```c
/* Implementation of bignums by Henrik.Johansson@Nexus.Comm.SE in 1991
 * version 1.2
 */

#include "bignum.h"

#define DIGIT BIGNUM_DIGIT
#define DIGIT2 BIGNUM_TWO_DIGITS

#define TRUE (1 == 1)
#define FALSE (1 != 1)

#define MIN_ALLOC ((sizeof(long) / sizeof(DIGIT)) << 1)

/* Don't expect the "sign" field to have the right value (BIG_SIGN_0) at
 * all times, so compare real content of bignum with zerop.
 * Maybe it would be better to be sure at all times that the sign is right?
 */
#define zerop(BIG) ((*(BIG)->dp == 0) && ((BIG)->dgs_used == 1))
#define uonep(BIG) ((*(BIG)->dp == 1) && ((BIG)->dgs_used == 1))
#define NEGATE_SIGN(SGN) (-(SGN))

#define DIGIT_BITS BIGNUM_DIGIT_BITS
#define DIGIT2_BITS BIGNUM_TWO_DIGITS_BITS

#define DIGIT_PART(N) ((DIGIT)((N) & ((((DIGIT2)1) << DIGIT_BITS) - 1)))
#define RESULT_MINUSP(RES) (((RES) & ((DIGIT2)1 << (DIGIT2_BITS - 1))) != 0)
#define SHIFT_DIGIT_DOWN(N) (DIGIT_PART((N) >> DIGIT_BITS))
#define SHIFT_DIGIT_UP(N) ((DIGIT2)((N) << DIGIT_BITS))

#define DIGIT_BASE (((DIGIT2)1) << DIGIT_BITS)
#define MAX_DIGIT ((((DIGIT2)1) << DIGIT_BITS) - 1)

#define uint unsigned int
#define ulong unsigned long

extern char *malloc();
extern int free();

char *last_string, *big_end_string;
ulong length_last_string;
char big_base_digits[] =
 "00112233445566778899AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz";
char error_string[] = "(big_errno != 0, something has gone wrong)";

bignum big_one;
bignum tmp_string, tmp_add, tmp_mul, tmp_q, tmp_r, tmp_rand, tmp_round;
DIGIT *tmp_digits;
ulong tmp_ulong;

/* #define COUNT_ALLOC */
#ifdef COUNT_ALLOC
ulong dgs_alloc = 0, digits_freed = 0;
#endif

int big_errno = BIG_OK;
```

```
double log2tbl[] =
{
    -1.0, -1.0,
    1.000000000000000, 1.584961891174317, 2.000000000000002, 2.321928024291994,
    2.584960937500003, 2.807353973388675, 3.000000000000004, 3.169923782348637,
    3.321928024291997, 3.459430694580083, 3.584960937500005, 3.700439453125006,
    3.807353973388678, 3.906888961791999, 4.000000000000014, 4.087459564208999,
    4.169921875000016, 4.247924804687517, 4.321926116943377, 4.392314910888691,
    4.459430694580098, 4.523559570312520, 4.584960937500021, 4.643856048584007,
    4.700439453125023, 4.754886627197290, 4.807353973388697, 4.857978820800807,
    4.906887054443386, 4.954193115234403, 5.000000000000028, 5.044391632080107,
    5.087459564209015, 5.129280090332062, 5.169921875000032
};

struct digit_blocks
{
    int digCnt;
    DIGIT dig_base;
} big_block[37];                /* The code uses index 2 to 36 */


/* -----------------------------------------------------------------------
 * All static utility functions are placed first in this file.
 */

static char *                   /* You may remove this function if you */
strchr(str_ptr, ch)             /* have it in your standard library */
char *str_ptr;
char ch;
{
    do
    {
        if (*str_ptr == ch)
        {
            return str_ptr;
        }
    } while (*str_ptr++ != '\0');
    return NULL;
}

#if 0

extern int printf();

static void
print_digits(prefix, a)
char *prefix;
bignum *a;
{
    unsigned long i;
    ulong tmp;

    printf("%s", prefix);
    if (a->sign == BIG_SIGN_MINUS)
    {
        printf("- ");
    }
    else
    {
        if (a->sign == BIG_SIGN_PLUS)
        {
            printf("+ ");
        }
        else
        {
            printf("+/- ");
        }
    }
    for (i = a->dgs_used - 1; i > 0; i--)
    {
        tmp = a->dp[i];
        printf("%lu, ", tmp);
    }
    tmp = *a->dp;
    printf("%lu\n", tmp);
}

#else

#define print_digits(prefix, big) /* */

#endif

static void
init_digit_blocks()
{
    uint digcnt, base;
    DIGIT maxdigit, curdigit, tmp;

    for (base = 2; base <= 36; base++)
    {
        tmp = ((1 << (DIGIT_BITS - 1)) / base);
        maxdigit = tmp * 2;
        curdigit = 1;
        digcnt = 0;
        while (curdigit < maxdigit)
        {
            digcnt++;
            curdigit *= base;
        }
        big_block[base].digCnt = digcnt;
        big_block[base].dig_base = curdigit;
    }
}

#ifdef COUNT_ALLOC
static void
free_digits(dp, count)
DIGIT *dp;
ulong count;
{
    digits_freed += count;
    free((char *)dp);
```

```
}
#else
#define free_digits(DP,CNT) free((char *)DP)
#endif

static DIGIT *
allocate_digits(alloclen)
ulong alloclen;
{
    DIGIT *digit_ptr;

    if (big_errno != BIG_OK)
    {
        return NULL;
    }
#ifdef COUNT_ALLOC
    dgs_alloc += alloclen;
#endif
    digit_ptr = (DIGIT *)malloc(alloclen * sizeof(DIGIT));
    if (digit_ptr == NULL)
    {
        big_errno = BIG_MEMERR;
        return NULL;
    }
    return digit_ptr;
}

static bigerr_t
newsize(dp_p, cursize_p, minsz, newsz)
DIGIT **dp_p;
ulong *cursize_p;
ulong minsz;
ulong newsz;
{
    if (*cursize_p >= minsz)
    {
        return big_errno;
    }
    free_digits(*dp_p, *cursize_p);
    if (newsz < MIN_ALLOC)
    {
        newsz = MIN_ALLOC;
    }
    if ((*dp_p = allocate_digits(newsz)) == NULL)
    {
        return big_errno;
    }
    *cursize_p = newsz;
    return big_errno;
}

/* `memcpy' uses an `int' as counter.  If an int can't hold a big enough
 * counter, then `digits_cpy' becomes a function (and a little slower,
 * probably.  Unfortunately `memcpy' takes a signed counter, but _that_
 * size of numbers are almost too big!  Or isn't it?
 */
#ifdef MEMCPY_LONG_COUNTER
```

```
extern char *memcpy();
#define digits_cpy(dst, src, count) memcpy((char *)(dst), (char *)(src), \
                                           (count) * sizeof(DIGIT))
#else
static void
digits_cpy(dst, src, count)
DIGIT *dst;
DIGIT *src;
ulong count;
{
    while (count-- > 0)
    {
        *dst++ = *src++;
    }
}
#endif

static DIGIT *
copy_digits(src, cp_count, alloclen)
DIGIT *src;
ulong cp_count;
ulong alloclen;
{
    DIGIT *dst;

    if (big_errno != BIG_OK)
    {
        return NULL;
    }
    if ((dst = allocate_digits(alloclen)) == NULL)
    {
        return NULL;
    }
    digits_cpy(dst, src, cp_count);
    return dst;
}

static void
add_digit(a, d)
bignum *a;
DIGIT d;
{
    DIGIT2 res = d;
    DIGIT *last_digit, *digit_ptr = a->dp, *digvect;

    last_digit = digit_ptr + a->dgs_used;
    while (digit_ptr < last_digit)
    {
        res = *digit_ptr + res;
        *digit_ptr = DIGIT_PART(res);
        res = SHIFT_DIGIT_DOWN(res);
        digit_ptr++;
        if (res == 0)
        {
            break;
        }
    }
    if (res != 0)
    {
        if (a->dgs_used < a->dgs_alloc)
        {
            *digit_ptr = DIGIT_PART(res);
        }
        else
        {
            if ((digvect = copy_digits(a->dp, a->dgs_used,
                                       a->dgs_used + 4)) == NULL)
            {
                return;         /* big_errno will be set to last error */
            }
            digvect[a->dgs_used] = DIGIT_PART(res);
            free_digits(a->dp, a->dgs_alloc);
            a->dgs_alloc = a->dgs_used + 4;
            a->dp = digvect;
        }
        a->dgs_used++;
    }
}

static DIGIT
vect_div_digit(digvect, len, d)
DIGIT *digvect;
ulong *len;
DIGIT d;
{
    DIGIT *digit_ptr = digvect + *len - 1, newval;
    DIGIT2 rest = 0;

    if (d == 0)
    {
        big_errno = BIG_DIV_ZERO;
        return -1;
    }
    if (d == 1)
    {
        return 0;
    }
    while (digit_ptr >= digvect)
    {
        rest = (DIGIT2)SHIFT_DIGIT_UP(rest);
        newval = DIGIT_PART((*digit_ptr + rest) / d);
        rest = (*digit_ptr + rest) % d;
        *digit_ptr = newval;
        digit_ptr--;
    }
    if (*len > 1)
    {
        if (digvect[*len - 1] == 0)
        {
            *len -= 1;
        }
    }
    return DIGIT_PART(rest);
}
```

```
static DIGIT
udiv_digit(a, d)
bignum *a;
DIGIT d;
{
    DIGIT res;

    res = vect_div_digit(a->dp, &a->dgs_used, d);
    if (zerop(a))
    {
        a->sign = BIG_SIGN_0;
    }
    return res;
}

static DIGIT
vect_mul_digit(digvect, len, x)
DIGIT *digvect;
ulong len;
DIGIT x;
{
    DIGIT *digit_ptr = digvect + len;
    DIGIT2 res = 0;

    while (digvect < digit_ptr)
    {
        res += *digvect * (DIGIT2)x;
        *digvect = DIGIT_PART(res);
        res = SHIFT_DIGIT_DOWN(res);
        digvect++;
    }
    return DIGIT_PART(res);
}

static void
umul_digit(a, x)
bignum *a;
DIGIT x;
{
    DIGIT ovfl, *digvect;

    ovfl = vect_mul_digit(a->dp, a->dgs_used, x);
    if (ovfl != 0)
    {
        if (a->dgs_used < a->dgs_alloc)
        {
            a->dp[a->dgs_used] = ovfl;
        }
        else
        {
            digvect = copy_digits(a->dp,
                                  a->dgs_used,
                                  a->dgs_used + 4);
            digvect[a->dgs_used] = ovfl;
            free_digits(a->dp, a->dgs_alloc);
            a->dgs_alloc = a->dgs_used + 4;
            a->dp = digvect;
        }
        a->dgs_used++;
    }
}

static int
ucompare_digits(a, b)
bignum *a;
bignum *b;
{
    DIGIT *a_ptr, *b_ptr;

    if (a->dgs_used  == b->dgs_used)
    {
        a_ptr = a->dp + a->dgs_used - 1;
        b_ptr = b->dp + b->dgs_used - 1;
        while ((*a_ptr == *b_ptr) && (a_ptr >= a->dp))
        {
            a_ptr--;
            b_ptr--;
        }
        if (a_ptr < a->dp)
        {
            return 0;
        }
        else
        {
            return (*a_ptr > *b_ptr) ? 1 : -1;
        }
    }
    return (a->dgs_used > b->dgs_used) ? 1 : -1;
}

static void
uadd_digits(a, b, c)
bignum *a;
bignum *b;
bignum *c;
{
    DIGIT *dp_x, *dp_y, *dp_z, *res_dp, *end_x, *end_y;
    ulong len_x, len_y;
    DIGIT2 res = 0;

    if (a->dgs_used > b->dgs_used)
    {
        dp_x = a->dp;
        len_x = a->dgs_used;
        dp_y = b->dp;
        len_y = b->dgs_used;
    }
    else
    {
        dp_x = b->dp;
        len_x = b->dgs_used;
        dp_y = a->dp;
        len_y = a->dgs_used;
    }
    end_x = dp_x + len_x;
    end_y = dp_y + len_y;
    if (c->dgs_alloc >= len_x)
    {
        dp_z = c->dp;
    }
    else
    {
        if (newsize(&tmp_add.dp, &tmp_add.dgs_alloc,
                    len_x, len_x + 4) != BIG_OK)
        {
            return;
        }
        dp_z = tmp_add.dp;
    }
    res_dp = dp_z;
    while (dp_y < end_y)
    {
        res += ((DIGIT2)*dp_x++) + *dp_y++;
        *res_dp++ = DIGIT_PART(res);
        res = SHIFT_DIGIT_DOWN(res);
    }
    while (dp_x < end_x)
    {
        res += *dp_x++;
        *res_dp++ = DIGIT_PART(res);
        res = SHIFT_DIGIT_DOWN(res);
    }
    if (res != 0)
    {
        *res_dp++ = DIGIT_PART(res);
    }

    if (dp_z != c->dp)
    {
        tmp_digits = c->dp;
        c->dp = tmp_add.dp;
        tmp_add.dp = tmp_digits;

        tmp_ulong = c->dgs_alloc;
        c->dgs_alloc = tmp_add.dgs_alloc;
        tmp_add.dgs_alloc = tmp_ulong;
    }
    c->dgs_used = res_dp - c->dp;
}

static void
usub_digits(a, b, c)
bignum *a;
bignum *b;
bignum *c;
{
    DIGIT *dp_x, *dp_y, *dp_z, *res_dp, *end_x, *end_y;
    ulong len_x, len_y;
    DIGIT2 res = 0;
    dp_x = a->dp;
    len_x = a->dgs_used;
    dp_y = b->dp;
    len_y = b->dgs_used;

    end_x = dp_x + len_x;
    end_y = dp_y + len_y;
    if (c->dgs_alloc >= len_x)
    {
        dp_z = c->dp;
    }
    else
    {
        if (newsize(&tmp_add.dp, &tmp_add.dgs_alloc,
                    len_x, len_x + 2) != BIG_OK)
        {
            return;
        }
        dp_z = tmp_add.dp;
    }
    res_dp = dp_z;
    while (dp_y < end_y)
    {
        res = ((DIGIT2)*dp_x++) - *dp_y++ - RESULT_MINUSP(res);
        *res_dp++ = DIGIT_PART(res);
    }
    while (dp_x < end_x)
    {
        res = *dp_x++ - RESULT_MINUSP(res);
        *res_dp++ = DIGIT_PART(res);
    }
#ifdef BIG_CHECK_LIMITS
    if (RESULT_MINUSP(res) != 0)
    {
        big_errno = BIG_ALGERR;
        return;
    }
#endif
    while ((*--res_dp == 0) && (res_dp > dp_z))
    {
        /* Move pointer down until we reach a non-zero */
    }
    if (dp_z != c->dp)
    {
        tmp_digits = c->dp;
        c->dp = tmp_add.dp;
        tmp_add.dp = tmp_digits;

        tmp_ulong = tmp_add.dgs_alloc;
        tmp_add.dgs_alloc = c->dgs_alloc;
        c->dgs_alloc = tmp_ulong;
    }
    c->dgs_used = res_dp - dp_z + 1;
}

/*
 *   This (pseudo) random number generator is not very good.  It has a long
 * period [ 2 ^ (DIGIT_BITS * 2) (?)] before it starts with the same sequence
 * again, but the lower bits are "less random" than they should be.  I've
 * solved it by using word of double length, and returning the upper bits.
 * Please mail me if you know how to make it into a "more random" generator.
 *   One important thing though: it will have to reasonably fast.
 *   The good thing with this one is that it is very portable, but that doesn't
 * help much when you want _good_ random numbers.
 *                                                Henrik.Johansson@Nexus.Comm.SE
 */
static DIGIT
rand()
{
    static DIGIT2 x1 = 33, x2 = 45; /* Just give them two starting values */

    if (x2 = BIG_RAND_A2 * x2 + BIG_RAND_C2, x2 == 45)
    {
        x1 = BIG_RAND_A1 * x1 + BIG_RAND_C1;
    }
    return SHIFT_DIGIT_DOWN(x1 + x2); /* Skip least significant bits */
}

/* ------------------------------------------------------------------------
 * All external functions comes here.
 */

bigerr_t
big_init_pkg()
{
    init_digit_blocks();
    big_create(&tmp_string);
    big_create(&tmp_add);
    big_create(&tmp_mul);
    big_create(&tmp_q);
    big_create(&tmp_r);
    big_create(&tmp_rand);
    big_create(&big_one);
    big_set_long((long)1, &big_one);
    length_last_string = 10;
    if ((last_string = malloc(length_last_string)) == NULL)
    {
        big_errno = BIG_MEMERR;
    }
    return big_errno;
}

void
big_release_pkg()
{
    big_destroy(&tmp_string);
    big_destroy(&tmp_add);
    big_destroy(&tmp_mul);
    big_destroy(&tmp_q);
    big_destroy(&tmp_r);
    big_destroy(&tmp_round);
    big_destroy(&big_one);
    free(last_string);
#ifdef COUNT_ALLOC
```

```
    printf("Allocated digits: %lu\n", dgs_alloc);
    printf("Freed digits:     %lu\n", digits_freed);
#endif
}

bigerr_t
big_create(a)
bignum *a;
{
    if (big_errno != BIG_OK)
    {
        return big_errno;
    }
    a->sign = BIG_SIGN_0;
    a->dgs_used = 1;
    if ((a->dp = allocate_digits((long)sizeof(long))) == NULL)
    {
        return big_errno;
    }
    a->dgs_alloc = sizeof(long);
    *a->dp = 0;
    return big_errno;
}

void
big_destroy(a)
bignum *a;
{
    free_digits(a->dp, a->dgs_alloc);
}

ulong
big_bitcount(a)
bignum *a;
{
    int bits = 0;
    DIGIT high_digit;

    if (big_errno != BIG_OK)
    {
        return 0;
    }
    high_digit = a->dp[a->dgs_used - 1];
    while (high_digit != 0)
    {
        bits++;
        high_digit >>= 1;
    }
    return DIGIT_BITS * (a->dgs_used - 1) + bits;
}

bigerr_t
big_set_big(a, b)
bignum *a;
bignum *b;
{
    if ((big_errno != BIG_OK) || (a == b))
    {
        return big_errno;
    }
    if (newsize(&b->dp, &b->dgs_alloc, a->dgs_used, a->dgs_used) != BIG_OK)
    {
        return big_errno;
    }
    b->dgs_used = a->dgs_used;
    b->sign = a->sign;
    digits_cpy(b->dp, a->dp, a->dgs_used);
    return big_errno;
}

void
big_set_ulong(n, a)
ulong n;
bignum *a;
{
    int i;

    if (big_errno != BIG_OK)
    {
        return;
    }
    if (n == 0)
    {
        *a->dp = 0;
        a->dgs_used = 1;
        a->sign = BIG_SIGN_0;
    }
    else
    {
        a->dgs_used = 0;
        for (i = 0; i < sizeof(long) / sizeof(DIGIT); i++)
        {
            if (n == 0)
            {
                break;
            }
            a->dgs_used++;
            a->dp[i] = DIGIT_PART(n);
            n = SHIFT_DIGIT_DOWN(n);
        }
        a->sign = BIG_SIGN_PLUS;
    }
    print_digits("set_(u)long: a = ", a);
}

void
big_set_long(n, a)
long n;
bignum *a;
{
    ulong m;

    m = (n < 0) ? - n : n;
    big_set_ulong(m, a);
    if (!(n >= 0))
    {
        a->sign = BIG_SIGN_MINUS;
    }
}

bigerr_t
big_set_string(numstr, base, a)
char *numstr;
int base;
bignum *a;
{
    char *src = numstr, *maxdigit, *chrptr;
    DIGIT dig_base, dig_sum, last_base;
    int cnt, maxcnt;

    if (big_errno != BIG_OK)
    {
        return big_errno;
    }
    big_end_string = numstr;
    if ((base < 2) || (base > 36))
    {
        big_errno = BIG_ARGERR;
        return big_errno;
    }

    maxdigit = big_base_digits + (base << 1);

    maxcnt = big_block[base].digCnt;
    dig_base = big_block[base].dig_base;
    a->dgs_used = 1;
    *a->dp = 0;
    a->sign = BIG_SIGN_PLUS;    /* Assume it will be positive */
    while (strchr(" \t\n\r", *src) != NULL) /* Skip whitespaces */
    {
        src++;
    }
    if ((*src == '-') || (*src == '+')) /* First non-white is a sign? */
    {
        a->sign = (*src == '-') ? BIG_SIGN_MINUS : BIG_SIGN_PLUS;
        src++;
    }
    chrptr = strchr(big_base_digits, *src);
    if ((chrptr == NULL) || (chrptr >= maxdigit)) /* Next chr not a digit? */
    {
        big_end_string = src;
        big_errno = BIG_ARGERR;
        return big_errno;
    }
    while (*src == '0')         /* Next chr a '0'? */
    {
        src++;                  /* Read past all '0'es */
    }
    chrptr = strchr(big_base_digits, *src);
    if ((chrptr == NULL) || (chrptr >= maxdigit)) /* Next char not a digit */
    {
        big_end_string = src;
        a->sign = BIG_SIGN_0;   /* It was just a plain 0 */
        return big_errno;
    }
    dig_sum = 0;
    cnt = 0;
    while ((chrptr = strchr(big_base_digits, *src)),
           (chrptr != NULL) && (chrptr < maxdigit))
    {
        dig_sum = dig_sum * base + ((chrptr - big_base_digits) >> 1);
        if (++cnt >= maxcnt)
        {
            umul_digit(a, dig_base);
            add_digit(a, dig_sum);
            dig_sum = 0;
            cnt = 0;
        }
        src++;
    }
    if (cnt > 0)
    {
        last_base = base;
        while (--cnt > 0)
        {
            last_base *= base;
        }
        umul_digit(a, last_base);
        add_digit(a, dig_sum);
    }
    big_end_string = src;
    return big_errno;
}

int
big_ulong(a, n)
bignum *a;
ulong *n;
{
    ulong old_n;
    DIGIT *dp;

    if (big_errno != BIG_OK)
    {
        return FALSE;
    }
    if (a->dgs_used > sizeof(ulong) / sizeof(DIGIT))
    {
        return FALSE;
    }
    dp = a->dp + a->dgs_used - 1;
    old_n = *n = *dp--;
    while ((dp >= a->dp) && (old_n < *n))
    {
        old_n = *n;
        *n = SHIFT_DIGIT_UP(*n) + *dp--;
    }
    if (old_n >= *n)
    {
        return FALSE;
    }
    return FALSE;
}

int
big_long(a, n)
bignum *a;
long *n;
{
    long old_n;
    DIGIT *dp;

    if (a->dgs_used > sizeof(ulong) / sizeof(DIGIT))
    {
        return FALSE;
    }
    dp = a->dp + a->dgs_used - 1;
    old_n = *n = *dp--;
    while ((dp >= a->dp) && (old_n <= *n))
    {
        old_n = *n;
        *n = SHIFT_DIGIT_UP(*n) + *dp--;
    }
    if (old_n >= *n)
    {
        return FALSE;
    }
    if (a->sign == BIG_SIGN_MINUS)
    {
        *n = -*n;;
    }
    return FALSE;
}

char *
big_string(a, base)
bignum *a;
int base;
{
    ulong bit_length, str_length;
    char *dst;
    DIGIT dig_sum, dig_base, rem;
    int cnt, maxcnt;

    if (big_errno != BIG_OK)
    {
        return error_string;
    }
    big_set_big(a, &tmp_string);
                            /* Need more room than minimum here... */
    bit_length = tmp_string.dgs_used * DIGIT_BITS;
    /*    bit_length = big_bitcount(&tmp_string); */
    str_length = (ulong)(bit_length / log2tbl[base] + 4);
    if (str_length > length_last_string)
    {
        free(last_string);
        if ((last_string = malloc(str_length)) == NULL)
        {
            big_errno = BIG_MEMERR;
            return NULL;
        }
        length_last_string = str_length;
    }

    dst = last_string + length_last_string - 1;
    *dst-- = '\0';
    maxcnt = big_block[base].digCnt;
    dig_base = big_block[base].dig_base;
    while (tmp_string.dgs_used > 1)
    {
        rem = udiv_digit(&tmp_string, dig_base);
        for (cnt = 0; cnt < maxcnt; cnt++)
        {
            *dst-- = big_base_digits[(rem % base) << 1];
            rem /= base;
        }
    }
    rem = *tmp_string.dp;
    do
    {
        *dst-- = big_base_digits[(rem % base) << 1];
        rem /= base;
    } while (rem != 0);

    if (a->sign == BIG_SIGN_MINUS)
    {
        *dst = '-';
    }
    else
    {
        dst++;
    }
    return dst;
}

bigerr_t
big_negate(a, b)
bignum *a;
bignum *b;
{
    big_set_big(a, b);
    b->sign = NEGATE_SIGN(a->sign);
    return big_errno;
}

int
big_sign(a)
bignum *a;
{
    return a->sign;
}
```

```
bigerr_t
big_abs(a, b)
bignum *a;
bignum *b;
{
    big_set_big(a, b);
    if (a->sign == BIG_SIGN_MINUS)
    {
        b->sign = NEGATE_SIGN(a->sign);
    }
    return big_errno;
}

int
big_compare(a, b)
bignum *a;
bignum *b;
{
    if (a->sign == b->sign)
    {
        if (a->sign == 0)
        {
            return 0;
        }
        return
            (a->sign == BIG_SIGN_MINUS)
            ? -ucompare_digits(a, b)
            : ucompare_digits(a, b);
    }
    return b->sign - a->sign;
}

int
big_lessp(a, b)
bignum *a;
bignum *b;
{
    return big_compare(a, b) < 0;
}

int
big_leqp(a, b)
bignum *a;
bignum *b;
{
    return !(big_compare(a, b) > 0);
}

int
big_equalp(a, b)
bignum *a;
bignum *b;
{
    return big_compare(a, b) == 0;
}

int
```

```
big_geqp(a, b)
bignum *a;
bignum *b;
{
    return !(big_compare(a, b) < 0);
}

int
big_greaterp(a, b)
bignum *a;
bignum *b;
{
    return big_compare(a, b) > 0;
}

int
big_zerop(a)
bignum *a;
{
    return a->sign == BIG_SIGN_0;
}

int
big_evenp(a)
bignum *a;
{
    return ((*a->dp & 0x01) == 0);
}

int
big_oddp(a)
bignum *a;
{
    return ((*a->dp & 0x01) == 1);
}

bigerr_t
big_add(a, b, c)
bignum *a;
bignum *b;
bignum *c;
{
    int cmp;

    if (big_errno != BIG_OK)
    {
        return big_errno;
    }
    print_digits("add:\ta = ", a);
    print_digits("\tb = ", b);
    if (a->sign == b->sign)
    {
        uadd_digits(a, b, c);
        c->sign = a->sign;
    }
    else
    {
        cmp = ucompare_digits(a, b);
        if (cmp < 0)
        {
            usub_digits(b, a, c);
            if (zerop(c))
            {
                c->sign = BIG_SIGN_0;
            }
            else
            {
                c->sign = b->sign;
            }
        }
        else if (cmp > 0)
        {
            usub_digits(a, b, c);
            if (zerop(c))
            {
                c->sign = BIG_SIGN_0;
            }
            else
            {
                c->sign = a->sign;
            }
        }
        else
        {
            c->dgs_used = 1;
            *c->dp = 0;
            c->sign = BIG_SIGN_0;
        }
    }
    print_digits("\tc = ", c);
    return big_errno;
}

bigerr_t
big_sub(a, b, c)
bignum *a;
bignum *b;
bignum *c;
{
    int cmp;

    if (big_errno != BIG_OK)
    {
        return big_errno;
    }
    print_digits("sub:\ta = ", a);
    print_digits("\tb = ", b);
    if (a->sign == BIG_SIGN_0)
    {
        big_set_big(b, c);
        big_negate(c, c);
        print_digits("\tc = ", c);
        return big_errno;
    }
    if (b->sign == BIG_SIGN_0)
    {
        big_set_big(a, c);
        print_digits("\tc = ", c);
        return big_errno;
    }

    cmp = ucompare_digits(a, b);
    if (cmp <= 0)
    {
        if (a->sign != b->sign)
        {
            uadd_digits(a, b, c);
            c->sign = a->sign;
        }
        else
        {
            usub_digits(b, a, c);
            c->sign = (zerop(c) ? BIG_SIGN_0 : NEGATE_SIGN(a->sign));
        }
    }
    else if (cmp > 0)
    {
        if (a->sign != b->sign)
        {
            uadd_digits(a, b, c);
            c->sign = a->sign;
        }
        else
        {
            usub_digits(a, b, c);
            c->sign = (zerop(c) ? BIG_SIGN_0 : b->sign);
        }
    }
    else
    {
        c->dgs_used = 1;
        *c->dp = 0;
        c->sign = BIG_SIGN_0;
    }
    print_digits("\tc = ", c);
    return big_errno;
}

bigerr_t
big_mul(a, b, c)
bignum *a;
bignum *b;
bignum *c;
{
    DIGIT *dp_x, *dp_xstart, *dp_xend;
    DIGIT *dp_y, *dp_ystart, *dp_yend;
    DIGIT *dp_z, *dp_zstart, *dp_zend, *dp_zsumstart;
    ulong len_x, len_y, len_z;
    DIGIT2 res;
    DIGIT tmp_res;          /* Without use of this, gcc (v 1.39) generates */
                            /* erroneous code on a sun386 machine */
```

```
                              /* Should be removed with #ifdef's, when */
                              /* not on a sun386, but... */

    if (big_errno != BIG_OK)
    {
        return big_errno;
    }
    print_digits("mul:\ta = ", a);
    print_digits("\tb = ", b);

    if (zerop(a) || zerop(b))
    {
        c->sign = BIG_SIGN_0;
        c->dgs_used = 1;
        *c->dp = 0;
        print_digits("(a=0 || b=0)c = ", c);
        return big_errno;
    }
    if (uonep(a))
    {
        big_set_big(b, c);
        c->sign = (a->sign == b->sign) ? BIG_SIGN_PLUS : BIG_SIGN_MINUS;
        print_digits("(abs(a)=1)c = ", c);
        return big_errno;
    }
    if (uonep(b))
    {
        big_set_big(a, c);
        c->sign = (a->sign == b->sign) ? BIG_SIGN_PLUS : BIG_SIGN_MINUS;
        print_digits("(abs(b)=1)c = ", c);
        return big_errno;
    }

    if (a->dgs_used < b->dgs_used)
    {
        dp_xstart = a->dp;
        len_x = a->dgs_used;
        dp_ystart = b->dp;
        len_y = b->dgs_used;
    }
    else
    {
        dp_xstart = b->dp;
        len_x = b->dgs_used;
        dp_ystart = a->dp;
        len_y = a->dgs_used;
    }
    if ((c == a) || (c == b))
    {
        if (newsize(&tmp_mul.dp, &tmp_mul.dgs_alloc,
                    len_x + len_y, len_x + len_y + 2) != BIG_OK)
        {
            return big_errno;
        }
        dp_zsumstart = tmp_mul.dp;
        len_z = tmp_mul.dgs_alloc;
    }
    else
    {
        if (newsize(&c->dp, &c->dgs_alloc,
                        len_x + len_y, len_x + len_y + 2) != BIG_OK)
        {
            return big_errno;
        }
        dp_zsumstart = c->dp;
        len_z = c->dgs_alloc;
    }

    dp_xend = dp_xstart + len_x;
    dp_yend = dp_ystart + len_y;
    dp_zend = dp_zsumstart + len_y;

    for (dp_z = dp_zsumstart; dp_z < dp_zend; dp_z++)
    {
        *dp_z = 0;              /* Zero out rightmost digits */
    }

    dp_zstart = dp_zsumstart;

    for (dp_x = dp_xstart; dp_x < dp_xend; dp_x++)
    {
        dp_z = dp_zstart;
        tmp_res = 0;
        for (dp_y = dp_ystart; dp_y < dp_yend; dp_y++)
        {
            res = (DIGIT2)(*dp_x) * (*dp_y) + (*dp_z) + tmp_res;
            *dp_z++ = DIGIT_PART(res);
            tmp_res = SHIFT_DIGIT_DOWN(res);
        }
        *dp_z = tmp_res;
        dp_zstart++;
    }
    if (dp_zsumstart != c->dp)
    {
        tmp_digits = c->dp;
        c->dp = tmp_mul.dp;
        tmp_mul.dp = tmp_digits;

        tmp_ulong = c->dgs_alloc;
        c->dgs_alloc = tmp_mul.dgs_alloc;
        tmp_mul.dgs_alloc = tmp_ulong;
    }
    if (*dp_z == 0)
    {
        dp_z--;
    }
    c->dgs_used = dp_z - dp_zsumstart + 1;
    c->sign = a->sign * b->sign;
    print_digits("\tc = ", c);
    return big_errno;
}

bigerr_t
big_trunc(a, b, q, r)
```

```
bignum *a;
bignum *b;
bignum *q;
bignum *r;
{
    DIGIT *v_end, *q_end, *r_end, *src, *dst;
    DIGIT norm, qhat, t1, t2, t3, v1, v2, u1, u2, u3;
    DIGIT2 temp, res, carry;
    long a_l, b_l, q_l, r_l;
    ulong i, j, m, n;
    int cmp, q_eq_a, q_eq_b, r_eq_a, r_eq_b;

    if (big_errno != BIG_OK)
    {
        return big_errno;
    }
    print_digits("div:\ta = ", a);
    print_digits("\tb = ", b);
    if (zerop(b))
    {
        big_errno = BIG_DIV_ZERO;
        return big_errno;
    }

    if (q == r)
    {
        big_errno = BIG_ARGERR;
        return big_errno;
    }

    if (b->dgs_used == 1)
    {
        big_set_big(a, q);
        q->sign = ((a->sign == b->sign) ? BIG_SIGN_PLUS : BIG_SIGN_MINUS);
        *r->dp = udiv_digit(q, *b->dp);
        r->dgs_used = 1;
        r->sign = (zerop(r) ? BIG_SIGN_0 : a->sign);
        print_digits("\t3:q = ", q);
        print_digits("\t  r = ", r);
        return big_errno;
    }

    if (big_long(a, &a_l))      /* Pretend it is a signed value */
    {
        big_long(b, &b_l);      /* |a| < |b| so this will succeed */
        q_l = a_l / b_l;        /* Compute with unsigned operators */
        r_l = a_l % b_l;
        big_set_long((long)q_l, q);
        big_set_long((long)r_l, r);
        print_digits("\t4:q = ", q);
        print_digits("\t  r = ", r);
        return big_errno;
    }

    cmp = ucompare_digits(a, b); /* Unsigned compare, that is... */
    if (cmp < 0)
    {
        big_set_big(a, r);      /* r = a (don't care about big_errno here) */
        q->sign = BIG_SIGN_0;   /* q = 0 */
        *q->dp = 0;
        q->dgs_used = 1;
        print_digits("\t1:q = ", q);
        print_digits("\t  r = ", r);
        return big_errno;
    }
    else
    if (cmp == 0)
    {
        q->sign = ((a->sign == b->sign) ? BIG_SIGN_PLUS : BIG_SIGN_MINUS);
        *q->dp = 1;     /* q = 1 */
        q->dgs_used = 1;
        r->sign = BIG_SIGN_0;   /* r = 0 */
        *r->dp = 0;
        r->dgs_used = 1;
        print_digits("\t2:q = ", q);
        print_digits("\t  r = ", r);
        return big_errno;
    }

    q_eq_a = (q == a);
    q_eq_b = (q == b);
    if (q_eq_a || q_eq_b)
    {
        q = &tmp_q;
    }

    r_eq_a = (r == a);
    r_eq_b = (r == b);
    if (r_eq_a || r_eq_b)
    {
        r = &tmp_r;
    }

    if (newsize(&r->dp, &r->dgs_alloc, /* At least one more dig in r */
                a->dgs_used + 1, a->dgs_used + 2) != BIG_OK)
    {
        return big_errno;
    }
    big_set_big(a, r);
    r->dp[a->dgs_used] = 0; /* In case no overflow in mult. */

    n = b->dgs_used;
    v_end = b->dp + n - 1;
    norm = DIGIT_PART((DIGIT_BASE / ((DIGIT2)*v_end + 1)));
    if (norm != 1)
    {
        umul_digit(r, norm);
        umul_digit(b, norm);
        print_digits("r = ", r);
        print_digits("b = ", b);
    }
    m = a->dgs_used + 1 - b->dgs_used;
    r_end = r->dp + a->dgs_used;
    if (newsize(&q->dp, &q->dgs_alloc, m, m + 2) != BIG_OK)
    {
        return big_errno;
    }
    q_end = q->dp + m - 1;

    v1 = *v_end;
    v2 = *(v_end - 1);

    for (j = 0; j < m; j++)                     /* m steps through division */
    {
        u1 = *r_end;                              /*  routine */
        u2 = *(r_end - 1);
        u3 = *(r_end - 2);
        qhat = ((u1 == v1) ?
                MAX_DIGIT :
                DIGIT_PART(((DIGIT2)u1 * DIGIT_BASE + u2) / v1));
        while (1)
        {
            t3 = DIGIT_PART(temp = (DIGIT2)qhat * v2);
            t2 = DIGIT_PART(temp = SHIFT_DIGIT_DOWN(temp) + v1 * (DIGIT2)qhat);
            t1 = DIGIT_PART(SHIFT_DIGIT_DOWN(temp));
#if 0
            printf("t1 = %lu, ", (ulong)t1);
            printf("t2 = %lu, ", (ulong)t2);
            printf("t3 = %lu\n", (ulong)t3);
#endif
            if (t1 < u1) break;
            if (t1 > u1) {--qhat; continue; }
            if (t2 < u2) break;
            if (t2 > u2) { --qhat; continue; }
            if (t3 <= u3) break;
            qhat--;
        }

        /* This is a tricky one - multiply and subtract simultaneously */
        carry = 1;
        res = 0;
        src = b->dp;
        dst = r->dp + m - j - 1;
        while (src <= v_end)
        {
            res = (DIGIT2)qhat * *(src++) + SHIFT_DIGIT_DOWN(res);
            carry += (DIGIT2)(*dst) + MAX_DIGIT - DIGIT_PART(res);
            *(dst++) = DIGIT_PART(carry);
            carry = DIGIT_PART(SHIFT_DIGIT_DOWN(carry));
        }
        carry += (DIGIT2)(*dst) + MAX_DIGIT - SHIFT_DIGIT_DOWN(res);
        *dst = DIGIT_PART(carry);
        carry = DIGIT_PART(SHIFT_DIGIT_DOWN(carry));

        if (carry == 0)
        {
            qhat--;
            src = b->dp;
            dst = r->dp + m - j - 1;
            while (dst <= r_end)
            {
                carry = (DIGIT2)(*dst) + *src++ + SHIFT_DIGIT_DOWN(carry);
                *dst++ = DIGIT_PART(carry);
            }
            *dst = 0;
        }
        *(q_end - j) = DIGIT_PART(qhat);
        r_end--;
    }
    r->sign = a->sign;
    i = r->dgs_used;
    while ((*r_end == 0) && (r_end > r->dp))
    {
        r_end--;
    }
    if (r_end == r->dp)
    {
        r->dgs_used = 1;
        r->sign = BIG_SIGN_0;
    }
    else
    {
        r->dgs_used = r_end - r->dp + 1;
        r->sign = a->sign;
    }
    if (norm != 1)
    {
        udiv_digit(b, norm);
        udiv_digit(r, norm);
    }
    while ((*q_end == 0) && (q_end > q->dp)) /* This is not needed!(?) */
    {
        q_end--;
    }
#if 0
    i = m - 1;
    while ((i > 0) && (q->dp[i--] == 0))
    {
        /* Loop through all zeroes */
    }
#endif
    q->dgs_used = q_end - q->dp + 1;
    q->sign =  ((a->sign == b->sign) ? BIG_SIGN_PLUS : BIG_SIGN_MINUS);

    if (q_eq_a)
    {
        big_set_big(q, a);
    }
    else
    if (q_eq_b)
    {
        big_set_big(q, b);
    }

    if (r_eq_b)
    {
        big_set_big(r, b);
    }
    else
    if (r_eq_a)
    {
        big_set_big(r, a);
    }

    print_digits("\t5:q = ", q);
    print_digits("\t  r = ", r);
    return big_errno;
}

bigerr_t
big_floor(a, b, q, r)
bignum *a;
bignum *b;
bignum *q;
bignum *r;
{
    int b_eq_qr, sign_eq;

    if (b_eq_qr = ((b == q) || (b == r)))
    {
        big_set_big(b, &tmp_mul);
    }
    sign_eq = a->sign == b->sign;
    big_trunc(a, b, q, r);
    if (sign_eq)
    {
        return big_errno;
    }
    if (!zerop(r))
    {
        if (b_eq_qr)
        {
            big_add(r, &tmp_mul, r);
        }
        else
        {
            big_add(r, b, r);
        }
        print_digits("big_one = ", &big_one);
        big_sub(q, &big_one, q);
    }
    return big_errno;
}

bigerr_t
big_ceil(a, b, q, r)
bignum *a;
bignum *b;
bignum *q;
bignum *r;
{
    int b_eq_qr, sign_diff;

    if (b_eq_qr = ((b == q) || (b == r)))
    {
        big_set_big(b, &tmp_mul);
    }
    sign_diff = a->sign != b->sign;
    big_trunc(a, b, q, r);
    if (sign_diff)
    {
        return big_errno;
    }
    if (!zerop(r))
    {
        if (b_eq_qr)
        {
            big_sub(r, &tmp_mul, r);
        }
        else
        {
            big_sub(r, b, r);
        }
        big_add(q, &big_one, q);
    }
    return big_errno;
}

/* This one doesn't work to 100%.  I was a little braindamaged when I wrote
 * this, but I'll eventually fix it.   Zzzzzzz.
 */
bigerr_t
big_round(a, b, q, r)
bignum *a;
bignum *b;
bignum *q;
bignum *r;
{
    int b_eq_qr, b_neg_p, a_sgn_neq_b_sgn;

    if (b_eq_qr = ((b == q) || (b == r)))
    {
        big_set_big(b, &tmp_round);
    }
    b_neg_p = b->sign == BIG_SIGN_MINUS;
    a_sgn_neq_b_sgn = a->sign != b->sign;
    big_trunc(a, b, q, r);

    big_set_big(r, &tmp_add);
    umul_digit(&tmp_add, 2);
    if (ucompare_digits(&tmp_add, b) > 0) /* |2 * r| > |b| */
    {
        if (q->sign == BIG_SIGN_0)
        {
            if (a_sgn_neq_b_sgn)
            {
                big_sub(q, &big_one, q);
            }
            else
            {
                big_add(q, &big_one, q);
            }
        }
        else
        {
            if (q->sign == BIG_SIGN_MINUS)
            {
                big_sub(q, &big_one, q);
            }
            else
            {
                big_add(q, &big_one, q);
            }
        }
        if (b_eq_qr)
        {
            if (q->sign == BIG_SIGN_PLUS)
            {
                big_sub(r, &tmp_round, r);
            }
            else
            {
                big_add(r, &tmp_round, r);
            }
        }
        else
        {
            if (q->sign == BIG_SIGN_PLUS)
            {
                big_sub(r, b, r);
            }
            else
            {
                big_add(r, b, r);
            }
        }
    }
    return big_errno;
}

bigerr_t
big_random(a, b)
bignum *a;
bignum *b;
{
    unsigned long i;
    int a_sgn = a->sign;

    if (big_errno != BIG_OK)
    {
        return big_errno;
    }
    if (zerop(a))               /* a = 0 -> big_random => 0 (special case) */
    {
        *b->dp = 0;
        b->dgs_used = 1;
        b->sign = a_sgn;
        return big_errno;
    }
    if (newsize(&tmp_rand.dp, &tmp_rand.dgs_alloc,
                a->dgs_used + 1, a->dgs_used + 1) != BIG_OK)
    {
        return big_errno;
    }
    for (i = 0; i <= a->dgs_used; i++)
    {
        tmp_rand.dp[i] = rand();
    }
    while (tmp_rand.dp[a->dgs_used] == 0) /* Make sure high digit is non-0 */
    {
        tmp_rand.dp[a->dgs_used] = rand();
    }
    tmp_rand.dgs_used = a->dgs_used + 1;
    tmp_rand.sign = BIG_SIGN_PLUS;
    a->sign = BIG_SIGN_PLUS;
    big_trunc(&tmp_rand, a, &tmp_q, b); /* Dangerous to use tmp_q here... */
    a->sign = a_sgn;
    b->sign = zerop(&tmp_rand) ? BIG_SIGN_0 : a_sgn;
    return big_errno;
}

/* ------------------------------------------------------------------------
 * External functions that do not need to know anything about the internal
 * representation of a bignum.
 */

int
big_expt(a, z, x)
bignum *a;
unsigned long z;
bignum *x;
{
    bignum b;

    big_create(&b);
    big_set_big(a, &b);
    big_set_long((long)1, x);

    while ((z != 0) && (big_errno == BIG_OK))
    {
        while ((z & 0x01) == 0)
        {
            z >>= 1;
            big_mul(&b, &b, &b);
        }
        z -= 1;
        big_mul(x, &b, x);
    }

    big_destroy(&b);
    return big_errno;
}

int
big_exptmod(a_in, z_in, n, x)
```

```
bignum *a_in;
bignum *z_in;
bignum *n;
bignum *x;
{
    bignum a, z, b0, b1, b2, dummy;

    big_create(&a);
    big_create(&z);
    big_create(&b0);
    big_create(&b1);
    big_create(&b2);
    big_create(&dummy);

    big_set_big(a_in, &a);
    big_set_big(z_in, &z);
    big_set_long((long)1, x);
    big_set_long((long)0, &b0);
    big_set_long((long)1, &b1);
    big_set_long((long)2, &b2);

    /* No foolproof testing on big_errno - it really ought to be done */
    while ((big_compare(&z, &b0) != 0) && (big_errno == BIG_OK))
    {
        while (big_evenp(&z) && (big_errno == BIG_OK))
        {
            big_trunc(&z, &b2, &z, &dummy);
            big_mul(&a, &a, &a);
            big_trunc(&a, n, &dummy, &a);
        }
        big_sub(&z, &b1, &z);
        big_mul(x, &a, x);
        big_trunc(x, n, &dummy, x);
    }

    big_destroy(&dummy);
    big_destroy(&b2);
    big_destroy(&b1);
    big_destroy(&b0);
    big_destroy(&z);
    big_destroy(&a);

    return big_errno;
}

int
big_gcd(a, b, g)
bignum *a;
bignum *b;
bignum *g;
{
    bignum a1, b1, tmp;

    if (big_zerop(b))
    {
        big_abs(a, g);
        return big_errno;
    }

    big_create(&a1);
    big_create(&b1);
    big_create(&tmp);

    big_abs(a, &a1);
    big_abs(b, &b1);

    while (!big_zerop(&b1) && (big_errno == BIG_OK))
    {
        big_floor(&a1, &b1, &tmp, &a1);
        if (big_zerop(&a1))
        {
            break;
        }
        big_floor(&b1, &a1, &tmp, &b1);
    }

    if (big_zerop(&a1))
    {
        big_set_big(&b1, g);
    }
    else
    {
        big_set_big(&a1, g);
    }

    big_destroy(&tmp);
    big_destroy(&b1);
    big_destroy(&a1);

    return big_errno;
}
```