients
     * array.
     */

    g->torsion_coefficients[i] = m;
    g->torsion_coefficients[j] = n;
}

/*
 * Delete torsion coefficients of one, and adjust
 * g->num_torsion_coefficients accordingly.
 */

/*
 * First set ii to be the index of the first non-one, if any.
 */

for (    ii = 0;
        ii < g->num_torsion_coefficients && g->torsion_coefficients[ii] == 1L;
        ii++)
    ;

/*
 * Now shift all the non-ones to the start of the array,
 * overwriting the ones. (Note that this algorithm is valid
 * even if the array contains all ones, or no ones.)
 */

for (i = 0; ii < g->num_torsion_coefficients; i++, ii++)
    g->torsion_coefficients[i] = g->torsion_coefficients[ii];

/*
 * Set g->num_torsion_coefficients to its correct value;
 */

g->num_torsion_coefficients = i;
}

void free_abelian_group(
    AbelianGroup *g)
{
    if (g != NULL)
    {
        if (g->torsion_coefficients != NULL)
            my_free(g->torsion_coefficients);
        my_free(g);
    }
}

```

```
    }  
}
```