GMP and pari library programming

Both GMP (Gnu Multi-Precision library) and Pari's library are powerful tools for C programming. Generally, GMP is not as featureful, but it sits very close to the metal. Pari gives you much wider range of basic types and functions on those types.

GMP programming

GMP has three basic types: floating point, integers, and rationals.

Functions are also divided by the same three classes.

GMP programming

The types are identified by the following naming convention:

```
mpz_t    #  type for integers
mpz_*    #  names for integer functions

mpf_t    #  type for floats
mpf_*    #  names for floating point functions

mpq_t    #  type for rationals
mpq_*    #  names for rational functions
```

Writing a GMP program

Writing a C program with GMP is easy if a bit tedious. First, you need to pull in the headers:

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>
```

Writing a GMP program

Next you declare variables:

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>

int main()
{
   mpz_t x, y;    // types are simple to use
}
```

Writing a GMP program

Now you **must** initialize any variables before use:

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>

int main()
{
   mpz_t x, y;

   mpz_init(x);   // critical, otherwise errors are unpredictable
   mpz_init(y);   //
}
```

Writing a GMP program

Compiling and linking is simple:

```
gcc -o prog prog.c -lgmp
```

Writing a GMP program

When creating a subroutine, make sure you clear the variables after you finish using them (despite the static declaration, that's just a pointer to the actual dynamically allocated memory for the variable):

Writing a GMP program

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>

void func()
{
   mpz_t x;
   mpf_t y;

   mpz_init(x);
   mpf_init(y);

   mpz_clear(x);      // otherwise you have a memory leak!
   mpf_clear(y);      //
   return;
}
```

Simple example program

```
#include <unistd.h>
#include <gmp.h>

char *answers[3]  =  { "composite", "probably prime", "prime" } ;

int main(int argc, char *argv[])
{
   int result;
   mpz_t n;
   mpz_init(n);
   mpz_set_str(n,argv[1],10);  // set the value of n from a string in base 10

   result = mpz_probab_prime_p(n,20);  // do a primality test with 20 repetitions
   gmp_printf("%Zd is %s\n",n,answers[result]);
}
```

Integer functions: assignment

```
void mpz_set (mpz_t result, mpz_t op)   # z = z
void mpz_set_ui (mpz_t result, unsigned long int op) # z = uint
void mpz_set_si (mpz_t result, signed long int op) # z = signed int
void mpz_set_d (mpz_t result, double op) # z = double
void mpz_set_q (mpz_t result, mpq_t op) # z = q  (via truncation)
void mpz_set_f (mpz_t result, mpf_t op) # z = f  (via truncation)

int mpz_set_str (mpz_t result, char *str, int base)
   # return 0 means string was completely a number
   # in the indicated base, -1 means that it wasn't

void mpz_swap (mpz_t result1, mpz_t result2) # swap two values
```

Integer functions: arithmetic

```
void mpz_add (mpz_t sum, mpz_t op1, mpz_t op2) # z = z + z
void mpz_add_ui (mpz_t sum, mpz_t op1, unsigned long int op2) # z = z + uint
void mpz_sub (mpz_t diff, mpz_t op1, mpz_t op2) # z = z - z
void mpz_sub_ui (mpz_t diff, mpz_t op1, unsigned long int op2) # z = z - unit
void mpz_ui_sub (mpz_t diff, unsigned long int op1, mpz_t op2) # z = uint - z
void mpz_mul (mpz_t result, mpz_t op1, mpz_t op2) # z = z * z
void mpz_mul_si (mpz_t result, mpz_t op1, long int op2) # z = z * signed int
void mpz_mul_ui (mpz_t result, mpz_t op1, unsigned long int op2) # z = z * uint
void mpz_neg (mpz_t result, mpz_t op) # z = -z
void mpz_abs (mpz_t result, mpz_t op) # z = |z|
```

Rational number functions: arithmetic

```
void mpq_add (mpq_t sum, mpq_t addend1, mpq_t addend2) # q = q + q
void mpq_sub (mpq_t difference, mpq_t minuend, mpq_t subtrahend) # q = q - q
void mpq_mul (mpq_t product, mpq_t multiplier, mpq_t multiplicand) # q = q * q
void mpq_div (mpq_t quotient, mpq_t dividend, mpq_t divisor) # q = q / q
void mpq_neg (mpq_t negation, mpq_t operand) # q = - q
void mpq_abs (mpq_t result, mpq_t op) # q = |q|
void mpq_inv (mpq_t inverted_number, mpq_t number) # q = 1 / q
```

Floating point functions: arithmetic

```
void mpf_add (mpf_t sum, mpf_t op1, mpf_t op2) # f = f + f
void mpf_add_ui (mpf_t sum, mpf_t op1, unsigned long int op2) # f = f + uint
void mpf_sub (mpf_t diff, mpf_t op1, mpf_t op2) # f = f - f
void mpf_ui_sub (mpf_t diff, unsigned long int op1, mpf_t op2) # f = uint - f
void mpf_sub_ui (mpf_t diff, mpf_t op1, unsigned long int op2) # f = f - uint
void mpf_mul (mpf_t result, mpf_t op1, mpf_t op2) # f = f * f
void mpf_mul_ui (mpf_t result, mpf_t op1, unsigned long int op2) # f = f *uint
void mpf_div (mpf_t result, mpf_t op1, mpf_t op2) # f = f / f
void mpf_ui_div (mpf_t result, unsigned long int op1, mpf_t op2) # f = uint / f
void mpf_div_ui (mpf_t result, mpf_t op1, unsigned long int op2) # f = f / uint
void mpf_sqrt (mpf_t root, mpf_t op) # f = sqrt(f)
void mpf_sqrt_ui (mpf_t root, unsigned long int op) # f = sqrt(uint)
void mpf_pow_ui (mpf_t result, mpf_t op1, unsigned long int op2) # f = f ^ f
void mpf_neg (mpf_t negation, mpf_t op) # f = - f
void mpf_abs (mpf_t result, mpf_t op) # f = |f|
```

Comparison functions

```
int mpz_cmp (mpz_t op1, mpz_t op2) # returns negative if op1 < op2,
                                   # 0 if op1 == op2
                                   # positive if op1 > op2
int mpz_cmp_ui (mpz_t op1, unsigned long int op2) # same for uint
int mpf_cmp (mpf_t op1, mpf_t op2) # same for floats
int mpf_cmp_ui (mpf_t op1, unsigned long int op2) # same
int mpq_cmp (mpq_t op1, mpq_t op2) # same for rationals
int mpq_cmp_ui (mpq_t op1, unsigned long int num2, unsigned long int den2
```

Other useful functions

```
int mpz_probab_prime_p (mpz_t N, int repetitions)
          # returns 0 if N definitely composite
          # 1 if probably prime
          # 2 if definitely prime
void mpz_nextprime (mpz_t result, mpz_t N)
          # result is next prime greater than N
void mpz_gcd (mpz_t result, mpz_t op1, mpz_t op2)
          # result is GCD(op1,op2)
int mpz_jacobi (mpz_t a, mpz_t b)
          # jacobi (a/b)    Calculate the Jacobi symbol (a/b). This is defined only for
int mpz_legendre (mpz_t a, mpz_t p)
          # legendre (a/p)
```

Other useful functions

```
unsigned long int mpz_remove (mpz_t result, mpz_t op, mpz_t f)
          # result = divide out all of a given factor f from op
void mpz_fac_ui (mpz_t result, unsigned long int op)
          # result = op!
void mpz_bin_ui (mpz_t rop, mpz_t n, unsigned long int k)
          # computes the binomial coefficient n over k
void mpz_fib_ui (mpz_t fn, unsigned long int n)
          # computes the nth fibonacci number
```