GNU MP

GNU 多倍長精度演算ライブラリ
Edition 6.1.2
16 December 2016

Torbjörn Granlund と GMP 開発チーム

このマニュアルは GNU 多倍長演算ライブラリ(GNU MP) Version 6.1.2のインストール方法と使用方法について記述したものです。

目次

GNU MP の著作権

このライブラリは**フリー***(free)*なものです。「フリー」とは使用や配布の権利が，自由な基盤の上において万人に存在する，ということです。このライブラリはパブリックドメインなものではありません。つまり，著作権によって守られている著作物であり，再配布に関しては制限がある，ということを意味します。しかし，良き協力者たる市民が望むところにおいては，再配布を妨げられることはありません。貴方が本ライブラリそのものや改変物などをシェアする行為を妨げることは認められていません。

我々としては特に，以下のことを貴方に確認してほしいと考えています。まず，貴方には本ライブラリを放出する権利があります。そして，入手を望むソースコードその他のものを受け取る権利があります。本ライブラリを改変したり，この一部を利用して他のフリーなプログラムを作る権利があります。最後に，貴方はこれらのことが可能であるということを知る権利があります。

万人がこれらの権利を持っていることを知らしめるため，我々はこれらの権利を剥奪するような行為を厳禁します。例えば，GNU MP ライブラリの配布に際しては，配布者が所持する権利一切を，ライブラリの受取人にも付与しなくてはなりません。また，ソースコードも受け取る権利があることを周知しなくてはいけません。更に，これらの諸権利を万人に告知しなくてはいけません。

加えて，配布者を守るために，この GNU MP ライブラリの使用に際しては一切の保証はない，ということも万人に周知しておかなければなりません。本ライブラリを改良し，それを配布したとしても，それは我々が作成したものではなく，それ故に混入した問題に関しては我々の世評を害するものではない，ということも知っておいて欲しいと思います。

正確を期しておくと，GNU MP ライブラリは二重ライセンス方式で配布されており，一つは GNU Lesser General Public License version 3 (`COPYING.LESSERv3`参照), もう一つは GNU General Public License version 2 (`COPYINGv2`参照)になります。どちらを選ぶかはライブラリの利用者が決めることで，これらのライセンスに後から加わった条項を付加することもできます。(二重ライセンス方式を取っているのは，GPL Version 2 ライセンスのプログラムを含んでいるライブラリも使用可能にする必要があるからです。歴史的な事情等があり，GPL の以降のバージョンでは利用できないケースが存在しているのです。)

本ライブラリ全体を含まず，一部だけ使っている，デモ用，GMP テストスイート等のプログラムは GNU General Public License version 3 (`COPYINGv3`参照)以降のライセンスが適用されています。

1 GNU MP について

GNU MP (GMP)はさまざまな環境で動作する C ライブラリで，整数，有理数，浮動小数点数の任意精度演算をサポートします。C が直接サポートする基本的なデータ型よりも高精度な演算を必要とするアプリケーションソフトウェアのために，最高速の計算ルーチンを提供することを目的としています。

大体のアプリケーションはせいぜい数百ビット程度の精度しか使わないのが普通ですが，何千，何百万ビットの精度を必要とするアプリケーションも，数は少ないにしろ存在しています。GMP は，そのどちらのアプリケーションに対しても適した計算速度を提供できるよう，オーバーヘッドを最小化する設計がなされています。

GMP の高速さは，これら 3 つの基本演算型と，定義されたこれらの型に割り当てられたワード全てをフルに使用し，洗練されたアルゴリズムを選び抜き，様々な CPU に対して注意深く最適化されたアセンブラプログラムを用いて使用頻度の高い内部ループを記述することで達成されたものです（その反面，コードの記述は分かりづらいです）。

現在は下記の CPU 用のアセンブラコードが用意されています。ARM Cortex-A9, Cortex-A15, 汎用 ARM, DEC Alpha 21064, 21164, 21264, AMD K8 と K10 (Athlon64, Phenom, Opteron 等々，いろんなブランド名で販売展開されています) Bulldozer と Bobcat, Intel Pentium, Pentium Pro/II/III, Pentium 4, Core2, Nehalem, Sandy bridge, Haswell, 汎用 x86, Intel IA-64, Motorola/IBM PowerPC 32 と 64 (POWER970, POWER5, POWER6, POWER7), MIPS 32-bit と 64-bit, SPARC 32-bit と全ての UltraSPARC 向けサポート付きの SPARC 64-bit。廃止されてしまった CPU 用のアセンブラコードもそのまま残っています。

GMP の最新情報は下記の GMP Web ページで確認して下さい。

`https://gmplib.org/`

GMP ライブラリの最新バージョンは下記のところからダウンロードできます。

`https://ftp.gnu.org/gnu/gmp/`

世界中にある‘`ftp.gnu.org`’のミラーサイトから，ご自身の近くにあるものを選んで使って下さい。ミラーサイト一覧は`https://www.gnu.org/order/ftp.html`に載っています。

GMP の公開メーリングリストは 3 つあります。一つはリリース情報アナウンス用，もう一つは GMP に関する質問や議論を行うためのもの，三つめはバグ報告用のものです。詳細情報は下記をご覧下さい。

`https://gmplib.org/mailman/listinfo/`.

バグ報告の正式な送り先は`gmp-bugs@gmplib.org`です。バグ報告についての情報はChapter 4 [Reporting Bugs], p. 31をご覧下さい。

1.1 このマニュアルの使い方

まずはChapter 3 [GMP Basics], p. 18を読んで下さい。ライブラリのインストール方法を知りたい時にはChapter 2 [Installing GMP], p. 3をご覧下さい。複数 ABI に対応したシステムを使う場合は，コンパイラオプションの指定が必要になるので，Section 2.2 [ABI and ISA], p. 8を参照して下さい。

後々の参考になると思いますので，これ以降の内容も一通り眺めておくことをお勧めします。

2 GMP のコンパイルとインストール

GMP は設定構築のために autoconf/automake/libtool を使っています。UNIX および互換 OS 上では次のコマンドを使うことで基本的な設定構築を行うことができます。

```
./configure
make
```

make の設定自己チェックは次のようにして行います。

```
make check
```

インストールは次のようにして行います（デフォルト設定では/usr/localにインストール)。

```
make install
```

何か問題が起こった時にはgmp-bugs@gmplib.orgに報告して下さい。その際にはChapter 4 [Reporting Bugs], p. 31を参照し，有用なバグ報告になるように，必要な事柄をきちんと書いて下さい。

2.1 ビルド時のオプション

GMP では通常の autoconf の設定オプションをすべて利用することができるようになっており，‘./configure --help’でその概要を読むことができます。INSTALL.autoconfファイルにも一般的なインストールのための情報が記述されています。

ツール類
: ‘configure’スクリプトは沢山の UNIX ツールを使用します。Section 2.4 [Notes for Particular Systems], p. 13を参照してください。UNIX ではない OS 上での設定方法を記述してあります。

  ‘configure’スクリプトを使わなくても，全てのコードが揃っていれば，ビルドすることは可能ではあります。但し，ツールの代わりに全部自力で設定を行わなければなりません。

ビルドするディレクトリ
: ビルドするディレクトリを別のところに設定する際には，cdコマンドでそこへ移動し，プレフィックスを，GMP のソースが置いてあるディレクトリへのパスをくっつけて configure を実行します。例えば次のように行います。

  ```
  cd /my/build/dir
  /my/sources/gmp-6.1.2/configure
  ```

  この場合，‘make’プログラムがコンパイルに必要な全ての情報(VPATH)を保持しているわけではありません。特に，SunOS やクソトロい Solaris のmakeコマンドはバグのおかげで上記のように別のディレクトリでのビルドができません。やりたければ GNU makeを使って下さい。

--prefix と--exec-prefix
: --prefixオプションを使うと，GMP をインストールするディレクトリを指定できます。デフォルトのディレクトリは‘/usr/local’です。

  --exec-prefixオプションを指定すると，libgmp.aのように CPU アーキテクチャごとに生成されるファイルを異なるディレクトリに設置できます。このオプションではドキュメントのようにアーキテクチャに依存しないファイルも指定ディレクトリに設置できますが，分離することも可能です。gmp.hはlibgmpを生成する際に作られますので，アーキテクチャ依存のファイルになります。従ってビルド時には$prefix/includeと$exec_prefix/includeの両方をコンパイラに周知しておく必要があります。

`--disable-shared`, `--disable-static`
: デフォルトでは静的ライブラリも動的ライブラリも両方ビルドされますが，片方だけビルドすることもできます。動的ライブラリは実行ファイルを小さくしたい時に有効で，動作しているプロセス間で同じ GMP のコードを共有します。但し，CPU のタイプによっては，関数呼び出しに要する時間のため，若干低速になります。

ネイティブコンパイル, `--build=CPU-VENDOR-OS`
: 通常のネイティブコンパイルを行うには，対象システムを‘`--build`’で指定できます。デフォルトでは‘`./configure`’時において，‘`./config.guess`’の実行結果を利用します。‘`./config.guess`’で正確な CPU タイプを知ることもできますが，コンパイル時にこのオプションを使って明示的に CPU タイプを指定することもできます。例えば下記のように指定します。

```
./configure --build=ultrasparc-sun-solaris2.7
```

どんなケースでも‘`OS`’の部分は重要で，これによって libtool が動的ライブラリを生成する方法を指定できます。‘`OS`’が不明の場合は‘`./config.guess`’を動作することで簡単に判明します。

クロスコンパイル, `--host=CPU-VENDOR-OS`
: クロスコンパイルを行う時には，コンパイルに使用したいシステムを‘`--build`’に指定し，ライブラリを動かしたいシステムを‘`--host`’に指定します。下記の例では，FreeBSD Athlon システムを利用してコンパイルし，GNU/Linux m68k 用のバイナリを生成しています。

```
./configure --build=athlon-pc-freebsd3.5 --host=m68k-mac-linux-gnu
```

コンパイラツールは最初にホストシステムのタイプを見に行きます。例えば`m68k-mac-linux-gnu-ranlib`が指定された場合，標準の`ranlib`が使用されます。これによって，ネイティブツールと共存しているクロスコンパイル用のツール群が指定できるようになります。プレフィックスは‘`--host`’に指定でき，これは‘`m68k-linux`’のようにエイリアスが利用できます。但し，ツール類はここで指定するのではなく，クロスコンパイル用の`cc`を含むパス(`PATH`)が通っているだけで十分です。

同じファミリーに属する異なる CPU に対してコンパイルを行うには，クロスコンパイルとして行うこともできますが，ネイティブコンパイラにオプションをつけるだけで済ますことも可能です。どんなケースでも，‘`./configure`’はビルドするシステムの影響は受けずに実行できますので，ビルドを実行するシステム上では動作しない，新しい CPU 向けのバイナリの生成も可能です。

いかなる場合でも，コンパイラは標準 C の`main`関数を含むソースから，実行ファイル（の形式はともかく）を生成できなければなりません。オブジェクトファイルは`libgmp`の構成物になるだけですが，‘`./configure`’コマンドは，例えば実行対象のホストシステムで動作する関数を確定するために，リンクテストを使うことになります。

現状ではクロスコンパイルの際に‘`--build`’オプションの指定がないと警告が出ます。これは，`PATH`が通ったディレクトリにクロスコンパイル用の`cc`しかない場合は正しくビルドシステムのタイプを認識できない可能性があるからです。

‘`--target`’オプションは，GMP のクロスコンパイル用としては適切ではありません。これはビルド用のコンパイラツールを指定するだけのもので，‘`--host`’によって動作する環境を，‘`--target`’ によってコード生成の対象となるアーキテクチャを指定しています。GMP のような通常のプログラムやライブラリは，‘`--host`’部分だけ指定すれば事足ります(以前の GMP は‘`--target`’を不正確に使っていました)。

CPU の種別
: 一般的に，ライブラリをできるだけ高速に動作させたい時には，GMP の設定を，使用している正確な CPU 種別に合わせる必要があります。とはいえ，その CPU ファミリーの古いタイプで動かすこともあるのか，他の CPU ファミリーでは遅くなるこ

ともあり得るのか，古い CPU か新しい CPU か，といったことを示すことになりますが，最良の方法は，正しいマシンタイプに対して GMP をビルドすることです。

下記に挙げる CPU は特別なサポートがなされています。どんなコードでどんなコンパイラオプションが使えるのかは，`configure.ac`で確認して下さい。

- Alpha: ‘`alpha`’, ‘`alphaev5`’, ‘`alphaev56`’, ‘`alphapca56`’, ‘`alphapca57`’, ‘`alphaev6`’, ‘`alphaev67`’, ‘`alphaev68`’ ‘`alphaev7`’
- Cray: ‘`c90`’, ‘`j90`’, ‘`t90`’, ‘`sv1`’
- HPPA: ‘`hppa1.0`’, ‘`hppa1.1`’, ‘`hppa2.0`’, ‘`hppa2.0n`’, ‘`hppa2.0w`’, ‘`hppa64`’
- IA-64: ‘`ia64`’, ‘`itanium`’, ‘`itanium2`’
- MIPS: ‘`mips`’, ‘`mips3`’, ‘`mips64`’
- Motorola: ‘`m68k`’, ‘`m68000`’, ‘`m68010`’, ‘`m68020`’, ‘`m68030`’, ‘`m68040`’, ‘`m68060`’, ‘`m68302`’, ‘`m68360`’, ‘`m88k`’, ‘`m88110`’
- POWER: ‘`power`’, ‘`power1`’, ‘`power2`’, ‘`power2sc`’
- PowerPC: ‘`powerpc`’, ‘`powerpc64`’, ‘`powerpc401`’, ‘`powerpc403`’, ‘`powerpc405`’, ‘`powerpc505`’, ‘`powerpc601`’, ‘`powerpc602`’, ‘`powerpc603`’, ‘`powerpc603e`’, ‘`powerpc604`’, ‘`powerpc604e`’, ‘`powerpc620`’, ‘`powerpc630`’, ‘`powerpc740`’, ‘`powerpc7400`’, ‘`powerpc7450`’, ‘`powerpc750`’, ‘`powerpc801`’, ‘`powerpc821`’, ‘`powerpc823`’, ‘`powerpc860`’, ‘`powerpc970`’
- SPARC: ‘`sparc`’, ‘`sparcv8`’, ‘`microsparc`’, ‘`supersparc`’, ‘`sparcv9`’, ‘`ultrasparc`’, ‘`ultrasparc2`’, ‘`ultrasparc2i`’, ‘`ultrasparc3`’, ‘`sparc64`’
- x86 family: ‘`i386`’, ‘`i486`’, ‘`i586`’, ‘`pentium`’, ‘`pentiummmx`’, ‘`pentiumpro`’, ‘`pentium2`’, ‘`pentium3`’, ‘`pentium4`’, ‘`k6`’, ‘`k62`’, ‘`k63`’, ‘`athlon`’, ‘`amd64`’, ‘`viac3`’, ‘`viac32`’
- Other: ‘`arm`’, ‘`sh`’, ‘`sh2`’, ‘`vax`’,

上記のリストに上がっていない CPU については汎用 C コードが適用されます。

汎用 C コードのビルド

アセンブラコードの利用に問題が発生したり，利用したくない時には`--disable-assembly`を指定し，汎用 C コードを使用してビルドして下さい。

汎用 C コードを使うと計算速度がガクッと遅くなってしまいますが，可搬性は上がりますし，多少問題含みの環境でも動かすことはできるようになります。

ファットバイナリ, `--enable-fat`

`--enable-fat`オプションを使うと x86 系のシステムでは“ファットバイナリ”ビルドができるようになります。これによって，CPU に対して最適化された基盤関数群が利用できるようになります。つまり，全ての x86 CPU 用のコードを持ちつつ，パフォーマンスの最適化が適宜できるようになる訳です（このオプションは将来的に他のアーキテクチャに対しても使えるようになる予定です）。

`ABI` ある種のシステム上では，GMP は複数の ABI（Application Binary Interface）をサポートしています。これはデータ型のサイズや関数呼び出しに関係してきます。デフォルトでは GMP は利用できる ABI のうち最適なものを選択しますが，下記のように特定の ABI を指定することもできます。

```
./configure --host=mips64-sgi-irix6 ABI=n32
```

使用できるオプション指定や CPU，アプリケーション側でできること等については，Section 2.2 [ABI and ISA], p. 8をご覧下さい。

`CC`, `CFLAGS`

デフォルトの C コンパイラは，`gcc`をはじめ，通常使われることの多いコンパイラの中から選択して指定します。通常は‘`CC=whatever`’というオプションで‘`./configure`’に渡され，デフォルトでは選ばれないものも使用可能になります。

デフォルトのコンパイラのフラグは，CPU や C コンパイラによって変わってきます。通常は‘CFLAGS="-whatever"’というオプションで‘./configure’に渡され，デフォルトとは異なるフラグや，GMP が感知しえない適切なフラグを指定することができるようになります。

‘CC’ や‘CFLAGS’といったオプションを使うと，‘./configure’実行中にその指定値が表示され，生成されたMakefileの中に書き込まれます。オプションのの値を変えたり何か追加したりしたい時にはまずこのデフォルト値を確認するのがいいでしょう。

‘CC’ や‘CFLAGS’といったオプション値を，複数の ABI をサポートするシステムで指定したい時には，明示的に‘ABI=whatever’というオプションをしてしておくことで，GMP が間違ったフラグやアセンブラコードを各 ABI に指定することを防ぎます。

‘CC’だけを指定した場合は，そのコンパイラに対してのデフォルト値を‘CFLAGS’に与えます（GMP がその値を知っていた場合）。例えば‘CC=gcc’とすると，強制的に GCC が C コンパイラとして指定され，そのデフォルト値（とデフォルトの ABI）が付加されます。

CPPFLAGS
: プリプロセッサが必要とするフラグ，例えば‘-D’による定義や，‘-I’によるインクルードファイルのディレクトリ指定は，‘CFLAGS’ではなく，‘CPPFLAGS’で指定して下さい。‘CPPFLAGS’ と‘CFLAGS’の両方を指定してコンパイルすることもできますが，プリプロセッサは前者しか使いません。プリプロセッサとコンパイラの指定可能なフラグには相違があるのが普通だからです。いくつかの configure 時のテストでも，プリプロセッサは分離して実行されます。

CC_FOR_BUILD
: ビルド時に利用されるプログラムはコンパイルされ，ホスト指定のデータテーブルを生成するために実行されます。‘CC_FOR_BUILD’オプションはこの目的のために使用されるコンパイラの指定ができます。どんな ABI やモードでも特段指定の必要がないオプションで，実行ファイルを構築するためだけのオプションです。デフォルトでは，指定済みの‘CC’，探索して見つかった動作可能な C コンパイラ，例えば‘cc’や‘gcc’になります。

  単純に‘cc foo.c’のようにフラグなしでコンパイルするだけなので，‘CC_FOR_BUILD’に対するフラグは利用できません。どうしてもフラグ指定をしたければ，‘CC_FOR_BUILD="cc -whatever"’のように，このオプション指定に取り込んで下さい。

C++のサポート, --enable-cxx
: GMP を C++で使えるようにするには‘--enable-cxx’オプションでコンパイルします。これにより C++コンパイラを要求するようになります。‘--enable-cxx=detect’オプションを使うと，C++コンパイラが使用可能な時のみサポートされますので便利です。C++利用のためにはlibgmpxx.laライブラリと，gmpxx.hヘッダファイルがセットで必要となります(see Section 3.1 [Headers and Libraries], p. 18)。

  libgmpxx.laを分離してあるのは，C++オブジェクトをlibgmp.laに同梱するよりも使いやすいからです。これにより，ダイナミックリンクを利用している C プログラムが C++標準ライブラリを要求したり，C++コンパイラが C プログラムのリンクを要求したりすることがなくなります。

  libgmpxx.laは内部でligmp.laを使っているため，どちらも同じ GMP バージョンのものでなくてはなりません。バージョンアップの度にルーチンの名前変更などがありますので，異なるバージョンの GMP を使うと，挙動がおかしくなるというより，シンボル未定義が頻発することが多くなると思います。

  一般にlibgmpxx.laはビルド時に C++コンパイラを必要としますので，違うコンパイラで GMP を使うと名前修飾(name mangling)やランタイムサポートの違いで必ずトラブります。

CXX, CXXFLAGS
: C++を利用する際には C++コンパイラとそのオプションを‘CXX’ と‘CXXFLAGS’に指定するのが普通のやり方です。‘CXX’のデフォルト値は望ましい C++コンパイラの候補の先頭にあるものが選ばれます。g++が使える場合はこれが筆頭になります。‘CXXFLAGS’のデフォルト値は‘CFLAGS’のうち，‘-g’を除いたものになります。例えば，g++の場合は‘-g -O2’が‘-O2’になり，他のコンパイラでは‘-g’が単なるオプションなし，となります。‘CFLAGS’を流用するのは‘gcc’と‘g++’を共に利用する際に便利だからで，コンパイラオプションが大体共通なのです。

  ちなみに，‘CXX’と‘CC’を全く同一の値にすることは望ましくありません。例えばgccの場合，foo.ccを C++プログラムであると理解はしますが，g++のように正しくリンカを実行して，C++オブジェクトファイルから実行ファイルや共有ライブラリを生成することはできません。

一時メモリ領域, --enable-alloca=<choice>
: GMP は一時メモリ領域を，下記に挙げる 3 つの方法で確保します。方法の選択は例えば‘--enable-alloca=malloc-reentrant’というオプションで指定可能です。

  - ‘alloca’ - C ライブラリ，もしくはコンパイラに内包されている関数
  - ‘malloc-reentrant’ - 再入可能な形でヒープに確保
  - ‘malloc-notreentrant’ - グローバル変数として使用可能なヒープ領域

  オプション指定を簡便にするため，下記のような簡易指定も可能です。例えば‘--disable-alloca’は‘no’と指定したものと同じ意味になります。

  - ‘yes’ - ‘alloca’と同じ
  - ‘no’ - ‘malloc-reentrant’と同じ
  - ‘reentrant’ - 使用可能であればallocaを指定し，そうでなければ‘malloc-reentrant’を指定。これがデフォルトです。
  - ‘notreentrant’ - 使用可能であればallocaを指定子，そうでなければ‘malloc-notreentrant’を指定。

  allocaは再入可能で，高速に実行できるのでお勧めです。スタック上に小さいメモリブロックを確保します。もっと大きなメモリブロックが欲しい場合は malloc-reentrant を利用して下さい。

  ‘malloc-reentrant’はその名前の通り，再入可能でスレッドセーフです。‘malloc-notreentrant’は高速だが再入可能性が不要の時に使用します。

  この二つの malloc を利用する方法は，mp_set_memory_functionsで指定したメモリ割り当て関数を使います。デフォルトではmalloc関数やその類似の関数が使われます。(See Chapter 13 [Custom Allocation], p. 91)

  追加して使えるオプションとしては‘--enable-alloca=debug’があり，メモリに関連するデバッグを行いたい時には重宝します(see Section 3.12 [Debugging], p. 25)。

FFT 乗算, --disable-fft
: デフォルトでは，乗算は Karatsuba, 3-way Toom-Cook, 高次 Toom-Cook，フェルマー FFT アルゴリズムを使って実行されます。FFT は特別デカいオペランドに対してのみ適用されるものなので，コードサイズを小さくしたいのであれば，このオプションを使って使用不可にしておくこともできます。

アサーションによるチェック, --enable-assert
: このオプションを利用してライブラリ内の整合性のチェックができるようになります。デバッグ時see Section 3.12 [Debugging], p. 25, に利用う可能です。

実行時プロファイル, --enable-profiling=prof/gprof/instrument
: プロファイリング方法はいくつかありますが see Section 3.13 [Profiling], p. 28，そのうちの一つがこのオプション指定で使えるようになります。

`MPN_PATH` mpn サブルーチンチン用のアセンブラプログラムは多様なものが提供されており，各 CPU 用のものは自動で検索されますが，このオプションを使うと mpn 用のコードのフォルダを指定できるようになります。例えば，下記は‘`sparcv8`’用の指定です。

```
MPN_PATH="sparc32/v8 sparc32 generic"
```

この例では，まず最初に v8 コードを読みに行き，次に sparc32 コード(v7 用)，最後に汎用 C コードを読みに行きます。ハードウェアに詳しいユーザが特定の目的で利用したい時には，異なるパスを指定しても結構です。通常は無用のオプションになります。

文書類 あなたが今読んでいるこのマニュアルのソースは`doc/gmp.texi`にあり，Texinfo フォーマットで記述されています。詳細は*Texinfo*を読んで下さい。

Info フォーマットの文書‘`doc/gmp.info`’はこの GMP の TAR ボールの中に入っています。通常の automake を利用して，PostScript, DVI, PDF, HTML を生成することができます（但し，TEXや Texinfo ツールが使える環境であること）。

DocBook や XML 形式の文書は Texinfo の`makeinfo`プログラムから生成できます。詳細はSection “Options for `makeinfo`” in *Texinfo*をご覧下さい。

追記的なメモについては`doc`以下に置いてあります。

2.2 ABI と ISA

ABI (Application Binary Interface の略)は機能の相互呼び出し規約のことで，どのレジスタを使うのか，どの C データ型を使うのか，ということを規定したものです。ISA (Instruction Set Architecture の略)は CPU が使用できる命令セットやレジスタを規定したものです。

ある種の 64-bit ISA CPU は 64-bit ABI と 32-bit ABI の両方が定義されており，32-bit ABI に関しては同じファミリー内の古い CPU と互換性があります。GMP はこのような二つの ABI を持つ CPU もサポートしています。実際，GMP ‘`ABI`’の内部では ABI の組み合わせも可能ですし，どの ABI を使うのかを選択する機能も有しています。例えばある種の 32-bit ABI については，32-bit 長`long`型のリムと 64-bit 長`long long`型のリムのどちらかを選択して使うことができるようになっています。

デフォルトでは，GMP は使用システムに応じて最適な ABI を選択するようになっています。そのため，自動的に高速な動作を可能にしてくれます。他のライブラリやアプリケーションの都合に合わせて GMP を明示的に構成したい時には，次のように ABI オプションを指定します。

```
./configure ABI=32
```

どんな場合でも，与えられたプログラム内で使われるすべてのオブジェクトコードが，同一の ABI に対してコンパイルされるようになっている，ということが重要です。

GMP のリムは常に`long`型として実装されています。`long long`型のリムが使用されるのは，生成された`gmp.h`内でそのように定義されている時に限られます。このような扱いはアプリケーションにとっては都合が良いのですが，`gmp.h`が多種多様になり，それを他の環境にコピペして使い回すことはできないということを意味します。特定の ABI を対象としたコンパイラに対しては同一のリム型を使うことが期待される以上，`gmp.h`がコンパイラ依存になっていまうことは止むを得ません。

現状では，特定の ABI を対象としたライブラリのインストールやヘッダファイルの生成は行っていません。複数のシステム向け GMP の生成や，特定の‘`libdir`’を使う設定をできる限り単純化したりするぐらいは可能ですが，異なる ABI に対してのビルドは個別に行い，それぞれ`./configure`や`make`を新たに生成して実行する必要がある，ということを覚えておいて下さい。

AMD64 (‘x86_64’)

AMD64 システム上では，32 ビット，64 ビットモードの両方をサポートしており，下記の ABI が利用可能です。

‘ABI=64’ 64bit ABI は 64bit リムとポインタを使用してその性能を最大限引き出しており，これがデフォルトになります。アプリケーションは特にコンパイラ用のフラグを設定する必要はありませんが，下記のオプションで指定ができます。

```
gcc  -m64
```

‘ABI=32’ 32bit ABI は普通の i386 用です。比較的低速になってしまうので，64bit 環境との相互運用性が必要ないのであれば，これを使用することはお勧めしません。

```
gcc  -m32
```

(GCC 2.95 以前では，‘-m32’オプションは利用できず，これが唯一のモードでした。)

‘ABI=x32’ x32 ABI は 64bit リムと 32bit ポインタを利用します。64bit ABI のような環境では，これで CPU の演算性能を最大限活用できるようになります。この ABI が使えない OS 環境も存在します。

```
gcc  -mx32
```

HPPA 2.0 (‘hppa2.0*’, ‘hppa64’)

‘ABI=2.0w’

2.0w ABI は 64-bit リムとポインタを使用し，HP-UX 11 以降の環境で使用可能です。アプリケーションは次のようにしてコンパイルしなければいけません。

```
gcc [built for 2.0w]
cc  +DD64
```

‘ABI=2.0n’

2.0n ABI とは，32bit の HPPA 1.0 ABI と関数呼び出し形式を意味します。但し，関数の内部では 64bit 命令も使用可能です。GMP では 64bit 長の`long long`型のリムを使用します。この ABI は hppa64 GNU/Linux と HP-UX 10 以降の環境で利用可能です。この ABI を使うにはアプリケーション生成時には次のようにコンパイルしなければなりません。

```
gcc [built for 2.0n]
cc  +DA2.0 +e
```

現在の GCC(Verison 3.2 など)では，`long long`用の 64bit 命令を生成しませんので，2.0w に対しては低速になります(GMP アセンブラコードも同様です)。

‘ABI=1.0’ HPPA 2.0 CPU は，32 ビット HPPA 1.0 ABI である全ての HPPA 1.0, 1.1 のコードを動作させることができます。この場合は特別なコンパイラオプションを付加する必要はありません。

この 3 種類の ABI は‘hppa2.0w’, ‘hppa2.0’，‘hppa64’の CPU タイプすべてで利用可能です。但し，‘hppa2.0n’の CPU は 2.0n もしくは 1.0 しか利用できません。

HP-UX 上の GCC は，HP `cc`とは違い，API として 2.0n か 2.0w を選択するためのオプションを持っておらず，どちらか片方の ABI でビルドする必要があります。GMP はビルド方法を検出し，対応する‘ABI’を選択します。。

HP-UX 環境下の IA-64 (‘ia64*-*-hpux*’, ‘itanium*-*-hpux*’)

HP-UX は IA-64 用に二つの ABI をサポートしており，GMP のパフォーマンスはどちらも同程度です。

‘ABI=32’ 32-bit ABI では，ポインタ，int型，long型は全て 32 bit 長で，GMP では 64 bit 長のlong long型をリムとして使用しています。特に何もオプション指定を行わないと，この ABI が HP C でも GCC でもデフォルトで使用されます。オプションとしてして明示したいのであれば，下記のように指定して下さい。

```
gcc  -milp32
cc   +DD32
```

‘ABI=64’ 64-bit ABI では，long型とポインタは 64 bit 長になり，GMP はこのlong型をリムとして使用します。アプリケーション生成時には下記のようにオプション指定してコンパイルして下さい。

```
gcc  -mlp64
cc   +DD64
```

他の IA-64 システム，例えば GNU/Linux は‘ABI=64’だけ指定すれば十分です。

IRIX 6 環境下の MIPS (‘mips*-*-irix[6789]’)

IRIX 6 では常に 64bit MIPS 3 以上の CPU を使用しており，ABI としては o32, n32, 64 が利用可能です。n32 か 64 を使用することをお勧めします。GMP のパフォーマンスはどちらも同程度です。デフォルトは n32 になります。

‘ABI=o32’ o32 ABI は 32bit ポインタと整数になり，64bit 命令はサポートしません。従って，GMP は n32 や 64 より遅くなりますので，GCC 2.7.2 のような古いコンパイラをサポートするためだけに存在しています。特にフラグを指定せずに新旧のコンパイラを実行する際には下記のオプションを指定して下さい。

```
gcc  -mabi=32
cc   -32
```

‘ABI=n32’ n32 ABI は 32bit ポインタと整数となりますが，long long型が使えるので 64bit リムになります。コンパイル時は次のようにオプション指定を行います。

```
gcc  -mabi=n32
cc   -n32
```

‘ABI=64’ 64bit ABI は 64bit 長のポインタと整数になります。コンパイル時のオプションは下記のようになります。

```
gcc  -mabi=64
cc   -64
```

カーネル version 2.2 の MIPS GNU/Linux では n32 もしくは 64 のサポートの必要はありませんので，32bit 長のリムと MIPS 2 用のコード生成だけを行います。

PowerPC 64 (‘powerpc64’, ‘powerpc620’, ‘powerpc630’, ‘powerpc970’, ‘power4’, ‘power5’)

‘ABI=mode64’

AIX 64 ABI は 64-bit リムとポインタを使いますので，PowerPC 64 ‘*-*-aix*’システム上ではこれがデフォルトとなります。アプリケーションをコンパイルする時のオプションは下記の通りです。

```
gcc  -maix64
xlc  -q64
```

64-bit GNU/Linux, BSD, Mac OS X/Darwin 上では下記のように指定します。

```
gcc  -m64
```

‘ABI=mode32’
: ‘mode32’ ABI は 64-bit long longリムを使用するものの，CPU は 32-bit モードで動作し，32bit の関数呼び出しを行います。真の 64-bit ABI が利用できない環境ではこれがデフォルトになります。コンパイラのオプション指定は必要ありません。こ n ABI は AIX 環境下では使用できません。

‘ABI=32’
: 基本的な 32bit PowerPC ABI で，32bit リムを使用します。この ABI を使う際には，特別なコンパイラオプションを付加する必要はありません。

GMP の速度は‘mode64’ ABI が最高で，次に‘mode32’ ABI です。‘ABI=32’では 32-bit ISA しか使えませんので，64bit CPU を有効に使用できません。

Sparc V9 (‘sparc64’, ‘sparcv9’, ‘ultrasparc*’)

‘ABI=64’
: 64-bit V9 ABI は様々な spar64 BSD，最新の Sparc64 GNU/Linux, Solaris 2.7 以上(カーネルが 64bit モードの場合）で利用可能です。GCC 3.2 以上，もしくは Sun ccが利用可能です。GNU/Linux 上では，デフォルトのgccのモードに依存しますが，下記のようにオプションをつけてコンパイルして下さい。

```
gcc  -m64
```

Solaris 上では次のように指定します。

```
gcc  -m64 -mptr64 -Wa,-xarch=v9 -mcpu=v9
cc   -xarch=v9
```

BSD sparc64 上では，64bit ABI しか使えませんので，オプション指定は不要です。

‘ABI=32’
: 基本的な 32bit ABI では，GMP は V9 ISA を利用します。Sun のドキュメントによると，この組み合わせは v8plus と呼ばれているようです。GNU/Linux 上では，デフォルトのgccのモードにもよりますが，下記のオプションを指定します。

```
gcc  -m32
```

Solaris 上では下記のように指定することが望ましいわけですが，特段オプションは付けなくても大丈夫です。(gcc 2.8 以前の場合は‘-mv8’になります）。

```
gcc  -mv8plus
cc   -xarch=v8plus
```

GMP の速度は‘ABI=64’が最大になるので，使用可能であればこの ABI がデフォルトになります。64bit 環境では追加のレジスタが利用でき，それ用のコードを持っているので最速になる訳です。

コンパイラに渡す‘-m’ と‘-x’ オプションをごっちゃにしないで下さい。どちらも‘arch’を指定するものですが，ABI と ISA の両方を効果的に指定できます。

Solaris 2.6 以前では，カーネルが全てのレジスタを保持しないので，‘ABI=32’だけが利用可能です。

Solaris 2.7 で 32bit モードのカーネルを使うと，普通のネイティブビルドを行うと実行可能ファイルが動作しなくなるので，‘ABI=64’オプションは受け付けません。下記のように，クロスコンパイルする要領でオプション指定すれば，‘ABI=64’でビルドできます。

```
./configure --build=none --host=sparcv9-sun-solaris2.7 ABI=64
```

2.3 バイナリ配布のためのビルドに関する注意

GMP は極力バイナリ配布ファイルの生成に関しては面倒が起きないようにしています。

Libtool はライブラリのビルドに使用されており，GMP 3.0 では‘3:0:0’のように，‘-version-info’で正しく GMP のバージョン番号が設定されるようになります(see Section “Library interface versions” in *GNU Libtool*)。

GMP Version 4 ではどのリリースでも，Version 3 と上位バイナリ互換になっています。とはいえ，リリースごとに関数が追加されていますので，libtool のバージョンチェック機能が起動時に完璧に働いていないシステムでは，アプリケーション側で，動的リンク可能なバージョンの GMP になっているかどうか，確認する別の機能が必要になるでしょう。

また，libgmpxx.la(--enable-cxxを使った場合, see Section 2.1 [Build Options], p. 3) が必要とする同じバージョンのlibgmp.laの存在を確認する機能も必要になります。というのも，libtool のバージョンチェックはこの確認作業を実行しないからです。GMP のバージョン違いはリンカ，もしくはローダで，関数等の名前解決不可能エラー(unresolved symbols)を発生させることになります。

ある CPU ファミリー用にパッケージをビルドする時には，‘--host’ (または‘--build’) オプションを使い，このパッケージを利用することになる CPU 間の最小公倍数(LCD)を注意深く選ばなくてはなりません。例えば，SPARC ファミリー用には‘sparc’ (V7 を意味します)をこのオプションで与えておくことになります。

x86 CPU ファミリーには，--enable-fatを使ってファットバイナリを作ることができるようになります。これにより，最適化された低レベルルーチンを実行時に選択して使うことができるようになります。x86 チップを対象とした広い範囲のパッケージングを行うにはこのオプションが役立ちます。

スピードを重視するのであれば，GMP ビルドは正確な CPU 型を指定し，使える限りの最適化手法の中から最高のものを使いたいと思うのが当然です。作り直しの機能もあれば便利でしょう。このためは‘--build’ (と‘--host’)オプションの指定を無視し，‘./config.guess’で自動的に CPU の型を探索することができるようします。とはいえ‘./config.guess’の探索が正確に行えないシステムでは，‘--build’オプションの手動指定を行わなければなりません。

複数 ABI を持つシステム上では，パッケージのビルドにおいてどの ABI を選ぶのかを決める必要が出てきます（詳細はSection 2.2 [ABI and ISA], p. 8参照）。‘./configure’を実行して得られるのは一つの ABI 用のバイナリだけです。別の ABI 用のバイナリが欲しい時には，クリーンにしたディレクトリツリー(‘make distclean’)を用意し，もう一度そこで‘./configure’を実行しなければなりません。

“ABI と ISA”の節の注記にもある通り，‘ABI=32’の時は/usr/libに，‘ABI=64’の時は/usr/lib/sparcv9にインストール先を ABI 毎に自動的に変更するという虫の良い機能はありません。一つのパッケージは‘libdir’ と標準的な場所に，必要に応じて上書きされていくことになります。

gmp.hはビルドによって生成されるファイルなので，CPU アーキテクチャと ABI に依存しているものであることを覚えておいて下さい。同時に二つの ABI をインストールしようとする時には，アプリケーションのコンパイル時に正しい ABI に対応した gmp.h を使うようにしておくことが大事です。もしコンパイラに対して異なる ABI を選べるようにインクルードパスを設定していないのであれば，プリプロセッサが関数や変数をテストしたり，正しいgmp.hを選べるようにするため，/usr/include/gmp.hを生成できるようにしておく必要があります。

2.4 特定のシステムにおける注意

AIX 3 と 4

‘*-*-aix[34]*’上では共有ライブラリがデフォルトでは使用不可になっています。ネイティブarが利用できないからなのですが，共有ライブラリとしてビルドしたい時には次のようにオプション指定を行います。

```
./configure --enable-shared --disable-static
```

‘--disable-static’が必要になるのは，共有ライブラリビルド時に libtoolがlibgmp.aをlibgmp.soへの symlink として生成してしまい，.aしか認識しない古いldが利用できるようになるからです。つまり，不幸にしてフル機能が使えるldでない限りは共有ライブラリの利用が出来ないことになります。

ARM ‘arm*-*-*’システム上では，GCC 2.95.3 以上では符号なし除算に関してバグが存在しており，結果がおかしくなることがあります。GMP の‘./configure’では GCC 2.95.4 以降であることを要求します。

Compaq C++

Compaq C++ を OSF 5.1 上で使用すると，2 種類のiostreamが使用でき，一つは標準のもの，もう一つは古い標準のものです(see ‘man iostream_intro’). GMP は前者でしか利用できませんが，不幸にして後者がデフォルトになっていますので，利用する時にはきちんと__USE_STD_IOSTREAMを指定して次のように設定を行います。

```
./configure --enable-cxx CPPFLAGS=-D__USE_STD_IOSTREAM
```

浮動小数点演算モード

ハードウェアが直接実行する浮動小数点演算では，特定の演算精度の設定が可能な制御モードが存在するケースがあります。例えば，x86 システムにおける，単精度，倍精度，拡張倍精度のようなものです(x87 浮動小数点命令)。double型を使用する GMP 関数は，単精度モードではフル精度の保障ができません。この場合はもちろん GMP に限らず，全てのアプリケーションに影響が及びます。

FreeBSD 7.x, 8.x, 9.0, 9.1, 9.2

上記の FreeBSD リリースに含まれるm4は 2 番目と 3 番目の引数を無視してくれる評価関数を持っており，.asmファイルを処理するのには向いていません。‘./configure’ではこの問題を見つけてPATHから別の m4 を見つけて利用するようになっています。このバグは FreeBSD 9.3 と 10.0 で解決されていますので，OS をアップグレードするか GNU m4 を使うようにして下さい。FreeBSD パッケージシステムでは，GNU m4 を‘gm4’という名前でインストールしますが，この名前では GMP は認識しません。

FreeBSD 7.x, 8.x, 9.x

GMP 6.0 以降では‘ABI=32’を FreeBSD/amd64 1.0 以降の環境ではサポートしません。limits.hが壊れているせいで，GMP が動作しなくなってしまいます。

MS-DOS と MS Windows

MS-DOS 上では DJGPP を使い，MS Windows 上では Cygwin, DJGPP, MINGW で GMP をビルドできます。この 3 つのコンパイラ環境は，いずれも GCC と GNU ツールの優れた移植です。

```
http://www.cygwin.com/
http://www.delorie.com/djgpp/
http://www.mingw.org/
```

Microsoft は Interix “Services for Unix”をリリースしており，この環境下で Windows 用の GMP をビルドできます（通常の‘./configure’を使う)。但しこれは有料ソフトウェアです。

MS Windows DLL
: ‘*-*-cygwin*’, ‘*-*-mingw*’，‘*-*-pw32*’環境下では，オプションなしで実行すると静的ライブラリしかビルドできません。DLL を生成する時には次のように指定します。

```
./configure --disable-static --enable-shared
```

静的ライブラリと動的ライブラリ(DLL)を両方ビルドすることはできず，それぞれに対しては異なるディレクティブを含むgmp.hを使うことになります。

MINGW 環境下でビルドした DLL 版 GMP は Microsoft C で利用可能です。Libtool は.libフォーマットのインポートライブラリを生成しませんので，MS のlibコマンドを使って下記のように生成し，インストールディレクトリにコピーして下さい。libmp やlibgmpxxに対しても同様にインポートライブラリを生成できます。

```
cd .libs
lib /def:libgmp-3.dll.def /out:libgmp-3.lib
```

MINGW は C のランタイムライブラリである‘msvcrt.dll’を I/O 用に使いますので，GMP の I/O ルーチンを利用するアプリケーションは必ず‘cl /MD’オプション付きでコンパイルして下さい。これ以外の MS C のランタイムライブラリが必要な場合，例えば GMP の文字列関数やアプリケーションへの I/O が必要な場合も，同様にリンクして下さい。

Motorola 68k タイプの CPU
: ‘m68k’は 68000 を意味します。‘m68020’以上の CPU では更に性能向上が見込めます。‘m68360’は CPU32 シリーズの CPU です。‘m68302’は“ドラゴンボール（Dragonball）”シリーズの CPU で，これらは全て‘m68000’の CPU ファミリーに属します。

NetBSD 5.x
: これらの NetBSD のリリースに含まれるm4は 2 番目と 3 番目の引数を無視する評価関数が入っており，.asmファイルを処理するには適切ではありません。‘./configure’はこの問題を認識してPATHから使える他の m4 を探して適用します。このバグは NetBSD 6 で取れましたので，OS をアップグレードするか GNU m4 を利用して下さい。NetBSD のパッケージシステムは GNU m4 を‘gm4’という名前でインストールしますので，GMP が認識できなくなります。

OpenBSD 2.6
: この OpenBSD リリースに含まれるm4はevalのバグがあり，.asmファイルの処理に向いていません。‘./configure’はこの問題に対処するため，PATHが通っている他の m4 を見つけて使用します。このバグは OpenBSD 2.7 で取れましたので，OS をアップグレードするか GNU m4 を利用して下さい。

Power CPU ファミリー
: GMP では，‘power*’ や‘powerpc*’といった CPU タイプごとに，相互に使用できない命令を使っています。従って，正しい CPU を指定することは大切です。現在の GMP 実装では，アセンブラコードで共通に使用できる命令サブセットをサポートしていません。どちらの CPU タイプでも実行できるようにするには，汎用 C コードを使用し(--disable-assembly), 適切なコンパイラオプション(gcc用には‘-mcpu=common’等）を指定して下さい。CPU ‘rs6000’ (CPU ではなくワークステーションの種別）はconfig.subで利用できますが，これは現状，--disable-assemblyと等価ではありません。

Sparc CPU ファミリー
: ‘sparcv8’や‘supersparc’ファミリーは，標準の‘sparc’用の V7 コードを高速に実行できます。

Sparc App Regs
: 32-bit と 64-bit Sparc の両方に対応した GMP のアセンブラコードは，“アプリケーションレジスタ” g2, g3，g4を痛めつけます（?）。同様のことは GCC のデフ

ォルトオプションである‘-mapp-regs’を使っても起こります(see Section “SPARC Options” in *Using the GNU Compiler Collection (GCC)*)。

このアセンラコードは特別な V9 に対するオプション‘-mcmodel=embmedany’ (g4レジスタをデータセグメントのポインタとして利用）にも，これらのレジスタを特別な目的で使用する予定のアプリケーションにも向いていません。こういう場合は GMP ビルド時に--disable-assemblyを指定し，アセンブラコードを使わないようにした方がいいでしょう。

SunOS 4
: /usr/bin/m4が入っていないので，.asmファイルの処理ができません。そのため，‘./configure’は自動的に/usr/5bin/m4を使用するようになっており，大体これで大丈夫かと思います(ダメなら GNU m4 を使います)。

x86 CPU ファミリー
: ‘i586’, ‘pentium’，‘pentiummmx’用のコードは P5 Pentium CPU にも有用ですが，Intel P6 クラスの CPU ではかえって低速になります(PPro, P-II, P-III). どちらでも使えるバイナリを作りたいのであれば，‘i386’を選択して下さい。

x86 MMX と SSE2 用コード
: CPU では MMX 命令を使用可能でもアセンブラがサポートしていない場合は，警告が出て MMX が使用できません。低レベルのビルドを行って MMX コードを適用すれば，標準の整数コードよりも高速に実行できるようになります。同じことは SSE2 にも言えます。

古い‘gas’は MMX 命令をサポートしておらず，特に FreeBSD 2.2.8 やもっと新しい OpenBSD 3.1 に入っている version 1.92.3 はダメです。

Solaris 2.6 と 2.7 のasは，movq命令を実行するとレジスターに対して不正なオブジェクトコードを生成してしまいます。従って，MMX 命令も使えません。使いたいのであれば，新しいgasをインストールして下さい。

2.5 ビルド時における問題点

更新情報はhttps://gmplib.org/で見ることができます。

コンパイル時のリンクオプション
: 現状の libtool は，共有ライブラリをリンクする時にコンパイラオプションを外してしまいます。将来的にはこの制限はなくなる予定ですが，現段階ではスクリプトを書き換えてこの制限をなくし，次のように configure 時に指定すればいいでしょう。

```
./configure CC=gcc-with-my-options
```

DJGPP (‘*-*-msdosdjgpp*’)
: DJGPP に含まれるbash 2.03は‘configure’スクリプトの実行ができません。動作開始時のメッセージだけをconfig.logに書き残して逝ってしまいます。bash 2.04 以降を使って下さい。

‘make all’すると，libgmp.laとリンクしてテストを行う段階で，64MB フルに利用できたとしてもメモリ溢れを起こします。‘make libgmp.la’と直接指定してコンパイルし，各サブディレクトリに移動して各テストを実行するようにします。

Version 2.12 以前の GNU binutils strip
: GNU binutils 2.11 以前に含まれるstripは，静的ライブラリlibgmp.a とlibmp.aに対しては使用しないで下さい。同じ名前のアーカイブファイル名が重複していると，その全部を削除するのではなく，最後のものだけを削除してしまいます。libgmp.aに 3 種類のinit.oが入っているケースがそれに当たります。Binutils 2.12 以降では正しく削除を実行できます。

共有ライブラリlibgmp.so やlibmp.soに対してはこのような現象は起きず，どのバージョンのstripを使っても大丈夫です。

makeの文法エラー
: SCO OpenServer 5 と IRIX 6.5 のいくつかのバージョンでは，ネイティブmakeはlibgmp.laに対する長い依存指定を処理できません。この症状は，Makefileの最初に登場する下記の記述を“syntax error”（文法エラー）にしてしまいます。

```
libgmp.la: $(libgmp_la_OBJECTS) $(libgmp_la_DEPENDENCIES)
```

回避するためには GNU Make を利用するか，この行から$(libgmp_la_DEPENDENCIES)を削除します（最初のビルドはこれでクリアしますが，もう一度コンパイルするとlibgmp.laが生成されなくなります）。

MacOS X (‘*-*-darwin*’)
: 現在の Libtool は，ネイティブcc(GCC の改良版)を使って MasOS X 上の共有ファイルを生成することしかできず，普通の GCC は使えません。静的ライブラリだけを生成したいのであれば，‘--disable-shared’を使って下さい。

NeXT 3.3 以前
: 古いバージョンの NeXT のシステムコンパイラは廃止され，古い GCC がccの代わりに使われています。このコンパイラは GMP をビルドできないので，新しい GCC をインストールする必要があります（NeXT は release 3.3 でこの不具合を修正する予定です）。

POWER と PowerPC
: GCC 2.7.2 (と 2.6.3)にあったバグのせいで，POWER や PowerPC 上で GMP のコンパイルができませんでした。GCC を使ってビルドしたければ，GCC 2.7.2.1 以降のものを入手して下さい。

Sequent Symmetry
: システムのアセンブラは深刻なバグがありますので，GNU assembler を使って下さい。

Solaris 2.6
: libtool でlibgmp.laをビルドする際，sedは“Output line too long”(出力行が長すぎる）というエラーを吐きます。このせいでメンドクサイ影響が出るわけではありませんが，GNU sedの利用をお勧めします。

Sparc Solaris 2.7 上で gcc 2.95.2 を使用する際の‘ABI=32’指定時
: GMP の共有ライブラリビルド時に，結合処理でコケます。正確に言うと，ビルドはできますがテストで失敗します。gmp_randinit_lc_2exp_size関数内のデータ再配置が正しく行われないためと思われますが，実際のところは不明です。‘--disable-shared’オプションの利用をお勧めします。

2.6 パフォーマンスの最適化

最適なパフォーマンスを達成するためには，ターゲットとするシステムの CPU 型を正確に指定して GMP をビルドするようにして下さい。

大多数の他のプログラムとは異なり，GMP ではアセンブラコードを最重要箇所に使用しているため，コンパイラはあまり役に立っていません。

長時間実行したり，超巨大な精度の数を使う GMP 利用のアプリケーションでは，tuneディレクトリにあるtuneupプログラムを使うことが肝要です。例えば

```
cd tune
make tuneup
./tuneup
```

とすることで，gmp-mparam.hファイル内のパラメータがより良いものになるかもしれません。

そういう結果を得たければ，‘Parameters for ...’ヘッダ内で示唆されているファイルの出力を使い，最初からコンパイルをやり直して下さい。

tuneupプログラムは有用なパラメータを提供しています。例えば‘-f NNN’は FFT 乗算パラメータをどの程度の長さにすべきかをプログラムに指示できます。超巨大な数を GMP で扱うのであれば，tuneupに超巨大な NNN 値を与えて実行してみたくなるはずです。

3 GMP の基本

本マニュアルに記述されていない関数，マクロ，データ型等々を使うことは厳に慎んで下さい。さもないと，あなたの作ったアプリケーションが将来の **GMP** を利用できなくなるかもしれません。

3.1 ヘッダファイルとライブラリ

GMP を使うために必要な宣言は全て`gmp.h`に集約されており，C と C++コンパイラのどちらでも動くように記述されています。

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

但し，GMP 関数のプロトタイプ宣言のうち，`FILE *`パラメータを含むものは`<stdio.h>` をインクルードしている時のみ使用できます。

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

同様に，`va_list`を含む関数，例えば`gmp_vprintf`関数のプロトタイプ宣言用に`<stdarg.h>`をインクルードしておく必要があります。また，`gmp_obstack_printf` のような関数が使えるようにしておくには，`struct obstack`パラメータを含みますので，`<obstack.h>`をインクルードしておきます。

GMP を利用するプログラムは例外なく`libgmp`ライブラリをリンクしなくてはなりません。例えば UNIX および UNIX 互換 OS では‘`-lgmp`’オプションをコンパイル時に指定します。

```
gcc myprogram.c -lgmp
```

GMP の C++関数は`libgmpxx`ライブラリに分離して格納されています。C++の利用を可能にしたい場合(see Section 2.1 [Build Options], p. 3)には，これをビルドしてインストールしておいて下さい。例えば次のようにリンクして使用します。

```
g++ mycxxprog.cc -lgmpxx -lgmp
```

GMP は Libtool を使ってビルドします。アプリケーションは必要に応じてそれをリンク時に利用することができます（*GNU Libtool*参照）。

GMP がデフォルトではないディレクトリ位置にインストールされている時には，‘`-I`’や‘`-L`’オプションでコンパイラに正しいパスを教えておく必要があります。また，共有ライブラリを使う際にはそのランタイムパスを指定しなければなりません。

3.2 用語とデータ型

本マニュアルでは，**整数**(*integer*)は常に GMP ライブラリで定義している多倍長精度の整数を指す言葉として使用します。C のデータ型としては`mpz_t`として定義してあります。（多倍長）整数は次のように宣言して使います。

```
mpz_t sum;

struct foo { mpz_t x, y; };

mpz_t vec[20];
```

有理数(*Rational number*)もまた，多倍長精度の分数を指す言葉として使用します。C のデータ型は`mpq_t`で，次のように使います。

```
mpq_t quotient;
```

浮動小数点数(*Floating point number*または*Float*)は任意精度の仮数部(mantissa) と，固定精度の指数部(exponent)を持ちます。C のデータ型は`mpf_t`で，次のように利用します。

```
mpf_t fp;
```

浮動小数点演算関数は，指数部を表現する`mp_exp_t`型の値のやり取りを行います。現状では通常`long`型そのものですが，性能向上のために，いくつかのシステム上では`int`型になっていることもあります。

リム(*limb*)とは，多倍長精度型の値を構成する 1 マシンワード分を意味する用語です。この用語を選んだ理由は，多倍長精度値の構成がちょうど人間の四肢(limb)に似ていて，複数の指(digit)を持っているところもそっくり，ということです。通常，リムは 32 ビットもしくは 64 ビット長で，C のデータ型としては`mp_limb_t`型になります。

1 つの多倍長精度数を構成するリムの数は，`mp_size_t`型で表現します。現状では`long`型ですが，性能向上のために`int`型の場合もありますし，将来は`long long`型にすることもあるでしょう。

1 つの多倍長精度数が何ビットになるのかを表現する際には`mp_bitcnt_t`型を使います。現状では必ず`unsigned long`型になっていますが，将来は`unsigned long long`型にすることもあるでしょう。

乱数生成状態(*Random state*)は，選択した乱数生成アルゴリズムの種類と現在の生成状態を意味する用語です。データ型としては`gmp_randstate_t`型として定義されており，次のように使います。

```
gmp_randstate_t rstate;
```

一般に，`mp_bitcnt_t`型はビット長と値の範囲を表現するの使いますが，`size_t`型はバイト単位，もしくは，文字単位の長さを表現するために使用します。

3.3 関数の種別

GMP ライブラリには 6 種類の関数があります。

1. `mpz_`から始まるのは符号付き（多倍長）整数演算を行う関数です。扱うデータ型は`mpz_t`型です。総計約 150 個の関数があります。(see Chapter 5 [Integer Functions], p. 32)
2. `mpq_`から始まるのは（多倍長）有理数演算を行う関数です。扱うデータ型は`mpq_t`型です。大体 35 個の関数があります。分子と分母はそれぞれ別個に整数演算関数で計算することができるようになっています。(see Chapter 6 [Rational Number Functions], p. 48)
3. `mpf_`から始まるのは（多倍長）浮動小数点演算関数です。扱うデータ型は`mpf_t`型です。大体 70 個の関数があります。(see Chapter 7 [Floating-point Functions], p. 52)
4. 基盤的な高速自然数演算を担う関数群です。これは上記 3 つの関数群の内部で使用されるものですが，極力高速な処理を実現したいユーザが直接自分のプログラムに使用することもできます。この関数群は`mpn_`から始まる関数名になり，扱うデータ型は`mp_limb_t`の配列です。良く使うのは大体 60 個の関数です。(see Chapter 8 [Low-level Functions], p. 59)
5. 上記の 4 種類に加えて，メモリ割り当てのためのカスタマイズ関数(see Chapter 13 [Custom Allocation], p. 91)や，乱数生成用の関数(see Chapter 9 [Random Number Functions], p. 71)があります。

3.4 変数の利用法

GMP が提供する関数は大概，入力引数より前に出力のための引数を置きます。これは代入演算子(=) に似せるためです。BSD MP 互換関数の流儀はこれと異なり，出力のための引数は後ろに置かれます。

GMP は 1 回の関数呼び出しで同じ変数を同時に入力値，出力値に指定することができます。例えば整数の乗算を行う関数mpz_mulは，次のように指定することで，xの 2 乗を計算してxに書き戻すことができます。

```
mpz_mul (x, x, x);
```

GMP の変数を使用可能にする前に，変数型ごとに決められた初期化関数を呼び出す必要があります。変数を使い終わった後には，変数消去のための関数を呼び出す必要があります。詳細は整数関数，有理数関数，浮動小数点数関数の章をそれぞれ参照して下さい。

変数の初期化は 1 回だけ行って下さい。また，再度初期化する際は，その前に一度消去しておいて下さい。変数初期化の後は何度でも値の代入ができるようになります。

変数の初期化と消去の繰り返しは，動作遅延の元になりますので避けて下さい。以下の例のように，なるべく変数初期化は関数のスタート時に，消去は終了時に行うようにして下さい。

```
void
foo (void)
{
  mpz_t  n;
  int    i;
  mpz_init (n);
  for (i = 1; i < 100; i++)
    {
      mpz_mul (n, ...);
      mpz_fdiv_q (n, ...);
      ...
    }
  mpz_clear (n);
}
```

3.5 パラメータの利用法

GMP の変数を関数の引数として使用する際は，参照渡しすると遅延が防げますが，これを行うと，関数内で値を代入すると，オリジナルの値とは別のものに書き換わってしまいます。入力値のままにしておきたい場合は，const宣言をしておくと，意図しない代入の際にはコンパイルエラーもしくは警告が出るようになります。

GMP のデータ型を返り値とする関数を作る時には，GMP が提供する関数群がやっているように，引数に値をセットするようにして下さい。複数の値を返したい時は，複数の出力用引数を作ります。

下記は，mpz_t型の引数を取って計算を行い，結果を引数に渡す関数の例です。

```
void
foo (mpz_t result, const mpz_t param, unsigned long n)
{
  unsigned long  i;
  mpz_mul_ui (result, param, n);
  for (i = 1; i < n; i++)
    mpz_add_ui (result, result, i*7);
}

int
main (void)
{
```

```
  mpz_t  r, n;
  mpz_init (r);
  mpz_init_set_str (n, "123456", 0);
  foo (r, n, 20L);
  gmp_printf ("%Zd\n", r);
  return 0;
}
```

foo関数は，paramとresultに同じ変数を指定しても，他の GMP 関数同様きちんと動作します。たまにこのような場合の動作がおかしかったり，この種の場合のサポートをサボったりすることもあるでしょうが。

興味のある人向きに解説しておくと，GMP のmpz_tのような多倍長数のデータ型は，これを定義した構造体の配列 1 個分として実装されています。このようにしている理由は，型宣言した時点で GMP が必要とする構造体の構成要素を持ったオブジェクトを生成するからです。また同時に，引数としてこれを使うと，オブジェクトへのポインタを渡すことになるからでもあります。それぞれのmpz_tのような GMP の多倍長数のデータ型に含まれる構成要素は内部的に使われるだけで，直接アクセスして良いものではありません。そのような使い方をすると，将来の GMP に対しては非互換になってしまう可能性が出てきます。

3.6 メモリ管理

GMP が定義するmpz_tのような多倍長数のデータ型は，データサイズとデータそのものへのポインタしか保持していませんので，データ量としては少ないものです。初期化を一度行うと，GMP はメモリ領域についての管理を行うようになります。十分なメモリ領域を確保できない時にはいつでも追加のメモリスペースを割り当てます。

mpz_t型とmpq_t型の変数は割り当てられたメモリ領域を減らすことはできません。頻繁に割り当てを繰り返す愚を避けるためにはこの方が普通は良い訳です。逆にアプリケーション側でヒープ領域が必要になった時には，mpz_realloc2関数を使って明示的に再割り当てができますし，不必要になった変数を消去することも可能です。

現状の実装では，mpf_t変数に対しては，初期化の段階で精度に応じた固定サイズのメモリ領域を割り当てます。サイズの変更は自動的には行いません。

全てのメモリ領域は，malloc関数や同様のメモリ確保関数を使って確保するものですが，使用する関数を変えることも可能です。詳細はChapter 13 [Custom Allocation], p. 91をご覧下さい。スタックのための一時変数(alloca関数で確保)も使用できますが，これもビルド時に変更することができます。詳細はSection 2.1 [Build Options], p. 3を参照して下さい。

3.7 再入可能性

GMP は再入可能(reentrant)かつスレッドセーフですが，次の例外があります。

- alloca関数が利用できない時に--enable-alloca=malloc-notreentrant (もしくは--enable-alloca=notreentrant)を使ってコンパイルすると，当然 GMP は再入可能ではなくなります。
- mpf_set_default_prec関数とmpf_init関数は精度指定のためにグローバル変数を使ってしまいます。代わりとして，mpf_init2や，C++での明示的なmpf_classコンストラクタへの精度指定が使用可能です。
- mpz_random関数と，他にも古い乱数関数はグローバル変数を使うので，結果として再入可能ではなくなります。gmp_randstate_t型のパラメータを受け付ける新しい乱数関数をその代わりに使うことが可能です。
- gmp_randinit 関数(廃止予定)はグローバル関数を介したエラー表示を行うのでスレッドセー

フではありません。その代わり，gmp_randinit_default関数やgmp_randinit_lc_2exp関数の使用をお勧めします。

- mp_set_memory_functions関数はグローバル変数を使って指定されたメモリ割り当て関数名を保存しています。
- メモリ割り当て関数がmp_set_memory_functions関数(もしくはmalloc関数とその仲間)で指定でされていて再入可能でなければ，GMP も再入可能でなくなります。
- fwrite関数のように標準 I/O 関数が再入可能でなければ，それを使っている GMP の I/O 関数も再入可能でなくなります。
- 二つのスレッドで同時に GMP の変数を読み出すことはできますが，片方が書きこんでいる時にもう片方が読み出したり，同時に書き込みを行うとスレッドセーフではなくなります。また，二つのスレッドで同じgmp_randstate_t変数を使って同時に乱数を生成すると，同時にこの変数をアップデートしてしまいますので，スレッドセーフではなくなります。

3.8 利用可能なマクロと定数

`const int mp_bits_per_limb` [Global Constant]
1 リム(limb)あたりのビット数。

`__GNU_MP_VERSION` [マクロ]
`__GNU_MP_VERSION_MINOR` [マクロ]
`__GNU_MP_VERSION_PATCHLEVEL` [マクロ]
GMP バージョンのメジャー番号，マイナー番号，パッチレベルを表わす整数です。GMP i.j の場合はそれぞれ i, j, 0 となります。GMP i.j.k の場合は i, j, k となります。

`const char * const gmp_version` [Global Constant]
GMP のバージョン番号で，“i.j.k”という形式の，NULL で終端となる文字列で表現されます。本文書を含む GMP のバージョンは"6.1.2"です。k がゼロの場合は“i.j”という形式も 4.3.0 以前では使われていました。

`__GMP_CC` [マクロ]
`__GMP_CFLAGS` [マクロ]
GMP のコンパイル時に使うコンパイラの指定と，コンパイラに与えるフラグ。文字列として与えます。

3.9 古い GMP との互換性

本バージョンの GMP は，GMP 5.x, 4.x, 3.x すべてに対してバイナリレベルで上位互換であり，ソースコードレベルでは 2.x と上位互換性を持っています。但し，下記の例外があります。

- mpn_gcd関数は，GMP 3.0 のものと同様に大本の引数を交換してしまいます。これは他のmpn関数と組み合わせて使っても問題ないようにするためです。
- mpf_get_prec関数は GMP 3.0 と 3.0.1 とで若干精度桁数の計算に違いがあります。3.1 では 2.x のスタイルに戻しました。
- GMP 4 の preliminary で予告した通り，mpn_bdivmod関数は削除されました。

GMP 1 と GMP 2 における相互互換性は，GMP 1 から GMP 5 へアプリケーションを移植する際にも引き継がれています。詳細は GMP 2 のマニュアルを参照して下さい。

3.10 サンプルプログラム

demosディレクトリには GMP を用いたサンプルプログラムが入っています。基本的にはビルドもインストールもされませんが，Makefileを使って明示的にビルドすることができます。例えば下記のようなプログラムをコンパイルして実行できます。

```
make pexpr
./pexpr 68^975+10
```

これを含め，下記のようなデモ用プログラムが提供されています。

- 'pexpr' は式計算プログラムで，GMP Web ページから使うことができます。
- 'calc'ディレクトリには，lexとyaccを用いた簡単な式計算プログラムがあります。
- 'expr'ディレクトリには別の式計算ツールが入っており，C のプログラムから使いやすい形のライブラリとして提供されています。詳細はdemos/expr/READMEを参照して下さい。
- 'factorize'は Pollard-Rho 因数分解を実行するプログラムです。
- 'isprime'はmpz_probab_prime_p関数を実行するためのコマンドラインインターフェースプログラムです。
- 'primes'は篩法(sieve)を使って指定区間における素数の数を数えたり，素数のリストを作るプログラムです。
- 'qcn'は 2 次位数(quadratic class number)を評価するためのmpz_kronecker_ui関数の使用事例です。
- 'perl'ディレクトリには GMP を使うための Perl インターフェースが用意されています。詳細はdemos/perl/INSTALLをご覧ください。POD フォーマットの文書はdemos/perl/GMP.pmにあります。

以下は閑話です。GMP ライブラリ内でのある種の式計算に要する時間を想像してみて下さい。ユーザが定義した関数，ループ，変数操作のための固定精度値(fixnum)等々・・・これらは GMP の中では考慮されないコストです(インタプリタやコンパイラ側の問題。See Chapter 14 [Language Bindings], p. 93)。上記のexprやpexprを組み合わせて使うことで，数値データの入力は簡単になるかもしれませんが，今のところは上記の式計算プログラム類は，GMP の使い方の解説用として提供されているものとお考え下さい。

3.11 高速計算のための諸注意

小さい値に対する演算
: 扱う値が小さい時には，計算時間よりも関数呼び出しやメモリ割り当てに要する時間の方がシビアに効きます。標準データ型を使う際には不可避のことではありますが，GMP では，桁の長短に応じてこのオーバーヘッドを調節して効率化を図っています。

スタティックライブラリの利用
: CPU によっては，特に，x86 系統では，スタティックライブラリlibgmp.aを使うと高速に実行できるようになります。これは，ダイナミックライブラリにおける PIC コードで，関数やグローバルデータの呼び出しに要する小規模なオーバーヘッドが発生することに起因します。大方のプログラムでは大した問題にはなりませんが，長時間にわたる計算を実行するとじわじわ効いてくるでしょう。

変数の初期化と解放
: 変数に対して初期化や解放を連続的に行わないようして下さい。加算のように軽い演算に比べ，結構な時間を食われてしまいます。

  インタプリタの場合，解放した変数リストや，初期化済みの使用可能な変数のスタックを保持している場合があり，これによってガベージコレクションが可能になります。

メモリ空間の再確保
: mpz_t型とmpq_t型の変数は逐次的に桁を増やしていくものなので，その都度reallocに関数を使ってメモリ空間を再確保することになります。当然，低速になったり，メモリ空間の断片化が起こったりしますが，その程度は C ライブラリ次第です。アプリケーション側で予め最大サイズが分かっているのであれば，mpz_init2関数

かmpz_realloc2関数を呼び出して，最初から必要スペースを確保しておくこともできます。(see Section 5.1 [Initializing Integers], p. 32).

mpz_init2関数やmpz_realloc2関数で確保したサイズが小さすぎるということは気にする必要はなく，全ての関数は必要に応じて桁が収まるようにメモリ空間の再確保を行います。むしろ，メモリを余分に取り過ぎる方が無駄というものです。

2exp関数 mpz_mul_2expのような関数を適切な所に使うとアプリケーションの性能向上が見込めます。とはいえ，こういう被演算数が特殊な場合を演算ごとに確認するのは時間の無駄なので，mpz_mul関数のような汎用演算関数を 2 のべき乗や高速化が見込める形の演算用として使うのは止めましょう。

ui関数とsi 関数

ui関数や，少数ながら存在するsi関数は，利用可能なケースには便利です。とはいえ，例えばmpz_t型に値が入っていて，これがunsigned long型に収まる桁数である場合は，わざわざこれをunsigned long型に取り出して使うメリットはなく，そのまま通常のmpz関数を使った方が得策です。

同じ変数に対する演算

mpz_abs, mpq_abs, mpf_abs, mpz_neg, mpq_neg, mpf_negといった関数は，mpz_abs(x,x)というように，同じ変数に対する処理は高速に行います。現在の実装では，xという一つのフィールドだけを変更するだけで済むからです。これに向いているコンパイラ（例えば GCC)においてはインラインとして実行できます。

mpz_add_ui, mpz_sub_ui, mpf_add_ui，mpf_sub_uiといった関数も，mpz_add_ui(x,x,y)という一変数演算に向いており，xの 1，2 リム変更するだけで済みます。yの桁数が短い時には同様のことがフル精度のmpz_add等の関数にも言えます。yが長い桁であっても，全桁計算しますがキャッシュの効果が期待できます。

mpz_mul関数は上記の関数群とは真逆で，現在の実装では，演算結果の格納先は別変数になっていた方が多少マシになります。mpz_mul(x,x,y)のように使用すると，yがたとえ 1 リム長であっても，xの一時的な複写が行われてから演算が実行されます。通常，この複写は乗算に比べれば僅かな時間で済みますので，あまり問題にはならないとは思いますが。

mpz_set, mpq_set, mpq_set_num, mpf_set等の関数は，代入先が元の変数かどうかの確認は行いませんので，mpz_set(x,x)などというのは無駄以外の何物でもありません。こんな書き方にはならないように，二つのポインタが同じ値を指示しているかどうか，下記のように確認しておきましょう。

```
if (x != y)
  mpz_set (x, y);
```

無駄なmpz_set呼び出しを導入しないよう，GMP におけるデータの行先には注意を払って下さい。

(小さい整数による)除算可能性のテスト

mpz_divisible_ui_p関数とmpz_congruent_ui_p関数は，mpz_t型の数が，小さい整数で割り切れるかどうかを確認するための最良の関数で，mpz_tdiv_ui関数より高速なアルゴリズムを使っています。但し，剰余の情報を得ることはできないので，使えるのはきっちり割り切れるときだけです（もしくは，剰余が既知の時のみ）。

複数の小さい整数で割り切れるかどうかの確認を行う時には，それらの積の剰余を求めるのが最良で，これによって多倍長演算量を節約できるようになります。例えば，ある数が 23, 29, 31 で割り切れるかどうかを知りたい時には，$23 \times 29 \times 31 = 20677$ の剰余を求め，これが割り切れるかどうかを確認します。

mpz_tdiv_q_ui関数のように，商と剰余を同時に求める除算関数は，mpz_tdiv_ui関数のように剰余のみ求める関数より少々低速になります。商が必要になることはあまりないので，剰余だけ求めておき，必要に応じて商を計算するようにした方がいいでしょう(剰余がゼロならmpz_divexact_ui関数が使えます)。

有理数演算

mpq関数は，既約分数であるmpq_t型の値を使って演算を行います。約分できるのであれば必ず実行する必要があり，こうすることで，なるべく桁数を減らして後続の演算を軽くできます。

とはいえ，アプリケーション側が約分できるかどうかの情報を持っている場合は，いちいち GCD を確認するのは避けたくなります。例えば，素数を乗じた場合，GCD を確認するよりは分母にその素数因子があるかどうかだけを見れば十分なわけです。また，積が長くなる場合も，殆ど約分できないことが分かっているのであれば，約分は最後までとっておきたくなります。

mpq_numrefとmpq_denrefマクロは，分子と分母へのアクセスを提供するためのもので，mpq関数の枠外で個別に操作ができるようになります(See Section 6.5 [Applying Integer Functions], p. 50)。

有理数の標準形はmpq_tと整数の加減算を混ぜて使うことができます。例えば次のようになります。

```
/* mpq++ */
mpz_add (mpq_numref(q), mpq_numref(q), mpq_denref(q));

/* mpq += unsigned long */
mpz_addmul_ui (mpq_numref(q), mpq_denref(q), 123UL);

/* mpq -= mpz */
mpz_submul (mpq_numref(q), mpq_denref(q), z);
```

数列

mpz_fac_ui, mpz_fib_ui，mpz_bin_uiuiといった関数は，数列中の特定の値を求めるためのものです。一定範囲の数列全て欲しい場合は，初期値だけこの関数を使って求め，残りは漸化式で求めるのが最良です。

テキストの入出力

16 進，もしくは 8 進表現はテキストの入出力時に指定できます。このように 2 のべき乗が基数である時には，それ以外の基数の表現よりも効率的に変換ができます。長い桁の数に対しては，処理時間はどのみち長くなりますので，基数は大して影響しません。

スウェーデン王国のチャールズ 12 世とその後継者が提案しているように，そのうち 8 進数が日常生活で普通に利用されるようになってほしいモンですな。（Knuth 本の Vol.2, 4.1 節参照）

3.12 デバッグ

スタックオーバーフロー

システムによりますが，セグメンテーション違反(segmentation violation)やバスエラー(bus error)が出ると，スタックオーバーフローが発生していることを意味します。この発生個所を特定するには，Section 2.1 [Build Options], p. 3の‘--enable-alloca’を指定して下さい。

GCC の比較的新しいバージョンでは，‘-fstack-check’オプションが利用でき，これでシステムがダメージを受ける前に，スタックオーバーフローを感知できるようになります。また，‘-fstack-limit-symbol’ や‘-fstack-limit-register’を使うと，システム側では何もしなくてもチェックできるようになります(see Section “Options for Code Generation” in *Using the GNU Compiler Collection (GCC)*)。これらのオプションは GMP のビルド時に‘CFLAGS’フラグに指定しておく必要があります(see Section 2.1 [Build Options], p. 3)。アプリケーション側にはこれらのオプション指定の影響はない筈ですが，関数呼び出しやスタック確保時にオーバーヘッドが多少加わって遅くなるかもしれません。

ヒープ領域の問題

GMP を使ってアプリーションを作ると，問題の多くがヒープ領域の破壊に起因することが分かります。GMP の変数の初期化に失敗すると，予測不能の影響が出て GMP の実行そのものにも及んできたりします。GMP 変数を 2 回以上初期化したり，変数の解放に失敗したりすると，メモリリークの原因になります。

このような場合は，`malloc`デバッガーの利用をお勧めします。GNU や BSD システムには標準 C ライブラリの`malloc`関数に診断機能が付いおり，詳細はSection “Allocation Debugging” in *The GNU C Library Reference Manual*, もしくは‘`man 3 malloc`’を参照して下さい。他にも，順不同に示すと下記のようなツールがあります（訳注：アクセス不可の URL 多し)。

```
http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/
http://dmalloc.com/
http://www.perens.com/FreeSoftware/ (electric fence)
http://packages.debian.org/stable/devel/fda
http://www.gnupdate.org/components/leakbug/
http://people.redhat.com/~otaylor/memprof/
http://www.cbmamiga.demon.co.uk/mpatrol/
```

GMP のデフォルトのメモリ割り当てルーチンは`memory.c`にあり，ここには簡単な防人スキームが組み込まれていて，`#define DEBUG`をこのファイルで有効にすると使えるようになります。基本的には GMP 開発時にバッファオーバーランを検知するための機構ですが，他の目的にも役に立つでしょう。

スタックのバックトレース

ある種のシステムでは, GMP がデフォルトで使用するコンパイラオプションがデバッグの邪魔をする可能性があります。特に x86 系や 68k 系では，‘`-fomit-frame-pointer`’オプションが使えますが，これを指定するとスタックのバックトレースが妨げられてしまいます。メモリの問題を無視したり，コンパイラのバグを隠してしまう可能性はありますが，デバッグ時には，このオプションを付けずに再コンパイルしておいた方がいいでしょう。

GDB, GNU デバッガ

サンプルとして`.gdbinit`が GMP の TAR ボールに同梱されており，これでドキュメント化されていないダンプ関数を呼び出して GMP 変数の中身を GDB で表示させる方法が分かるようになっています。これらの関数はアプリケーションの最終実行コードには含まれていません。ドキュメント化していないので，将来の GMP では互換性のない形で変更される可能性があります。

ソースプログラムへのパス

GMP は同名のファイルを多数持っており，それぞれ別のディレクトリに入っています。`mpz`，`mpz`, `mpf`を例に挙げると，これらのディレクトリには`init.c`がそれぞれ含まれています。デバッガはどれを指しているのか決められませんので，それぞれの C ファイルへの絶対パスを付加してビルドしておくと，どの`init.c`か分かるようになります。やり方としては，オブジェクトファイルを生成するディレクトリを別に作り，そこからソースがあるディレクトリへの絶対パスを下記のように指定します。

```
cd /my/build/dir
/my/source/dir/gmp-6.1.2/configure
```

これは`VPATH`経由でも実行できますが, GNU `make`が必要となります。他にも，`.c.lo`の生成規則を変更することでできるようになります。

アサーションによるチェック

`--enable-assert`というビルド時のオプションを付加することで，ライブラリの整合性チェックが使えるようになります(Section 2.1 [Build Options], p. 3参照)。大方のアプリケーションに対して，値の制限ができるようになります。アサーションエ

ラーが発生すると，ライブラリかコンパイラのバグのように，メモリ破壊として検知されるようになります。

基盤的mpn関数を使ったアプリケーションの場合，--enable-assertを使うと，関数の引数のチェックを行うことができ，大変便利です。こういうアプリケーションは微妙な制限を付ける必要があるからです。しかしながら，このアサーションチェックは汎用 C コードだけで利用でき，アセンブラコードでは使えませんので，最大限活用したいのであれば，--disable-assemblyオプションをつけてビルドして下さい。

一時メモリ領域のチェック

ビルド時のオプション--enable-alloca=debugを使うと，GMP の一時メモリ領域の各ブロックが，別々のmalloc (もしくはmp_set_memory_functionsで指定された関数)を使って確保されるようになります。

これによって，malloc デバッガが，内部エリアの外側からのアクセスや，解放されていないメモリを検知できるようになります。通常のビルドを行うと，GMP の一時メモリ領域のブロックは，その利用範囲内に限定されたバラバラのものとして確保されたり，コンパイラにビルドインされているalloca関数で確保されるので，malloc デバッガのフックが効かなくなってしまいます。

デバッグしやすくするために

今まで縷々説明してきたことを要約すると，最大限デバッグしやすくしようとするならば，GMP のビルド時のオプション指定は次のように行って下さい。

```
./configure --disable-shared --enable-assert \
  --enable-alloca=debug --disable-assembly CFLAGS=-g
```

C++の場合は‘--enable-cxx CXXFLAGS=-g’もお忘れなく。

チェッカー

GCC チェッカー(https://savannah.nongnu.org/projects/checker/)は GMP で利用可能です。これはスタブライブラリを持っており，GMP のアプリケーションにこのチェッカーを付けてコンパイルすると，通常の GMP ビルドも使用できるようにしてくれます。

GMP の内部にチェッカーを付加して GMP ビルド行うことも可能ですが，結果としてものすごく低速になってしまいます。GNU/Linux 上では次のようにビルド時の設定を行います。

```
./configure --disable-assembly CC=checkergcc
```

GMP のアセンブラコードにはチェック機構が入っていませんので，--disable-assemblyでビルドしなくてはなりません。現在のチェッカーのバージョン(0.9.9.1)では標準 C++ライブラリをサポートしていませんので，GMP C++でもチェック機構は利用できません。

Valgrind Valgrind (http://valgrind.org/)は x86, ARM, MIPS, PowerPC, S/390 用のメモリチェッカーです。各マシン用の命令を変換してエミュレートし，未初期化データ（ビット単位のレベルまで）や，不正ポインタやメモリリークが発生しているメモリアクセスに対して強力なチェックを行います。

Valgrind は全ての命令をサポートしている訳ではなく，特に，最近 ISA に追加された命令はサポートしていません。従って，Valgrind のエミュレーションは最新の GMP，もしくは最新の GCC でコンパイルされた GMP に対しては有効でない可能性があります。

GMP のアセンブラコードは読み込むリムが大きくなっても効率良く実行できるようにするためのものです。GMP は，リム配列の最初から最後まで読み込みを行い，自然なアラインメントを行って大きなデータタイプにも対応することになります。つまり，あらかじめ確保されたメモリエリアの外側を読み込むこともあり得るので，この際には Valgrind が警告を発することになります。ウザいようなら Valgrind の‘--partial-loads-ok=yes’オプションを使って下さい。

その他の問題
: GMP そのものに潜むバグは，アプリケーション側の問題ではないということを確認する必要があります。Chapter 4 [Reporting Bugs], p. 31を参照して下さい。

3.13 プロファイル

プロファイラの配下でプログラムを実行すると，その中のどの部分で処理時間を食っているのか，どこを改善すべきかが判明します。GMP ビルド時にプロファイリングの有無を指定するオプションは下記の通りです。

‘--disable-profiling’
: デフォルトではプロファイリングに関しては何も指定しません。

  プログラムのメイン部分に-pを付加してコンパイルすることで，profコマンドを使って，プログラムに埋め込まれたカウンタのサンプリングを行ってプロファイリング情報を取得できるようになります。GMP のアセンブラコードには大体，このために必要となるシンボルが入っています。

  こういうやり方をすることで，通常のプログラム処理に干渉する手間を最小限にとどめることができますが，大概のシステムのサンプリング時間間隔は非常に長い（例えば 10 ミリ秒間隔）ので，長時間実行しないと正確な情報が掴めません。

‘--enable-profiling=prof’
: システム標準のプロファイラprofを利用可能にします。‘-p’はコンパイラのフラグ‘CFLAGS’に付加されます。

  このオプションを指定することで，プログラムのカウント間隔の指定だけでなく，呼び出し回数のカウントができるようになり，一番呼び出される回数の多いルーチンの特定や，指定関数の平均処理時間を知ることができるようになります。

  x86 アセンブラコードはこのオプションをサポートしていますが，他のプロセッサ上では，アセンブラルーチンは‘-p’オプションナシでビルドした形になり，結果として呼び出し回数のカウントはできなくなります。

  GNU/Linux のようなシステムでは，‘-p’は事実上‘-pg’と同義となり，下記で説明する‘--enable-profiling=gprof’を指定した方がいいでしょう。

‘--enable-profiling=gprof’
: gprofが使えるようにビルドします。‘-pg’は‘CFLAGS’に追加するフラグです。

  このオプションは，呼び出し回数を行ったり，その調査間隔を指定したりするのに加えて，グラフを構成します。これによって，異なる場所からの呼び出し回数を数えることができるようになります。例えば，mpz_mul関数とmpf_mul関数からの，mpn_mul関数の呼び出し回数を比較したいとします。調査間隔が同じ場合，グラフなしではmpn_mul関数で要する計算時間はまとめて積算され，それぞれの関数毎の呼び出し分を分離することはできなくなってしまいます。

  x86 アセンブラコードではこのオプションをサポートしています。他のプロセッサ上では，アセンブラルーチンは，‘-pg’オプションナシでコンパイルされ，関数呼び出し回数を数えることはできません。

  x86 系と m68k 系のシステムでは‘-pg’ と‘-fomit-frame-pointer’に互換性はありませんので，後者はデフォルトフラグからは除かれ，効率の悪いコードが生成される可能性があります。

  ついでに言うと，‘--enable-profiling=prof’オプション付きでビルドしたとしても，gprofは利用可能にしておくべきでしょう。‘gprof -p’だけだとフラットプロファイルと呼びだし回数カウントが行えるようになりますが，‘gprof -q’を指定しないと呼び出しグラフを得ることはできません。

‘--enable-profiling=instrument’

GCC のオプションである‘-finstrument-functions’を‘CFLAGS’に追加します(see Section “Options for Code Generation” in *Using the GNU Compiler Collection (GCC)*)。

このオプションは，関数の最初と最後に実行する特別な命令を追加できるようになります。これによって正確な時間計測と，呼び出しグラフ構造が判明します。

この命令は標準のシステムにはない機能ですので，下記のような外部ライブラリのサポートがないと使えません。

```
http://sourceforge.net/projects/fnccheck/
```

このオプションは，テストプログラムにリンクできるよう，GMP の confiugre 時に‘LIBS’に含めておく必要があります。

```
./configure --enable-profiling=instrument LIBS=-lfc
```

GNU システム上では，C ライブラリはダミーの命令関数を提供しており，このオプションをつけてコンパイルしたプログラムはこれをリンクできるようになります。この場合は，アプリケーションのリンク時に正しいライブラリを指定するだけで済みます。

x86 アセンブラコードはこのオプションをサポートしていますが，他のプロセッサのアセンブラコードは‘-finstrument-functions’オプションなしでコンパイルされるものと考えて下さい。

3.14 Autoconf

Autoconf を利用するアプリケーションは，GMP がインストールされているかどうかを簡単に確認できます。気をつけなればいけないのは，Version 3 以降の GMP ライブラリのシンボル（関数名，変数名など）は__gmpzというような接頭詞を持つようになっている，ということです。従って，下記のようなテストを行うことになります。

```
AC_CHECK_LIB(gmp, __gmpz_init)
```

この例では，GMP の有無を調べるデフォルトのAC_CHECK_LIB の動作を利用しています。GMP が不可欠のアプリケーションの場合は，GMP が見つからない場合にエラーを吐くようにしたい訳です。

```
AC_CHECK_LIB(gmp, __gmpz_init, ,
  [AC_MSG_ERROR([GNU MP not found, see https://gmplib.org/])])
```

特定バージョンの GMP かどうかを確認したい時には，下記のようにして，そのバージョンで使用できる関数の一つを指定すれば良いわけです。例えば，mpz_mul_si関数は GMP 3.1 で追加されましたので，下記のように指定します。

```
AC_CHECK_LIB(gmp, __gmpz_mul_si, ,
  [AC_MSG_ERROR(
  [GNU MP not found, or not 3.1 or up, see https://gmplib.org/])])
```

他の方法としては，AC_EGREP_CPPを利用してgmp.hに記述してあるバージョン番号を調べることでも実現できます。この方が，サブマイナーリリース番号まで，正確なバージョン確認ができます。

新しい GMP は新しい機能を提供していますので，アプリケーション作成時には新しい GMP を利用しましょう。

アプリケーションが望むのは，設定時やプリプロセス時におけるデータ型のサイズであって，実行時のsizeofの値ではありません。これはmp_limb_t型を使えば済む話で，GMP 4.0 以降では最良の方法です。以前のバージョンでは，long long型のリムを使用してるシステム上では‘-D’オプションが必要とされていました。

```
AC_CHECK_SIZEOF(mp_limb_t, , [#include <gmp.h>])
```

3.15 Emacs

C-h C-i (info-lookup-symbol)を使うと，文書編集しながら C 関数を見つけることができて便利です(see Section “Info Documentation Lookup” in *The Emacs Editor*)。

下記のようにして，この GMP マニュアルを，貴方が使用している.emacsに追記しておくことができます。

```
(eval-after-load "info-look"
  '(let ((mode-value (assoc 'c-mode (assoc 'symbol info-lookup-alist))))
     (setcar (nthcdr 3 mode-value)
             (cons '("(gmp)Function Index" nil "^ -.* " "\\>")
                   (nth 3 mode-value)))))
```

4 バグ報告

GMP ライブラリにバグを発見したと思った時には，それを調査の上，報告して下さい。我々開発陣は本ライブラリを提供している訳ですから，バグ報告を使用者のあなたにお願いしても罰は当たらんでしょう。

バグ報告の前に，それが既知のものでないかどうかをSection 2.5 [Known Build Problems], p. 15で確認して頂くか，特定の環境で起きている現象なのかをSection 2.4 [Notes for Particular Systems], p. 13で確認して下さい。また，`https://gmplib.org/`でこのバージョン用のパッチが出ていないかどうかもチェックして下さい。

バグ報告には下記の内容を必ず書いて下さい。

- GMP のバージョン番号。正式リリース前のものか，パッチを当てたものであればそれも明記して下さい。
- バグを再現できるテストプログラム。このプログラムを実行するための手順も書いて下さい。
- 不具合の内容。結果が不正確になるとか，クラッシュしてしまうというのであれば，どういう手順を踏むとそうなってしまうのか，記述して下さい。
- クラッシュするようであれば，デバッガによるスタックのバックトレース結果(`gdb`出力の該当部分，もしくは，`adb`の‘`$C`’)を入れて下さい。
- コアダンプファイルや実行ファイル，`strace`を送りつけないで下さい。
- GMP ビルド時に指定した‘`configure`’のオプション。
- ‘`configure`’の標準出力結果と使用オプション。
- 使用コンパイラ名とバージョン番号。`gcc`であれば，バージョン番号は‘`gcc -v`’で取得できますし，他のコンパイラであれば‘`what `which cc‘’等で判明するかと思います。
- ‘`uname -a`’実行時の出力。
- ‘`./config.guess`’と‘`./configfsf.guess`’の実行時の出力（多分同じものになる筈）。
- ‘`configure`’絡みのバグであれば，`config.log`を圧縮して送って下さい。
- アセンブルしていない`asm`ファイルに関連するものであれば，`config.m4`の内容と，一時ファイル`mpn/tmp-<file>.s`の気になる部分。

一読して内容が把握できる，テストやデバッグもできるファイル類を添付したバグ報告を書くようにして下さい。曖昧な要望や断片的な記述では，バグの修正は難しくなり，開発陣のやる気も起きません。

コンパイラのバグによって問題が起きるということはあまりありませんが，GMP のコード自体がコンパイラの重箱の隅をつつくようなことは割とあります。

バグ報告が良いものであれば，その修正を行うべく最大限の努力を行いますが，そうでない場合は放置されても仕方ありません（もっと分かりやすく説明してくれとお願いすることになると思いますが）。

バグ報告は`gmp-bugs@gmplib.org`に送って下さい。

このマニュアルで不明瞭なところや大ウソがあったり，改善すべき文章があれば，上記アドレス宛にお知らせ下さい。

(訳注：日本語訳に関しての問題は`kouya.tomonori@sist.ac.jp`宛にお知らせ下さい。)

5 整数関数

本章では多倍長整数演算を行う GMP の関数群について解説します。これらの関数はmpz_から始まる名前が付いています。

GMP の多倍長整数はmpz_tというデータ型のオブジェクトとして保持されます。

5.1 初期化関数

多倍長整数演算を行う関数群は，全ての多倍長整数型のオブジェクトが初期化されているものとして扱います。下記のようにmpz_init関数を使うことで初期化が実行できます。

```
{
  mpz_t integ;
  mpz_init (integ);
  ...
  mpz_add (integ, ...);
  ...
  mpz_sub (integ, ...);

  /* Unless the program is about to exit, do ... */
  mpz_clear (integ);
}
```

上記の例で分かる通り，一度初期化した多倍長整数オブジェクトには何度でも新しい値を代入することができます。

`void mpz_init (`*mpz_t* `x)` [関数]
多倍長整数xを初期化し，値をゼロにセットします。

`void mpz_inits (`*mpz_t* `x`, ...`)` [関数]
NULL 文字を終端とするmpz_t型変数のリストを初期化し，全ての変数の値をゼロにセットします。

`void mpz_init2 (`*mpz_t* `x`, *mp_bitcnt_t* `n)` [関数]
n bit 長のxを初期化し，値をゼロにセットします。mpz_init関数やmpz_inits関数で一度初期化した変数に対しては，必要に応じて値を保持するために必要となる領域再確保を，GMP 側で自動で行いますので，この関数を呼び出す必要はありません。

この関数によって初期化時に確保されたn bit 長の変数xについても，足りなくなれば値保持に必要なだけ自動的に領域再確保を行いますが，最大長があらかじめ分かっている場合は領域再確保を避けることができます。

GMP では演算準備段階で，1 リム以上の領域を確保することがあります。これを避けたいのであれば，xに対して，nに 1 リム(mp_limb_t)分のビット数を加えておく必要があります。

`void mpz_clear (`*mpz_t* `x)` [関数]
xが確保していた領域を解放します。どんな手段で確保されたmpz_t型変数であっても利用可能です。

`void mpz_clears (`*mpz_t* `x`, ...`)` [関数]
NULL 文字を終端とするmpz_t型変数のリストが保持する領域全てを解放します。

`void mpz_realloc2` (*mpz_t* ***x***, *mp_bitcnt_t* ***n***) [関数]
変数*x*用に*n* bit 分のメモリ領域を再確保します。*x*が保持する値が代入できる長さであればそのまま保持されますが，足りない時にはゼロがセットされます。

GMP は必要に応じて自動的にメモリ領域の再確保を行いますので，この関数は基本的に不要なものです。繰り返し再領域確保が行われるようであったり，不要なメモリ領域をヒープに戻したいような状況でのみ有用な関数です。

5.2 代入関数

ここでは，一度初期化(see Section 5.1 [Initializing Integers], p. 32)された整数変数に新しい値を代入する関数群を解説します。

`void mpz_set` (*mpz_t* ***rop***, *const mpz_t* ***op***) [関数]
`void mpz_set_ui` (*mpz_t* ***rop***, *unsigned long int* ***op***) [関数]
`void mpz_set_si` (*mpz_t* ***rop***, *signed long int* ***op***) [関数]
`void mpz_set_d` (*mpz_t* ***rop***, *double* ***op***) [関数]
`void mpz_set_q` (*mpz_t* ***rop***, *const mpq_t* ***op***) [関数]
`void mpz_set_f` (*mpz_t* ***rop***, *const mpf_t* ***op***) [関数]
*op*の値を*rop*に代入します。

`mpz_set_d`, `mpz_set_q`，`mpz_set_f`関数は*op*の値を切り捨てて整数値に丸めます。

`int mpz_set_str` (*mpz_t* ***rop***, *const char* ****str***, *int* ***base***) [関数]
基数を*base*として表現された Null 文字終端子の C 文字列*str*を，*rop*にセットします。ホワイトスペース（空白，タブなど）が入っていても単に無視されます。

基数*base*は 2 から 62 まで指定することができます。これが 0 の場合は先頭文字列から基数を判断します。`0x`や`0X`から開始されていれば 16 進数，`0b`や`0B`であれば 2 進数，`0`なら 8 進数，それ以外はすべて 10 進数と解釈します。

基数が 36 以下であれば，大文字小文字の区別はせず，どちらも同じ値として解釈します。基数が 37 以上 62 以下であれば，大文字(A, B, ..., Z)は 10 から 35 までを表わし，小文字(a, b, ..., z)は 36 から 61 までの値であると解釈します。

この関数の返り値が 0 の時は，基数*base*の表現として正しい文字列であることを示します。1 文字でも不正であれば，−1 を返します。

`void mpz_swap` (*mpz_t* ***rop1***, *mpz_t* ***rop2***) [関数]
*rop1*と*rop2*の値を高速に交換します。

5.3 初期化代入関数

GMP では，変数の初期化と値の代入をまとめて実行する便利な関数(`mpz_init_set...`から始まる関数名)が用意されています。

下記に使い方の例を示します。

```
{
  mpz_t pie;
  mpz_init_set_str (pie, "3141592653589793238462643383279502884", 10);
  ...
  mpz_sub (pie, ...);
  ...
  mpz_clear (pie);
}
```

多倍長整数をmpz_init_set...関数で一度初期化すると，他の多倍長整数関数で利用可能な状態にできます。但し，一度初期化した変数に対しては実行しないようにして下さい。

void mpz_init_set (*mpz_t* **rop**, *const mpz_t* **op**) [関数]
void mpz_init_set_ui (*mpz_t* **rop**, *unsigned long int* **op**) [関数]
void mpz_init_set_si (*mpz_t* **rop**, *signed long int* **op**) [関数]
void mpz_init_set_d (*mpz_t* **rop**, *double* **op**) [関数]
*rop*をリム格納スペース付きで初期化し，*op*の値を初期値として代入します。

int mpz_init_set_str (*mpz_t* **rop**, *const char* ***str**, *int* **base**) [関数]
*rop*を初期化してmpz_set_str関数同様に文字列を変換して値をセットします(詳細は本文書の当該関数を参照のこと)。

文字列が基数*base*で表現される正しい値であれば 0 を返し，エラーになれば −1 を返します。エラーになったとしても*rop*の初期化は実行されます（リリース時にはmpz_clear関数を呼び出す必要あり）。

5.4 変換関数

この節では GMP の多倍長整数を C 標準のデータ型に変換する関数について解説します。GMP の多倍長整数に変換する関数についてはSection 5.2 [Assigning Integers], p. 33とSection 5.12 [I/O of Integers], p. 42で解説しています。

unsigned long int mpz_get_ui (*const mpz_t* **op**) [関数]
*op*の値をunsigned long型にして返します。

*op*の値がunsigned long型に収まり切れない場合は，収まる程度の小さい方の有効ビットだけを返すことになります。*op*の符号は無視されますので，必ず絶対値になります。

signed long int mpz_get_si (*const mpz_t* **op**) [関数]
*op*がsigned long int型に収まる場合は*op*の値がそのまま返ってきます。収まり切れない時には，*op*と同じ符号の，小さい方の有効桁数部分が返ってきます。

*op*がsigned long int型としては大きすぎる場合はあまり役に立つ結果にはなりませんが，収まる値になっているかどうかはmpz_fits_slong_p関数を使うことで判明します。

double mpz_get_d (*const mpz_t* **op**) [関数]
*op*の値をdouble型に切り捨てモード(rounding towards zero)で丸めて変換します。

変換の際に指数部が大きすぎる場合，返り値は実行環境によって変化します。無限大が使えるようなら無限大となります。ハードウェアによるオーバーフロー停止が起きるかどうかも環境次第です。

double mpz_get_d_2exp (*signed long int* ***exp**, *const mpz_t* **op**) [関数]
*op*の値をdouble型に切り捨てモード(rounding towards zero)で丸めて変換し，その指数部も切り離して返します。

返り値は必ず$0.5 \le |d| < 1$の範囲に収まる値（仮数部）になり，指数部の値は**exp*に格納されます。*op*がゼロであれば0.0 が返り値となり，**exp*には 0 が格納されます。

この関数は標準 C のfrexp関数と同じ機能を有しています(see Section “Normalization Functions” in *The GNU C Library Reference Manual*)。

char * mpz_get_str (*char* ***str**, *int* **base**, *const mpz_t* **op**) [関数]
*op*の値を基数*base*とする整数文字列に変換します。基数として使えるのは 2 から 62，もしくは −2 から −36 までの値です。

基数*base*が 2 から 36 までの場合は，数字と小文字を一桁分として使用し，−2 から −36 までの場合は数字と大文字が使用されます。基数が 37 から 62 の範囲では，数が大きい順に子文字↦大文字↦数字で一桁が表現されていきます(例えば*base*=62 の時は, 61 = z, ..., 36 = a, 35 = Z, ..., 10 = A, 9, 8, ..., 1, 0 となる)。

*str*が`NULL`ポインタであれば，返り値となる文字列は現時点におけるメモリ割り当て関数(see Chapter 13 [Custom Allocation], p. 91)を使って確保された`strlen(str)+1`バイト分の領域に格納され，NULL 終端子付きの文字列として表現されます。

*str*が`NULL`ポインタでなく，`mpz_sizeinbase (op, base) + 2`分のスペースがあれば，そこに返り値として格納されます。2 バイトの冗長分はマイナス符号と NULL 終端子用です。

返り値の文字列へのポインタは，割り当てられた領域か，*str*へのポインタとなります。

5.5 演算関数

`void mpz_add` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
`void mpz_add_ui` (*mpz_t* **rop**, *const mpz_t* **op1**, *unsigned long int* **op2**) [関数]
op1 + *op2* を実行し，結果を*rop*に格納します。

`void mpz_sub` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
`void mpz_sub_ui` (*mpz_t* **rop**, *const mpz_t* **op1**, *unsigned long int* **op2**) [関数]
`void mpz_ui_sub` (*mpz_t* **rop**, *unsigned long int* **op1**, *const mpz_t* **op2**) [関数]
op1 − *op2*を実行し，結果を*rop*に格納します。

`void mpz_mul` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
`void mpz_mul_si` (*mpz_t* **rop**, *const mpz_t* **op1**, *long int* **op2**) [関数]
`void mpz_mul_ui` (*mpz_t* **rop**, *const mpz_t* **op1**, *unsigned long int* **op2**) [関数]
op1 × *op2* を実行し，結果を*rop*に格納します。

`void mpz_addmul` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
`void mpz_addmul_ui` (*mpz_t* **rop**, *const mpz_t* **op1**, *unsigned long int* **op2**) [関数]
rop + *op1* × *op2* を実行し，結果を*rop*に格納します。

`void mpz_submul` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
`void mpz_submul_ui` (*mpz_t* **rop**, *const mpz_t* **op1**, *unsigned long int* **op2**) [関数]
rop − *op1* × *op2* を実行し，結果を*rop*に格納します。

`void mpz_mul_2exp` (*mpz_t* **rop**, *const mpz_t* **op1**, *mp_bitcnt_t* **op2**) [関数]
$op1 \times 2^{op2}$ を実行し，結果を*rop*に格納します。この演算は*op2*ビット分の左 bit シフトと同じです。

`void mpz_neg` (*mpz_t* **rop**, *const mpz_t* **op**) [関数]
−*op*を*rop*に格納します。

`void mpz_abs` (*mpz_t* **rop**, *const mpz_t* **op**) [関数]
*op*の絶対値を*rop*に格納します。

5.6 除算関数

割る数がゼロの時の商は定義されていません。除算関数や剰余関数(剰余べき乗関数mpz_powmやmpz_powm_uiも含む）にゼロ除算がセットされると，通常の C の整数演算と同様に，算術例外が発生します。

`void mpz_cdiv_q` (*mpz_t* `q`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`void mpz_cdiv_r` (*mpz_t* `r`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`void mpz_cdiv_qr` (*mpz_t* `q`, *mpz_t* `r`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`unsigned long int mpz_cdiv_q_ui` (*mpz_t* `q`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_cdiv_r_ui` (*mpz_t* `r`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_cdiv_qr_ui` (*mpz_t* `q`, *mpz_t* `r`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_cdiv_ui` (*const mpz_t* `n`, *unsigned long int* `d`) [関数]
`void mpz_cdiv_q_2exp` (*mpz_t* `q`, *const mpz_t* `n`, *mp_bitcnt_t* `b`) [関数]
`void mpz_cdiv_r_2exp` (*mpz_t* `r`, *const mpz_t* `n`, *mp_bitcnt_t* `b`) [関数]

`void mpz_fdiv_q` (*mpz_t* `q`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`void mpz_fdiv_r` (*mpz_t* `r`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`void mpz_fdiv_qr` (*mpz_t* `q`, *mpz_t* `r`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`unsigned long int mpz_fdiv_q_ui` (*mpz_t* `q`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_fdiv_r_ui` (*mpz_t* `r`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_fdiv_qr_ui` (*mpz_t* `q`, *mpz_t* `r`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_fdiv_ui` (*const mpz_t* `n`, *unsigned long int* `d`) [関数]
`void mpz_fdiv_q_2exp` (*mpz_t* `q`, *const mpz_t* `n`, *mp_bitcnt_t* `b`) [関数]
`void mpz_fdiv_r_2exp` (*mpz_t* `r`, *const mpz_t* `n`, *mp_bitcnt_t* `b`) [関数]

`void mpz_tdiv_q` (*mpz_t* `q`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`void mpz_tdiv_r` (*mpz_t* `r`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`void mpz_tdiv_qr` (*mpz_t* `q`, *mpz_t* `r`, *const mpz_t* `n`, *const mpz_t* `d`) [関数]
`unsigned long int mpz_tdiv_q_ui` (*mpz_t* `q`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_tdiv_r_ui` (*mpz_t* `r`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_tdiv_qr_ui` (*mpz_t* `q`, *mpz_t* `r`, *const mpz_t* `n`, [関数]
unsigned long int `d`)
`unsigned long int mpz_tdiv_ui` (*const mpz_t* `n`, *unsigned long int* `d`) [関数]
`void mpz_tdiv_q_2exp` (*mpz_t* `q`, *const mpz_t* `n`, *mp_bitcnt_t* `b`) [関数]
`void mpz_tdiv_r_2exp` (*mpz_t* `r`, *const mpz_t* `n`, *mp_bitcnt_t* `b`) [関数]

nをdで割り，qに商を，rに剰余を格納します。2exp付きの関数は，$d = 2^b$ で除算を行います。丸めの方法は下記の通り 3 種類あり，使用アプリケーションごとに使い分けができます。

- cdiv除算関数はqを$+\infty$ 方向に丸め，rはdとは逆符号の値にします。cは“ceil”(天井関数)を意味します。
- fdiv除算関数は，qを$-\infty$ 方向に丸め，rはdと同符号の値にします。fは“floor”(床関数)を意味します。

- tdiv除算関数は，qをゼロ方向に切り捨て，rはnと同符号の値にします。tは“truncate”(切り捨て)を意味します。

上記の除算関数は必ず，qとrは$n = qd + r$ を，rは$0 \le |r| < |d|$ を満足するように値を決めます。

q除算関数は商のみ，r除算関数は剰余のみ，qr関数は商と剰余の両方を計算します。qr関数では，qとrに同じ変数を指定することはできません。

ui除算関数は返り値が剰余となります。従って，これらの関数はdiv_ui除算関数と同じ返り値となります。tdivとcdiv関数については，剰余がマイナスになることもありますが，その場合は剰余の絶対値を返り値とします。

2exp除算関数は，割る数（除数）が2^b となりますので，実際の計算は右シフトとビットマスクで行います。丸め方は他の除算関数と同じです。

正のnに対しては，mpz_fdiv_q_2exp関数もmpz_tdiv_q_2exp関数も，同じビット単位の右シフトだけを実行します。負のnの場合，mpz_fdiv_q_2exp関数はビット演算関数と同様にnを2 の補数として扱って効率的に右シフトを行い，mpz_tdiv_q_2exp関数はnを符号と絶対値だけのものとして扱います。

`void mpz_mod` (*mpz_t* **r**, *const mpz_t* **n**, *const mpz_t* **d**) [関数]
`unsigned long int mpz_mod_ui` (*mpz_t* **r**, *const mpz_t* **n**, *unsigned long int* **d**) [関数]
n mod dを求め，rにセットします。割る数の符号は無視されますので，結果は必ず非負整数となります。

mpz_mod_ui関数は，上記のmpz_fdiv_r_ui関数と同一のもので，rに剰余がセットされます。返り値が必要な場合はmpz_fdiv_ui関数を使ってみて下さい。

`void mpz_divexact` (*mpz_t* **q**, *const mpz_t* **n**, *const mpz_t* **d**) [関数]
`void mpz_divexact_ui` (*mpz_t* **q**, *const mpz_t* **n**, *unsigned long* **d**) [関数]
n/dの値をqに格納します。nがdで割り切れる場合にのみ，正しい商を返します。

これらの関数は，他の除算関数よりかなり高速なので，正確な除算が可能であると分かっている場合，例えば有理数を小さい値で約分するケースには最適です。

`int mpz_divisible_p` (*const mpz_t* **n**, *const mpz_t* **d**) [関数]
`int mpz_divisible_ui_p` (*const mpz_t* **n**, *unsigned long int* **d**) [関数]
`int mpz_divisible_2exp_p` (*const mpz_t* **n**, *mp_bitcnt_t* **b**) [関数]
nが正確にdで割り切れる場合は非ゼロを返します。mpz_divisible_2exp_p関数は2^b で割り切れるかどうかで判断します。

$n = qd$ を満足する整数dが存在していれば，nはdで割り切れることになります。他の除算関数とは違い，$d = 0$ でも実行でき，この場合はゼロだけがゼロで割り切れるものとして扱います。

`int mpz_congruent_p` (*const mpz_t* **n**, *const mpz_t* **c**, *const mpz_t* **d**) [関数]
`int mpz_congruent_ui_p` (*const mpz_t* **n**, *unsigned long int* **c**, *unsigned long int* **d**) [関数]
`int mpz_congruent_2exp_p` (*const mpz_t* **n**, *const mpz_t* **c**, *mp_bitcnt_t* **b**) [関数]
nがc mod dに対して合同であれば非ゼロを返します。mpz_congruent_2exp_pはdの代わりに mod 2^b で判断します。

$n = c + qd$ を満足する整数qが存在していれば，nは$c \bmod d$に対して合同であると言えます。他の除算関数とは異なり，$d = 0$ であっても実行でき，この場合はnとcが同一の値である時のみ，mod 0 に対して合同であると判断します。

5.7 べき乗関数

`void mpz_powm (`*mpz_t* **rop**, *const mpz_t* **base**, *const mpz_t* **exp**, *const mpz_t* **mod**`)` [関数]

`void mpz_powm_ui (`*mpz_t* **rop**, *const mpz_t* **base**, *unsigned long int* **exp**, *const mpz_t* **mod**`)` [関数]

$base^{exp} \bmod mod$ を計算し，*rop*に格納します。

$base^{-1} \bmod mod$ の逆数が存在する場合，*exp*が負の値でも問題ありません(Section 5.9 [Number Theoretic Functions], p. 39の`mpz_invert`参照)。逆数が存在しない場合はゼロ除算が発生します。

`void mpz_powm_sec (`*mpz_t* **rop**, *const mpz_t* **base**, *const mpz_t* **exp**, *const mpz_t* **mod**`)` [関数]

$base^{exp} \bmod mod$ を計算して*rop*に格納します。

$exp > 0$ かつ*mod*が奇数であるという前提が必要な関数です。

この関数は，同時刻に同サイズの引数に対して同時キャッシュアクセスするために作られています。関数発行時に引数は同じ場所にあり，計算機の状態も同一である，ということも前提となっています。この関数は暗号化を目的としており，サイドチャンネル攻撃に対する耐性が求められるからです。

`void mpz_pow_ui (`*mpz_t* **rop**, *const mpz_t* **base**, *unsigned long int* **exp**`)` [関数]

`void mpz_ui_pow_ui (`*mpz_t* **rop**, *unsigned long int* **base**, *unsigned long int* **exp**`)` [関数]

$base^{exp}$ を計算して*rop*に格納します。0^0 の場合は 1 を返します。

5.8 べき乗根関数

`int mpz_root (`*mpz_t* **rop**, *const mpz_t* **op**, *unsigned long int* **n**`)` [関数]

$\lfloor \sqrt[n]{op} \rfloor$, 即ち，*op*の*n*乗根を求めて整数部のみ取り出し，*rop*に格納します。計算に丸めが発生しなければ，つまり，*op*がピッタリ*op*の*n*乗根であればそのまま*rop*に格納します。

`void mpz_rootrem (`*mpz_t* **root**, *mpz_t* **rem**, *const mpz_t* **u**, *unsigned long int* **n**`)` [関数]

$\lfloor \sqrt[n]{u} \rfloor$, 即ち，*u*の*n*乗根を求めて整数部のみ取り出し，*root*に格納します。*rem*には剰余，即ち，$(u - root^n)$ が格納されます。

`void mpz_sqrt (`*mpz_t* **rop**, *const mpz_t* **op**`)` [関数]

$\lfloor \sqrt{op} \rfloor$, 即ち，*op*の平方根の整数部のみ取り出して*rop*に格納します。

`void mpz_sqrtrem (`*mpz_t* **rop1**, *mpz_t* **rop2**, *const mpz_t* **op**`)` [関数]

$\lfloor \sqrt{op} \rfloor$, 即ち，`mpz_sqrt`関数同様，*op*の平方根を求め，その整数部を*rop1*に格納します。*rop2*には剰余，即ち，$(op - rop1^2)$ が格納されます。もし*op*がピッタリ平方根であれば，この値はゼロになります。

*rop1*と*rop2*が同じ変数の時は結果は不定になります。

`int mpz_perfect_power_p` (*const mpz_t* **op**) [関数]
*op*がべき乗数，即ち，$b > 1$ を満足する整数a とb が存在して$op = a^b$ となっていれば，非ゼロを返します。

0 も 1 も完全なべき乗数とみなします。*op*は負数でもよく，この場合は奇数べき乗かどうかを確認するだけとなります。

`int mpz_perfect_square_p` (*const mpz_t* **op**) [関数]
*op*が完全な平方数，即ち，*op*の平方根が整数である場合は非ゼロを返します。0 と 1 は完全な平方数とみなします。

5.9 数論関数

`int mpz_probab_prime_p` (*const mpz_t* **n**, *int* **reps**) [関数]
*n*が素数かどうかを決定します。返り値が 2 の場合は*n*は確実に素数，1 の場合は*n*は確率的に素数（確定的ではない），0 の時は確実に非素数であることを意味します。

この関数は複数の除算を伴う，Miller-Rabin 確率的素数テストを行います。高次の*reps*値を使うことで，非素数（合成数）を「確率的素数」と認識してしまう率を下げることができ，その誤認識率は4^{-reps} 以下です。適当な*reps*は大体 15 〜 50 です。

`void mpz_nextprime` (*mpz_t* **rop**, *const mpz_t* **op**) [関数]
*op*より次に大きい素数を*rop*に格納します。

この関数は素数判定に確率的アルゴリズムを使用します。実用的には十分で，合成数を選んでしまう可能性は非常に小さいと思われます。

`void mpz_gcd` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
op1 と*op2*の GCD（Greatest Common Divisor, 最大公約数)を求めて*rop*に格納します。入力値に負の値が入っていても，結果は常に正数になります。どちらもゼロであれば，$gcd(0,0) = 0$ となります。

`unsigned long int mpz_gcd_ui` (*mpz_t* **rop**, *const mpz_t* **op1**, *unsigned long int* **op2**) [関数]
op1 と*op2*の GCD を計算します。*rop*が`NULL`でなければ，結果はここに格納されます。

GCD が`unsigned long int`の範囲に収まるものであれば，これが返り値になります。収まらないようであれば 0 が返り値となり，結果は*op1*に格納されます。*op2*がゼロでなければ，この計算結果は常に`unsigned long int`に収まる筈です。

`void mpz_gcdext` (*mpz_t* **g**, *mpz_t* **s**, *mpz_t* **t**, *const mpz_t* **a**, *const mpz_t* **b**) [関数]
*a*と*b*の最大公約数を求めて*g*に格納し，更に$as + bt = g$ を満足する係数*s*と*t*を求めて格納します。*g*は常に正数で，*a*と*b*が負数であっても関係ありません（両方ゼロならばゼロになります)。*s*と*t*の値は通常，$|s| < |b|/(2g)$ かつ$|t| < |a|/(2g)$ を満足するように定まり，結果として*s*と*t*は一意になります。但し，下記の例外があります。

$|a| = |b|$ の時は，$s = 0$，$t = sgn(b)$ となります。

それ以外では，$b = 0$ もしくは$|b| = 2g$ の時は$s = sgn(a)$ ，$a = 0$ もしくは$|a| = 2g$ の時は$t = sgn(b)$ となります。$g = |b|$，即ち，b がaを割り切ったり，$a = b = 0$ となる時には，必ず$s = 0$ となります。

*t*が`NULL`の時は*t*の計算を行いません。

`void mpz_lcm` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
`void mpz_lcm_ui` (*mpz_t* **rop**, *const mpz_t* **op1**, *unsigned long* **op2**) [関数]
op1 と*op2*の LCM(Least Common Multiple，最小公倍数)を計算し，*rop*に格納します。*op1*と*op2*の符号とは無関係に，*rop*は常に正の値になります。*op1*か*op2*のどちらかがゼロの場合は，*rop*もゼロになります。

`int mpz_invert` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
*op1*のモジュロ*op2*演算の逆数を求め，*rop*に格納します。逆数が存在していれば非ゼロを返し，*rop*は$0 \le rop < |op2|$ ($|op2| = 1$ の時は$rop = 0$ となる，つまりゼロ環に退化）を満たすように決まります。逆数が存在していない時にはゼロを返し，*rop*は不定になります。*op2*がゼロの時のことはこの関数では考慮していません。

`int mpz_jacobi` (*const mpz_t* **a**, *const mpz_t* **b**) [関数]
Jacobi 記号$\left(\frac{a}{b}\right)$ の計算を行います。これは*b*が奇数の時のみ定義された値です。

`int mpz_legendre` (*const mpz_t* **a**, *const mpz_t* **p**) [関数]
Legendre 記号$\left(\frac{a}{p}\right)$ の計算を行います。*p*が正の素数である時のみ定義されており，Jacobi 記号と同じ値になります。

`int mpz_kronecker` (*const mpz_t* **a**, *const mpz_t* **b**) [関数]
`int mpz_kronecker_si` (*const mpz_t* **a**, *long* **b**) [関数]
`int mpz_kronecker_ui` (*const mpz_t* **a**, *unsigned long* **b**) [関数]
`int mpz_si_kronecker` (*long* **a**, *const mpz_t* **b**) [関数]
`int mpz_ui_kronecker` (*unsigned long* **a**, *const mpz_t* **b**) [関数]
Kronecker 拡大付きの Jacobi 記号$\left(\frac{a}{b}\right)$ の計算を行います。*a* が奇数の時は$\left(\frac{a}{2}\right) = \left(\frac{2}{a}\right)$，偶数の時は$\left(\frac{a}{2}\right) = 0$ になります。

*b*が奇数の時は，Jacobi 記号と Kronecker 記号は同じものになります。`mpz_kronecker_ui`以下の関数は Jacobi 記号の混合精度計算に利用できます。

詳細については Henri Cohen のテキストの 1.4.2 節(see 付録 B [References], p. 128)か，数論のテキストをご覧ください。`mpz_kronecker_ui`関数を使ったサンプルプログラム`demos/qcn.c`も参照して下さい。

`mp_bitcnt_t mpz_remove` (*mpz_t* **rop**, *const mpz_t* **op**, *const mpz_t* **f**) [関数]
*op*から因数*f*を全て抜き取り，その結果を*rop*に格納します。返り値は，取り除かれた回数です。

`void mpz_fac_ui` (*mpz_t* **rop**, *unsigned long int* **n**) [関数]
`void mpz_2fac_ui` (*mpz_t* **rop**, *unsigned long int* **n**) [関数]
`void mpz_mfac_uiui` (*mpz_t* **rop**, *unsigned long int* **n**, *unsigned long int* **m**) [関数]
*n*の階数を求め，*rop*に格納します。`mpz_fac_ui`関数は階数$n!$を，`mpz_2fac_ui`関数は 2 重階数$n!!$を，`mpz_mfac_uiui`は*m*重階数$n!^{(m)}$ を計算します。

`void mpz_primorial_ui` (*mpz_t* **rop**, *unsigned long int* **n**) [関数]
*n*の素数階乗，即ち*n*以下の全ての正の素数の積を求め，*rop*に格納します。

`void mpz_bin_ui` (*mpz_t* **rop**, *const mpz_t* **n**, *unsigned long int* **k**) [関数]
`void mpz_bin_uiui` (*mpz_t* **rop**, *unsigned long int* **n**, *unsigned long int* **k**) [関数]
2 項係数$\binom{n}{k}$ を求め，*rop*に格納します。*n*が負数の時は$\binom{-n}{k} = (-1)^k \binom{n+k-1}{k}$ を計算します。Knuth 本の volume 1，1.2.6 節の part G を参照して下さい。

`void mpz_fib_ui` (*mpz_t* **fn**, *unsigned long int* **n**) [関数]
`void mpz_fib2_ui` (*mpz_t* **fn**, *mpz_t* **fnsub1**, *unsigned long int* **n**) [関数]
`mpz_fib_ui`関数は*fn*にF_n, 即ち，*n*項目の Fibonacci 数を計算して格納します。`mpz_fib2_ui`関数は*fn*にF_n を，*fnsub1*にF_{n-1} を計算して格納します。

これらの関数は，単独の Fibonacci 数列の値を計算するためのものです。数列全体を欲しい時には，`mpz_fib2_ui`関数から出発し，それ以降は$F_{n+1} = F_n + F_{n-1}$ を繰り返し実行すると高速に数列全体が得られます。

`void mpz_lucnum_ui` (*mpz_t* **ln**, *unsigned long int* **n**) [関数]
`void mpz_lucnum2_ui` (*mpz_t* **ln**, *mpz_t* **lnsub1**, *unsigned long int* **n**) [関数]
`mpz_lucnum_ui`関数は*ln*にL_n, 即ち, Lucas 数の*n*番目の値を求めて格納します。`mpz_lucnum2_ui`関数は*ln*にL_n を，*lnsub1*にL_{n-1} を求めて格納します。

これらの関数は単独の Lucus 数を計算するためのものです。数列全体を欲しい時には，`mpz_lucnum2_ui`関数から出発し，それ以降は$L_{n+1} = L_n + L_{n-1}$ を繰り返し実行すると高速に数列全体が得られます。

Fibonacci 数や Lucas 数は漸化式で定義されていますので，`mpz_fib2_ui`関数と`mpz_lucnum2_ui`関数を一度に呼び出す必要はありません。Fibonacci 数から Lucas 数への変換を行う方法はSection 15.7.5 [Lucas Numbers Algorithm], p. 114に書いておきました。その逆の変換は自明です。

5.10 比較関数

`int mpz_cmp` (*const mpz_t* **op1**, *const mpz_t* **op2**) [Function]
`int mpz_cmp_d` (*const mpz_t* **op1**, *double* **op2**) [Function]
`int mpz_cmp_si` (*const mpz_t* **op1**, *signed long int* **op2**) [マクロ]
`int mpz_cmp_ui` (*const mpz_t* **op1**, *unsigned long int* **op2**) [マクロ]
op1 と*op2*を比較し，$op1 > op2$ ならば正数を, $op1 = op2$ ならばゼロを, $op1 < op2$ ならば負数を返します。

`mpz_cmp_ui`と`mpz_cmp_si`はマクロとなっており，引数を複数回評価します。`mpz_cmp_d`関数は無限大との比較も可能ですが，非数(NaN)は正常に扱えません。

`int mpz_cmpabs` (*const mpz_t* **op1**, *const mpz_t* **op2**) [Function]
`int mpz_cmpabs_d` (*const mpz_t* **op1**, *double* **op2**) [Function]
`int mpz_cmpabs_ui` (*const mpz_t* **op1**, *unsigned long int* **op2**) [Function]
op1 と*op2*を絶対値として比較し，$|op1| > |op2|$ ならば正数を，$|op1| = |op2|$ ならばゼロを，$|op1| < |op2|$ ならば負数を返します。

`mpz_cmpabs_d`関数は無限大との比較も可能ですが，非数は正常に扱えません。

`int mpz_sgn` (*const mpz_t* **op**) [マクロ]
$op > 0$ の時は+1 を, $op = 0$ の時はゼロを, $op < 0$ の時は−1 を返します。

この関数はマクロで実装されていますので，引数を複数回評価します。

5.11 ビット操作関数

ここで解説する関数群は，2 の補数演算として利用できます（実際には符号付きで実装されていますが）。最小有効 bit は 0 番目となります。

`void mpz_and` (*mpz_t* **rop**, *const mpz_t* **op1**, *const mpz_t* **op2**) [関数]
*op1*と*op2*のビット論理積を*rop*に格納します。

`void mpz_ior` (*mpz_t rop*, *const mpz_t op1*, *const mpz_t op2*) [関数]
*rop*に*op1*と*op2*のビット論理和を格納します。

`void mpz_xor` (*mpz_t rop*, *const mpz_t op1*, *const mpz_t op2*) [関数]
*op1*と*op2*のビット排他的論理和を格納します。

`void mpz_com` (*mpz_t rop*, *const mpz_t op*) [関数]
*op*の 1 の補数を*rop*に格納します。

`mp_bitcnt_t mpz_popcount` (*const mpz_t op*) [関数]
$op \geq 0$ の時には*op*の 2 進表現における"1"ビットの数を返します。$op < 0$ の時は，"1"の数は無限大となり，返り値は`mp_bitcnt_t`の最大値となります。

`mp_bitcnt_t mpz_hamdist` (*const mpz_t op1*, *const mpz_t op2*) [関数]
op1 と*op2* が共に≥ 0 か< 0 である時，この 2 数のハミング距離を返します。ハミング距離とは，*op1* と*op2*の 2 進表現において異なるビット数のことです。片方が≥ 0 でもう片方が< 0 の時は，異なるビット数は無限大となり，返り値は`mp_bitcnt_t`の最大数となります。

`mp_bitcnt_t mpz_scan0` (*const mpz_t op*, *mp_bitcnt_t starting_bit*) [関数]
`mp_bitcnt_t mpz_scan1` (*const mpz_t op*, *mp_bitcnt_t starting_bit*) [関数]
*op*を*starting_bit*ビット目から，有効ビット数の大きい方向に調べていき，最初の 0，もしくは 1 が見つかったら終了します。返り値は発見したビット数の位置です。

*starting_bit*ビット目における値が既に発見されたものである時には，*starting_bit* が返されます。

指定値のビットが見つからなかった時には`mp_bitcnt_t`の最大値が返されます。`mpz_scan0`関数の場合は負数の最後，`mpz_scan1`関数の場合は非負数の最後を過ぎたところでこれが発生します。

`void mpz_setbit` (*mpz_t rop*, *mp_bitcnt_t bit_index*) [関数]
*rop*の*bit_index*ビット目に 1 を代入します。

`void mpz_clrbit` (*mpz_t rop*, *mp_bitcnt_t bit_index*) [関数]
*rop*の*bit_index*ビット目に 0 を代入します。

`void mpz_combit` (*mpz_t rop*, *mp_bitcnt_t bit_index*) [関数]
*rop*の*bit_index*ビット目を反転します。

`int mpz_tstbit` (*const mpz_t op*, *mp_bitcnt_t bit_index*) [関数]
*op*の*bit_index*ビット目を調べてそのビット値(0 もしくは 1)を返します。

5.12 入出力関数

ここでは stdio ストリームに対して`mpz`数の入出力を行う関数群を解説します。*stream*引数に*NULL*が渡されると，入力は`stdin`から，出力は`stdout`に行います。

この入出力関数を使うときには，`gmp.h`より前に`stdio.h`をインクルードしておいて下さい。`gmp.h`でこの入出力関数のプロトタイプ宣言が使えるようになります。

Chapter 10 [Formatted Output], p. 73, とChapter 11 [Formatted Input], p. 78も併せて参照して下さい。

`size_t mpz_out_str` (*FILE *stream*, *int base*, *const mpz_t op*) [関数]
*op*を stdio ストリーム*stream*に，基数*base*の文字列として出力します。基数は 2 以上 62 以下，もしくは，−36 以上 −2 以下に設定できます。

*base*が 2 以上 36 以下の時は，数字，その次に，小文字が使われます。−36 以上 −2 以下の時は，数字，次に，大文字が使われます。37 以上 62 以下の時には，数字，次に大文字，最後に小文字が使われます。

返り値は出力したバイト数です。エラーが起こった時には 0 が返ります。

`size_t mpz_inp_str` (*mpz_t rop*, *FILE *stream*, *int base*) [関数]
ホワイトスペースを先頭に含む基数*base*の文字列を stdio ストリーム*stream*から読み取り，*rop*に格納します。

*base*は 2 以上 62 以下に設定できます。*base*が 0 の場合は，先頭文字で基数を判断します。`0x`や`0X`は 16 進数，`0b`や`0B`は 2 進数，`0`は 8 進数，特に指定がないか，これ以外の指定があれば 10 進数と判断します。

36 以下の基数の場合，大文字小文字の区別は行わず，全て同じ値とみなします。37 以上 62 以下の基数の場合は，大文字は 10 〜 35，小文字は 36 〜 61 と見なします。

返り値は読み込んだバイト数で，エラーが発生した時には 0 が返ります。

`size_t mpz_out_raw` (*FILE *stream*, *const mpz_t op*) [関数]
2 進形式の値として，stdio ストリーム*stream*に*op*を出力します。整数はどの環境でも利用可能な形式で出力され，4 バイトのサイズ情報にリムの列が続きます。サイズやリムは有効桁数の小さい順に（即ち，ビッグエンディアンで）出力されます。

出力結果は`mpz_inp_raw`関数で読み取ることができます。

返り値は書き出したバイト数で，エラーが発生した時にはゼロが返ります。

この出力結果は，GMP 1 では 32bit 環境と 64bit 環境の互換性の問題があり，`mpz_inp_raw`関数で読み取ることができなくなりました。

`size_t mpz_inp_raw` (*mpz_t rop*, *FILE *stream*) [関数]
`mpz_out_raw`関数で書き込んだフォーマットに従って stdio ストリーム*stream*からの入力を行い，その結果を*rop*に格納します。返り値は読み込んだバイト数で，エラーが発生した時には 0 が返ります。

この関数は`mpz_out_raw`関数の出力を読み取ることができます。GMP 1 では，32bit 環境と 64bit 環境の食い違いを正すための変更が必要になります。

5.13 乱数関数

GMP の乱数関数は，2 つのグループに分類されます。一つは古い関数で，グローバル変数に乱数の状態を保存するタイプ，もう一つは進化した新しいグループで，読み書きされる状態変数をパラメータとして受け取るタイプです。利用方法や使うべきではない乱数関数についての詳細はChapter 9 [Random Number Functions], p. 71を参照して下さい。

`void mpz_urandomb` (*mpz_t rop*, *gmp_randstate_t state*, *mp_bitcnt_t n*) [関数]
0 以上$2^n - 1$ 以下の一様整数乱数を生成します。

この関数を使う前に，変数*state*は`gmp_randinit`関数群のどれか一つで初期化されていなければなりません(Section 9.1 [Random State Initialization], p. 71)。

void mpz_urandomm (*mpz_t* **rop**, *gmp_randstate_t* **state**, *const mpz_t* **n**) [関数]
0 以上$n-1$ 以下の一様整数乱数を生成します。

この関数を使う前に，変数*state*は`gmp_randinit`関数群のどれか一つで初期化されていなければなりません(Section 9.1 [Random State Initialization], p. 71)。

void mpz_rrandomb (*mpz_t* **rop**, *gmp_randstate_t* **state**, *mp_bitcnt_t* **n**) [関数]
0 と 1 がランダムに並ぶ長い 2 進表現列を生成します。関数やアルゴリズムに潜む発現しづらいバグを見つけるために役立つ乱数です。生成される乱数の範囲は2^{n-1} 以上，2^n-1 以下です。

この関数を使う前に，変数*state*は`gmp_randinit`関数群のどれか一つで初期化されていなければなりません(Section 9.1 [Random State Initialization], p. 71)。

void mpz_random (*mpz_t* **rop**, *mp_size_t* **max_size**) [関数]
*max_size*リムを使用する整数乱数を生成します。但し，この乱数はランダム性に関しては保証されません。負の乱数を生成したい時には*max_size*を負数に設定します。

この関数は廃止予定です。`mpz_urandomb`関数か，`mpz_urandomm`関数を使って下さい。

void mpz_random2 (*mpz_t* **rop**, *mp_size_t* **max_size**) [関数]
0 と 1 がランダムに並ぶ，*max_size*リム長の 2 進表現の整数を生成します。関数やアルゴリズムに潜む発現しづらいバグを見つけるために役立つ乱数です。負の乱数を生成したい時には*max_size*を負数に設定します。

この関数は廃止予定です。`mpz_rrandomb`関数を使って下さい。

5.14 整数のインポートとエクスポート

`mpz_t`型の変数は，ここで述べる関数を使って任意長のバイナリデータと相互に変換することができます。

void mpz_import (*mpz_t* **rop**, *size_t* **count**, *int* **order**, *size_t* **size**, *int* **endian**, *size_t* **nails**, *const void* ***op**) [関数]
*op*におけるワードデータの配列を*rop*にインポートして格納します。

パラメータはデータ形式を規定しています。1 ワード*size*バイトの*count*個分のワードが飲み込まれます。*order*が 1 の場合は最大有効ワードから，-1 の場合は最小有効ワードから読み込まれます。ワードはそれぞれ*endian*の値に従い，1 の場合は最大有効バイトから，-1 の場合は最小有効バイトから，0 の場合はホスト CPU のエンディアンに従って読み込まれます。各ワードのうち最大有効*nails*ビット分は読み飛ばされますので，0 の場合は全てのワードが読み込まれることになります。

このデータに対しては符号は扱いません。*rop*は基本的には正数となります。符号についてはアプリケーション側で，例えば`mpz_neg`関数を使って付与して下さい。

*op*についてのデータアラインメントは関係ありませんので，どんなアドレスでも利用可能です。

以下，`unsigned long`型の配列に入っているデータを最大有効要素から，ホストのバイトオーダーで値を読み出して変換する例を示します。

```
unsigned long  a[20];
/*  z と aを初期化 */
mpz_import (z, 20, 1, sizeof(a[0]), 0, 0, a);
```

この例では，sizeofバイト分のデータを全て利用しており，unsigned long型が利用できる環境であれば実行可能です。例外は Cray ベクトルマシンで，short型もint型も(sizeofの返り値に従って）8 バイトで格納されますが，そのうち 32bit もしくは 46bit 分しか利用しません。*nails*が利用できる時にはこの分をカウントして8*sizeof(int)-INT_BIT bit 分だけ渡すようにできます。

void * mpz_export (*void *rop*, *size_t *countp*, *int order*, *size_t size*, *int endian*, *size_t nails*, *const mpz_t op*) [関数]

*rop*に*op*のワードデータを詰め込みます。

パラメータは書き出されるデータ形式を規定しています。1 ワード*size*バイトの*count*個分のワードが書き出されます。*order*が 1 の場合は最大有効ワードから，-1 の場合は最小有効ワードから書き出されます。ワードはそれぞれ*endian*の値に従い，1 の場合は最大有効バイトから，-1 の場合は最小有効バイトから，0 の場合はホスト CPU のエンディアンに従って書き出されます。各ワードのうち最大有効*nails*ビット分は利用されず，ゼロが詰め込まれます。*nails*が 0 場合は全てのワードが書き出されます。

書き出されるワード数は*countp*に格納され，*countp*がNULLの場合はワード数をカウントしません。*rop*は書き出すデータが入る分確保されていなければなりません。*rop*がNULLの場合は現状の GMP のメモリ割り当て関数(see Chapter 13 [Custom Allocation], p. 91)を使って必要なだけ配列が確保されます。どちらのケースでも，返り値は*rop*か，確保されたメモリブロックへのポインタになります。

*op*が非ゼロの場合は，書き出される最大有効ワードも非ゼロになります。*op*がゼロであれば，返されるカウントはゼロで，*rop*には何も書き出されません。更に*rop*が*NULL*であれば，メモリブロックは確保されず，*NULL*が返されます。

*op*の符号は無視され，絶対値だけがエクスポートされます。アプリケーションが側でmpz_sgn関数で符号を確認し，必要であればそれを格納しておいて下さい。(see Section 5.10 [Integer Comparisons], p. 41)

*rop*におけるデータアラインメントの制限はありませんので，どんなアドレスでも利用可能です。

アプリケーション側で必要なメモリ空間を確保するには，次のように必要サイズを計算します。mpz_sizeinbase関数は常に最小 1 は返してきますので，このcountも最小 1 となり，たとえzがゼロで実際にはメモリスペースを必要としない場合でも，malloc(0)となる問題は発生しません。

```
numb = 8*size - nail;
count = (mpz_sizeinbase (z, 2) + numb-1) / numb;
p = malloc (count * size);
```

5.15 その他の関数

int mpz_fits_ulong_p (*const mpz_t op*) [関数]
int mpz_fits_slong_p (*const mpz_t op*) [関数]
int mpz_fits_uint_p (*const mpz_t op*) [関数]
int mpz_fits_sint_p (*const mpz_t op*) [関数]
int mpz_fits_ushort_p (*const mpz_t op*) [関数]
int mpz_fits_sshort_p (*const mpz_t op*) [関数]

*op*がそれぞれ，unsigned long int, signed long int, unsigned int, signed int, unsigned short int, signed short int型に合致する時のみ非ゼロを返します。

`int mpz_odd_p` (*const mpz_t op*) [マクロ]
`int mpz_even_p` (*const mpz_t op*) [マクロ]
それぞれ，*op*が偶数か奇数かを判断します。イエスの時は非ゼロ，ノーの時はゼロが返ります。このマクロは引数を複数回評価します。

`size_t mpz_sizeinbase` (*const mpz_t op, int base*) [関数]
*op*のサイズを，基数*base*の桁数換算で返します。*base*は 2 以上 62 以下の整数です。*op*の符号は無視され，絶対値のみ使用します。結果は桁数そのものか，大きすぎる時は 1 を返します。*base*が 2 のべき乗の時は結果は常に桁数そのものになります。*op*がゼロの時は常に 1 が返ってきます。

この関数は，*op*を文字列に変換する時にどのぐらいのスペースを要するのかを知りたい時に使います。確保されるサイズは，`mpz_sizeinbase`関数の返り値より 2 文字分だけ大きいのが普通で，これはマイナス符号と NULL 終端子の部分になります。

`mpz_sizeinbase(op,2)`という使い方をすると，*op*の最大有効ビットを 1 からカウントしていきます。(ビット演算関数では 0 からカウントします。(See Section 5.11 [Logical and Bit Manipulation Functions], p. 41)

5.16 特殊目的の関数

この節で述べる関数は，様々な特殊な目的のために用意されています。普通はアプリケーションで利用することはないでしょう。

`void mpz_array_init` (*mpz_t integer_array, mp_size_t array_size,* [関数]
mp_size_t fixed_num_bits)
この関数は廃止予定です。使用しないで下さい。

`void * _mpz_realloc` (*mpz_t integer, mp_size_t new_alloc*) [関数]
*integer*用のメモリ空間を*new_alloc*リムに変更します。*interger*に入っている値はそのまま保持されますが，メモリ空間が足りないようなら 0 になります。返り値は無用のものなので無視しましょう。

この関数を使うよりは`mpz_realloc2`関数の方がいいでしょう。`mpz_realloc2`関数もこの`_mpz_realloc`関数も同じようにメモリ空間の付け替えが可能ですが，`_mpz_realloc`関数の方は，リムのサイズの指定ができます。

`mp_limb_t mpz_getlimbn` (*const mpz_t op, mp_size_t n*) [関数]
*op*の*n*番目のリムを返します。*op*の符号は無視され，絶対値だけを使用します。最小有効リムは 0 番目です。

`mpz_size`関数を使うと，*op*が何リムから構成されているか分かります。*n*が 0 以上`mpz_size(op)-1`リムの範囲外にある時には，`mpz_getlimbn`関数はゼロを返します。

`size_t mpz_size` (*const mpz_t op*) [関数]
*op*を構成するリム数を返します。*op*がゼロであれば，返り値もゼロになります。

`const mp_limb_t * mpz_limbs_read` (*const mpz_t x*) [関数]
*x*の絶対値を構成するリム配列へのポインタを返します。配列の大きさは`mpz_size(x)`で分かります。この関数は読み取りしかできません。

`mp_limb_t * mpz_limbs_write` (*mpz_t x, mp_size_t n*) [関数]
`mp_limb_t * mpz_limbs_modify` (*mpz_t x, mp_size_t n*) [関数]
書き込み可能なリム配列へのポインタを返します。配列は必要に応じて，*n*リム分入るように再確保されます。この関数を実行する際には$n > 0$でなければなりません。`mpz_limbs_modify`関数

はxの古い絶対値はそのまま残しつつ，再確保されたポインタを返します。mpz_limbs_write関数の場合は古い値は破壊され，何が入っているか分からない配列へのポインタを返します。

void mpz_limbs_finish (*mpz_t x*, *mp_size_t s*) [関数]
xの内部サイズフィールドの値を書き換えます。mpz_limbs_write関数やmpz_limbs_modify関数でリム配列を書き換えた後にこの関数を適用し，書き換え処理を完了します。この配列には$|s|$ 分の有効リムが確保されてxの絶対値が書きこまれ，xの符号にはsの符号が格納されます。この関数はxを再確保しませんので，リムポインタはそのまま使用できます。

```
void foo (mpz_t x)
{
  mp_size_t n, i;
  mp_limb_t *xp;

  n = mpz_size (x);
  xp = mpz_limbs_modify (x, 2*n);
  for (i = 0; i < n; i++)
    xp[n+i] = xp[n-1-i];
  mpz_limbs_finish (x, mpz_sgn (x) < 0 ? - 2*n : 2*n);
}
```

mpz_srcptr mpz_roinit_n (*mpz_t x*, *const mp_limb_t *xp*, *mp_size_t xs*) [関数]
既存のリム配列とそのサイズを使用してxを初期化します。xは書き込み不可として扱わなければなりません。他の mpz 関数への入力値として安全な渡し方ができるものなので，出力先としては使用しないで下さい。xpは最低でも読み込み可能なリムが一つ以上で，$|xs|$ 個分のものとします。返り値はxですが，定数ポインタ型にキャストして返されます。

```
void foo (mpz_t x)
{
  static const mp_limb_t y[3] = { 0x1, 0x2, 0x3 };
  mpz_t tmp;
  mpz_add (x, x, mpz_roinit_n (tmp, y, 3));
}
```

mpz_t MPZ_ROINIT_N (*mp_limb_t *xp*, *mp_size_t xs*) [マクロ]
このマクロは mpz_t 変数への代入を行う初期化関数として展開されます。リム配列xpは少なくとも読み取り可能なリムへのポインタでなければならず，mpz_roinit_n関数とはこの点が異なります。配列は正規化，つまり，xsが非ゼロであれば，$xp[|xs|-1]$も非ゼロでなければなりません。このマクロは定数値となりますので，非定数値に対して適用するのであれば C99 をサポートする C コンパイラが必要となります。

```
void foo (mpz_t x)
{
  static const mp_limb_t ya[3] = { 0x1, 0x2, 0x3 };
  static const mpz_t y = MPZ_ROINIT_N ((mp_limb_t *) ya, 3);

  mpz_add (x, x, y);
}
```

6 有理数関数

本章では GMP の有理数演算を実行するための関数群を解説します。これらの関数は`mpq_`から始まる名前になります。

有理数は`mpq_t`型のデータオブジェクトとして格納されます。

全ての有理数演算関数は，オペランドは標準形であることを仮定しており，演算結果も標準形にします。ここでいう標準形とは既約分数（これ以上通分できない分数形）であり，分母は必ず正として与えます。ゼロの表現は 0/1 に統一されています。

単純な代入関数は，与えられた値を標準化することはありません。演算が実行される前に，値を標準化しておくのはユーザーの責務になります。

`void mpq_canonicalize (`*mpq_t op*`)` [関数]
*op*に対して通分を実行し，分母を正にします。

6.1 初期化関数と代入関数

`void mpq_init (`*mpq_t x*`)` [関数]
*x*を初期化して 0/1 を代入します。変数は一度だけ普通に初期化・消去(`mpq_clear`)しておく必要があります。

`void mpq_inits (`*mpq_t x, ...*`)` [関数]
NULL 文字終端子を持つ`mpq_t`変数のリストを初期化し，それぞれに 0/1 をセットします。

`void mpq_clear (`*mpq_t x*`)` [関数]
*x*が確保していた領域を解放します。一度使ったすべての`mpq_t`変数に対してこの関数を呼び出して消去したことを確認して下さい。

`void mpq_clears (`*mpq_t x, ...*`)` [関数]
NULL 文字終端子を持つ`mpq_t`変数のリストが確保していた領域を消去します。

`void mpq_set (`*mpq_t rop, const mpq_t op*`)` [関数]
`void mpq_set_z (`*mpq_t rop, const mpz_t op*`)` [関数]
*op*の値を*rop*に代入します。

`void mpq_set_ui (`*mpq_t rop, unsigned long int op1, unsigned long int op2*`)` [関数]
`void mpq_set_si (`*mpq_t rop, signed long int op1, unsigned long int op2*`)` [関数]
op1/*op2*を*rop*にセットします。*op1*と*op2*に共通因子がある場合は，*rop*を使った演算を実行する前に`mpq_canonicalize`関数を呼び出しておく必要があります。

`int mpq_set_str (`*mpq_t rop, const char *str, int base*`)` [関数]
NULL 終端子のある文字列*str*を，*base*進数として解釈し，*rop*にセットします。

例えば“41”という整数や，“41/152”という表記の有理数が使えます。有理数は標準形であることを想定していますので，そうでないときには`mpq_canonicalize`関数で標準化しておきます。

分子と（必要があれば）分母は`mpz_set_str`関数と同様の処理を行って読み取りされます(see Section 5.2 [Assigning Integers], p. 33)。ホワイトスペースが文字列に入っていても単に無視されます。*base*は 2 以上 62 以下で設定できますし，他のやり方としては，文字列の先頭が 0 ならば 8 進数，`0x` もしくは`0X`ならば 16 進数，`0b` もしくは`0B`ならば 2 進数，それ以外

は 10 進数として解釈できます。分母と分子の基数は独立に設定でき，`0xEF/100` は 239/100, `0xEF/0x100` は 239/256 と解釈されます。

文字列全体が正常に変換できれば 0 を返します。不正文字列の場合は −1 を返します。

`void mpq_swap` (*mpq_t* ***rop1***, *mpq_t* ***rop2***) [関数]
*rop1*と*rop2*の値を高速に交換します。

6.2 変換関数

`double mpq_get_d` (*const mpq_t* ***op***) [関数]
*op*を`double`型に変換し，必要があれば打ち切ります（つまり，ゼロへの丸めを実行する）。

指数部が`double`型の範囲を超えたものになるときの挙動はシステムによります。無限大がサポートされていれば，大きすぎる場合には無限大が返ってきますし，小さすぎるときには0.0に変換されます。ハードウェアのオーバーフロー，アンダーフロー，非正規化エラーは設定次第で発生したりしなかったりします。

`void mpq_set_d` (*mpq_t* ***rop***, *double* ***op***) [関数]
`void mpq_set_f` (*mpq_t* ***rop***, *const mpf_t* ***op***) [関数]
*op*の値を*rop*に代入します。正確に変換できる場合は丸めは発生しません。

`char * mpq_get_str` (*char* ***str***, *int* ***base***, *const mpq_t* ***op***) [関数]
*op*を，基数*base*の文字列に変換します。基数は 2 以上 36 以下で設定できます。文字列は‘`num/den`’という形式になりますが，分母が 1 の時は‘`num`’と表現されます。

*str*が`NULL`の時は，変換結果を格納するメモリ領域を現在のメモリ割り当て関数を使って確保して格納されます(see Chapter 13 [Custom Allocation], p. 91)。領域は`strlen(str)+1`バイト確保され，ここに文字列と NULL 終端子が入ります。

*str*が`NULL`でなければ，結果を格納するのに十分なメモリブロックが確保されていなければなりません。つまり

```
mpz_sizeinbase (mpq_numref(op), base)
+ mpz_sizeinbase (mpq_denref(op), base) + 3
```

だけ必要になります。

余分に 3 バイト必要になるのは，マイナス符号，スラッシュ，NULL 終端子を格納するためです。

返り値は変換結果を格納する文字列へのポインタで，新たに確保された所か，既に確保されている*str*になります。

6.3 演算関数

`void mpq_add` (*mpq_t* ***sum***, *const mpq_t* ***addend1***, *const mpq_t* ***addend2***) [関数]
addend1 + *addend2*を求め，*sum*に格納します。

`void mpq_sub` (*mpq_t* ***difference***, *const mpq_t* ***minuend***, *const mpq_t* ***subtrahend***) [関数]
minuend − *subtrahend*を求め，*difference*に格納します。

`void mpq_mul` (*mpq_t* ***product***, *const mpq_t* ***multiplier***, *const mpq_t* ***multiplicand***) [関数]
multiplier × *multiplicand* を求め，*product*に格納します。

`void mpq_mul_2exp` (*mpq_t* **rop**, *const mpq_t* **op1**, *mp_bitcnt_t* **op2**) [関数]
$op1 \times 2^{op2}$ を求め，*rop*に格納します。

`void mpq_div` (*mpq_t* **quotient**, *const mpq_t* **dividend**, *const mpq_t* **divisor**) [関数]
dividend/*divisor*を求め，*quotient*に格納します。

`void mpq_div_2exp` (*mpq_t* **rop**, *const mpq_t* **op1**, *mp_bitcnt_t* **op2**) [関数]
$op1/2^{op2}$ を求め，*rop*に格納します。

`void mpq_neg` (*mpq_t* **negated_operand**, *const mpq_t* **operand**) [関数]
−*operand*を*negated_operand*に格納します。

`void mpq_abs` (*mpq_t* **rop**, *const mpq_t* **op**) [関数]
*op*の絶対値を*rop*に格納します。

`void mpq_inv` (*mpq_t* **inverted_number**, *const mpq_t* **number**) [関数]
1/*number*を求めて*inverted_number*に格納します。新たな分母(*number*の分子)がゼロの場合はゼロ除算が発生してしまいます。

6.4 比較関数

`int mpq_cmp` (*const mpq_t* **op1**, *const mpq_t* **op2**) [関数]
`int mpq_cmp_z` (*const mpq_t* **op1**, *const mpz_t* **op2**) [関数]
op1 と*op2*を比較します。$op1 > op2$ であれば正数，$op1 = op2$ であればゼロ，$op1 < op2$ であれば負数を返します。

この二つの有理数が等しいかどうかを決めるときには，`mpq_equal`関数の方が高速に判定できます。

`int mpq_cmp_ui` (*const mpq_t* **op1**, *unsigned long int* **num2**, *unsigned long int* **den2**) [マクロ]
`int mpq_cmp_si` (*const mpq_t* **op1**, *long int* **num2**, *unsigned long int* **den2**) [マクロ]
op1 と*num2*/*den2*を比較します。$op1 > num2/den2$ であれば正数，$op1 = num2/den2$ であればゼロ，$op1 < num2/den2$ であれば負数を返します。

*num2*と*den2*が共通因子を持っていても正常に比較できます。

これらの関数はマクロとして実装されているので，引数を複数回評価します。

`int mpq_sgn` (*const mpq_t* **op**) [マクロ]
$op > 0$ であれば+1 を, $op = 0$ であればゼロを，$op < 0$ であれば−1 を返します。

この関数はマクロとして定義されていますので，引数は複数回評価されます。

`int mpq_equal` (*const mpq_t* **op1**, *const mpq_t* **op2**) [関数]
*op1*と*op2*が等しければ正数を，等しくなければゼロを返します。`mpq_cmp`関数も同様の比較ができますが，この関数の方が高速です。

6.5 整数から有理数への変換

`mpq`関数の種類はとても少なく，特に入出力関数が殆どありません。ここで述べる関数は`mpq_t`型変数の分子と分母に直接アクセスを行うものです。

本章の最初に述べたように，分子と分母をそれぞれ代入すると，mpq_t型の値が標準形から外れたものになり得るので(see Chapter 6 [Rational Number Functions], p. 48)，mpq関数を適用する前にmpq_canonicalize関数で標準形に直しておく必要があります。

mpz_t mpq_numref (*const mpq_t op*) [マクロ]
mpz_t mpq_denref (*const mpq_t op*) [マクロ]
*op*の分子と分母への参照をそれぞれ返します。この参照を使ってmpz関数を直接利用することができるようになります。

void mpq_get_num (*mpz_t numerator*, *const mpq_t rational*) [関数]
void mpq_get_den (*mpz_t denominator*, *const mpq_t rational*) [関数]
void mpq_set_num (*mpq_t rational*, *const mpz_t numerator*) [関数]
void mpq_set_den (*mpq_t rational*, *const mpz_t denominator*) [関数]
有理数の分子と分母を代入したり，取り出したりすることができます。これらの関数は，mpq_numref やmpq_denrefを呼び出し，mpz_set関数を組み合わせることで実現しています。この関数を使うよりは直接mpq_numref やmpq_denrefを使うことをお勧めします。

6.6 入出力関数

ここに述べる関数は，標準入出力ストリームとの入出力行うものです。NULLポインタが*stream*引数に渡されると，stdinやstdoutとの入出力を行います。

この関数群を使う際には，gmp.hより前にstdio.hをインクルードしておき，必要となるプロトタイプ宣言が利用できるようにしておきます。

Chapter 10 [Formatted Output], p. 73, やChapter 11 [Formatted Input], p. 78も参照して下さい。

size_t mpq_out_str (*FILE *stream*, *int base*, *const mpq_t op*) [関数]
*op*を stdio ストリーム*stream*に，基数*base*の文字列として出力します。基数は 2 以上 36 以下に設定できます。出力形式は‘num/den’となりますが，分母が 1 の時は‘num’だけになります。

返り値は書きこんだバイト数です。エラーが起こった時にはゼロが返されます。

size_t mpq_inp_str (*mpq_t rop*, *FILE *stream*, *int base*) [関数]
*stream*から文字列を読み込んで有理数に変換し，*rop*に格納します。先頭のホワイトスペースは読み込みますが影響はしません。返り値は読み込んだ文字数（ホワイトスペースも含む）で，有理数として読み取れない場合はゼロが返されます。

入力できるのは‘17/63’という有理数の形式か，‘123’という整数の形式です。この形式に当て嵌まらない文字があると読み込みを停止します。ホワイトスペースが文字列中に入ることは認められていません。入力が標準形でない場合は，mpq_canonicalize関数を呼び出す必要があります(see Chapter 6 [Rational Number Functions], p. 48)。

*base*は 2 以上 36 以下に設定できます。0 の場合は，先頭の文字で基数を判断します。‘0x’や‘0X’は 16 進，‘0’は 8 進，何もなければ 10 進と解釈します。この先頭文字による基数の解釈は分子と分母で別々に行いますので，例えば‘0x10/11’ は16/11，‘0x10/0x11’ は16/17 となります。

7 浮動小数点演算関数

GMP の多倍長浮動小数点数は，mpf_t型のオブジェクトとして格納され，演算関数はmpf_から始まる関数名を持ちます。

浮動小数点数変数ごとに仮数部には，メモリの上限いっぱいまでユーザー指定の精度が設定できます。各変数は独自の桁数を保持し，いつでも変更が可能です。リム一つ分の仮数部が，精度の下限値になります。

計算結果の精度は，計算前の変数や入力数値の精度で決まります。入力変数の精度を変更しても，計算には影響を与えません（変数割り当て時に変数に入る値が変更されていれば別）。

浮動小数点数毎に保持する指数部は固定桁で，普通はマシンワード 1 つ分となります。現在の実装では，32bit マシンでは$2^{-68719476768}$ から$2^{68719476736}$ の範囲(訳注:$-(2^{36}+2^5)$ ～2^{36})となり，64bit マシンではもっと広がります。mpf_get_str関数については，mp_exp_t型に収まる指数部を返しますが，mpf_set_str関数の場合，現在の実装ではlong型より大きい指数部は受け付けませんので注意して下さい。

浮動小数点変数はそれぞれ実際に使用している仮数部の履歴を保持しています。つまり，浮動小数点数が数ビットで正確に表現できる場合は，変数の指定桁数がそれより大きくても，計算に必要な bit 数しか使用しません。これで計算結果に影響を与えずに，パフォーマンスを最適化することができます。

GMP の内部では，必要に応じて結果を格納する変数の桁数以上の精度で計算を行い，誤差の増大を抑えています。最終結果は常に結果を格納する変数の精度に切り捨てて丸められます。

仮数部表現は 2 進です。つまり，0.1 のような 10 進小数は正確に表現できません。これは IEEE double型浮動小数点数でも同様です。正確な 10 進小数表現が求められるお金の計算のようなものには向いていません（整数に直すか，有理数演算を使うのが良いでしょう）。

mpf関数と変数は，無限大(Inf)や非数(NaN)用の表現形式を扱いませんので，アプリケーション側で指数部のオーバーフローが起こらないように留意しなければなりません。さもないと何が起こるかわかりません（訳注：Segmentation fault が発生したりする）。

mpf関数は IEEE P754 演算を拡張したものではありませんので，ワードサイズの異なる環境で実行された結果は一致しないことがあります。

多倍長浮動小数点数を使用する新規のプロジェクトを行うにあたっては，GMP の拡張ライブラリである MPFR(http://mpfr.org) を使って下さい。MPFR はしっかりした精度管理と，正確な丸め処理を行っており，IEEE P754 の自然な拡張を行っています。

7.1 初期化関数

void mpf_set_default_prec (*mp_bitcnt_t prec*) [関数]
最小でも*prec* bit 確保すべく，デフォルト精度をセットします。この関数を実行して以降のmpf_init関数は，このデフォルト精度が使用されます。それ以前に初期化された変数は影響ありません。

mp_bitcnt_t mpf_get_default_prec (*void*) [関数]
現在使用されているデフォルト精度を返します。

mpf_tオブジェクトは，値を最初に格納する前に初期化されている必要があります。mpf_init関数やmpf_init2関数を使って初期化して下さい。

`void mpf_init` (*mpf_t x*) [関数]
変数*x*を初期化してゼロを格納します。通常は，変数の初期化と，`mpf_clear`関数を使った解放は一度だけ行うようにして下さい。*x*の精度は，`mpf_set_default_prec`関数でデフォルト精度がセットされていない限り，定義されません。

`void mpf_init2` (*mpf_t x, mp_bitcnt_t prec*) [関数]
変数*x*を初期化してゼロを格納し，精度を*prec* bit に設定します。通常は，変数の初期化と，`mpf_clear`関数を使った解放は一度だけ行うようにして下さい。

`void mpf_inits` (*mpf_t x, ...*) [関数]
NULL 終端子を持った`mpf_t`変数の列を初期化し，すべてにゼロをセットします。初期化された変数リストの精度は，`mpf_set_default_prec`関数でデフォルト精度が設定されていない限り不定となります。

`void mpf_clear` (*mpf_t x*) [関数]
*x*が確保していたメモリスペースを解放します。使用してた`mpf_t`型変数すべてに対してこの関数で解放したか，しっかり確認して下さい。

`void mpf_clears` (*mpf_t x, ...*) [関数]
NULL 終端子付きの`mpf_t`型変数リストが確保していたメモリスペースを全て解放します。

下記は，浮動小数点変数を初期化するサンプルプログラムです。

```
{
  mpf_t x, y;
  mpf_init (x);           /* デフォルト精度を使う */
  mpf_init2 (y, 256);     /* 精度は最小でも 256 bit */
  ...
  /* プログラムが中断されなければここを実行 ... */
  mpf_clear (x);
  mpf_clear (y);
}
```

以降の関数は，計算中に使用する精度を変更するために利用できます。良くある使用例としては，Newton-Raphon 反復中の精度の調整や，有効桁数に合わせた計算精度の変更があります。

`mp_bitcnt_t mpf_get_prec` (*const mpf_t op*) [関数]
*op*の現在の精度桁を bit 数で返します。

`void mpf_set_prec` (*mpf_t rop, mp_bitcnt_t prec*) [関数]
*rop*の精度を*prec* bit にセットします。*rop*に入っていた値は新しい精度に切り捨てられます。

この関数は`realloc`関数を使って実装されていますので，あまり繰り返し使わないで下さい。

`void mpf_set_prec_raw` (*mpf_t rop, mp_bitcnt_t prec*) [関数]
*rop*の精度を*prec* bit にセットします。既に確保されたメモリはそのまま使用します。

*prec*は，*rop*の精度，即ち初期化時の精度や`mpf_set_prec`で設定した精度より大きくしてはいけません。

*rop*の値は変化せずに保持されます。*prec* bit 以上の精度を持っていた時には，有効桁数の高い方が保持されます。新しい値は*rop*に書きこまれ，*prec* bit の精度になります。

mpf_clear関数やmpf_set_prec関数を呼び出す前であれば，mpf_set_prec_raw関数を呼び出して，元の精度で*rop*に書き戻さなくてはなりません。これが失敗すると何が起こるかわかりません。

mpf_set_prec_raw関数を呼び出す前であれば，mpf_get_prec関数で元の精度を取り出すことができます。呼び出し後だと，*prec*精度の値になってしまいます。

mpf_set_prec_raw関数は，計算の途中でもmpf_t型変数の精度を素早く変更することができ，ループの中で精度を増やしたり，計算中に別の目的で様々な精度を設定したりするときに有用です。

7.2 代入関数

この代入関数は新しい値を，初期化済みの浮動小数点数(see Section 7.1 [Initializing Floats], p. 52)に代入するためのものです。

void mpf_set (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
void mpf_set_ui (*mpf_t* **rop**, *unsigned long int* **op**) [関数]
void mpf_set_si (*mpf_t* **rop**, *signed long int* **op**) [関数]
void mpf_set_d (*mpf_t* **rop**, *double* **op**) [関数]
void mpf_set_z (*mpf_t* **rop**, *const mpz_t* **op**) [関数]
void mpf_set_q (*mpf_t* **rop**, *const mpq_t* **op**) [関数]
*rop*に*op*の値を代入します。

int mpf_set_str (*mpf_t* **rop**, *const char* ***str**, *int* **base**) [関数]
文字列*str*から変換して*rop*に値を代入します。文字列は'M@N'という表現形式か，10 以下の基数であれば'MeN'という形式に対応しています。'M'は仮数部，'N'は指数部を表わします。仮数部は常に指定された基数*base*を使います。指数部は指定基数か，*base*が負数であれば 10 進と判断して扱います。小数点は現在のロケールに依存したものが使用でき，localeconvで設定できます。

引数*base*は 2 以上 62 以下の値が設定できます。−62 以上−2 以下の負数であれば，指数部は 10 進数であると判断します。

36 以下の基数であれば，大文字小文字の区別はせずに同じ値とみなします。37 以上 62 以下であれば，大文字は 10 〜 35，小文字は 36 〜 61 を表現しているものとみなします。

同様の機能を持つmpz関数とは異なり，*base*がゼロであっても先頭文字で基数を判断したりはしません。従って，'0.23'のような小数を 8 進数とは解釈しません。

ホワイトスペースが文字列に入っていても単純に無視されます。[というのは完全には正しくない。文字列の先頭や仮数部の中に入ったホワイトスペースは確かに無視されるが，その他の場所，例えばマイナス記号の後や，指数部の中に入った場合は別。この関数の定義を変えて，入力値にホワイトスペースが入ることを厳禁するようにしたいと考えているので，ご意見をお寄せ頂きたい。"3 14"を 314 と解釈してもいいと思う？]

この関数は，文字列全体が正しい基数*base*の数として表現されていればゼロを返し，失敗した時には −1 を返します。

void mpf_swap (*mpf_t* **rop1**, *mpf_t* **rop2**) [関数]
*rop1*と*rop2*の値と精度を素早く交換します。

7.3 初期化代入関数

GMP は初期化と代入を同時に行う便利な関数群を提供しています。これらの関数名はmpf_init_set...で始まります。

浮動小数点変数が`mpf_init_set...`で一度初期化されると，通常の浮動小数点数関数同様に，入力元変数か出力先変数が使用できるようになります。一度初期化した浮動小数点変数に対しては，これらの初期化代入関数を使わないようにして下さい。

`void mpf_init_set` (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
`void mpf_init_set_ui` (*mpf_t* **rop**, *unsigned long int* **op**) [関数]
`void mpf_init_set_si` (*mpf_t* **rop**, *signed long int* **op**) [関数]
`void mpf_init_set_d` (*mpf_t* **rop**, *double* **op**) [関数]
*rop*を初期化して*op*の値を代入します。

*rop*の精度は，`mpf_set_default_prec`関数でセットしたデフォルト精度となります。

`int mpf_init_set_str` (*mpf_t* **rop**, *const char* ***str**, *int* **base**) [関数]
*rop*を初期化して，文字列*str*の値を変換して代入します。代入操作については`mpf_set_str`関数の解説をご覧下さい。

*rop*はエラーが発生した時でも初期化は行われます。(即ち，`mpf_clear`を呼び出さなくてはならない。)

*rop*の精度は，`mpf_set_default_prec`等の関数でセットされた現在有効なデフォルト精度になります。

7.4 変換関数

`double mpf_get_d` (*const mpf_t* **op**) [関数]
*op*を`double`型に変換します。必要があれば切り捨て(即ち，ゼロ方向への丸め）ます。

*op*の指数部が`double`型に対して大きすぎたり小さすぎたりする場合は，システムごとに結果が異なってきます。無限大が使えるときには，指数部溢れで無限大が返ってきますし，小さすぎるときには0.0 が返されます。ハードウェアのオーバーフロー，アンダーフロー，非正規化エラーは設定次第で発生したりしなかったりします。

`double mpf_get_d_2exp` (*signed long int* ***exp**, *const mpf_t* **op**) [関数]
*op*を`double`型に変換し，必要があれば切り捨て処理を行います（つまり，ゼロへの丸め）。指数部は別の場所に返されます。

返り値は$0.5 \leq |d| < 1$ の範囲になります。指数部は`*exp`に格納されます。$d \times 2^{exp}$ は切り捨てられた*op*の値です。*op*がゼロであれば，返り値は0.0 となり，`*exp`にはゼロが格納されます。

この関数は標準 C の`frexp`関数と同じような機能を持ちます(see Section “Normalization Functions” in *The GNU C Library Reference Manual*)。

`long mpf_get_si` (*const mpf_t* **op**) [関数]
`unsigned long mpf_get_ui` (*const mpf_t* **op**) [関数]
*op*を`long`や`unsigned long`型に変換し，小数部は切り捨てます。*op*が変換型としては大きすぎる場合の結果は定義されていません。

`mpf_fits_slong_p` や`mpf_fits_ulong_p`(see Section 7.8 [Miscellaneous Float Functions], p. 58)も参照して下さい。

`char * mpf_get_str` (*char* ***str**, *mp_exp_t* ***expptr**, *int* **base**, *size_t* **n_digits**, *const mpf_t* **op**) [関数]
*op*を基数*base*の文字列に変換します。基数は 2 以上 36 以下，もしくは −36 以上 −2 以下に設定できます。表現できるのは*n_digits*桁までで，以降のゼロは無視されます。*op*を正確に表現できる桁数までしか表示されません。*n_digits*がゼロの時は，最大桁数まで表示されます。

*base*が 2 以上 36 以下の時は，数字と小文字で表現されます。−36 以上 −2 以下の時は，数字と大文字が使用されます。37 以上 62 以下の時は数字，大文字，小文字が順に使用されます。

*str*が`NULL`であれば，現在のメモリ割り当て関数(see Chapter 13 [Custom Allocation], p. 91)を使って必要な文字列分，即ち，NULL 終端子を含む`strlen(str)+1`バイト分が確保されます，

*str*が`NULL`でなければ，*n_digits* + 2 バイト分の領域があればそこに格納されます。これは仮数部，マイナス符号，NULL 終端子を含む領域です。*n_digits*がゼロの時は，有効桁全て表示されますが，アプリケーションがどのぐらいの桁数になるかを事前に知ることができませんので，この場合は*str*を`NULL`にしておくべきでしょう。

生成された文字列は小数で，小数点は先頭の桁の左にあるものと考えます。指数部は*expptr*ポインタのところに書きこまれます。例えば，3.1416 は`"31416"`という文字列（小数部）と，指数部 1 になって返されます。

*op*がゼロの時は，空文字列と指数部ゼロが返ってきます。

返り値は文字列へのポインタで，新規に確保された領域か，あらかじめ与えられた*str*となります。

7.5 演算関数

`void mpf_add` (*mpf_t* **rop**, *const mpf_t* **op1**, *const mpf_t* **op2**) [関数]
`void mpf_add_ui` (*mpf_t* **rop**, *const mpf_t* **op1**, *unsigned long int* **op2**) [関数]
op1 + *op2* を求めて*rop*に格納します。

`void mpf_sub` (*mpf_t* **rop**, *const mpf_t* **op1**, *const mpf_t* **op2**) [関数]
`void mpf_ui_sub` (*mpf_t* **rop**, *unsigned long int* **op1**, *const mpf_t* **op2**) [関数]
`void mpf_sub_ui` (*mpf_t* **rop**, *const mpf_t* **op1**, *unsigned long int* **op2**) [関数]
op1 − *op2*を計算して*rop*に格納します。

`void mpf_mul` (*mpf_t* **rop**, *const mpf_t* **op1**, *const mpf_t* **op2**) [関数]
`void mpf_mul_ui` (*mpf_t* **rop**, *const mpf_t* **op1**, *unsigned long int* **op2**) [関数]
op1 × *op2* を計算して*rop*に格納します。

割る数がゼロの時の除法の結果は未定義で，内部的にはゼロ除算が発生することになります。他の演算例外同様に，ユーザー側で演算例外を制御できるようにしておきましょう。

`void mpf_div` (*mpf_t* **rop**, *const mpf_t* **op1**, *const mpf_t* **op2**) [関数]
`void mpf_ui_div` (*mpf_t* **rop**, *unsigned long int* **op1**, *const mpf_t* **op2**) [関数]
`void mpf_div_ui` (*mpf_t* **rop**, *const mpf_t* **op1**, *unsigned long int* **op2**) [関数]
op1/*op2*を求めて*rop*に格納します。

`void mpf_sqrt` (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
`void mpf_sqrt_ui` (*mpf_t* **rop**, *unsigned long int* **op**) [関数]
$\sqrt{op}$ を求め，*rop*に格納します。

`void mpf_pow_ui` (*mpf_t* **rop**, *const mpf_t* **op1**, *unsigned long int* **op2**) [関数]
$op1^{op2}$ を求め，*rop*に格納します。

`void mpf_neg` (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
−*op*を*rop*に格納します。

`void mpf_abs` (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
*op*の絶対値を*rop*に格納します。

`void mpf_mul_2exp` (*mpf_t* **rop**, *const mpf_t* **op1**, *mp_bitcnt_t* **op2**) [関数]
op1 × 2^{op2} を求め，*rop*に格納します。

`void mpf_div_2exp` (*mpf_t* **rop**, *const mpf_t* **op1**, *mp_bitcnt_t* **op2**) [関数]
op1/2^{op2} を求め，*rop*に格納します。

7.6 比較関数

`int mpf_cmp` (*const mpf_t* **op1**, *const mpf_t* **op2**) [関数]
`int mpf_cmp_z` (*const mpf_t* **op1**, *const mpz_t* **op2**) [関数]
`int mpf_cmp_d` (*const mpf_t* **op1**, *double* **op2**) [関数]
`int mpf_cmp_ui` (*const mpf_t* **op1**, *unsigned long int* **op2**) [関数]
`int mpf_cmp_si` (*const mpf_t* **op1**, *signed long int* **op2**) [関数]
op1 と*op2*を比較します。*op1* > *op2* の時は正数を，*op1* = *op2* の時はゼロを，*op1* < *op2* の時は負数を返します。

`mpf_cmp_d`は無限大を引数に取ることもできますが，NaN の時の結果は定義されていません。

`int mpf_eq` (*const mpf_t* **op1**, *const mpf_t* **op2**, *mp_bitcnt_t op3*) [関数]
この関数は数学的には正しくないものなので，なるべく使わないで下さい。

*op1*と*op2*の先頭の*op3* bit 分が等しい時には非ゼロを返し，等しくない時にはゼロを返します。この関数の比較方法では，例えば，256 (2 進表現 100000000)と 255 (11111111) は常に等しくないと判断します。またゼロと等しいと判断できるのはゼロ自身だけです。

`void mpf_reldiff` (*mpf_t* **rop**, *const mpf_t* **op1**, *const mpf_t* **op2**) [関数]
op1 と*op2*の相対差異を計算し，*rop*に格納します。相対差異とは|*op1* − *op2*|/*op1* のことです。

`int mpf_sgn` (*const mpf_t* **op**) [マクロ]
op > 0 ならば+1 を，*op* = 0 であればゼロを，*op* < 0 であれば−1 を返します。

この関数はマクロとして定義されていますので，引数を複数回評価します。

7.7 入出力関数

ここで述べる関数は，`mpf`値に対して stdio ストリームとの入出力を行います。`stream`引数に`NULL`ポインタを渡すと，`stdin`からの入力，`stdout`への出力をそれぞれ行います。

これらの関数を使うときには，`gmp.h`より前に`stdio.h`をインクルードしておき，必要となるプロトタイプ宣言が使えるようにしておきます。

Chapter 10 [Formatted Output], p. 73, やChapter 11 [Formatted Input], p. 78も参照して下さい。

`size_t mpf_out_str` (*FILE* ***stream**, *int* **base**, *size_t* **n_digits**, *const mpf_t* **op**) [関数]
*stream*に*op*を文字列として出力します。返り値は書きこまれたバイト数で，エラーが発生した時にはゼロが返されます。

仮数部の先頭は‘`0.`’から始まり，基数*base*は 2 以上 36 以下，または −36 以上 −2 以下に設定できます。指数部は‘`e`’で区切られて表示されますが，基数が 10 以上であれば区切り文字は`@`になります。指数部は常に 10 進表現になります。小数点は現在のロケールに従いますので，`localeconv`で設定可能です。

基数*base*が 2 以上 36 以下の場合は数字と小文字だけで表現されます。−36 以上 −2 以下の場合は，数字と大文字が使われます。37 以上 62 の時は数字，大文字，小文字がこの順で使用されます。

仮数部は，*op*の精度桁以内の*n_digits*桁まで出力されます。*n_digits*がゼロの時は最大桁数まで表示されます。

`size_t mpf_inp_str` (*mpf_t* **rop**, *FILE* ***stream**, *int* **base**) [関数]
基数*base*の値として*stream*から文字列として読み込み，浮動小数点数に変換して*rop*に格納します。'`M@N`'の形式か，10 進表現であれば'`MeN`'という形式であれば読み取ることができます。'`M`'は仮数部，'`N`'は指数部を意味します。仮数部は常に指定された基数表現となります。指数部は指定された基数表現，もしくは，*base*が負数の時は 10 進表現となります。小数点は現在のロケールのものが使用され，`localeconv`で設定できます。

引数*base*は 2 以上 36，もしくは −36 以上 −2 以下に設定できます。負数の場合は指数部は 10 進表現と指定したことになります。

対応する`mpz`関数とは異なり，*base*がゼロの時は，基数を先頭の文字で指定することができません。従って，'`0.23`'のような数字を 8 進数と解釈することはありません。

返り値は読み込んだバイト数になります。エラー発生時にはゼロが返されます。

7.8 その他の関数

`void mpf_ceil` (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
`void mpf_floor` (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
`void mpf_trunc` (*mpf_t* **rop**, *const mpf_t* **op**) [関数]
*op*を整数に丸めて*rop*に格納します。`mpf_ceil`関数は隣接整数の大きい方に，`mpf_floor`は小さい方に，`mpf_trunc`関数は切り捨てて整数に丸めます。

`int mpf_integer_p` (*const mpf_t* **op**) [関数]
*op*が整数であれば非ゼロを返します。

`int mpf_fits_ulong_p` (*const mpf_t* **op**) [関数]
`int mpf_fits_slong_p` (*const mpf_t* **op**) [関数]
`int mpf_fits_uint_p` (*const mpf_t* **op**) [関数]
`int mpf_fits_sint_p` (*const mpf_t* **op**) [関数]
`int mpf_fits_ushort_p` (*const mpf_t* **op**) [関数]
`int mpf_fits_sshort_p` (*const mpf_t* **op**) [関数]
*op*を整数に切り捨てた時に，対応する C データ型に収まるようであれば非ゼロを返します。

`void mpf_urandomb` (*mpf_t* **rop**, *gmp_randstate_t* **state**, *mp_bitcnt_t* **nbits**) [関数]
*rop*に一様浮動小数点乱数をセットします。値の範囲は$0 \le rop < 1$ で，*rop*の精度が小さい時には有効*nbits*ビット以下の仮数部になるようにします。

*state*変数は，この関数で使用する前に`gmp_randinit`関数(Section 9.1 [Random State Initialization], p. 71)で初期化しておく必要があります。

`void mpf_random2` (*mpf_t* **rop**, *mp_size_t* **max_size**, *mp_exp_t* **exp**) [関数]
2 進表現としては 0 と 1 がランダムに並ぶ，最大*max_size*リム長の浮動小数点乱数を生成します。指数部は −*exp* から*exp* (リム換算で)となります。この関数は，滅多に起こり得ないケースのバグを見つけるためにアルゴリズムや関数をチェックするのに役立ちます。*max_size*が負数の時には負の乱数を生成します。

8 基盤関数

本章では，GMP の上位関数群を下支えする基盤関数群を解説します。少しでも計算時間を短縮したい時にも有用なものです。

基盤関数は`mpn_`から始まる名前になります。

`mpn`関数は，高速化に特化して作られており，首尾一貫したインターフェースを提供するものではありません。同じような機能を持つ関数が複数提供されているのは，使い回すのが難しいケースに対応しているからです。実際の多倍長計算に有用なものとして機能しており，万人に使いこなせる代物ではありません。

計算元となる引数は，最小有効リムへのポインタと，リムの総数を指定しています。計算結果を格納する先は，ポインタだけで指定します。つまり，基盤関数の使用者が，計算結果の格納先に十分なメモリ領域を確保していることを保証しなければなりません。

このように計算元と計算結果の格納先を規定することで，引数リム数の範囲内（subrange)で計算を実行することができ，計算結果もリム数の範囲内に格納することができるようになります。

全ての基盤関数は，少なくとも計算元には最低 1 リムの領域が確保されていることだけを要求しています。サイズゼロの場合はゼロと見なします。特に明記されていない限り，同位置（in-place）演算，即ち，計算元と結果の格納先が同じであっても処理が可能になります。但し，一部だけ重なっているようなケースはダメです。

`mpn`関数は`mpz_`, `mpf_`, `mpq_`関数の実装の基盤となっています。

下記の例は，*s1p*から始まる自然数と，*s2p*から始まる自然数の和と求め，*destp*に格納しています。全ての自然数格納エリアは*n*リム確保されています。

```
cy = mpn_add_n (destp, s1p, s2p, n)
```

`mpn`関数は，それぞれの引数のどこのリムがゼロであるか，特殊な形態になっているかということは全く感知しません。ランダムなデータに対して，いちいちチェックなぞしていたら時間の無駄です。アプリケーション側でデータの情報が欲しければ，計算途中に確認のためのステップを設ければ済む話です。

以下に述べる基盤関数の解説のうち，計算元の引数は最小有効リムへのポインタとリムの数，例えば{*s1p*, *s1n*} というセットで与えます。

`mp_limb_t mpn_add_n` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *const mp_limb_t* ***s2p**, *mp_size_t* **n**) [関数]

{*s1p*, *n*} と{*s2p*, *n*} を和を求め，和の小さい方の*n*リム分を*rp*に書き込みます。桁上がりがあれば返り値は 1，なければ 0 になります。

この関数は加算の土台になっており，大方の CPU に対してアセンブラ言語で記述されていることから，加算を実行する時にはこの関数を使うことをお勧めします。変数そのものに足し込みたい時(即ち*s1p* と*s2p*が同じものである時)には，`mpn_lshift`関数で 1bit シフトするだけで済むのでこちらの方が高速に実行できます。

`mp_limb_t mpn_add_1` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *mp_size_t* **n**, *mp_limb_t* ***s2limb**) [関数]

{*s1p*, *n*} と*s2limb*の和を求め，最小リムの方から*n*リム分を*rp*に書き込みます。桁上がりがあれば返り値は 1 ，なければ 0 になります。

`mp_limb_t mpn_add` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t s1n, const mp_limb_t *s2p, mp_size_t s2n*) [関数]

{*s1p*, *s1n*} と{*s2p*, *s2n*} の和を求め，*s1n*個分の最小有効リムを*rp*に書き込みます。桁上がりがあれば返り値は 1，なければ 0 になります。

演算を実行するには，*s1n*が*s2n*以上である必要があります。

`mp_limb_t mpn_sub_n` (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [関数]

{*s1p*, *n*} から{*s2p*, *n*} を引き，その結果を最小有効リムの方から*n*個分*rp*に書き込みます。桁の借り上げがあれば 1 を，なければ 0 を返します。

この関数は減算の土台になります。大方の CPU 用にアセンブラで記述されていますので，減算を実装する時にはこの関数を使って下さい。

`mp_limb_t mpn_sub_1` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n, mp_limb_t s2limb*) [関数]

{*s1p*, *n*} から*s2limb*を引き，最小有効リムの*n*個分を*rp*に書き込みます。桁の借り上げがあれば 1 を，なければ 0 を返します。

`mp_limb_t mpn_sub` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t s1n, const mp_limb_t *s2p, mp_size_t s2n*) [関数]

{*s1p*, *s1n*} から{*s2p*, *s2n*} を引き，その結果を最小有効リムの*s1n*個分を*rp*に書き込みます。桁の借り上げがあれば 1 を，なければ 0 を返します。

この演算を実行するためには，*s1n*が*s2n*以上でなければなりません。

`mp_limb_t mpn_neg` (*mp_limb_t *rp, const mp_limb_t *sp, mp_size_t n*) [関数]

{*sp*, *n*} にマイナスをつけて{*rp*, *n*} に書き戻します。この関数は，`mpn_sub_n`関数を用いて，*n*リム分のゼロから{*sp*, *n*} を差し引く演算と同じ結果になります。桁の借り上げがあれば 1 を，なければ 0 を返します。

`void mpn_mul_n` (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [関数]

{*s1p*, *n*} と{*s2p*, *n*} の積を求め，2*n*リム長の積を*rp*に書き込みます。

積を格納する領域には，積の有効リムの多数がゼロであっても，2*n*リム分のメモリ領域が不可欠です。この 3 つの引数のメモリ領域は互いに重複してはいけません。

乗じる 2 数が同じものである場合は，`mpn_sqr`関数を使って下さい。

`mp_limb_t mpn_mul` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t s1n, const mp_limb_t *s2p, mp_size_t s2n*) [関数]

{*s1p*, *s1n*} と{*s2p*, *s2n*} の積を求め，(*s1n*+*s2n*)長の積を*rp*に書き込みます。返り値は積の最大有効リムになります。

積を格納する領域は，積の有効桁数リムの多数がゼロであっても，*s1n* + *s2n*リム分のメモリ領域が不可欠です。この 3 つのメモリ領域は互いに重複してはいけません。

この演算を実行する際には，*s1n*は*s2n*以上でなければなりません。

`void mpn_sqr` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n*) [関数]

{*s1p*, *n*} の 2 乗を求め，2*n*リム長の結果を*rp*に書き込みます。

2 乗を書き込むメモリ領域には，2 乗の有効リムの多数がゼロであっても，2*n*リム分必要になります。計算元と格納先のメモリ領域は互いに重複してはいけません。

`mp_limb_t mpn_mul_1` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *mp_size_t* **n**, *mp_limb_t* **s2limb**) [関数]

{*s1p*, *n*} と*s2limb*の積を求め，積の最小*n*リム分を*rp*に書き込みます。返り値は積の最大有効リムです。{*s1p*, *n*} と{*rp*, *n*} のメモリ領域は，$rp \leq s1p$ であれば重なりがあっても構いません。

この基盤関数は，GMP における他の演算同様，一般的な乗算の土台になります。大部分のコードは大方の CPU 向けにアセンブラで記述されています。

*s2limb*が 2 のべき乗の場合は，この関数を使わず，*s2limb*の log と等しいシフト値を与えて`mpn_lshift`を使うとより高速な処理が可能になります。

`mp_limb_t mpn_addmul_1` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *mp_size_t* **n**, *mp_limb_t* **s2limb**) [関数]

{*s1p*, *n*} と*s2limb*の積を求め，その積の最小*n*リム分を{*rp*, *n*} に加え，結果を*rp*に書き戻します。返り値は，積の最大有効リムに加算における桁上がり分が加えられたものです。{*s1p*, *n*} と{*rp*, *n*} は，$rp \leq s1p$ であれば，同じメモリ位置でも問題ありません。

この基盤関数は，GMP の他の演算同様，乗算を構築するための土台になります。大部分の CPU 向けにアセンブラルーチンが提供されています。

`mp_limb_t mpn_submul_1` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *mp_size_t* **n**, *mp_limb_t* **s2limb**) [関数]

{*s1p*, *n*} と*s2limb*の積を求め，その積の最小*n*リム分を{*rp*, *n*} から引き，結果を*rp*に書き戻します。返り値は，積の最大有効リムに減算における桁借り上げ分が加えられたものです。{*s1p*, *n*} と{*rp*, *n*} は，$rp \leq s1p$ であれば，同じメモリ位置でも問題ありません。

この基盤関数は，GMP の他の演算同様，乗算と除算を構築するための土台になります。大部分の CPU 向けにアセンブラルーチンが提供されています。

`void mpn_tdiv_qr` (*mp_limb_t* ***qp**, *mp_limb_t* ***rp**, *mp_size_t* **qxn**, *const mp_limb_t* ***np**, *mp_size_t* **nn**, *const mp_limb_t* ***dp**, *mp_size_t* **dn**) [関数]

{*np*, *nn*} を{*dp*, *dn*} で割り，商を{*qp*, *nn*−*dn*+1} に，剰余を{*rp*,*dn*} に格納します。商は切り捨て(0 への丸め)られます。

*np*と*rp*を除き，他の引数のメモリ領域が重なっていてはいけません。割られる数のリム数*nn*は，割る数のリム数*dn*以上でなければなりません。割る数の最小有効リ）ムは非ゼロでなくてはなりません。*qxn*はゼロでなければなりません（訳注：不要な引数?)。

`mp_limb_t mpn_divrem` (*mp_limb_t* ***r1p**, *mp_size_t* **qxn**, *mp_limb_t* ***rs2p**, *mp_size_t* **rs2n**, *const mp_limb_t* ***s3p**, *mp_size_t* **s3n**) [関数]

[この関数は廃止予定です。もっと効率の良い`mpn_tdiv_qr`関数を使って下さい。]

{*rs2p*, *rs2n*} を{*s3p*, *s3n*} で割り，商を*r1p*に書き込みますが，最大有効リムは別に扱われ，この関数の返り値になります。剰余は*rs2p*に上書きされ，*s3n*リムの長さとなります(即ち，割る数(divisor)と同じリム長となる)。

整数の商に加えて*qxn*長の小数部のリムが格納されます。大概は*qxn*はゼロになります。

*rs2n*は*s3n*以下でなければなりません。また，割る数の最大有効ビットは 1 でなければなりません。

商が不要であれば，*r1p*として*rs2p* + *s3n*を渡して下さい。特別な場合を除き，引数のメモリ空間の重複があってはいけません。

返り値は商の最大有効リムになりますので，0 か 1 です。

*r1p*のエリアは*rs2n* − *s3n* + *qxn*リム長必要です。

`mp_limb_t mpn_divrem_1` (*mp_limb_t* ***r1p***, *mp_size_t* ***qxn***, *mp_limb_t* ***s2p***, *mp_size_t* ***s2n***, *mp_limb_t* ***s3limb***) [Function]
`mp_limb_t mpn_divmod_1` (*mp_limb_t* ***r1p***, *mp_limb_t* ***s2p***, *mp_size_t* ***s2n***, *mp_limb_t* ***s3limb***) [マクロ]

{*s2p*, *s2n*} を*s3limb*で割り，商を*r1p*に格納し，剰余を返します。

整数の商は{*r1p*+*qxn*, *s2n*} に書き込まれ，小数部*qxn*が計算されて，{*r1p*, *qxn*} に書き込まれます。*s2n* と*qxn*はゼロでも構いません。大概のケースでは*qxn*はゼロになります。

`mpn_divmod_1`が互換性のために残されていますが，これは*qxn*をゼロと設定した`mpn_divrem_1`関数のマクロです。

r1p と*s2p*のメモリ領域は，同一か，完ぺきに分離している必要があります。

`mp_limb_t mpn_divmod` (*mp_limb_t* ***r1p***, *mp_limb_t* ***rs2p***, *mp_size_t* ***rs2n***, *const mp_limb_t* ***s3p***, *mp_size_t* ***s3n***) [関数]

[この関数は廃止予定です。もっと効率の良い`mpn_tdiv_qr`関数を使って下さい。]

`void mpn_divexact_1` (*mp_limb_t* * ***rp***, *const mp_limb_t* * ***sp***, *mp_size_t* ***n***, *mp_limb_t* ***d***) [関数]

{*sp*, *n*} は*d*で割り切れるという前提の元で除算を実行し，その商を{*rp*, *n*} に格納します。*d*では割り切れない場合は，{*rp*, *n*} に格納される値は不定です。*rp*と*sp*のメモリ領域は完全に別のところに確保し，重なりがないようにして下さい。

`mp_limb_t mpn_divexact_by3` (*mp_limb_t* ***rp***, *mp_limb_t* ***sp***, *mp_size_t* ***n***) [マクロ]
`mp_limb_t mpn_divexact_by3c` (*mp_limb_t* ***rp***, *mp_limb_t* ***sp***, *mp_size_t* ***n***, *mp_limb_t* ***carry***) [Function]

{*sp*, *n*} を 3 で割り，正確に割り切れれば，その結果を{*rp*, *n*} に書き込みます。この場合は返り値はゼロになります。 3 で割り切れなければ，返り値は非ゼロで，結果は不正確なものになります。

`mpn_divexact_by3c`関数は初期桁上がり・桁下がりパラメータを持ち，これは前回呼び出し時の返り値を代入できます。それによって，低次桁から高次桁へ，一桁ずつ大きな数の計算ができるようになります。`mpn_divexact_by3`関数は，`mpn_divexact_by3c` 関数に桁上がりパラメータをゼロとしたときのマクロ実装です。

これらのルーチン群は逆数との乗算を行っており，高速な乗算と遅い除算を組み合わせている`mpn_divrem_1`より高速です。

入力値a, 計算結果q, サイズn, 初期桁上がりi,返り値c は$cb^n + a - i = 3q$ という関係式を満足します。ここで$b = 2^{\texttt{GMP_NUMB_BITS}}$ です。返り値c は 0, 1, 2 のどれかで，初期桁上がりi も 0, 1, 2 のどれかになります(後者二つは桁上がりの意)。$c = 0$ であれば，明らかに$q = (a - i)/3$ です。$c \neq 0$ であれば，剰余$(a - i) \bmod 3$ は，$b \equiv 1 \bmod 3$ なので（`mp_bits_per_limb`が偶数の場合。現行バージョンでは常に偶数），$3 - c$ となります。

`mp_limb_t mpn_mod_1` (*const mp_limb_t* ***s1p***, *mp_size_t* ***s1n***, *mp_limb_t* ***s2limb***) [関数]

{*s1p*, *s1n*} を*s2limb*で割り，剰余を返します。*s1n*はゼロでも構いません。

`mp_limb_t mpn_lshift` (*mp_limb_t* ***rp**, *const mp_limb_t* ***sp**, *mp_size_t* **n**, *unsigned int* **count**) [関数]

{*sp*, *n*} を左方向に*count* ビット分シフトし，その結果を{*rp*, *n*} に書き込みます。左シフトの結果，はみ出た部分，即ち，最小*count*ビット分が返り値になります（返り値の残りの部分はゼロとなります）。

*count*は 1 以上，`mp_bits_per_limb`−1 以下でなければなりません。{*sp*, *n*}，{*rp*, *n*} のメモリ領域は重なりがあっても，$rp \geq sp$ であれば問題ありません。

この関数は大方の CPU に対応したアセンブラで記述されています。

`mp_limb_t mpn_rshift` (*mp_limb_t* ***rp**, *const mp_limb_t* ***sp**, *mp_size_t* **n**, *unsigned int* **count**) [関数]

{*sp*, *n*} を右方向に*count*ビットシフトし，結果を{*rp*, *n*} に書き込みます。右側シフトの結果はみ出た最大*count*ビット分が返り値となります(それ以外の部分はゼロです）。

*count*は 1 以上`mp_bits_per_limb`−1 以下でなければなりません。{*sp*, *n*} と{*rp*, *n*} のメモリ領域は，$rp \leq sp$ であれば，重なり部分があっても構いません

この関数は大方の CPU 向けにアセンブラコードが提供されています。

`int mpn_cmp` (*const mp_limb_t* ***s1p**, *const mp_limb_t* ***s2p**, *mp_size_t* **n**) [関数]

{*s1p*, *n*} と{*s2p*, *n*} を比較し，$s1 > s2$ であれば正数を，等しければゼロを，$s1 < s2$ であれば負数を返します。

`int mpn_zero_p` (*const mp_limb_t* ***sp**, *mp_size_t* **n**) [関数]

{*sp*, *n*} を調べ，ゼロの時は 1 を，それ以外の時は 0 を返します。

`mp_size_t mpn_gcd` (*mp_limb_t* ***rp**, *mp_limb_t* ***xp**, *mp_size_t* **xn**, *mp_limb_t* ***yp**, *mp_size_t* **yn**) [関数]

{*xp*, *xn*} と{*yp*, *yn*} の最大公約数(Greatest Common Divisor, GCD)を求め，{*rp*, *retval*} に格納します。GCD は最大*yn*リムになりますので，返り値は実際の GCD のリム長となります。2 数については上書きされて元の値ではないものになります。

この関数は$xn \geq yn > 0$，かつ，{*yp*, *yn*} の最大有効リムが非ゼロでなければ実行できません。{*xp*, *xn*} と{*yp*, *yn*} のメモリ領域が互いに重なりあってもいけません。

`mp_limb_t mpn_gcd_1` (*const mp_limb_t* ***xp**, *mp_size_t* **xn**, *mp_limb_t* **ylimb**) [関数]

{*xp*, *xn*} と*ylimb*の最大公約数を返します。2 数は非ゼロでなければなりません。

`mp_size_t mpn_gcdext` (*mp_limb_t* ***gp**, *mp_limb_t* ***sp**, *mp_size_t* ***sn**, *mp_limb_t* ***up**, *mp_size_t* **un**, *mp_limb_t* ***vp**, *mp_size_t* **vn**) [関数]

U は{*up*, *un*}，V は{*vp*, *vn*} であるとします。

この関数はU とV の GCD であるG を計算すると共に，$G = US + VT$ を満足する cofactor であるS も求めます。2 番目の cofactor であるTは計算されませんが，$(G - US)/V$ が成立するので簡単に求めることができます(割算は正確に行えます)。$un \geq vn > 0$ かつ，{*vp*, *vn*} の最大有効リムは非ゼロである必要があります。

S は$S = 1$ か，$|S| < V/(2G)$ を満足します。V がU を割り切る時（即ち，$G = V$ の時)に限り，$S = 0$ となります。

G は*gp*に格納され，そのリム長は返り値になります。S は*sp*に格納され，|**sn*| がそのリム長になります。S が負数の時は，**sn*が負数になります。*gp*のメモリ空間は*vn*リム長，*sp*のリム長は$vn + 1$ 必要になります。

入力引数は破壊されます。

互換性に関する注意：GMP 4.3.0 と 4.3.1 ではS が若干異なっています。これ以前の GMP では以降の GMP 同様，S は上述のように記述されています。GMP 4.3.0 以前では更に余分のメモリスペースが入出力に際して必要でした。正確に言うと，{*up*, $un+1$} と{*vp*, $vn+1$} のエリアが破壊されます（即ち，余分の 1 リムが必要であったということ）。*gp* と*sp*が指し示すメモリエリアは$un+1$ リム分必要であったことになります。

`mp_size_t mpn_sqrtrem` (*mp_limb_t* ***r1p**, *mp_limb_t* ***r2p**, *const mp_limb_t* ***sp**, *mp_size_t* **n**) [関数]

{*sp*, *n*} の平方根を計算し，結果を{*r1p*, $\lceil n/2 \rceil$} に格納し，余りを{*r2p*, *retval*} に格納します。*r2p*はnリム分必要になります。何リム生成したのかは返り値で分かります。

{*sp*, *n*} の最大有効リムは非ゼロでなければなりません。{*r1p*, $\lceil n/2 \rceil$} と{*sp*, *n*} のメモリ空間は完全に分離されていなければなりません。{*r2p*, *n*} と{*sp*, *n*} は同一のエリアか，完全分離されたものかのどちらかでなければなりません。

剰余が不要であれば*r2p*は`NULL`に設定して下さい。この場合，剰余がゼロか非ゼロかによって返り値もそれにならいます。

返り値がゼロであれば，入力値は完全平方数です。`mpn_perfect_square_p`関数の解説も参照して下さい。

`size_t mpn_sizeinbase` (*const mp_limb_t* ***xp**, *mp_size_t* **n**, *int* **base**) [関数]

基数*base*である時の，{*xp*,*n*} の桁数を返します。*base*は 2 以上 62 以下に設定できます。$n > 0$ かつ$xp[n-1] > 0$ でなければなりません。返り値は正確な桁数か，大きすぎる場合は 1 になります。*base*が 2 のべき乗である時には，結果は常に正確なものになります。

`mp_size_t mpn_get_str` (*unsigned char* ***str**, *int* **base**, *mp_limb_t* ***s1p**, *mp_size_t* **s1n**) [関数]

{*s1p*, *s1n*} を unsigned char の配列である*str*に，基数*base*の文字列に変換して格納します。返り値は生成された文字列の長さです。文字列の先頭部分はゼロで埋められます。文字列は ASCII コードではないので，表示するためには‘`0`’ もしくは‘`A`’を加えて ASCII コード化する必要があります。加えるべき値は基数によります。基数*base*は 2 以上 62 以下で指定可能です。

入力引数{*s1p*, *s1n*} の最大有効リムは非ゼロである必要があります。*base*が 2 のべき乗であれば入力値{*s1p*, *s1n*} はそのままですが，それ以外の場合はぶち壊されます。

*str*のメモリ空間は*s1n*長のリム配列で表現でき，1 字分の余分エリアを加えて，最大の数を格納できるだけの広さが必要です。

`mp_size_t mpn_set_str` (*mp_limb_t* ***rp**, *const unsigned char* ***str**, *size_t* **strsize**, *int* **base**) [関数]

*base*を基数とする{*str*,*strsize*} バイト列を，*rp*のリムに変換します。

str[0] は入力値の最大有効バイト，*str*[*strsize* − 1] は最小有効バイトです。配列の各バイトは 0 以上*base* − 1 以下の整数値で，ASCII コードは受け付けません。基数*base*は 2 以上 256 以下です。

変換される値は，{*rp*,*rn*} となり，*rn*が返り値になります。最大有効バイト*str*[0] が非ゼロであれば，*rp*[*rn* − 1] が非ゼロになるか，*rp*[*rn* − 1] 以下の複数リムがゼロになる可能性があります。

*rp*のメモリ空間は，指定基数での最大桁数の数に余分の 1 リムを加えた分を格納できる*strsize*でなければなりません。

入力値は最小でも 1 バイトなければならず，{*str*,*strsize*} と結果を格納する*rp*のメモリ領域に重複があってはいけません。

`mp_bitcnt_t mpn_scan0` (*const mp_limb_t* ***s1p***, *mp_bitcnt_t* ***bit***) [関数]
*bit*ビット目から*s1p*を調べ，次の"0"ビットを見つけます。

この関数は，値を返すために，*s1p*の範囲内か，*bit*以降の位置に"0"ビットが見つかることを前提としています。

`mp_bitcnt_t mpn_scan1` (*const mp_limb_t* ***s1p***, *mp_bitcnt_t* ***bit***) [関数]
*bit*ビットから*s1p*を調べ，次の"1"を見つけます。

この関数は，*s1p*の範囲内か，*bit*以降のエリアに"1"が見つかることを前提としています。

`void mpn_random` (*mp_limb_t* ***r1p***, *mp_size_t* ***r1n***) [関数]
`void mpn_random2` (*mp_limb_t* ***r1p***, *mp_size_t* ***r1n***) [関数]
*r1n*リムの長さの乱数を生成し，*r1p*に格納します。この乱数の最大有効リムは常に非ゼロになります。`mpn_random`関数は一様分布リムデータを生成し，`mpn_random2`関数は，ランダムに 0 と 1 が並んだ長い 2 進表現列を生成します。

`mpn_random2`関数は，`mpn`関数の処理の正しさを確認するために作られました。

`mp_bitcnt_t mpn_popcount` (*const mp_limb_t* ***s1p***, *mp_size_t* ***n***) [関数]
{*s1p*, *n*} における"1"のビット数を数えます。

`mp_bitcnt_t mpn_hamdist` (*const mp_limb_t* ***s1p***, *const mp_limb_t* ***s2p***, [関数]
mp_size_t ***n***)
{*s1p*, *n*} と{*s2p*, *n*} のハミング距離を計算します。ハミング距離とは，2 数の 2 進表現におけるビット単位の差異の数です。

`int mpn_perfect_square_p` (*const mp_limb_t* ***s1p***, *mp_size_t* ***n***) [関数]
{*s1p*, *n*} が完全平方数である時のみ，非ゼロを返します。入力引数{*s1p*, *n*} の最大有効リムは非ゼロでなければなりません。

`void mpn_and_n` (*mp_limb_t* ***rp***, *const mp_limb_t* ***s1p***, *const mp_limb_t* [関数]
s2p, *mp_size_t* ***n***)
{*s1p*, *n*} と{*s2p*,*n*} のビット単位の論理積を求め，その結果を{*rp*, *n*} に書き込みます。

`void mpn_ior_n` (*mp_limb_t* ***rp***, *const mp_limb_t* ***s1p***, *const mp_limb_t* [関数]
s2p, *mp_size_t* ***n***)
{*s1p*, *n*} と{*s2p*, *n*} のビット単位の論理和を求め，その結果を{*rp*, n} に書き込みます。

`void mpn_xor_n` (*mp_limb_t* ***rp***, *const mp_limb_t* ***s1p***, *const mp_limb_t* [関数]
s2p, *mp_size_t* ***n***)
{*s1p*, *n*} と{*s2p*, *n*} のビット単位の排他的論理和を求め，その結果を{*rp*, *n*} に書き込みます。

`void mpn_andn_n` (*mp_limb_t* ***rp***, *const mp_limb_t* ***s1p***, *const mp_limb_t* [関数]
s2p, *mp_size_t* ***n***)
{*s1p*, *n*} と{*s2p*, *n*} の補数のビット単位の論理積を求め，その結果を{*rp*, *n*} に書き込みます。

void mpn_iorn_n (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [関数]
{*s1p*, *n*} と{*s2p*, *n*} の補数のビット単位の論理和を求め，その結果を{*rp*, *n*} に書き込みます。

void mpn_nand_n (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [関数]
{*s1p*, *n*} と{*s2p*, *n*} のビット単位の論理積を求め，その結果の補数を{*rp*, *n*} に書き込みます。

void mpn_nior_n (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [関数]
{*s1p*, *n*} と{*s2p*, *n*} のビット単位の論理和を求め，その結果の補数を{*rp*, *n*} に書き込みます。

void mpn_xnor_n (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [関数]
{*s1p*, *n*} と{*s2p*, *n*} のビット単位の排他的論理和を求め，更にその結果の補数を{*rp*, *n*} に書き込みます。

void mpn_com (*mp_limb_t *rp, const mp_limb_t *sp, mp_size_t n*) [関数]
{*sp*, *n*} のビット単位の補数を取り（訳注：ビット反転?），その結果を{*rp*, *n*} に書き戻します。

void mpn_copyi (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n*) [関数]
{*s1p*, *n*} から{*rp*, *n*} へ，高次リム方向からコピーします。

void mpn_copyd (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n*) [関数]
{*s1p*, *n*} から{*rp*, *n*} へ，低次リム方向からコピーします。

void mpn_zero (*mp_limb_t *rp, mp_size_t n*) [関数]
{*rp*, *n*} をゼロにします。

8.1 暗号処理のための基盤関数

mpn_sec_ やmpn_cnd_という名前で始まる基盤関数群は，二つの同じサイズの引数に対して，同じ基盤処理を実行し，同じキャッシュアクセスパターンが生じるように実装されたもので，同じ位置に引数が存在し，マシンの状態が監修の処理開始時から変化しないという前提の元に構築されています。というのも，この関数群は暗号処理を目的としており，サイドチャンネル攻撃(side-channel attack)を防御できるように設計してあるからです。

ここで述べる暗号処理用関数群は，同じ機能を持つ「リークしやすい」関数群よりパフォーマンスが劣り，同じサイズの引数で暗号化しようとすると，15% ～ 100% 程度，遅くなってしまいます。より大きなサイズの引数に対してはもっと不利になり，漸近的に初等的なアルゴリズムの演算量に引きずられていってしまいます。

ここで述べる関数群は，明示的なメモリ割り当てを行わず，中間処理のためのメモリ空間を要求する関数はそれに当たる引数をを受け付けます。これによって，あらかじめ指定されたメモリ領域でデータ処理が行えるようになります。しかしながら，コンパイラが，指定メモリ領域にあるはずのスカラー値があふれた時にはスタック領域も確保し，結果としてそこで注意すべきデータを扱ってしまう可能性があることは留意しておいて下さい。

暗号処理用基盤関数以外にも，サイドチャネル攻撃防御が可能な基盤関数群も存在します。mpn_add_n, mpn_sub_n, mpn_lshift, mpn_rshift, mpn_zero, mpn_copyi, mpn_copyd, mpn_com, 基盤論理関数(mpn_and_n等)がそれにあたります。

但し，サイドチャネル攻撃防御性に関してはいくつかの例外があります。(1) mpn_lshift関数のアセンブラ実装の中には，shift-by-one(1 によるシフト)が特殊ケースとして処理していることがあり，シフト値が注意すべきデータである時に限り，問題が発生します。(2)64 ビットリムを用いる Alpha ev6 と Pentium4 実装では，mpn_add_n関数とmpn_sub_n関数に「リークしやすい」問題があります。(3) Alpha ev6 の実装では，mpn_mul_1関数もリークしやすく，結果としてmpn_sec_mul関数が影響を受けてこのシステムの安全性を損なっています。

mp_limb_t mpn_cnd_add_n (*mp_limb_t* ***cnd***, *mp_limb_t* ***rp***, *const mp_limb_t* ***s1p***, *const mp_limb_t* ***s2p***, *mp_size_t* ***n***) [関数]

mp_limb_t mpn_cnd_sub_n (*mp_limb_t* ***cnd***, *mp_limb_t* ***rp***, *const mp_limb_t* ***s1p***, *const mp_limb_t* ***s2p***, *mp_size_t* ***n***) [関数]

この 2 つの関数は，条件付き加算と減算を実行します。*cnd*が非ゼロであれば，mpn_add_n関数やmpn_sub_n関数と同じ演算を実行します。*cnd*がゼロであれば，{*s1p*,*n*} を結果の格納先にコピーし，ゼロを返します。この関数は，条件値*cnd*とは独立に，データエリアの位置やサイズにのみ依存した計算時間やメモリアクセスパターンを持つように設計されています。大概のマシン上ではmpn_add_n関数やmpn_sub_n関数は，計算時間も実際のリムの値とは独立に決まります。

mp_limb_t mpn_sec_add_1 (*mp_limb_t* ***rp***, *const mp_limb_t* ***ap***, *mp_size_t* ***n***, *mp_limb_t* ***b***, *mp_limb_t* ***tp***) [関数]

mp_limb_t mpn_sec_sub_1 (*mp_limb_t* ***rp***, *const mp_limb_t* ***ap***, *mp_size_t* ***n***, *mp_limb_t* ***b***, *mp_limb_t* ***tp***) [関数]

*R*に*A* + *b* および*A* - *b*を格納しますs。ここで$R = \{rp,n\}$, $A = \{ap,n\}$ で，*b*は 1 リムです。返り値は桁上がり・桁下がりとなります。

リークしやすいmpn_add_1関数が平均$O(1)$ であるのに対し，これらの関数は$O(N)$ かかります。また，スクラッチスペースとしてmpn_sec_add_1_itch(n)リム長，mpn_sec_sub_1_itch(n)リム長必要になり，これらを*tp*に渡す必要があります。スクラッチスペースは*n*リム長あることが必須で，引数の長さに比例して増やせるようにしておかなければいけません。

void mpn_cnd_swap (*mp_limb_t* ***cnd***, *volatile mp_limb_t* ***ap***, *volatile mp_limb_t* ***bp***, *mp_size_t* ***n***) [関数]

*cnd*が非ゼロの時は，{*ap*,*n*} と{*bp*,*n*} の値を交換します。*cnd*がゼロの時は何もしません。リム上の論理演算を用いて実装されており，*cnd*の値とは無関係に，同じメモリアクセスだけを使用しています。

void mpn_sec_mul (*mp_limb_t* ***rp***, *const mp_limb_t* ***ap***, *mp_size_t* ***an***, *const mp_limb_t* ***bp***, *mp_size_t* ***bn***, *mp_limb_t* ***tp***) [関数]

mp_size_t mpn_sec_mul_itch (*mp_size_t* ***an***, *mp_size_t* ***bn***) [関数]

$A \times B$ を計算し，*R*に格納します。ここで$A = \{ap,an\}$, $B = \{bp,bn\}$, $R = \{rp,an + bn\}$ です。

この関数を実行するには$an \geq bn > 0$ でなければなりません。

*R*と入力引数とメモリ空間に重複があってはいけません。$A = B$ の場合は，mpn_sec_sqr関数の方が高速に実行できます。

この関数はmpn_sec_mul_itch(an, bn)リム長のスクラッチスペースが必要で，*tp*に渡します。このスクラッチスペースは，入力値の長さに単調に比例して長く確保しなければなりません。

`void mpn_sec_sqr` (*mp_limb_t* ***rp*, *const mp_limb_t* ***ap*, *mp_size_t* **an**, *mp_limb_t* ***tp*) [関数]
`mp_size_t mpn_sec_sqr_itch` (*mp_size_t* **an**) [関数]

A^2 を計算してRに格納します。ここで$A = \{ap,an\}$, $R = \{rp,2an\}$ です。

この関数を実行するには$an > 0$ でなければなりません。

Rと入力値に重複があってはいけません。

この関数は`mpn_sec_sqr_itch(an)`リム長のスクラッチスペースが必要で，tpに渡します。このスクラッチスペースのは，入力値の長さに単調に比例して長く確保しなければなりません。

`void mpn_sec_powm` (*mp_limb_t* ***rp*, *const mp_limb_t* ***bp*, *mp_size_t* **bn**, *const mp_limb_t* ***ep*, *mp_bitcnt_t* **enb**, *const mp_limb_t* ***mp*, *mp_size_t* **n**, *mp_limb_t* ***tp*) [関数]
`mp_size_t mpn_sec_powm_itch` (*mp_size_t* **bn**, *mp_bitcnt_t* **enb**, *size_t* **n**) [関数]

$B^E \bmod M$ を求めてRに格納します。ここで$R = \{rp,n\}$, $M = \{mp,n\}$, $E = \{ep,\lceil enb/\texttt{GMP\_NUMB\_BITS}\rceil\}$ です。

この関数を実行するには$B > 0$ であり，$M > 0$ が奇数であり，$E < 2^{enb}$ である必要があります。

Rと入力値とのメモリ空間における重複があってはいけません。

この関数は`mpn_sec_powm_itch(bn, enb, n)`リム長のスクラッチスペースが必要で，tpに渡します。このスクラッチスペースは，入力値の長さに単調に比例して長く取らなければなりません。

`void mpn_sec_tabselect` (*mp_limb_t* ***rp*, *const mp_limb_t* ***tab*, *mp_size_t* **n**, *mp_size_t* **nents**, *mp_size_t* **which**) [関数]

tabテーブルから$which$エントリを抜き出します。このテーブルは$nents$個のエントリがあり，それぞれnリム長あります。抜き出されたエントリはrpに格納されます s。

この関数はエントリテーブルを読み込み，サイドチャネル情報の漏えいを防止します。

`mp_limb_t mpn_sec_div_qr` (*mp_limb_t* ***qp*, *mp_limb_t* ***np*, *mp_size_t* **nn**, *const mp_limb_t* ***dp*, *mp_size_t* **dn**, *mp_limb_t* ***tp*) [関数]
`mp_size_t mpn_sec_div_qr_itch` (*mp_size_t* **nn**, *mp_size_t* **dn**) [関数]

$\lfloor N/D \rfloor$ とR to $N \bmod D$ を求め，それぞれQとRに格納します。ここで$N = \{np,nn\}$, $D = \{dp,dn\}$ です。Qの最大有効リムが返り値となり，残りのリムは$\{qp,nn\text{-}dn\}$, と$R = \{np,dn\}$ になります。

この関数を実行するに際しては，$nn \geq dn \geq 1$ であり，かつ$dp[dn-1] \neq 0$ でなければなりません。Nはゼロが詰まっている可能性があるので，この条件は$N \geq D$ であることを意味しません。

NとRに重複があっても構いません。他の引数との重複はダメです。N のスペースは全部上書きされます。

この関数は`mpn_sec_div_qr_itch(nn, dn)`リム長のスクラッチスペースが必要で，tpに渡します。

`void mpn_sec_div_r` (*mp_limb_t* ***np**, *mp_size_t* **nn**, *const mp_limb_t* ***dp**, *mp_size_t* **dn**, *mp_limb_t* ***tp**) [関数]

`mp_size_t mpn_sec_div_r_itch` (*mp_size_t* **nn**, *mp_size_t* **dn**) [関数]

$N \bmod D$ を計算し，Rに格納します。ここで$N = \{np,nn\}$, $D = \{dp,dn\}$, $R = \{np,dn\}$ です。

この関数を実行するに際しては，$nn \geq dn \geq 1$ かつ，$dp[dn-1] \neq 0$ でなければなりません。Nはゼロが詰まっている可能性があるので，この条件は$N \geq D$ であることを意味しません。

NとRに重複があっても構いません。他の引数との重複はダメです。N のスペースは全部上書きされます。

この関数は`mpn_sec_div_r_itch(nn, dn)`リム長のスクラッチスペースが必要で，tpに渡します。

`int mpn_sec_invert` (*mp_limb_t* ***rp**, *mp_limb_t* ***ap**, *const mp_limb_t* ***mp**, *mp_size_t* **n**, *mp_bitcnt_t* **nbcnt**, *mp_limb_t* ***tp**) [関数]

`mp_size_t mpn_sec_invert_itch` (*mp_size_t* **n**) [関数]

Rに$A^{-1} \bmod M$ を格納します。ここで$R = \{rp,n\}$, $A = \{ap,n\}$ かつ$M = \{mp,n\}$ です。この関数の引数は原始的なものです。

逆数が存在していれば返り値は 1，存在していなければ 0 で，この場合はRは不定になります。どちらの場合でも，入力値Aは破壊されます。

Mは奇数，かつ$nbcnt \geq \lceil \log(A+1) \rceil + \lceil \log(M+1) \rceil$ でなければなりません。. 安全策は$nbcnt = 2n \times$ `GMP_NUMB_BITS` ですが，MやAの先頭ビットに 0 が並ぶような場合は，小さい値を使うことでパフォーマンスが向上します。

この関数は，tpパラメータに渡すために，あらかじめ`mpn_sec_invert_itch(n)`分のメモリが必要になります。

8.2 ネイル

この節で述べていることは全て実験的なもので，将来の GMP のバージョンにおいては消去されるか，互換性を保持しない変更が行われる可能性があります。

ネイル(nails)とは，リム（`mp_limb_t`）ごとに置かれる，使用しない先頭部分の数ビットのことです。ネイルを置くことで，プロセッサによっては，桁上がり・桁下がりの操作を劇的に改善することができるようになります。

基盤`mpn`関数は全てリムデータを受け取り，ネイルビットにゼロがあると仮定します。さすれば，返す値も同様にネイルビットゼロの値を返します。リムの配列にも，単独リムでも同様にネイルビットを適用します。

ネイルは‘`--enable-nails`’オプション付きで設定すると使用できるようになります。デフォルトでは，プロセッサごとにネイルビット長を規定していますが，特定の長さにしたければ‘`--enable-nails=N`’オプションで長さを指定します。

mpn 関数レベルでは，ネイルビットをオンにしたビルドはソースレベルでもバイナリレベルでもネイルビット無の関数とは非互換になりますが，mpn 関数を使ってリムを処理するプログラムレベルでは，ネイルビットの有無は関係なく同じ動作を行います。本節で述べる下記の定義を使うことで，プログラムのソースレベルでの互換性を維持することができるようになります。

`mpz`等の高レベルの関数では，ネイルビット有の場合でも，ネイルビット無の場合との互換性は維持するようにしなくてはなりません。

GMP_NAIL_BITS [マクロ]
GMP_NUMB_BITS [マクロ]
GMP_LIMB_BITS [マクロ]
GMP_NAIL_BITSはネイルビット長を表わします。0 の場合はネイルビットを使用しません。GMP_NUMB_BITSはリムのビット長を表わします。GMP_LIMB_BITS はmp_limb_t型の全体のビット長を表わします。つまり下記のような関係式が成り立ちます。

```
GMP_LIMB_BITS == GMP_NAIL_BITS + GMP_NUMB_BITS
```

GMP_NAIL_MASK [マクロ]
GMP_NUMB_MASK [マクロ]
前者がリムのマスク長，後者がリムの残り部分のマスク長と表わします。GMP_NAIL_MASKがゼロの時はネイルを使用しません。

GMP_NAIL_MASKの使用頻度は高くありません。ネイル部分はx >> GMP_NUMB_BITSで取り出すことができ，これはさほど大きくない定数になるからです。RISC チップに対しては有効かもしれません。

GMP_NUMB_MAX [マクロ]
リム部分に格納できる最大の数を表わします。GMP_NUMB_MASKと同じものになりますが，ビット単位の操作よりも比較時に分かりやすくなります。

ネイル(nail)という用語は，指の爪に由来します。つまり，リム(腕や足)の先っぽ，ということです。‘‘numb"は数(number)の短縮形ですが，長時間，このような扱いづらい名前を見つけようとした後に開発者がどう感じるか(マヒする，凍える = numb)という意味も込めています。

将来（多分当分先）は，非ゼロのネイルを使うようになり，リム配列として表現する数には様々なネイルが入り込むことになるでしょう。さすれば，ベクトルプロセッサでは，桁上がり・桁下がりの操作を 1 ないし 2 リムだけで完結するようにできるからです。

9 乱数関数

GMP における準乱数列は`gmp_randstate_t`型の変数を使い，アルゴリズムを選択したり，乱数生成状態を保存しながら生成されます。この変数は`gmp_randinit`関数で初期化し，`gmp_randseed`関数で乱数の種(seed)を設定します。

実際に乱数を生成する関数についてはSection 5.13 [Integer Random Numbers], p. 43やSection 7.8 [Miscellaneous Float Functions], p. 58を参照して下さい。

以前の乱数関数は`gmp_randstate_t`変数ではなく，グローバル変数を利用しており，デフォルトのアルゴリズムや種の変更ができませんでした（が，そのうち変わるかも）。ここで述べる，`gmp_randstate_t`変数が利用可能な新しい乱数関数は，ランダム性も改善されているので，こちらの利用をお勧めします。

9.1 乱数状態変数の初期化

`void gmp_randinit_default (`*gmp_randstate_t* **`state`**`)` [関数]
*state*を初期化してデフォルトのアルゴリズムをセットします。アルゴリズムは乱数生成の速度とランダム性のバランスを決定しますので，特別な要求が不要のアプリケーションにはデフォルトのものを使用しましょう。現行の実装では`gmp_randinit_mt`を使います。

`void gmp_randinit_mt (`*gmp_randstate_t* **`state`**`)` [関数]
*state*を初期化してメルセンヌ・ツイスター(Mersenne Twister)アルゴリズムをセットします。このアルゴリズムは高速で良いランダム性を持っています。

`void gmp_randinit_lc_2exp (`*gmp_randstate_t* **`state`**`, `*const mpz_t* `a`*,* *unsigned long* **`c`**`, `*mp_bitcnt_t* **`m2exp`**`)` [関数]
*state*を初期化して線型合同法$X = (aX + c) \bmod 2^{m2exp}$ をセットします。

このアルゴリズムではX の低位ビットのランダム性があまり良くありません。最小有効ビットの周期は 2 以下で，2 番目のビットの周期は 4 以下，・・・となります。X の上位半分だけを使って生成することに起因します。

m2exp/2 bits 以上の乱数を生成する場合は，反復を繰り返して結果を繋ぎ合わせる他ありません。

`int gmp_randinit_lc_2exp_size (`*gmp_randstate_t* **`state`**`, `*mp_bitcnt_t* **`size`**`)` [関数]
*state*を初期化して線型合同法をセットし，`gmp_randinit_lc_2exp`関数に適用します。*a*，*c*，*m2exp*はテーブルから選び，X の*size*ビット以上を使用する，つまり$m2exp/2 \geq size$ であるようにします。

成功していれば返り値は非ゼロになります。*size*がテーブルのデータより大きい時にはゼロを返します。*size*の最大値は現状では 128 です。

`void gmp_randinit_set (`*gmp_randstate_t* **`rop`**`, `*gmp_randstate_t* **`op`**`)` [関数]
*rop*を初期化して，アルゴリズムや状態を*op*からコピーします。

`void gmp_randinit (`*gmp_randstate_t* **`state`**`, `*gmp_randalg_t* **`alg`**`, ...)` [関数]
この関数は廃止予定です。

*state*と初期化して*alg*で指定されたアルゴリズムをセットします。今のところ`GMP_RAND_ALG_LC`のみ指定可能で，`gmp_randinit_lc_2exp_size`関数が使われます。3 つ目のパラメー

タはunsigned long型変数で，関数に対するsizeを決めるものです。GMP_RAND_ALG_DEFAULT もしくは 0 と指定すると，どちらもGMP_RAND_ALG_LCと指定されたことになります。

gmp_randinit関数はグローバル変数であるgmp_errnoにビットをセットしてエラー発生が分かるようにします。algがサポートされていなければ，GMP_ERROR_UNSUPPORTED_ARGUMENTとなり，sizeパラメータが大きすぎるとGMP_ERROR_INVALID_ARGUMENTとなります。このエラー検知機能はスレッドセーフではありません（不都合であれば代わりにgmp_randinit_lc_2exp_size関数を使って下さい。）。

void gmp_randclear (*gmp_randstate_t* **state**) [関数]
*state*変数のメモリ領域を全て解放します。

9.2 乱数の種

void gmp_randseed (*gmp_randstate_t* **state**, *const mpz_t* **seed**) [関数]
void gmp_randseed_ui (*gmp_randstate_t* **state**, *unsigned long int* **seed**) [関数]
乱数の種(seed)を*state*にセットします。

種のサイズは，生成できる乱数系列の種類を決定します。種の「質」とは，以前に使用された種と比較してランダム性に変化があったかどうか，別の乱数系列のランダム性に影響があるかどうかということです。種を選択する手法は大変重要で，例えば暗号鍵を生成したりする重要なアプリケーションに影響を与えます。

伝統的に，種にはシステム時刻を使用することが普通でしたが，これには注意が必要です。システム時刻の粒度が足りないと，同じ乱数系列が繰り返し現れることがあります。また，システム時刻は簡単に類推できますので，乱数の種としてはあまりふさわしくありません。システムによっては，乱数の種用として特別な/dev/randomというデバイスを用意してある場合もあります。

9.3 その他の乱数生成関数

unsigned long gmp_urandomb_ui (*gmp_randstate_t* **state**, *unsigned long* **n**) [関数]
n bit の一様乱数を返します。値の範囲は 0 以上$2^n - 1$ 以下になります。nはunsigned long型のビット数以下でなければなりません。

unsigned long gmp_urandomm_ui (*gmp_randstate_t* **state**, *unsigned long* **n**) [関数]
0 以上$n - 1$ 以下の一様乱数を返します。

10 書式指定出力

10.1 書式指定文字列

gmp_printf等の出力関数は，標準 C 関数であるprintf (see Section “Formatted Output” in *The GNU C Library Reference Manual*)と同様の書式指定文字列が利用できます。指定方式は下記の通りです。

```
% [flags] [width] [.[precision]] [type] conv
```

GMP 用の書式指定子として，新たに‘Z’, ‘Q’，‘F’を，それぞれ，mpz_t, mpq_t，mpf_t型変数用に追加してあります。更に，‘M’はmp_limb_t型，‘N’はmp_limb_t型の配列用として追加されています。‘Z’, ‘Q’, ‘M’，‘N’は整数用の指定子です。‘Q’は必要があれば‘/’を分子と分母に挟んで出力します。‘F’は浮動小数点数用の指定子です。これらは次のように使います。

```
mpz_t z;
gmp_printf ("%s is an mpz %Zd\n", "here", z);

mpq_t q;
gmp_printf ("a hex rational: %#40Qx\n", q);

mpf_t f;
int   n;
gmp_printf ("fixed point mpf %.*Ff with %d digits\n", n, f, n);

mp_limb_t l;
gmp_printf ("limb %Mu\n", l);

const mp_limb_t *ptr;
mp_size_t       size;
gmp_printf ("limb array %Nx\n", ptr, size);
```

‘N’は，mpn関数(see Chapter 8 [Low-level Functions], p. 59)の処理がそうであるように，有効桁を最初に出力します。負数のサイズが指定されている時には，その値が負であることを示しています。

基本的に，標準 C のprintf関数の書式出力はそのまま利用でき，GMP 用の拡張書式出力と混ぜこぜにしても問題ありません。現在の実装では，標準の書式出力はprintf関数にそのまま引き渡し，GMP 用の拡張書式指定だけ直接制御しています。

書式指定におけるフラグは下記のものが使えます。GLIBC のスタイルである‘’’は，その C 言語のライブラリがサポートする限りは標準 C のデータ型に対してのみ利用でき，GMP では使えません。

0	スペースではなくゼロで詰める
#	基数の表示（‘0x’, ‘0X’，‘0’指定時）
+	必ず符号を表示
(space)	スペースか‘-’符号の表示
’	数桁ごとのグルーピング（GLIBC のスタイル）(GMP 型は不適用)

表示幅指定や桁数指定は標準printf関数と同様に，書式指定の中にその数字を埋め込んだり，int型に対しては‘*’で与えます。

標準データ型は次のように与えます。‘h’や‘l’は標準のものと同じで，それ以外のものはコンパイラやヘッダファイルに依存して決まります。

h	short
hh	char
j	intmax_t またはuintmax_t
l	long またはwchar_t
ll	long long
L	long double
q	quad_t またはu_quad_t
t	ptrdiff_t
z	size_t

GMP のデータ型に対しては下記のように与えます。

F	mpf_t, 浮動小数点形式
Q	mpq_t, 有理数形式(訳注:オリジナルは integer conversion)
M	mp_limb_t, 整数形式
N	mp_limb_t 配列, 整数形式
Z	mpz_t, 整数形式

出力形式の変更については下記のようなものが使えます。‘a’と‘A’はmpf_tに対しては常に使用可能ですが，出力形式は使用している C ライブラリの標準 C 浮動小数点数の形式に依存します。‘m’と‘p’の出力形式は C ライブラリに依存します。

a A	C99 形式の 16 進浮動小数点形式
c	文字
d	10 進整数
e E	科学技術形式（小数部 E 指数部)
f	固定小数点形式
i	dと同じ
g G	固定小数点形式か科学技術形式
m	GLIBC 形式のstrerror文字列
n	既に書きこまれた文字数
o	8 進整数
p	ポインタ
s	文字列
u	符号なし整数
x X	16 進整数

‘o’, ‘x’，‘X’は標準 C データ型を符号なし値として出力しますが，‘Z’, ‘Q’，‘N’に対しては符号つきで表示されます。

‘M’は C ライブラリが提供する‘l’もしくは‘L’の代替物で，そのサイズはmp_limb_tによります。符号なし形式で使うのが普通ですが，符号あり形式でも使用でき，その場合は 2 の補数による負数表現で解釈します。

‘n’は，GMP データ型も含むすべてのデータ型に対して指定できます。

printfで使えるその他のデータ出力形式はgmp_printf関数では使用できません。GLIBCのregister_printf_function関数で登録したものに関しても同様です。現状では POSIX '$'形式もサポートしていません（が，将来はサポートされるかも）。

精度指定部分は，整数に対する'Z'や，浮動小数点型に対する'F'でも同じ意味になりますが，現状では'Q'に対しては定義されていませんので，使わないようにして下さい。

mpf_tの出力形式は，表現すべき値を表現するために必要となる桁数を出力します。mpf_get_str関数の結果についても同様です。要求される表示桁数に満たない時にはゼロで埋めます。これはmpf_tの値が整数である場合に，'f'を指定した時にも同じようにゼロで埋めます。例えば2^{1024}が 128bits のmpf_t型に収められているとすると，約 40 桁しか表現できませんので残りはゼロで埋めます。表示桁数の指定が空の場合，例えば'%.Fe'や'%.Ff'の時は，有効桁をすべて表示します。ドット指定もない場合は，6 桁表示したと見なします。つまり，'%Ff', '%.Ff'，'%.0Ff'は全部異なる表示形式となります。

小数点を表わす文字（もしくは文字列）は使用システムのロケールに依存します。これはlocaleconvで分かります(see Section "Locales and Internationalization" in *The GNU C Library Reference Manual*)。C ライブラリは普通，標準浮動小数点出力と同じ形式になります。

書式指定文字列としては ASCII 文字だけで指定する必要があり，マルチバイト文字では認識しませんが，将来はできるようになるかもしれません。

10.2 書式指定出力関数

以下で示す関数はそれぞれ対応する C ライブラリの関数とほぼ同じ機能を持ちます。printfでは変数の可変リストを，vprintfは変数へのポインタをそれぞれ使用します。詳細はSection "Variadic Functions" in *The GNU C Library Reference Manual*か，'man 3 va_start'を参照して下さい。

書式指定が間違っていたり，型指定が食い違っていたりした時の出力は予測不能です。GCC の書式指定チェック機構は，GMP の拡張指定に対応していないので，役に立ちません。

ファイルに対する出力関数であるgmp_printfやgmp_fprintf関数は，書き込みエラー時には−1を返します。出力は"atomic"ではありませんので，書き込みエラー時には一部だけが書きこまれる可能性があります。C ライブラリのprintf関数やその派生関数が-1を返すようになっていれば，GMP の出力関数もそれにならってエラー時には−1 を返します。

`int gmp_printf (const char *fmt, ...)` [関数]
`int gmp_vprintf (const char *fmt, va_list ap)` [関数]
標準出力stdoutに表示を行います。通常は出力文字数を返し，エラー時には−1 を返します。

`int gmp_fprintf (FILE *fp, const char *fmt, ...)` [関数]
`int gmp_vfprintf (FILE *fp, const char *fmt, va_list ap)` [関数]
ファイルストリーム*fp*に出力します。通常は出力文字数を返し，エラー時には−1 を返します。

`int gmp_sprintf (char *buf, const char *fmt, ...)` [関数]
`int gmp_vsprintf (char *buf, const char *fmt, va_list ap)` [関数]
*buf*に NULL 終端子付きの文字列を形成します。返り値は，NULL 終端子を除いた出力文字数になります。

*buf*スペースと，*fmt*文字列がメモリ空間を共有してはいけません。

*buf*スペース以上の出力を防止する機能はありませんので，これらの関数の使用はお勧めしません。

`int gmp_snprintf` (*char *buf, size_t size, const char *fmt, ...*) [関数]
`int gmp_vsnprintf` (*char *buf, size_t size, const char *fmt, va_list ap*) [関数]
*buf*に NULL 終端子付きの文字列を形成します。*size*バイト以上の書き込みはできません。従って，出力結果を全て補損しておきたい時には，*size*は文字列長＋ NULL 終端子以上のスペースが必要になります。

返り値は，NULL 終端子文を除いた生成すべき文字全体の長さとなります。*retval* ≥ *size* であれば，出力結果は最初の*size* − 1 文字分に，NULL 終端子を足した長さになります。

{*buf*,*size*} と*fmt*文字列とのメモリ空間共有は行わないで下さい。

返り値は ISO C99 `snprintf`関数と同じです。C ライブラリの`vsnprintf`関数が古い GLIBC 2.0.x の形式であっても同様です。

`int gmp_asprintf` (*char **pp, const char *fmt, ...*) [関数]
`int gmp_vasprintf` (*char **pp, const char *fmt, va_list ap*) [関数]
実行時におけるメモリ割り当て関数(see Chapter 13 [Custom Allocation], p. 91)を使って確保されたメモリブロックに，NULL 終端子付きの文字列を形成します。メモリブロックは NULL 終端子を含む文字列分確保され，このブロックへのポインタは *pp*に保存されます。返り値は，NULL 終端子を除く生成文字列長となります。

C の`asprintf`関数とは異なり，*gmp_asprintf* はメモリ不足の時にも −1 を返さず，必要なだけメモリを確保しようとします。

`int gmp_obstack_printf` (*struct obstack *ob, const char *fmt, ...*) [関数]
`int gmp_obstack_vprintf` (*struct obstack *ob, const char *fmt, va_list ap*) [関数]
既存オブジェクト*op*に追記します。返り値は書きこまれた文字数です。NULL 終端子は付きません。

*fmt*は，更に増える可能性のある*op*とは別に確保しなくてはなりません。

これらの関数は C ライブラリが obstack を利用できる場合のみ使用できます。つまり GNU のシステム限定となります。詳細はSection “Obstacks” in *The GNU C Library Reference Manual*をご覧下さい。

10.3 C++ 書式指定出力

本節に示した関数は`libgmpxx` (see Section 3.1 [Headers and Libraries], p. 18)が提供しているもので，C++サポート(see Section 2.1 [Build Options], p. 3)を有効にしている時のみ使用可能です。型宣言は`<gmp.h>`で行っています。

`ostream& operator<<` (*ostream& stream, const mpz_t op*) [関数]
`ios`の書式指定に従って*op*を*stream*に出力します。出力後は`ios::width`はゼロにリセットされます。この辺の挙動は標準の`ostream operator<<`ルーチンと同じです。

１６進でも８進でも１０進でも，*op*は符号付きで表示されます。標準`operator<<`ルーチンの動作とは異なり，２の補数表示は使いません。

`ostream& operator<<` (*ostream& stream, const mpq_t op*) [関数]
`ios`の書式指定に従って*op*を*stream*に出力します。出力後は`ios::width`はゼロにリセットされます。この辺の挙動は標準の`ostream operator<<`ルーチンと同じです。

出力は‘`5/9`’のように分数表現となりますが，分母が 1 の時は‘`123`’のように単なる整数として表示します。

１６進でも８進でも１０進でも，*op*は符号付きで表示されます。`ios::showbase`がセットされている時には，分子と分母（の表示が必要な時には）にそれぞ適用されます

`ostream& operator<<` (*ostream& stream*, *const mpf_t op*) [関数]
１６進でも８進でも１０進でも，*op*は符号付きで表示されます。出力後は`ios::width`はゼロにリセットされます。この辺の挙動は標準の`ostream operator<<`ルーチンと同じです。

小数点は標準の`operator<<`演算子と同じように表示されます。*stream*の表示は`std::locale`に従って行われるのが近年では普通です。

*double*型に対する標準の`operator<<` とは異なり，１６進でも８進でも出力できます。仮数部は１６進か８進，指数は１０進になります。１６進の場合は仮数部と指数部の間の文字（デリミタ）が‘`@`’になります。これは`mpf_out_str`関数と同じ仕様です。

`ios::showbase`がサポートされており，仮数部の基数がセットされます。例えば 16 進は‘`0x1.8`’や‘`0x0.8`’，8 進は‘`01.4`’ or ‘`00.4`’のように出力されます。8 進表記が少し奇妙ですが，10 進表現と差異をつけるためにこのような表記を行っています。

以上のオペレーターは，GMP データ型を通常の C++の入出力方法で表示できるようにするものです。下記のように使います。

```
mpz_t  z;
int    n;
...
cout << "iteration " << n << " value " << z << "\n";
```

当たり前のことですが，`ostream`出力(と`istream`入力も, see Section 11.3 [C++ Formatted Input], p. 81)は，GMP データ型に対してこれらの演算子がオーバーロードされている時のみ有効で，例えば，`+`を`mpz_t`型に対して適用したりすると，その結果は予測不能なものになります。オーバーロードしてあるクラスについてはChapter 12 [C++ Class Interface], p. 82を参照して下さい。

11 書式指定入力

11.1 書式指定入力文字列

gmp_scanf関数等の入力関数群は，標準入力関数scanfと同様の書式指定入力文字列(see Section "Formatted Input" in *The GNU C Library Reference Manual*)の使用が可能です。書式指定は下記のように行います。

```
% [flags] [width] [type] conv
```

GMP はmpz_t型用に'Z'，mpq_t型用に'Q'，mpf_t型用に'F'をそれぞれ追加の書式指定子としました。このうち'Z'と'Q'は整数型のように使用できます。現在の仕様では'Q'の場合は'/'と分母を読み込み時に認識します。'F'は浮動小数点型のように使用できます。

GMP 型の変数は全てポインタになっていますので，gmp_scanf関数に引数として渡す際に&を付加する必要はありません。使い方としては次のようになります。

```
/* "a(5) = 1234"という形式で入力 */
int   n;
mpz_t z;
gmp_scanf ("a(%d) = %Zd\n", &n, z);

mpq_t q1, q2;
gmp_sscanf ("0377 + 0x10/0x11", "%Qi + %Qi", q1, q2);

/* "topleft (1.55,-2.66)"という形式で入力 */
mpf_t x, y;
char  buf[32];
gmp_scanf ("%31s (%Ff,%Ff)", buf, x, y);
```

標準 C のscanf関数で使えるデータ型は全て利用可能で，GMP データ型用の拡張指定と混ぜて使っても問題ありません。現在の実装では，標準データ型に対する書式指定はそのままscanfに渡しているだけで，GMP データ型用の拡張書式指定子のみ別個に処理を行っています。

使用できる書式指定フラグは下記の通りです。'a'と''については使用する C ライブラリの実装に依存します。''は GMP データ型に対しては利用できません。

*	読み込みは行うが格納はしない
a	バッファを確保する（文字列の変換用）
'	桁まとめ機能。GLIBC のスタイル(GMP データ型には利用不可)

使用できる標準データ型は下記の通りです。'h'と'l'はどの環境でも使えますが，それ以外のものはコンパイラ（もしくはインクルードファイル），C ライブラリに依存します。

h	short
hh	char
j	intmax_t またはuintmax_t
l	long int, double or wchar_t
ll	long long
L	long double
q	quad_t またはu_quad_t
t	ptrdiff_t
z	size_t

GMP データ型に対する書式指定入力子は次の通りです。

`F`	`mpf_t`, 浮動小数点数に変換
`Q`	`mpq_t`, 有理数に変換(訳注：オリジナルは interger？)
`Z`	`mpz_t`, 整数に変換

使用できる書式変換指定子は下記の通りです。‘`p`’と‘`l`’は C ライブラリに依存しますが，それ以外は標準 C と同じです。

`c`	文字，もしくは文字列
`d`	10 進整数
`e E f g`	浮動小数点数
`G`	
`i`	基数指定付きの整数
`n`	文字列
`o`	8 進整数
`p`	ポインタ
`s`	区切り文字なしの文字列
`u`	10 進整数
`x X`	16 進整数
`[`	ひとまとまりの文字列

‘`e`’，‘`E`’，‘`f`’，‘`g`’，‘`G`’は同一の書式指定ができ，固定小数点方式や浮動小数点方式が使用可能です。‘`E`’と‘`e`’は浮動小数点方式の指数部を表わします。

C99 スタイルの 16 進浮動小数点フォーマット(`printf %a`，see Section 10.1 [Formatted Output Strings], p. 73)は`mpf_t`型のためだけに使用されます。通常の標準浮動小数点型に適用できるかは C ライブラリ次第です。

‘`x`’と‘`X`’は同一の書式指定ができ，どちらも大文字小文字を区別せず 16 進数フォーマットを受け付けます。

‘`o`’, ‘`u`’, ‘`x`’，‘`X`’は全て，正数も負数も受け付けます。標準 C データ型に対しては符号なし(unsigned)型として解釈しますので，オーバーフローの扱いに関しては違いが出ます。負数は`strtoul`関数に渡されて処理が行われます（see Section “Parsing of Integers” in *The GNU C Library Reference Manual*)。GMP のデータ型に対してはオーバーフローは起こりませんので，‘`d`’も‘`u`’も同じ意味になります。

‘`Q`’は分子を読み込み，必要があれば分母を読み取ります。標準形ではない形式であれば，実際にその値を使う前に`mpq_canonicalize`関数で変換しておく必要があります(see Chapter 6 [Rational Number Functions], p. 48)。

‘`Qi`’は，分子と分母の基数を別個に読み取ることができます。例えば，‘`0x10/11`’は 16/11 となり，‘`0x10/0x11`’は 16/17 となります。

‘`n`’は，GMP データ型も含む上記の全ての書式指定に適用することができます。‘`*`’は無視したい文字列を指定する時に使います。

標準 C ライブラリの`scanf`関数で使用される他の書式指定子は，`gmp_scanf`関数では利用できません。

フィールドの前のホワイトスペース（空白文字）は読み込み時に無視されます。‘`c`’と‘`[`’の指定がある時にはそのまま読み込まれます。

浮動小数点数に対する書式指定子の場合，小数点を表わす文字は実行時のロケール次第です(localeconv関数で指定可能(see Section “Locales and Internationalization” in *The GNU C Library Reference Manual*)。標準浮動小数点入力に対しては通常の C ライブラリ関数でも同様です。

書式指定文字列は ASCII コードでなければダメで，今のところマルチバイト文字列は解釈できません。そのうち改善される予定です。

11.2 書式指定入力関数

ここで述べる関数は，対応する C ライブラリの関数と同じような機能を持ちます。標準のscanf関数に対しては，変数を引数リストに並べます。vscanf関数の場合は引数のポインタを与えます。詳細はSection “Variadic Functions” in *The GNU C Library Reference Manual*, もしくは‘man 3 va_start’を参照して下さい。

書式指定文字列が間違っていたり，引数が正しく対応していなかったりした場合の動作は予測不能です。GCC の書式文字列チェック機能は，GMP の拡張書式指定に対応していないので使用できません。

*fmt*書式指定文字列と，結果を格納するメモリ領域は重複しないようにして下さい。

`int gmp_scanf` (*const char *fmt, ...*) [関数]
`int gmp_vscanf` (*const char *fmt, va_list ap*) [関数]
標準入力stdinから読み取りを行います。

`int gmp_fscanf` (*FILE *fp, const char *fmt, ...*) [関数]
`int gmp_vfscanf` (*FILE *fp, const char *fmt, va_list ap*) [関数]
入力ストリーム*fp*から読み取りを行います。

`int gmp_sscanf` (*const char *s, const char *fmt, ...*) [関数]
`int gmp_vsscanf` (*const char *s, const char *fmt, va_list ap*) [関数]
NULL 終端子付き文字列*s*から読み取りを行います。

これらの関数の返り値は全て標準 C99 のscanf関数と同一で，正常に処理して格納できたフィールド数になります。‘%n’フィールドと‘*’で無視されたフィールドは返り値には反映されません。

書式指定のフィールドを処理する前に入力の終了（もしくはファイルエラー）になったり，有効なフィールドがなかったりした時には，0 ではなくEOFが返り値になります。書式指定文字列内のホワイトスペース文字はオプション的な扱いになりますので，入っていたからといってEOFを返すわけではありません。先頭部分のホワイトスペースは無視し，書式指定フィールドとしてはカウントしません。

GMP のデータ型に対しては，C99 の読み取り規則に則って処理され，一文字先読みされつつ，書式指定通りに読み取りされます。入力数が書式指定と一致していない時には関数が停止され，値が格納されていないフィールドが取り残され，読み取りできた分のみカウントされて返り値に反映されます。例えば，mpf_t型に対しては，入力が‘1.23e-XYZ’だとすると，‘X’まで読み取りされ，それ以降の文字は数字とは異なるので読み取りされません。‘1.23e-’という文字列は，‘e’の後に一桁以上の数字が求められますので，不正な値と判断されます。

標準 C のデータ型に対しては，現在の GMP の実装では C ライブラリのscanf関数が呼び出されますので，有効な入力は少し緩いものになります。

gmp_sscanf関数はgmp_fscanf関数と同一で，読み取りの際に一文字先読みするかどうかの違いだけです。入力データ全体を見ますが，基本的にはgmp_fscanf関数と同一の読み取りを行います。これは C99 のsscanf関数がfscanf関数で実装されているのと同じです。

11.3 C++書式指定入力

ここで述べる関数は`libgmpxx` (see Section 3.1 [Headers and Libraries], p. 18)で提供されており，C++サポートを有効にしている時のみビルドされるものです(see Section 2.1 [Build Options], p. 3)。関数プロトタイプは`<gmp.h>`で定義されています。

`istream& operator>>` (*istream& **stream**, mpz_t **rop***) [関数]
`ios`の書式指定文字列に則って*stream*を読み取って*rop*に格納します。

`istream& operator>>` (*istream& **stream**, mpq_t **rop***) [関数]
‘`123`’のような整数や‘`5/9`’のような分数を読み取ります。‘`/`’の前後にホワイトスペースが入ると正常に読み取れません。分数が標準形でない場合は，`mpq_canonicalize`関数を使って変換しておく必要があります(see Chapter 6 [Rational Number Functions], p. 48)。

整数の読み取りの際には，`ios::dec`, `ios::oct`，`ios::hex`という指定がなくとも，‘`0`’や‘`0x`’のように基数指定があれば読み取りできます。分数の場合は分子と分母の基数指定は別個に可能で，‘`0x10/11`’ は16/11，‘`0x10/0x11`’ は16/17 となります。

`istream& operator>>` (*istream& **stream**, mpf_t **rop***) [関数]
`ios`の書式指定に則り，*stream*を読み取って*rop*に格納します。

16 進浮動小数点数や 8 進浮動小数点数のサポートはありません。将来は分かりませんが，今のところは標準浮動小数点数を読み取る`operator>>`の機能だけサポートするのがベストと考えています。

`istream`ロケールで指定された桁まとめ機能は現状サポートしていません。そのうちサポートされるかもしれません。

これらのオペレータは，C++の普通のやり方で GMP データ型の読み取りができます。例えば下記のように使います。

```
mpz_t  z;
...
cin >> z;
```

`istream`入力(と`ostream` 出力, see Section 10.3 [C++ Formatted Output], p. 76)は GMP データ型に対してのみオーバーロードされているので，`mpz_t`に`+`を使った場合，結果は予測不能になります。オーバーロードされたクラスについてはChapter 12 [C++ Class Interface], p. 82を参照して下さい。

12 C++ クラスインターフェース

本章では GMP 用の C++クラスについて解説します。

GMP の C 言語用のデータ型や関数は全て C++プログラから利用できるよう，gmp.hが`extern "C"`でくくられています。しかし，C++のクラスを利用すると，関数・演算子の便利なオーバーロード機能が使えるようになります。

C++クラスインターフェースを実装するには，最近の C++コンパイラ，特に，名前空間(name space)，テンプレートの部分特殊化(partial specialization of templates)，メンバテンプレート(member template)が利用可能なものを必要とします。

本章で解説している諸々は基本的なもので，将来的には互換性を保証できないこともあります。

12.1 C++インターフェースの概要

C++クラスと関数を使うときには必ず下記のようにヘッダファイルを使って下さい。

```
#include <gmpxx.h>
```

C++プログラムは`libgmpxx`と`libgmp`を下記のように必ずリンクして下さい。

```
g++ mycxxprog.cc -lgmpxx -lgmp
```

定義済みクラスは下記の 3 つです。

`mpz_class` [Class]
`mpq_class` [Class]
`mpf_class` [Class]

標準的な演算子や関数群はこの定義済みクラスを用いてオーバーロードされており，下記のような使い方ができます。

```
int
main (void)
{
  mpz_class a, b, c;

  a = 1234;
  b = "-5678";
  c = a+b;
  cout << "sum is " << c << "\n";
  cout << "absolute value is " << abs(c) << "\n";

  return 0;
}
```

`a=b+c`のような式の場合，対応する演算関数である`mpz_add`を一度だけ呼びだせばよく，`b+c`の部分では一時変数は不要です。しかし，`a=b*c+d*e`のように一時変数を使わざるを得ない式の場合もあります。

普通のデータ型である`long`, `unsigned long`，`double`と同様に，GMP の定義済みクラスは自由に混ぜて式に利用して構いません。`int`や`float`のように短い長さのデータ型も利用でき，これらは対応する長いデータ型に変換されて使われます。

`bool`型を直接使うことはできませんので，最初に`int`型に明示的に変換しておく必要があります。C++は自動的にすべてのポインタは`bool`型に自動的に変換されるので，GMP が`bool`型を受

け付けてしまうと，不正なクラスやポインタのあらゆる組み合わせもコンパイルできてしまい，型に応じた変換操作ができなくなります。

GMP の定義済みクラスから通常の C++データ型への変換は自動的には行えません。get_siメンバ関数を利用するようにして下さい(詳細は以降の節を参照)。

同様に，クラスを GMP の C のデータ型に自動的に変換することもできません。下記の C データ型オブジェクトへの参照を返す下記の関数を使って下さい。

`mpz_t mpz_class::get_mpz_t ()` [関数]
`mpq_t mpq_class::get_mpq_t ()` [関数]
`mpf_t mpf_class::get_mpf_t ()` [関数]
上記の関数は C++クラスが使えない C の関数を使うために用意されています。例えばaに対してbとcの GCD を代入するには次のように上記の関数を使います。

```
mpz_class a, b, c;
...
mpz_gcd (a.get_mpz_t(), b.get_mpz_t(), c.get_mpz_t());
```

他のやり方としては，定義済みクラスを対応する GMP の C データ型として初期化，あるいはコンストラクタを明示的に呼び出して割り当てる，というものがあります。どちらの場合でも後腐れが生じないよう，値のコピーが作られます。次のように使えます。

```
mpz_t z;
// ... z を初期化して計算する ...
mpz_class x(z);
mpz_class y;
y = mpz_class (z);
```

gmpxx.hでは名前空間を定義しません。すべてのデータはそのままグローバル名前空間に放置されます。この流儀はgmp.hから発しており，互換性を保つための仕様です。gmpxx.hのその他の機能は GMP の名前変換（naming conversion）を保管しており，名前の衝突が起きないようにしています。

12.2 C++整数クラス

`mpz_class::mpz_class (`*type* `n)` [関数]
mpz_classのコンストラクタです。long longとlong doubleを除くすべての C++の標準データ型と，GMP C++クラスが利用可能です。ただし，mpq_classとmpf_classからの変換は明示的に行う必要があります。C データ型に変換する際には，例えばdoubleならmpz_set_d関数，というように対応する C データ型への関数を使って下さい（see Section 5.2 [Assigning Integers], p. 33参照)。

`explicit mpz_class::mpz_class (`*const mpz_t* `z)` [関数]
mpz_t型を利用したmpz_classコンストラクタです。一切の後腐れなく，*z*の値は新しいmpz_classにコピーされます。

`explicit mpz_class::mpz_class (`*const char* `*s`, *int* `base` = *0*`)` [関数]
`explicit mpz_class::mpz_class (`*const string&* `s`, *int* `base` = *0*`)` [関数]
mpz_set_str関数を用いて文字列からの変換を行うmpz_classコンストラクタです。(see Section 5.2 [Assigning Integers], p. 33)

文字列が正常な整数ではない場合，std::invalid_argument例外が発生します。operator=演算子でも同様です。

mpz_class operator"" _*mpz* (*const char* ***str***) [関数]
C++11 コンパイラを使うと，整数は123_mpz文法を用いて生成できます。これはmpz_class("123")と同じです。

mpz_class operator/ (*mpz_class* a, *mpz_class* ***d***) [関数]
mpz_class operator% (*mpz_class* a, *mpz_class* ***d***) [関数]
切り捨て方向に丸めるmpz_classの除算を行います。これはmpz_tdiv_q関数とmpz_tdiv_r関数で行います(see Section 5.6 [Integer Division], p. 36)。C99 の/演算子と%演算子に相当する演算です。

mpz_fdiv...関数，もしくはmpz_cdiv...関数は必要に応じていつでも呼びだせます。例えば下記のように使います。

```
mpz_class q, a, d;
...
mpz_fdiv_q (q.get_mpz_t(), a.get_mpz_t(), d.get_mpz_t());
```

mpz_class abs (*mpz_class* ***op***) [関数]
int cmp (*mpz_class* ***op1***, *type* ***op2***) [関数]
int cmp (*type* ***op1***, *mpz_class* ***op2***) [関数]
bool mpz_class::fits_sint_p (*void*) [関数]
bool mpz_class::fits_slong_p (*void*) [関数]
bool mpz_class::fits_sshort_p (*void*) [関数]
bool mpz_class::fits_uint_p (*void*) [関数]
bool mpz_class::fits_ulong_p (*void*) [関数]
bool mpz_class::fits_ushort_p (*void*) [関数]
double mpz_class::get_d (*void*) [関数]
long mpz_class::get_si (*void*) [関数]
string mpz_class::get_str (*int* ***base*** = *10*) [関数]
unsigned long mpz_class::get_ui (*void*) [関数]
int mpz_class::set_str (*const char* ***str***, *int* ***base***) [関数]
int mpz_class::set_str (*const string&* ***str***, *int* ***base***) [関数]
int sgn (*mpz_class* ***op***) [関数]
mpz_class sqrt (*mpz_class* ***op***) [関数]
mpz_class gcd (*mpz_class* ***op1***, *mpz_class* ***op2***) [関数]
mpz_class lcm (*mpz_class* ***op1***, *mpz_class* ***op2***) [関数]
void mpz_class::swap (*mpz_class&* ***op***) [関数]
void swap (*mpz_class&* ***op1***, *mpz_class&* ***op2***) [関数]
これらの関数は対応する GMP の C ルーチンに対する C++クラスラッパーです。

cmp関数は他のクラスや，long long型とlong double型を除く標準 C++データ型に対する比較が行えます。

オーバーロードされた演算子を使うと，mpz_classとdoubleの組み合わせは完ぺきに実行できますが，double型が正確な整数で表現できない場合，どこかで丸めが発生する可能性があり，将来のバージョンでは計算結果が変わるかもしれません。

mpz_class とdouble間の変換は，C 関数mpz_get_d とmpz_set_dで実行されます。比較についてはmpz_cmp_d関数で正確に実行されます。

12.3 C++有理数クラス

本節の全てのコンストラクタは，有理数が与えられると，mpq_class::canonicalize関数を呼び出さなくても必ず標準形に変換されます。

`mpq_class::mpq_class` (*type* **op**) [関数]
`mpq_class::mpq_class` (*integer* **num**, *integer* **den**) [関数]
`mpq_class`コンストラクタです。初期値が何らかのクラスの単独の値でも，有理数を表現する整数（`mpz_class`か C++標準整数)の組でも受け付けます（`mpf_class`からの変換は明示的に行う必要あり）。但し，`long long` と`long double` は受け付けません。下記のように使用できます。

```
mpq_class q (99);
mpq_class q (1.75);
mpq_class q (1, 3);
```

`explicit mpq_class::mpq_class` (*const mpq_t* **q**) [関数]
`mpq_t`型を利用した`mpq_class`コンストラクタです。一切の後腐れなく，*q*の値が新しい`mpq_class`にコピーされます。

`explicit mpq_class::mpq_class` (*const char* ***s**, *int* **base** = *0*) [関数]
`explicit mpq_class::mpq_class` (*const string&* **s**, *int* **base** = *0*) [関数]
`mpq_set_str`関数を用いた，文字列を変換して初期値として使う`mpq_class`コンストラクタです。(see Section 6.1 [Initializing Rationals], p. 48)

文字列が正常な有理数でない場合は`std::invalid_argument`例外が発生します。`operator=`演算子でも同様です。

`mpq_class operator"" _mpq` (*const char* ***str**) [関数]
C++11 コンパイラを使うと，有理数は`123_mpq`文法と等価になります。これは`mpq_class(123_mpz)`と同じです。他の有理数は`-1_mpq/2`もしくは`0xb_mpq/123456_mpz`のように構築することができます。

`void mpq_class::canonicalize` () [関数]
`mpq_class`クラスを標準形に直します(Chapter 6 [Rational Number Functions], p. 48参照)。すべての演算子は標準形のオペランドを要求し，演算結果も標準形にします。

`mpq_class abs` (*mpq_class* **op**) [関数]
`int cmp` (*mpq_class* **op1**, *type* **op2**) [関数]
`int cmp` (*type* **op1**, *mpq_class* **op2**) [関数]
`double mpq_class::get_d` (*void*) [関数]
`string mpq_class::get_str` (*int* **base** = *10*) [関数]
`int mpq_class::set_str` (*const char* ***str**, *int* **base**) [関数]
`int mpq_class::set_str` (*const string&* **str**, *int* **base**) [関数]
`int sgn` (*mpq_class* **op**) [関数]
`void mpq_class::swap` (*mpq_class&* **op**) [関数]
`void swap` (*mpq_class&* **op1**, *mpq_class&* **op2**) [関数]
これらの関数は GMP C 関数に対する C++クラスインターフェースを提供します。

`cmp`関数は，全てのクラスと，`long long`型と`long double`型を除く標準 C++データ型に対して比較を行います。

`mpz_class& mpq_class::get_num` () [関数]
`mpz_class& mpq_class::get_den` () [関数]
`mpq_class`の分子と分母への`mpz_class`参照を返します。読み書きのどちらにも使用できます。返される参照のオブジェクトが変更されると，元の`mpq_classs`も変更されます。

分子分母を直接操作して標準形から外れた値にしてしまった場合は，`mpq_class::canonicalize`を呼び出して以降の演算に影響がないようにして下さい。

`mpz_t mpq_class::get_num_mpz_t ()` [関数]
`mpz_t mpq_class::get_den_mpz_t ()` [関数]
`mpq_class`を構成する`mpz_t`型の分子と分母への参照を返します。これを使うと`mpz_t`を引数とする C 関数に値を渡せます。これらの`mpz_t`参照先を変更すると，元の`mpq_class`も変更されます。

直接操作した結果，標準形から外れた値になった時には，以降の操作に影響がないよう，`mpq_class::canonicalize`関数を呼び出すようにして下さい。

`istream& operator>> (`*istream& stream, mpq_class& rop*`);` [関数]
*stream*から*rop*を`ios`形式で読み取ります。`mpq_t operator>>`と同じです(see Section 11.3 [C++ Formatted Input], p. 81)。

*rop*が標準形でない場合は`mpq_class::canonicalize`を呼び出しておく必要があります。

12.4 C++浮動小数点数クラス

`f=g*h+x*y`のように，`mpf_class`クラスの一時中間変数を必要とする式の場合，これらの変数は最終結果を格納する`f`と同じ精度になります。この挙動が相応しくないときには，明示的にコンストラクタを呼び出して下さい。

`mpf_class::mpf_class (`*type op*`)` [関数]
`mpf_class::mpf_class (`*type op, mp_bitcnt_t prec*`)` [関数]
`mpf_class`コンストラクタです。`long long`型と`long double`型を除くすべての標準 C++データ型を利用できます。

*prec*が与えられると，初期精度 bit がこの値でセットされます。*prec*がセットされていないと，初期精度は*op*によって規定されます。`mpz_class`, `mpq_class`, C++標準のデータ型の場合はデフォルトの`mpf`精度が使用されます(see Section 7.1 [Initializing Floats], p. 52)。`mpf_class`や式の場合は，その変数の精度が適用されます。2 項から成る式の精度は，大きい方の精度が適用されます。

```
mpf_class f(1.5);        // default precision
mpf_class f(1.5, 500);   // 500 bits (at least)
mpf_class f(x);          // precision of x
mpf_class f(abs(x));     // precision of x
mpf_class f(-g, 1000);   // 1000 bits (at least)
mpf_class f(x+y);        // greater of precisions of x and y
```

`explicit mpf_class::mpf_class (`*const mpf_t f*`)` [関数]
`mpf_class::mpf_class (`*const mpf_t f, mp_bitcnt_t prec*`)` [関数]
`mpf_t`変数を利用した`mpf_class`コンストラクタです。*f*の値が新規生成された`mpf_class`変数にコピーされ，一切の参照関係は生じません。

*prec*が与えられていると，値の初期精度（ビット）となります。与えられていない場合は，初期精度は*f*の精度になります。

`explicit mpf_class::mpf_class (`*const char *s*`)` [関数]
`mpf_class::mpf_class (`*const char *s, mp_bitcnt_t prec, int base = 0*`)` [関数]
`explicit mpf_class::mpf_class (`*const string& s*`)` [関数]
`mpf_class::mpf_class (`*const string& s, mp_bitcnt_t prec, int base = 0*`)` [関数]
`mpf_set_str`関数を使って文字列を変換して`mpf_class`型にセットするコンストラクタです(see Section 7.2 [Assigning Floats], p. 54)。*prec*が与えられていれば，値の初期精度（ビット）となります。与えられていなければデフォルトの`mpf`精度(see Section 7.1 [Initializing Floats], p. 52)が使われます。

文字列が正常な浮動小数点数でない場合はstd::invalid_argument例外が発生します。operator=でも同様です。

mpf_class operator"" _mpf (*const char* ***str***) [関数]
C++11 コンパイラを使うと，浮動小数点数は1.23e-1_mpf文法で生成されます。これはmpf_class("1.23e-1")と同義です。

mpf_class& mpf_class::operator= (*type* ***op***) [関数]
与えられた*op*の値をmpf_classに変換してセットします。前述のコンストラクタ同様に，同じデータ型であればそのまま利用できます。

operator=は値を代入するだけで，元の変数の精度をコピーすることはなく，必要に応じて切り捨ても実行します。これはmpf_set関数等の代入関数と同様です。特に，mpf_classクラスのコピーを行うコンストラクタは，デフォルトのコンストラクタや割り当て関数とは異なる動作になります。

```
mpf_class x (y);   // x は y と同じ精度で生成

mpf_class x;       // x はデフォルトの精度で生成
x = y;             // y の値は x の精度に切り捨てられる
```

テンプレート化したプログラムを使うアプリケーションは，多様な精度を持つmpf_class変数を使う計算部において，何が行われるかということに留意する必要があります。例えば，標準の複素数テンプレートcomplexは標準の浮動小数点データ型を使うことしか想定されていませんが，同精度計算，異精度計算，どちらにも対応した実装が行われています（訳注：そうなの？）

mpf_class abs (*mpf_class* ***op***) [関数]
mpf_class ceil (*mpf_class* ***op***) [関数]
int cmp (*mpf_class* ***op1***, *type* ***op2***) [関数]
int cmp (*type* ***op1***, *mpf_class* ***op2***) [関数]
bool mpf_class::fits_sint_p (*void*) [関数]
bool mpf_class::fits_slong_p (*void*) [関数]
bool mpf_class::fits_sshort_p (*void*) [関数]
bool mpf_class::fits_uint_p (*void*) [関数]
bool mpf_class::fits_ulong_p (*void*) [関数]
bool mpf_class::fits_ushort_p (*void*) [関数]
mpf_class floor (*mpf_class* ***op***) [関数]
mpf_class hypot (*mpf_class* ***op1***, *mpf_class* ***op2***) [関数]
double mpf_class::get_d (*void*) [関数]
long mpf_class::get_si (*void*) [関数]
string mpf_class::get_str (*mp_exp_t&* ***exp***, *int* ***base*** = *10*, *size_t* ***digits*** = *0*) [関数]
unsigned long mpf_class::get_ui (*void*) [関数]
int mpf_class::set_str (*const char* ***str***, *int* ***base***) [関数]
int mpf_class::set_str (*const string&* ***str***, *int* ***base***) [関数]
int sgn (*mpf_class* ***op***) [関数]
mpf_class sqrt (*mpf_class* ***op***) [関数]
void mpf_class::swap (*mpf_class&* ***op***) [関数]
void swap (*mpf_class&* ***op1***, *mpf_class&* ***op2***) [関数]
mpf_class trunc (*mpf_class* ***op***) [関数]
これらの関数群は GMP の C ルーチンへの C++インターフェースを提供します。

cmp関数はlong long とlong doubleを除く，全ての標準 C++データ型の比較に利用できます。

hypot関数を実行した結果の精度については現在は保証されていません（訳注：精度保証が欲しければ MPFR の mpfr_hypot 関数を使いましょう。）。

mp_bitcnt_t mpf_class::get_prec () [関数]
void mpf_class::set_prec (*mp_bitcnt_t prec*) [関数]
void mpf_class::set_prec_raw (*mp_bitcnt_t prec*) [関数]
mpf_classクラスの現段階でのデフォルト精度をセットしたり入手したりできます。

mpf_set_prec_raw関数で述べた制限(see Section 7.1 [Initializing Floats], p. 52)はこのmpf_class::set_prec_raw関数にも当てはまります。特にmpf_classに対しては，破棄される前に割り当てられた精度で格納されます。この操作はアプリケーション側で実施されるべきものなので，自動的に行う機構は用意していません。

12.5 C++乱数生成関数

gmp_randclass [Class]
GMP 乱数生成関数用の C++クラスインターフェースはgmp_randclassクラスを使います。gmp_randstate_t型と同様に，乱数アルゴリズムの選択や，状態保存変数の利用が可能です。

gmp_randclass::gmp_randclass (*void* (**randinit*) (*gmp_randstate_t, ...*), ...) [関数]
gmp_randclassクラスのコンストラクタです。与えられた*randinit* 関数(see Section 9.1 [Random State Initialization], p. 71)を使って初期化します。引数は*randinit*関数に引き渡されるものですが，mpz_t型変数の代わりにmpz_classの変数で与えることができます。例えば下記のように使います。

```
gmp_randclass r1 (gmp_randinit_default);
gmp_randclass r2 (gmp_randinit_lc_2exp_size, 32);
gmp_randclass r3 (gmp_randinit_lc_2exp, a, c, m2exp);
gmp_randclass r4 (gmp_randinit_mt);
```

gmp_randinit_lc_2exp_size関数は，要求されたサイズが大きすぎるとエラーとなり，std::length_error例外を発生させます。

gmp_randclass::gmp_randclass (*gmp_randalg_t alg, ...*) [関数]
gmp_randinit (see Section 9.1 [Random State Initialization], p. 71)と同じ引数を持つgmp_randclassクラスのコンストラクタです。この関数は廃止対象なので，前述の*randinit*関数を引数として与えるスタイルのコンストラクタの方を利用して下さい。

void gmp_randclass::seed (*unsigned long int s*) [関数]
void gmp_randclass::seed (*mpz_class s*) [関数]
乱数生成用の種(seed)を与えます。どうすれば良い種が与えられるかについては see Chapter 9 [Random Number Functions], p. 71を参照して下さい。

mpz_class gmp_randclass::get_z_bits (*mp_bitcnt_t bits*) [関数]
mpz_class gmp_randclass::get_z_bits (*mpz_class bits*) [関数]
指定ビット数長の整数乱数を生成します。

mpz_class gmp_randclass::get_z_range (*mpz_class n*) [関数]
0 以上$n-1$ 以下の整数乱数を生成します。

`mpf_class gmp_randclass::get_f ()` [関数]
`mpf_class gmp_randclass::get_f (`*mp_bitcnt_t prec*`)` [関数]
$0 \le f < 1$ の範囲の多倍長浮動小数点乱数fを生成します。fは$prec$ビット精度になるか，指定されていない時は格納先の精度になります。

```
gmp_randclass  r;
...
mpf_class  f (0, 512);   // 512bit 精度で初期化
f = r.get_f();           // 512bit の乱数
```

12.6 C++インターフェースの制限

mpq_classとテンプレートの読み取り
: 一般的なテンプレートでは，mpq_classが，operator>>を用いた読み取りの際に標準形でない有理数であればcanonicalize関数の呼び出しを必要とする，ということを想定していません。このため，不正確な結果になることがあります。

  operator>>はなるべく高速になるように動作します。長いオペランドの場合は，標準形に直す操作に時間がかかるので，必要がない時にはなるべく避けるようにしましょう。

  この潜在的な遅延はmpq_classの使い勝手を悪くします。おそらくこのoperator>>に関する仕様は，プリプロセッサの定義やグローバルフラグ，iosフラグの利用で将来改善されるでしょう。あるいはmpq_class operator>>で標準形に直し，mpq_t operator>>ではそれをしない，という解決策を取るかもしれません。何か良い案があったらgmp-bugs@gmplib.orgにお寄せ下さい。

サブクラス化
: 現在動いている GMP C++クラスをサブクラス化することは現段階では推奨しません。

  サブクラス化された式表現は現在では（多分）正常に使えるでしょうが，標準的なC++のやり方ではサブクラスはコンストラクタや割り当てを継承しません。GMP のC++クラスにはこれが大量に入っているので，今のところこれらを全部含むサブクラス化の良い方法は提供されていません。

テンプレート式表現
: アプリケーションで定義したテンプレート関数を使った式表現を使うと，微妙におかしなことが起こり得ます。詳細については下記の，Tを使った数値データ型の事例を読んで考えて下さい。

  ```
  template <class T>
  T fun (const T &, const T &);
  ```

  生のmpz_class変数を使えば，Tはそのままmpz_classとして解釈され，上記のコードは正しく動作します。

  ```
  mpz_class f(1), g(2);
  fun (f, g);    // Good
  ```

  しかし，下記のように式表現を引数として与えてしまうと動作しません。

  ```
  mpz_class f(1), g(2), h(3);
  fun (f, g+h);  // Bad
  ```

  これはg+hがgmpxx.hの内部的な式テンプレート表現型となってしまうことで，C++テンプレートの解釈ルールでは自動的にmpz_classに変換できないという事由によるものです。解決策としては，強制的に型キャストしてしまうというものがあります。

  ```
  mpz_class f(1), g(2), h(3);
  fun (f, mpz_class(g+h));  // Good
  ```

同様に，funの内部では，テンプレート宣言されたfunc2の中で呼び出される時には，Tは式ではなくデータ型としてキャストされている必要があります。

```
template <class T>
void fun (T f, T g)
{
  fun2 (f, f+g);     // Bad
}

template <class T>
void fun (T f, T g)
{
  fun2 (f, T(f+g));  // Good
}
```

C++11 C++11 ではデータ型を推測する機能としてauto, decltype等が追加されています。大変便利なものですが，式テンプレートに混ぜ込んで使わないようにして下さい。下記の例ではsumがマクロのように定義され，加算が 2 回実行されてしまいます。

```
mpz_class z = 33;
auto sum = z + z;
mpz_class prod = sum * sum;
```

クラッシュしてしまう実例もお見せします。いくつかのコンパイラでは一見うまくいっているように見えますが，その実，z+zが評価される前にスコープ外に放り出されてしまいます。

```
mpz_class z = 33;
auto sum = z + z + z;
mpz_class prod = sum * 2;
```

こういう事情があるので，GMP C++式表現に対してはゆめゆめautoの使用は避けて下さい。

13 メモリ割り当て機能のカスタム化

GMP のデフォルトのメモリ割り当て関数は`malloc`, `realloc`，`free`です。これらがエラーを吐いた時には，標準エラー出力にメッセージを流し，プログラムを終了します。

デフォルトメモリ割り当て関数とは違うやり方でメモリ確保を行ったり，メモリが足りなくなった時の挙動を変えたりするために，他の関数を指定することも可能になっています。

`void mp_set_memory_functions (` [関数]
*void *(*`alloc_func_ptr`) (size_t)*,
*void *(*`realloc_func_ptr`) (void *, size_t, size_t)*,
void (`free_func_ptr`) (void *, size_t)*)

現在のメモリ割り当て関数群を引数に指定したものに置き換える。引数が`NULL`であれば，対応する関数はデフォルトのものがそのまま使用される。

ここで指定されたメモリ割り当て関数群は，GMP のあらゆるところで使用されることになります。但し，一時記憶領域は，GMP ビルド時に指定があり，利用可能である場合は，`alloca`関数で確保されます。

`mp_set_memory_functions`関数は，指定前のメモリ割り当て関数で指定された使用中の GMP オブジェクトが存在しない場合にのみ，呼び出すようにして下さい。つまり，全ての他の GMP 関数が実行される前に呼び出す必要がある，ということです。

こうして指定されたメモリ割り当て関数群は，下記のような型宣言をしておく必要があります。

`void * allocate_function (`*size_t `alloc_size`*`)` [Function]
新規に確保された*alloc_size*バイトのメモリ空間へのポインタを返します。

`void * reallocate_function (`*void *`ptr`, size_t `old_size`, size_t `new_size`*`)` [Function]
確保済みの*old_size*バイトのメモリブロック*ptr*を*new_size*バイトにリサイズします。

このメモリブロックは，必要があれば移動することもあります。例えば，*old_size*より*new_size*が小さい場合，新規に割り当てられたメモリ領域に*new_size*バイト分コピーして移動しなくてはなりません。返り値はリサイズ後のメモリブロックへのポインタで，移動していれば新規のメモリブロック，移動していなければ*ptr*がそのまま引き継がれます。

*ptr*が`NULL`になることはなく，常にここには既存のメモリブロックへのポインタが記憶されています。*new_size*は*old_size*より大きくても小さくても大丈夫です。

`void free_function (`*void *`ptr`, size_t `size`*`)` [Function]
ポインタ*ptr*に確保されたメモリ空間を解放します。

*ptr*が`NULL`になることはなく，常に確保済みの*size*バイト分のメモリエリアへのポインタが入ります。

ここでいう*byte*とは，`sizeof`オペレータが使用するメモリ単位を意味します。

*reallocate_function*関数の*old_size*パラメータと*free_function*関数の*size*パラメータは省力化のために渡されていますが，当然実装上必要でなければ無視されます。デフォルトのメモリ割り当て関数である`malloc`等が使用される時には不要なものです。

上記の関数群はエラーを返すことはありませんので，問題があった時には相応の処理を行う必要があります。*allocate_function*関数や*reallocate_function*関数が NULL を返すことはありません。

致命的なエラーに対する挙動は差異があった方が好ましいわけで，例えば，stderrにデフォルト的に出力するよりは，グラフィカルなダイアログを出す，というようにしましょう。メモリエラー(out of memory)は割と良くあることです。

例えばメモリエラー(out of memory)から回復するメモリ割り当て関数は今のところ定義されていませんので，起きた時にはプログラムを停止すべきです。longjmp関数や C++の例外発生が起こった時の挙動は未定義です。この仕様は将来変更されるかもしれません。

GMP はメモリブロックを割り当てて，他のメモリブロックへのポインタを保持することがあります。手堅いガベージコレクションスキームを構築したいのであれば，このようにしないことが望ましいということになります。

デフォルトの GMP メモリ割り当て関数はmalloc関数とその周辺の標準関数で，プログラムが最初に実行するのがmp_set_memory_functions関数であれば，これらの関数がリンクされるべきです。これが問題というのであれば，GMP のソースを書き変える必要があります。

`void mp_get_memory_functions (` [関数]
*void *(****alloc_func_ptr**) (size_t)*,
*void *(****realloc_func_ptr**) (void *, size_t, size_t)*,
*void (**__free_func_ptr__) (void *, size_t)*)

現在のメモリ割り当て関数を取り出し，引数に指定されたポインタにその関数を格納します。引数がNULLポインタであれば，格納はしません。

現在のメモリ解放関数を知りたい時には次のようにします。

```
void (*freefunc) (void *, size_t);

mp_get_memory_functions (NULL, NULL, &freefunc);
```

14 GMP が利用可能な言語

下記のパッケージやプロジェクトは，C 以外の言語から GMP の諸機能を使えるようにしてくれるものです。但し，使用できる機能の範囲やパフォーマンスについてはバラつきがあります。

C++

- GMP C++ クラスインターフェース, see Chapter 12 [C++ Class Interface], p. 82,
  クラスインターフェース，中間変数のない式テンプレートが利用可
- ALP `https://www-sop.inria.fr/saga/logiciels/ALP/`
  テンプレートを利用した線型代数や多項式演算
- Arithmos `http://cant.ua.ac.be/old/arithmos/`
  無限大や平方根が使える有理数演算
- CLN `http://www.ginac.de/CLN/`
  ハイレベルのクラスと演算子
- Linbox `http://www.linalg.org/`
  疎行列と疎ベクトル
- NTL `http://www.shoup.net/ntl/`
  C++ 数論ライブラリ

Eiffel

- Eiffelroom `http://www.eiffelroom.org/node/442`

Haskell

- Glasgow Haskell Compiler `https://www.haskell.org/ghc/`

Java

- Kaffe `https://github.com/kaffe/kaffe`

Lisp

- GNU Common Lisp `https://www.gnu.org/software/gcl/gcl.html`
- Librep `http://librep.sourceforge.net/`
- XEmacs (21.5.18 beta and up) `http://www.xemacs.org`
  GMP による多倍長整数，有理数，浮動小数点数のオプション機能

M4

- GNU m4 betas `http://www.seindal.dk/rene/gnu/`
  オプションで任意精度演算`mpeval`が使用可能

ML

- MLton compiler `http://mlton.org/`

Objective Caml

- MLGMP `http://opam.ocamlpro.com/pkg/mlgmp.20120224.html`
- Numerix `http://pauillac.inria.fr/~quercia/`
  GMP 利用可能なオプションが存在。

Oz

- Mozart `http://mozart.github.io/`

Pascal

- GNU Pascal Compiler `http://www.gnu-pascal.de/`
  GMP ユニットあり。

- Numerix http://pauillac.inria.fr/~quercia/
  フリーの Pascal で，オプションとして GMP が利用可能

Perl

- GMP モジュールについては GMP ソースのdemos/perlディレクトリを参照(see Section 3.10 [Demonstration Programs], p. 22)
- Math::GMP http://www.cpan.org/
  Math::BigInt と互換性あり。但し，GMP モジュールほど多機能ではない。
- Math::BigInt::GMP http://www.cpan.org/
  通常の Math::BigInt に対して Math::GMP が利用可能

Pike

- 通常のディストリビューションに mpz モジュールが含まれている, http://pike.ida.liu.se/

Prolog

- SWI Prolog http://www.swi-prolog.org/
  任意精度浮動小数点数のサポートあり。

Python

- GMPY https://code.google.com/p/gmpy/

Ruby

- http://rubygems.org/gems/gmp

Scheme

- GNU Guile https://www.gnu.org/software/guile/guile.html
- RScheme http://www.rscheme.org/
- STklos http://www.stklos.net/

Smalltalk

- GNU Smalltalk http://www.smalltalk.org/versions/GNUSmalltalk.html

Other

- Axiom https://savannah.nongnu.org/projects/axiom
  GCL を利用した計算代数システム
- DrGenius http://drgenius.seul.org/
  幾何学システムと数学プログラミング言語
- GiNaC http://www.ginac.de/
  CLN を利用した C++計算代数
- GOO https://www.eecs.berkeley.edu/~jrb/goo/
  動的オブジェクト指向言語
- Maxima https://www.ma.utexas.edu/users/wfs/maxima.html
  GCL を利用した Macsyma 計算代数システム
- Regina http://regina.sourceforge.net/
  トポロジー計算機
- Yacas http://yacas.sourceforge.net
  Yet another computer algebra system.

15 アルゴリズム

この章では，GMP で使われているアルゴリズムを紹介します。アルゴリズムを理解していないと，GMP のコードを読み解くことは困難です。

GMP の内部構造については後述しますが，将来の GMP と互換性を保てるアプリケーションを作りたいのであれば，この文書に書いてある関数だけを使うようにして下さい。

15.1 乗算アルゴリズム

N×N リムの乗算と 2 乗は下記の 7 つのアルゴリズムのうち 1 つを使って実行されます。N が増えるごとに 7 つのアルゴリズムに順次移行していきます。

アルゴリズム	下限閾値マクロ
筆算(Basecase)	(なし)
Karatsuba	`MUL_TOOM22_THRESHOLD`
Toom-3	`MUL_TOOM33_THRESHOLD`
Toom-4	`MUL_TOOM44_THRESHOLD`
Toom-6.5	`MUL_TOOM6H_THRESHOLD`
Toom-8.5	`MUL_TOOM8H_THRESHOLD`
FFT	`MUL_FFT_THRESHOLD`

2 乗の場合も`SQR`を接頭詞とする下限閾値マクロを使ってアルゴリズムを切り替えます。

N×M 乗算，即ち，被乗数のサイズが異なる場合，`MUL_TOOM22_THRESHOLD`を越えていると，サイズによりますが，Toom-cook に似たアルゴリズムか FFT を使って計算が行われます（see Section 15.1.8 [Unbalanced Multiplication], p. 102)。

15.1.1 筆算アルゴリズム

N×M を実行する筆算アルゴリズムは互いの積を計算し，長方形の枠に入れて和の計算をするというもので，手計算で長い桁の乗算を行う手法と同一のものです。そのため，スクールブック(schoolbook)法，グラマースクール(grammar school)法とも称されます。$O(NM)$ の手間のかかるアルゴリズムで，詳細は Knuth 本(see 付録 B [References], p. 128)の 4.3.1 節のアルゴリズム M を参照するか，`mpn/generic/mul_basecase.c`に記述したものを参考にして下さい。

`mpn_mul_basecase`関数をアセンブラで実装したものは汎用の C のコードと本質的に何ら変わるところはなく，相違点は普通のアセンブラの技法を使い，速度向上のための分かりづらいコーディングを行っているというところです。

2 乗の計算は，正方形の枠の上三角部分と下三角部分の積が同一のものとなるため，普通の乗算の約半分程度の時間で実行できます。対角成分以下の三角部分の積を作れば，その 2 倍（1bit 左シフト)を求め，対角成分の積を加えます。この 2 乗のアルゴリズムは`mpn/generic/sqr_basecase.c`で見ることができます。アセンブラによる実装も同様に行っており，本質的にはこの汎用 C コードと同一のものです。

	u0	u1	u2	u3	u4
u0	d				
u1		d			
u2			d		
u3				d	
u4					d

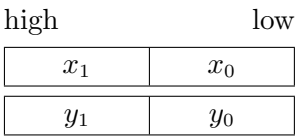
実際の 2 乗計算は乗算より 2 倍速くはならず，せいぜい 1.5 倍程度の高速化にとどまります。1.5 倍にもならないというのであれば，その CPU 向きの`mpn_sqr_basecase`関数の改良を要するということになります。

幾つかの CPU 上では，小さいサイズの値に対しては，`mpn_mul_basecase`関数の方が，汎用 C コードである`mpn_sqr_basecase`関数より高速になることもあります。`SQR_BASECASE_THRESHOLD`の値は，`mpn_sqr_basecase`関数を使うサイズを示しており，これがゼロに設定されていると，常にこの乗算関数を使うことを意味します。

15.1.2 Karatsuba 乗算

Karatsuba 乗算アルゴリズムは，Knuth 本の 4.3.3 節のパート A や他のテキストで解説されていますので，ここでは簡単な紹介に留めます。

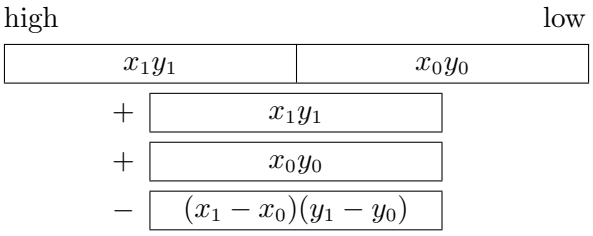
入力値がx とy である時，それぞれ，二つに均等分割します（N が奇数の時は大きい桁の方を 1 リムだけ短くします）。



今，b を分割ポイントである 2 のべき乗とします。即ち，もしx_0 がk 個のリムであるとすれば，(y_0 も同様)，$b = 2^{k*\texttt{mp\_bits\_per\_limb}}$ となります。$x = x_1 b + x_0$，$y = y_1 b + y_0$ であれば，下記の等式が成立します。

$$xy \;=\; (b^2+b)x_1y_1 \;-\; b(x_1-x_0)(y_1-y_0) \;+\; (b+1)x_0y_0$$

この等式は，N×N リムの乗算を計算する際，筆算アルゴリズムで実行すると 4 回の(N/2)×(N/2)乗算が必要になるところ，3 回の(N/2)×(N/2)で済んでしまうということを示しています。等式の係数である(b^2+b)，$-b$，$(b+1)$ は，3 つの項がどの位置で加算されるかを表現しているだけで，実際に乗算を行う必要はありません。



$(x_1-x_0)(y_1-y_0)$ の項は絶対値として計算し，符号に応じて加算するか減算するかを選ぶようにします。high(x_0y_0) + low(x_1y_1) の加算は 2 回必要になりますが，これは$6k$ リム長ではなく，$5k$ リム長の加算を実行することで済み，余計な関数呼び出しによるオーバーヘッド減らすことができます。

2 乗の計算も普通の乗法と同様のアルゴリズムで実行しますが，$x = y$ であることを利用すると，2 乗計算を 3 回実行するだけで済むようになります。

$$x^2 \;=\; (b^2+b)x_1^2 \;-\; b(x_1-x_0)^2 \;+\; (b+1)x_0^2$$

最終演算結果は上記の乗法の場合と同様に，3 つの 2 乗計算を足し込んで得られます。2 乗の場合は中間の項$(x_1-x_0)^2$ は常に正になります。

乗法も 2 乗計算も中間項を$(x_1+x_0)(y_1+y_0)$ という和の形で構成することも可能ですが，k リムを超える加算が必要になりますし，差の中間項を使った場合に比べて桁上がりとその和の処理が必要になる可能性が出てきてしまいます。

Karatsuba 乗算アルゴリズムは漸近的に$O(N^{1.585})$ の計算量を必要とします。ここで指数1.585...は$\log 3/\log 2$ であり，長さ1/2 の入力値の乗算を 3 回実行する，ということを意味しています。筆算アルゴリズムが$O(N^2)$ ですから，かなりの改善がなされたことになり，Karatsuba 乗算を実行することで必要となる加算のコストを上回る効果が期待できます。`MUL_TOOM22_THRESHOLD`は10 リム程度に設定しており，`SQR`(2 乗計算)の閾値は必ずこの約 2 倍になるように設定されています。

筆算アルゴリズムの計算時間は$M(N) = aN^2 + bN + c$ という形で与えられ，Karatsuba 乗算アルゴリズムでは$K(N) = 3M(N/2) + dN + e$ という形になり，これを展開すると$K(N) = \frac{3}{4}aN^2 + \frac{3}{2}bN + 3c + dN + e$ が得られます。a についている$\frac{3}{4}$ という係数は，筆算アルゴリズムの中の積を減少させた分量を意味しており，$M(N)$ が$K(N)$ よりも小さくなる閾値を押し上げます。逆に，b の係数$\frac{3}{2}$ は比例的に計算量を増加させることを意味しており，$K(N)$ の方が$M(N)$ より小さくなる閾値を押し上げます。後者のケースは，最適化された`mpn_sqr_diagonal`関数を`mpn_sqr_basecase`関数に持ち込んだ時にも当てはまります。どんなスピードアップ法を適用しても計算時間は減少しますので，アルゴリズム切り替えの閾値をどのように決めるかは学術的な課題に過ぎません。

15.1.3 Toom 3-Way 乗算

Karatsuba による定式化は分割統治法の最も単純な適用例で，Toom-Cook や FFT アルゴリズムはこの系統の発展形にあたります。Toom-Cook アルゴリズムについては Knuth 本の 4.3.3 節に記述があり，定理 A の後に 3-way 計算の実例も提示されています。ここでは GMP で使っている 3-way 計算について解説します。

引数がそれぞれ 3 等分割できるものとします（最初の有効桁部分は残りの 2 つに比べて 1, 2 リム短くても良い）。

high low

x_2	x_1	x_0
y_2	y_1	y_0

この 3 つのパートは下記の二つの多項式の係数として扱います。

$$X(t) = x_2t^2 + x_1t + x_0$$
$$Y(t) = y_2t^2 + y_1t + y_0$$

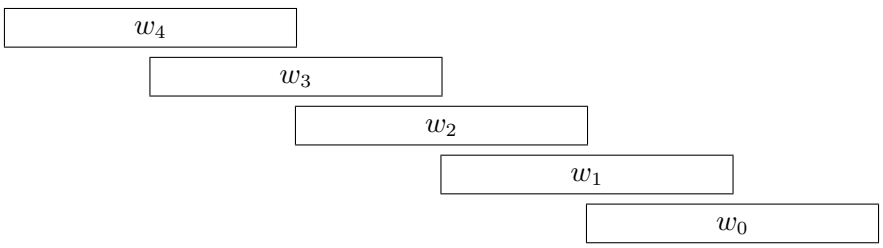
b は，x_0, x_1, y_0，y_1 と同じ長さの 2 のべき乗になります。つまり，これらのパートの長さがk リムだとすると，$b = 2^{k*\texttt{mp\_bits\_per\_limb}}$ となる訳です。ここで，$x = X(b)$，$y = Y(b)$ とします。

多項式$W(t) = X(t)Y(t)$ を作り，その係数が次のように表現できるとします。

$$W(t) = w_4t^4 + w_3t^3 + w_2t^2 + w_1t + w_0$$

この係数w_i は確定できますので，$xy = X(b)Y(b)$ より，最終形は$w = W(b)$ となります。この係数はそれぞれ大体b^2 となるので，最終形$W(b)$ は下記の和となります。

high low



係数w_i は単純な交差積として表現でき，$w_4 = x_2y_2$, $w_3 = x_2y_1+x_1y_2$, $w_2 = x_2y_0+x_1y_1+x_0y_2$ $\cdots$ となります。必要となるのは，$i, j = 0, 1, 2$ に対して 9 つのx_iy_j で，これは単なる筆算乗算アルゴリズムと同じになります。この代わりに，下記のようなアプローチを取ることもできます。

$X(t)$ と$Y(t)$ が得られており，5 点でこの積，即ち$W(t)$ の値が得られているものとします。GMP では下記の点を使用します。

補間点	補間点における値
$t = 0$	$x_0y_0 (= w_0)$
$t = 1$	$(x_2 + x_1 + x_0)(y_2 + y_1 + y_0)$
$t = -1$	$(x_2 - x_1 + x_0)(y_2 - y_1 + y_0)$
$t = 2$	$(4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0)$
$t = \infty$	$x_2y_2 (= w_4)$

$t = -1$ においては，負数となり得るので，絶対値を扱うようにして符号は別にしておきます。$t = \infty$ においては，実際の値は$\lim_{t\to\infty} \frac{X(t)Y(t)}{t^4}$ と評価されるので，単純にx_2y_2 はw_4 となるものと考えます（$t = 0$ においてはx_0y_0 はw_0 になります)。

$W(t) = w_4t^4 + \cdots + w_0$ に各点を代入して計算すると，係数w_i の線型結合が下記のように得られます。

$$\begin{array}{rcccccccccc}
W(0) &=& & & & & & & & & w_0 \\
W(1) &=& w_4 &+& w_3 &+& w_2 &+& w_1 &+& w_0 \\
W(-1) &=& w_4 &-& w_3 &+& w_2 &-& w_1 &+& w_0 \\
W(2) &=& 16w_4 &+& 8w_3 &+& 4w_2 &+& 2w_1 &+& w_0 \\
W(\infty) &=& w_4 & & & & & & & &
\end{array}$$

この結果，5 つの未知数を持つ 5 つの方程式が出ますので，単純な線型計算を行って各未知数w_i を導出します。それぞれの$W(t)$ の値に対して相互に加減算を行うと，結果として 2 のべき乗の除算が数回と，3 の除算 1 回行う必要があり，後者については`mpn_divexact_by3`関数で実行できます(see Section 15.2.5 [Exact Division], p. 103)。

$W(t)$ の値を係数に変換するには補間を使います。$W(t)$ は 4 次多項式なので，5 つの補間点を取れば唯一に決まります。補間点は任意に設定できるので，なるべく効率よく連立一次方程式が解けて係数w_i が導出できるように取りましょう。

2 乗についても，乗算と同じように計算することができますが，この場合は$X(t)$ が一つだけ与えられていればよいので，単純に 5 点取って，その値$W(t)$ の 2 乗を計算します。補間多項式は唯一に定まりますので，`toom_interpolate_5pts`サブルーチンを使って，2 乗と乗算を行います。

Toom-3 アルゴリズムは漸近的に$O(N^{1.465})$ となります。この指数の値は$\log 5/\log 3$ で，各サイズの 1/3 を 5 回再帰的に使用した結果です。これによって，Karatsuba 乗算の$O(N^{1.585})$ より改善できます。多項式の値評価や補間を行う手間を考慮しても，あるサイズ以上の乗算ではメリットが出てきます。

Toom-3 アルゴリズムと Karatsuba 乗算が交差するリムサイズ近くでは一定の幅でほとんど同じコストの部分がでてきます。`MUL_TOOM33_THRESHOLD`はその幅の中で適当に取っていますので，何度かチューニングのためのプログラムを実行してみると，多少上下にバラつきが出てきます。計算時間 vs. リムサイズのグラフを描いてみると，その様子が見て取れます。詳細は`tune/README`を参照して下さい。

Toom-3 の閾値がそこそこ小さいリムサイズを示している時には，Karatsuba と Toom-3 の漸近的アルゴリズムオーダーはあまり正確な時間予測を与えてくれません。当然，オーバーヘッドの影響が出てきますし，再帰ループが実行されていますので，その影響もあるからです。大きなリ

ムサイズであっても，キャッシュアーキテクチャ等のマシン固有の影響によって実際のパフォーマンスは予測していたものとは異なるケースもままあります。

Karatsuba アルゴリズム(see Section 15.1.2 [Karatsuba Multiplication], p. 96)が与える式は，5 回の乗算を実行する Toom-3 と同等のものになっていますが，複雑なので実際のところは良く分かっていません。

Toom-3 アルゴリズムに対しては別の見方が Zuras (see 付録 B [References], p. 128)によって提唱されており，これはx とy をベクトルで表現し，それを分割して，多項式評価と補間を行列乗算として表わすものです。この行列の逆行列を実際に使用するわけではありませんが，その要素は実際の補間ステップに現れるものよりもずっと大きなものになります。この Toom 3-way アルゴリズムを表現する見方は魅力的ですが，このやり方で実装する必要はあまりなく，例えば，6 の除算をビット操作で行うなどの工夫が必要になります。

15.1.4 Toom 4-Way 乗算

Karatsuba 乗算アルゴリズムと Toom-3 アルゴリズムはそれぞれ引数を 2 ないし 3 つの係数に分割しました。Toom-4 アルゴリズムでも同様に引数を 4 つの係数に分割します。Toom-3 乗算の記号を使うと，次のような多項式で表現できます。

$$X(t) = x_3t^3 + x_2t^2 + x_1t + x_0$$
$$Y(t) = y_3t^3 + y_2t^2 + y_1t + y_0$$

$X(t)$ と$Y(t)$ を 7 点で評価して乗算を行い，各点で$W(t)$ の値を出します。GMP では次の点を使用しています。

補間点	補間点における値
$t = 0$	$x_0y_0(= w_0)$
$t = 1/2$	$(x_3 + 2x_2 + 4x_1 + 8x_0)(y_3 + 2y_2 + 4y_1 + 8y_0)$
$t = -1/2$	$(-x_3 + 2x_2 - 4x_1 + 8x_0)(-y_3 + 2y_2 - 4y_1 + 8y_0)$
$t = 1$	$(x_3 + x_2 + x_1 + x_0)(y_3 + y_2 + y_1 + y_0)$
$t = -1$	$(-x_3 + x_2 - x_1 + x_0)(-y_3 + y_2 - y_1 + y_0)$
$t = 2$	$(8x_3 + 4x_2 + 2x_1 + x_0)(8y_3 + 4y_2 + 2y_1 + y_0)$
$t = \infty$	$x_3y_3(= w_6)$

Toom-4 の加算と減算の回数は，Toom-3 よりも大幅に増えますが，$t = 1$ や$t = -1$ では.例えば$x_2 + x_0$ のように再利用できるところが出てきます。

Toom-4 は漸近的に$O(N^{1.404})$ となり，この指数は$\log 7/\log 4$，即ち，元のリムサイズの 1/4 を 7 回再帰的に繰り返して実行した結果となります。

15.1.5 高次の Toom'n'half

以上述べてきた Toom-Cook アルゴリズム(see Section 15.1.3 [Toom 3-Way Multiplication], p. 97,see Section 15.1.4 [Toom 4-Way Multiplication], p. 99) は，入力値を任意の個数に分割して実行します。一般に，同じ長さの 2 つの入力値をr 個に分割すると，$2r - 1$ 個の点でそれぞれ多項式の評価と乗算を行うことになります。対称性をフルに活用できれば，4 点単位でまとめることができ，これによってより高次の Toom'n'half アルゴリズムが実行できるようになります。

Toom'n'half とは，一つの入力値に対してたくさんの分割を考えるという意味です。二つの入力値の長さが異なっていても，仮想的に考えることはできます。偶数r を選ぶと，Toom-$r\frac{1}{2}$ アルゴリズムは$2r$ 個の点を必要とするので，4 点の組ができます。

この 4 点の組は 0, ∞, +1, -1，$\pm 2^i$, $\pm 2^{-i}$ を含みます。各点においては，使い回しのできる多項式評価の計算と，各点における補間操作が入ります。更なる工夫としては，乗算アルゴリズム全体で，積と同じサイズのメモリバッファの利用回数を減らすということもできます。

現在の GMP 実装では Toom-6'n'half と Toom-8'n'half の両方を利用しています。

15.1.6 FFT 乗算

巨大な数の乗算を行うには，Fermat スタイルの FFT 乗算を使用します。これは Schönhage と Strassen (see 付録 B [References], p. 128)のアルゴリズムに基づいています。FFT は色々な形でたくさんのテキストで解説されており，例えば Knuth section 4.3.3 part C や Lipson chapter IX 等があります。ここでは GMP で使用されている形式のものを簡単に解説します。

実行する乗算はN が与えられた時の$xy \bmod 2^N+1$ です。積の完成形はxy ですが，これは$N \geq \text{bits}(x)+\text{bits}(y)$ となるようにN を選び，x とy の大きい方の有効リムにゼロを詰め込むと得ることができます。この剰余積は，このアルゴリズムの基本なので，このような操作が不可欠になります。

このアルゴリズムは，分割(split), 値評価(evaluate), 点ごとの乗算(pointwise multiply), 補間(interpolate)，Karatsuba アルゴリズムや Toom-Cook 3-way アルゴリズム同様の結合処理(combine) をこの順で実行します。パラメータk は分割処理を制御し，FFT-k 分割を行って，$M = N/2^k$ ビットごとに2^k 個に分割します。N は$2^k \times$ `mp_bits_per_limb` なので，この分割でリムの境界内に収めることができ，分割や結合処理におけるビット操作が不要になります。

値評価，点ごとの乗算，補間はすべてモジュロ$2^{N'}+1$ 上で実行されます。ここでN' は$2M+k+3$ を2^k 及び`mp_bits_per_limb`の倍数に切り上げられます。補間した結果，下記のように入力された分割パーツに対して準巡回畳み込みになります。N' はこの和が打ち切られない程度に長く取ります。

$$w_n = \sum_{\substack{i+j=b2^k+n \\ b=0,1}} (-1)^b x_i y_j$$

値評価を行う点はg^i ($i=0$ から2^k-1)となります。ここで$g = 2^{2N'/2^k}$ です。g は，$2^{N'}+1$ を法とする 1 の2^k 乗根で，補間の段階で必要となる計算の削減ができるようになります。また，これは 2 のべき乗なので，値の評価と補間に対して実行される Fourier 変換はビットシフト，2 進加算と符号反転だけで行えます。

この点ごとに行われる乗算は$2^{N'}+1$ を法として実行され，再帰的に FFT を繰り返し，最終的には通常の乗算を行って完了します（Toom-3, Karatsuba，筆算アルゴリズムを使用)。どれを行うかは，サイズN' に対して最適なアルゴリズムを選択します。補間処理は逆 FFT になります。結果として出てくる$x_i y_j$ の和の集合は，最終結果にふさわしいオフセットで加算されます。

2 乗の計算も同様に実行できますが，入力値x は一つだけになるので，値評価は一つで済み，点毎の乗算は 2 乗になります。補間についても同様です。

2^N+1 を法とする積に対しては，FFT-k アルゴリズムは$O(N^{k/(k-1)})$ となり，この指数は2^k 回，各$1/2^{k-1}$ のサイズの入力値に対して乗じたものになります。各k に対しては漸近的に改善が行われますが，オーバーヘッドも，サイズに応じて大きくなっていきます。このコードでは，各k が使われるところで，`MUL_FFT_TABLE`と`SQR_FFT_TABLE`を閾値として使用しています。新しいk に対しては効率的に乗算を，シフト，加算，オーバーヘッドに置き換えていきます。

2^N+1 を法とする積は通常の，$N \times N \to 2N$ bit 積と減算を行って求められますので，FFT と Toom-3 等のアルゴリズムは直接比較ができます。$k=4$ FFT は$O(N^{1.333})$ なので，$O(N^{1.465})$ となる Toom-3 よりは高速になることが期待できます。但し，実際には，`MUL_FFT_MODF_THRESHOLD`と`SQR_FFT_MODF_THRESHOLD`は 300 ～ 1000 リムの間で設定されており，CPU ごとに異なります。このサイズを越えた時のみ，大規模な FFT の再帰が各点における乗算ごとに実行されるようになっています。

FFT を用いて完全積を求めるときにはN を$2N$ に変更するだけで済み，理論的な計算量は与えられたk が変わらない限りは同じになります。しかし，FFT が真っ先に使われるような状況を考えると，FFT は通常の積を求めるために再帰呼び出しされ，その基盤上で，入力サイズ$1/2^{k-2}$ 毎に2^k 回乗算を実行することになり，結果として$O(N^{k/(k-2)})$ となります。これは$k=7$ の場合

は$O(N^{1.4})$ となり，Toom-3 より高速になることが期待できます。実際には，MUL_FFT_THRESHOLD とSQR_FFT_THRESHOLDは$k = 8$ の範囲で，大体 3000 ～ 10000 リムに設定されています。

N を2^k 個に分割すると，$2M + k + 3$ は2^k とmp_bits_per_limbの倍数に切り上げられますが，これは$2^k \geq$ mp_bits_per_limb である時に，N が効果的になるのは2^{2k-1} bit の倍数の時だからです。この$+k + 3$ の意味は，N が倍数よりも小さい時に，一番近い倍数に切り上げられるということを意味しています。このアルゴリズムの計算量は，望ましいサイスであり，丸めに際してもパディングが発生せず，付加的な$+k + 3$ bit は通常の FFT サイズでは無視できる程度の大きさということを仮定しています。

2^{2k-1} に制限した結果，階段効果(step-effect)が実際の計算速度に影響してきます。例えば$k = 8$ はN を 32768 bit の倍数に切り上げられ，32-bit リムの場合は 512 リムのグループに分割され，mpn_mul_nは同程度の計算速度になります。$k = 9$ の場合は 2048 リムのグループに，$k = 10$ の場合は 8192 リムのグループ，といった具合になります。実際には，各k は小さいサイズの倍数で使用されるので，階段効果の影響が計算時間 vs. サイズのグラフに如実に現れてきます。

閾値の値は現在の実装では各サイズステップの中央に設定されていますが，これはそこそこのチューニングにしかなっていません。というのは，新しいステップの出発地においては，前のk に戻った方がいいという場合があるからです。もう少し洗練されたMUL_FFT_TABLEとSQR_FFT_TABLEの設定方法が知りたいところです。

15.1.7 その他の乗算アルゴリズム

前述した Toom-Cook アルゴリズム(see Section 15.1.3 [Toom 3-Way Multiplication], p. 97, see Section 15.1.4 [Toom 4-Way Multiplication], p. 99)は任意の長さに入力値を切り分けますが，これは Knuth section 4.3.3 algorithm C に記述があります。現状の実装ではこのやり方は使っていません。ここではその理由について書きます。

一般に$r + 1$ 個に入力値を分割すると，値評価と点ごとの乗算が$2r + 1$ 個の点において発生します。4-way 法だとこれが 7 倍になり，5-way では 9 倍・・・となります。漸近的には，$(r + 1)$-way アルゴリズムを使うと$O(N^{log(2r+1)/log(r+1)})$ となります。点ごとの乗算回数は big-O 計算量に加えられていきますが，値の評価と補間に要する時間はr に比例して増え，実際の計算時間に多大な影響を与えます。各r の漸近的挙動はサイズが大きくなるごとに現実化していきます。オーバーヘッドは$O(Nr)$ となります。$r = 2^k$ FFT においては，$O(N \log r)$ ぐらいになります。

Knuth 本のアルゴリズム C では，各点, 0,1,2,. . . ,$2r$ 毎に値の評価を行っていますが，練習問題 4 にあるように$-r$,. . . ,0,. . . ,r を使うと，後に続く計算は値評価の時点で小さい値を乗じるだけで済みます(むしろ加算するだけで済む)。従って，補間の計算は半分程度で済むようになります。このアイディアは，最終結果を偶数番号の係数と奇数番号の係数に分割し，アルゴリズム C の C7 と C8 で個別に実行することができるようになる，というものです。C7 ステップにおける割る数はj^2 となり，C8 における乗じる数は$2tj - j^2$ となります。

正の点と負の点を通じて偶数と奇数の部分に分割すると，-1 を使う部分は 1 の平方根として考えることができるようになります。1 の 4 乗根が分かっていれば，更に分割してスピードアップできるようになりますが，1 以外の整数の平方根は通常利用できません。虚数単位$i = \sqrt{-1}$ を使って複素整数を導入しても役に立ちません。というのも，本質的には直交座標系では，複素乗算は 3 つの実数乗算が必要になるからです。1 の2^k 乗根が望ましい環や体の上に存在していれば，Fourier 変換は分割して実行でき，$O(N \log r)$ になります。

浮動小数点 FFT は，1 の N 乗根の近似複素数を使用します。CPU によっては，このような FFT 用に特別なサポートが備えられていますが，GMP では，下のビットがずれてしまう可能性が出て正確な乗算を保証できないため，浮動小数点 FFT を使っていません。最後の bit 値が 1 異なっても信号処理にはさしたる影響はありませんが，GMP では大変な問題になるのです。

15.1.8 不均衡乗算

乗算する 2 数の長さが異なる場合，MUL_TOOM22_THRESHOLD以下であれば，筆算アルゴリズムを使います(see Section 15.1.1 [Basecase Multiplication], p. 95)。

もっと長い桁であれば，FFT を直接使います。

この中間のサイズの場合は，Alberto Zanoni と Marco Bodrato が提唱した Toom-Cook に似たアルゴリズムを使います。考え方としては，2 数を分割して異なる次数の多項式を作ります。GMP の現在の実装では，小さい方の引数を 2 つの係数，即ち，1 次多項式に変換し，大きい方の引数は 2 〜 4 つの係数，即ち，1 〜 3 次多項式に変換します。

15.2 除算アルゴリズム

15.2.1 単一リムの除算

N×1 の除算は，2×1 の除算を，高い桁から低い桁へ繰り返し実行して行います。これはハードウェアの除算命令と逆数の乗算のうち，実行している CPU に適したものを選んで行います。

逆数の乗算は Möller と Granlund (see 付録 B [References], p. 128)の論文‘‘Improved division by invariant integers’’に基づいて実装されたgmp-impl.hにあるudiv_qrnnd_preinvマクロで行っています。考え方としては，$1/d$ (invert_limb参照)の固定小数点近似値を作り，割られる数の高い方のリムから(プラス 1bit 分)乗算して商q を求めるものです。d は正規化(高いビットが 1 になる)されているものとすると，q は 1 より大きくなることはありません。この割られる数からqd を引くと余りが得られるので，q もしくは$q-1$ が商となります。

この結果，除算が 2 回の乗算と 4，5 回の演算操作で実行できます。低いレイテンシの乗算機構を持つ CPU では，逆数の計算が必要にはなりますが，4，5 リム以上であればコスト的に引き合い，ハードウェアの除算命令より高速に実行できます。

割る数が確実に正規化されている時は，汎用 C ルーチンの__udiv_qrnnd_c，もしくは逆数の乗算を使って，$a2^k \div d2^k$ として行います。ここでa は割られる数で，k は$d2^k$ セットの高いビットを持つのに必要となる数です。割られる数に対するビットシフトは通常‘‘on the fly’’で実行されます。つまり，各ステップで適当なビットを抽出していくということです。こうすることで，商のリムが段々と計算されていきます。余りが欲しい時のみ，割られる数のシフトを戻して$r = a \bmod d2^k$ を計算し，最後のステップで$r2^k \bmod d2^k$ を行います。こうすることで，ビットシフトに時間がかかったり，レジスタの少ない CPU では高速化できます。

逆数の乗算は，一度に二つのリムをまとめて行うことができます。同じ計算を 2 回行うことになりますが，逆数は 2 つのリムで表現し，割る数の低いリムはゼロを詰め込んで扱います。余計な仕事が必要になり，逆数は 2×2 乗算が必要になりますが，4 つの 1×1 は独立に実行できますので，並列化可能です。同様にqd を求める 2×1 の計算も可能です。実質的には，2 リムを 2 回の乗算で実行した方が，1 回に 1 リムだけ扱うよりはレイテンシに対する効果が出てきます。これは一度に 3 〜 4 リムまで拡張します。逆数を適用する手間はかかりますが，乗算のスループットの限界まで高速化することができるようになります。

逆数については同様のアプローチを，割る数が半リムの場合は半リム計算にも適用することができます。この場合は，逆数に対する 1×1 乗算を 2 つの$\frac{1}{2\times1}$ 乗算に直すことができ，高速な半リム乗算を使うことで，パイプライン化されていなくても，CPU 資源の節約ができます。

15.2.2 筆算除算アルゴリズム

筆算 N×M 除算は手計算と同じ方法ですが，基数は$2^{\texttt{mp_bits_per_limb}}$ となります。詳細は Knuth section 4.3.1 algorithm D と，mpn/generic/sb_divrem_mn.cをご覧下さい。

簡単に解説すると，割られる数が割る数より大きい間は，商の高いリムが求められ，N×1 積であるqd が割られる数から引かれていきます。正規化した割る数(最大有効ビットが 1 ）を使うと，商を構成する各リムは 2×1 除算と 1×1 乗算に減算を何回か行って得ることができます。

2×1 除算は割る数の大きいリムで行われ，ハードウェアの除算命令か，逆数の乗算の高速な方を使います(Section 15.2.1 [Single Limb Division], p. 102と同じ)。商は大きくなり過ぎることもありますが，割る数を加減すれば防げますので，滅多に起こりません。

商のリムの個数である Q=N−M を用いると，これは$O(QM)$ のアルゴリズムになり，Q×M 筆算乗算アルゴリズムと同程度のスピードで実行できます。実際には，余計な乗算と除算が Q 個の商リムそれぞれに入りますので，多少違ってきます。

15.2.3 分割統治除算アルゴリズム

割る数が`DC_DIV_QR_THRESHOLD`より大きいと，除算は分割して実行されます。正確には，Moenck & Borodin, Jebelean, Burnikel & Ziegler (see 付録 B [References], p. 128)らによって提唱された分割統治法を再帰的に呼び出して計算します。

このアルゴリズムは，，2N×N の割算を筆算除法アルゴリズム(see Section 15.2.2 [Basecase Division], p. 102) を実行することで成立しているものです。ベースとして使用するのは N/2 リムであって，単一リムでないことに注意して下さい。こうして，(N/2)×(N/2)の乗算が使用できるようになり，Karatsuba 乗算や，他の多倍長乗算のメリットが生かせるようになります(see Section 15.1 [Multiplication Algorithms], p. 95). 商の 2「桁」分は，N×(N/2) 除算を繰り返し実行して導出します。

(N/2)×(N/2)乗算は，筆算アルゴリズムを用いて計算されますが，この計算量は筆算除法アルゴリズムと同程度です。但し，関数呼び出し回数が多くなり，乗法とは別に減算の手間が加わります。これらのオーバーヘッドは N/2 が`MUL_TOOM22_THRESHOLD`を超えて，分割統治アルゴリズムを使用するようになると発生します。

`DC_DIV_QR_THRESHOLD`は割る数のサイズ N で決まります。`MUL_TOOM22_THRESHOLD`の 2 倍を少し超える程度に設定してありますが，どの程度超えるかは CPU によります。最適化した`mpn_mul_basecase` 関数は，`mpn_submul_1`関数を繰り返し呼び出すことで有利にでき，結果として幾分低めの`DC_DIV_QR_THRESHOLD`になります。

分割統治アルゴリズムは漸近的に$O(M(N)\log N)$ となります。ここで$M(N)$ は FFT で実行される N×N 乗算の計算時間です。実際の計算時間は，Burnikel & Ziegler 論文の 2.2 節の最後で示されているように，繰り返し用いられるサイズの乗算時間の和になります。例えば，Toom-3 アルゴリズムの使用範囲では，分割統治法は$2.63M(N)$ になります。より高次のアルゴリズムを用いると，$M(N)$ の項を改善でき，そこに乗される値は小さくなり，結果として$\log N$ に近づけることができます。実際には，中〜大サイズのものに対しては，2N×N 乗算は 2 〜 4 倍，N×N 乗算よりも遅くなります。

15.2.4 ブロック Barrett 除算

大きな数で割る時には, ブロック Barrett 除算アルゴリズムを使います。ここで，割る数，割られる数のサイズによって精度を決め，割る数の逆数をこの精度で求めます。商リムのブロックは，割られる数にこの逆数をブロック単位で乗じることで生成されていきます。

我々の実装したブロックアルゴリズムは，素の Barrett アルゴリズムより小さい逆数を使いますので，サイズは$\lceil n/2 \rceil$ リム程度になります。

15.2.5 完全除算

いわゆる「完全除算」とは，割られる数が割る数の完全な倍数になっていることが既知の時の除算です。Jebelean の完全除算アルゴリズムは，この前提を利用して，いくつかの効果的な最適化を行います(see 付録 B [References], p. 128)。

考え方はこうです。例えば 10 進数の 368154 を 543 で割ることを考えましょう。割られる数の最後の桁は 4 ですので，商の最後の桁は 8 でなければなりません。これは$4\times7 \bmod 10$ から来ており，7 は 3 (割る数の最後の桁)の 10 を法とする逆数，即ち$3\times7 \equiv 1 \bmod 10$ となります。従っ

て，8×543 = 4344 は割られる数より差し引くことができ，363810 となります。結果として低い方の桁はゼロになります。

このやり方を 2 番目の桁にも適用すると，商の次の桁は 7 (1×7 mod 10)となり，7×543 = 3801 を引いて 325800 となります。最終的には商の 3 番目の桁である 6 (8×7 mod 10)が得られ，6×543 = 3258 を引いてゼロになります。従って，商は 678 となります。

しかしながら，乗法と減法は，割られる数の小さい 3 桁分に関しては必要ないことに気が付きます。3 桁の商を得るにはこれで十分だからです。ということで，商の最後の桁については減算も必要ありません。2N×N の除算についてもこれと同様にして実行でき，通常の筆算除算アルゴリズムの半分の労力で済みます。

N×M の完全除算は Q=N−M リムの商を生成するので，通常の筆算除算アルゴリズムを節約するために計算を 2 つのパートに分けます。まず最初に，商 Q の各リムは乗算 1 回で求められ，2×1 の除算や乗算は必要ありません。2 番目に，交差積は$Q > M$ ～$QM - M(M+1)/2$ であるか，$Q \leq M$ ～$Q(Q-1)/2$ であれば，減らすことができます。この節約は補足的なものです。Q が大きい時には多くの除算を減らすことができ，Q が小さければ，交差積の回数を減らすことができます。

使用するモジュラ逆数は`gmp-impl.h`の`binvert_limb`マクロで効率的に計算することができます。32bit リムでは 4 つ，64bit リムでは 6 つの積で得られます。`tune/modlinv.c`には，CPU にふさわしい，乗算の代わりのビット操作で行う別ルーチンを実装してあります。

準 2 次完全除算法は Jebelean が“Exact Division with Karatsuba Complexity”で提唱していますがまだ実装されていません。これは通常の除算に対して分割統治法的アプローチを取るものですが(see Section 15.2.3 [Divide and Conquer Division], p. 103)，低い桁から高い桁の方に計算を行っていきます。未実装ですが有望株としては“Bidirectional Exact Integer Division”（Krandick & Jebelean）があり，商リムを割られる数の両端の桁から求めていくもので，2N×N 除算に必要な交差積の回数を半減させることができます。

3 で割る時の完全除法の特別な場合は`mpn_divexact_by3`に実装してあります。これは Toom-3 乗法と`mpq` の標準化をサポートしています。3 のモジュラ逆数を乗じて商の桁を出していくものです(つまり`0xAA..AAB`)。次のリムに対して桁借り上げが必要になるかどうかを決める 2 回の比較も行います。桁借り上げの効用で乗算が不要になるので，パイプライン化した乗算機構が入っている CPU には有用でしょう。

15.2.6 完全剰余

完全除算アルゴリズムの各ステージで，フル減算が実行され，割られる数が割る数の倍数ではない時には，低い桁にはゼロリムが生成され，高いリムには剰余が生成されます。割られる数がa, 割る数がd, 商がq とすると，$b = 2^{\texttt{mp\_bits\_per\_limb}}$,より，剰余$r$ は次の形になります。

$$a = qd + rb^n$$

n は減算で生成されるゼロリムの数を表わしており，q に対して生成されるリムの数になります。r は$0 \leq r < d$ の範囲の数で，剰余とみなすことができますが，b^n の倍数でシフトすると得ることができます。

各ステージでフル減算を行って桁の借り上げを行うと，通常の除算として同数の交差積が実行されますが，いくつかの単一リム除算が節約できます。d が単一リムの時は，複数の簡易化が起き，プロセッサの数だけスピードアップできます。

`mpn_divexact_by3`関数, `mpn_modexact_1_odd`関数，内部関数の`mpn_redc_X` はそれぞれ微妙にr の返し方が違っており，上記の式における符号反転処理を先行して行っていたりしますが，基本的には同じ処理を行っています。

a がd の倍数であれば当然r はゼロになるので，通常の除算よりも，割り切れるかどうかのテストや合同かどうかのテストは高速になります。

r の因数b^n はd が奇数の時には GCD 内で無視できます。従って`mpn_gcd_1`や`mpz_kronecker_ui`等の関数を`mpn_modexact_1_odd` で利用できます(see Section 15.3 [Greatest Common Divisor Algorithms], p. 105)。

Montgomery の REDC 法は，モジュラ乗算を行う手法で，xb^{-n} やyb^{-n} という形のオペランドを利用します。$(xb^{-n})(yb^{-n})$ を計算する際には，b^n の倍数を完全剰余計算で利用し，$(xy)b^{-n}$(see Section 15.4.2 [Modular Powering Algorithm], p. 108)と同じ形の積を得ます。

r は一般的に通常の剰余$a \bmod d$ についての情報を得るには役立ちません，というのも$b^n \bmod d$ は何にでもなり得るからです。しかしながら$b^n \equiv 1 \bmod d$ であれば，r は通常の剰余の負数になります。これはd が$b^n - 1$ の因数であれば必ずそうなります。例えば 3 の場合は`mpn_divexact_by3`で起こります。32bit リムや 64bit リムでは，この因数が 5, 17，257 を含むので，このような場合に対する応用は効きません。

15.2.7 小さい商になる時の除算

N×M 除算において，商 Q=N−M のリムの数が小さい時には幾分最適化できます。

通常の筆算除法アルゴリズムは，割る数をシフトして正規化し，最大 bit の値を 1 にして，更に割られる数も同様にシフトして，計算の最後には剰余もシフトします。2，3 リムしかない商の場合はこれは無駄なので，代わりに，割られる数の上の方の2Q リム分を，割る数の上から Q リム分で割るようにして，お試し商を求めます。このためには最低でも割る数も割られる数も全てのリムを正規化する必要があります。

その後，乗算と減算をお試し商に適用し，割る数の M−Q 個の未使用リムと，割られる数の N−Q の未使用リム(ここでお試し商を出す除算の Q リム分を含む）を出します。最初のお試し商は 1，2 だけ大きくなる可能性がありますが，その場合は最初に，減算した結果得られる最大有効リムを比較することで大きくなっていることが分かります。加算して戻すのは商が 1 だけ大きくなった時です。

この計算全体は本質的には Q リムベースの筆算除法アルゴリズムの 1 ステップ分と同じですが，Q リム積の全体に行うのではなく，1 リム分にのみお試し除算を行うところが異なっています。この少し弱目のテストの正当性は，Knuth section 4.3.1 exercise 20 で確立されていますが，$v_2\hat{q} > b\hat{r} + u_2$ の条件については適度に緩められています。

15.3 最大公約数の計算

15.3.1 2 進 GCD

短い桁数の場合，GMP は$O(N^2)$ である 2 進スタイルの GCD アルゴリズムを使用します。これについては Knuth 本の 4.5.2 のアルゴリズム B を初めとして色々なテキストに解説が載っています。このアルゴリズムは，奇数a と奇数b に対しては，次の式を使ってリダクション処理を行っていくというものです。

$a, b = \mathrm{abs}\,(a - b), \min\,(a, b)$
a から因数 2 を取り除く

ユークリッド互除法は，Knuth 本のアルゴリズム E と A に解説されているように，商$q = \lfloor a/b \rfloor$ の計算を行い，a, b を$v, u - qv$ に置き換えを繰り返していきます。2 進アルゴリズムはこのユークリッド互除法よりも全体的に高速化できます。その理由は，各ステップにおける商が小さい値になり，1，2 回の減算を行うだけで，除算を行ったのと同じ効果が得られるからです。例えば，商が 1, 2，3 だったとすると，計算時間は 67.7% で済みます。詳細は Knuth section 4.5.3 Theorem E を参照して下さい。

この商が大きい数になると，b はa よりとても小さい値になるので，除算の労力が増大します。これによって，`mpn_gcd`と`mpn_gcd_1` (後者は N×1 と 1×1 の場合にも当てはまる)における，最

初の$a \bmod b$の労力が減ります。しかしこの労力の削減によって，大きな商が出現する率が減り，このチェックを行う必要もなくなります。

mpn_gcd_1 関数における最後の1×1 GCD は，前述の通り，汎用 C コードで実行しています。2 つの値が N-bit であるとすると，このアルゴリズムは 1bit あたり 0.68 回の反復が必要になります。処理を最適化するためには，a から因数 2 を除去する方法について考える必要があります。

まず最初に，2 の補数を取ると，$a - b$ の低い桁のゼロビット数は$b - a$ と同じになりますので，$\mathrm{abs}\,(a - b)$ の計算完了を待たずに，まず$a - b$ に対してゼロビットのカウントを行っていきます。

低い桁のゼロビット除去を行うループは，段々と分岐予測が難しくなっていきます。しかし，平均的には数ビット程度しかないので，1 ビット化 2 ビット除去する程度で済みます（AMD K6 ではそうしています）。必要であれば，テーブルを作って数ビットまとめてカウントすることもできます(AMD K7 の場合)。他の方法としては，a とb のどちらか一方が奇数になるようにして次のループを回します。

$a, b = \mathrm{abs}\,(a - b), \min\,(a, b)$
$a = a/2$ if a が偶数
$b = b/2$ if b が偶数

これによって，1bit あたり 1.25 回の反復が必要になりますが，各ステップにおける 1bit 除去処理における分岐を避けられます。1bit 除去を行うことで，大体 1bit あたり 0.9 回反復までコスト削減ができ，トレードオフの価値が出てきます。

このようなアプローチを取ると，一般的には大体 1bit あたり 6 サイクルの速度が出せますが，64bit GCD では 400 サイクルぐらいになり，あまり高速になったという感じではありません。GCD に除算可能性のチェックを組み合わせてもあまり効果がない，ということになりますね。

現在の実装では，$N < 3$ の時にのみ，2 進 GCD アルゴリズムを使うようになっています。

15.3.2 Lehmer のアルゴリズム

Euclid のアルゴリズムを改良した Lehmer の方法は，商の列の最初の部分が，入力値の最大有効桁にのみ依存して決まっているという観察に基づいています。GMP で使っているのは Lehmer のアルゴリズムの一種で，例えば Jebelean による(see 付録 B [References], p. 128)“A Double-Digit Lehmer-Euclid Algorithm” が示している通り，最大有効リムの二つを分割します。2 つの 2 リムの入力値の商は 2×2 行列に 1 リムごとに格納されます。これはmpn_hgcd2関数で行っています。この結果，行列はmpn_mul_1 やmpn_submul_1で使われる入力値に適用されます。通常は大体 1 リムごとに入力値を削減していきます。滅多にないことですが，商が大きい場合は大きい方の 2 有効リムを試すだけで計算がストップしてしまうことがあるので，この場合は商は通常の除算で求めるようにして下さい。

結果として得られるアルゴリズムは漸近的に$O(N^2)$ となり，Euclid アルゴリズムや 2 進分割法と同等程度になります。このアルゴリズムの 2 乗となる計算部分はmpn_mul_1とmpn_submul_1を呼び出して行っています。小さいサイズの場合は，ここよりも線型量となっている部分の方が比重が高くなります。つまり，大体N 回のmpn_hgcd2関数呼び出しが必要となります。この関数は重要な下記の 3 つの最適化を行っています。

- mpn_hgcd関数（次節を参照）と同じように厳密性を緩めた方法を使います。つまり，二つの大きな数の，最大有効リム 2 つ分を呼び出して生成した行列は，この 2 数の商の列の最初の部分と完全に一致しなくてもよい，とするものです。最終的な商は時々ずれることがあります。
- 商は常に小さい，という事実を利用します。どの CPU でも除算のアセンブラ命令は遅いので使用しません。（分かりずらい条件分岐をたくさん行えば改善できるとは思いますが）。

- 入力値のリムサイズを 2 から 1.5 リムにすることで，2 リムの計算から単一リムの計算に切り替え，コストを全体的に半分にします。

15.3.3 準 2 乗 GCD

入力値がGCD_DC_THRESHOLDよりも大きい時には，GCD は HGCD (Half GCD)関数を経由して計算します。これは Lehmer のアルゴリズムの一般形です。

入力値a, b のサイズをN リムとします。$S = \lfloor N/2 \rfloor + 1$ とすると，HGCD(a,b)は変換行列T を生成し，この要素は全て非負になります。さすれば$(c; d) = T^{-1}(a; b)$ にできます。こうして作られた減次された値c, d はS リムよりは大きくなりますが，その差$abs(c - d)$ はS リムに収まります。この行列要素のサイズも大体$N/2$ リムになります。

この HGCD は Lehmer のアルゴリズムを利用していますが，上記の停止条件を使うと，減次された値と対応する変換行列を半分のコストで返してきます。入力値がHGCD_THRESHOLDを超えていると，HGCD が再帰的に計算され，Möller (see 付録 B [References], p. 128)が“On Schönhage's algorithm and subquadratic integer GCD computation” で提唱している下記のような分割統治法が使用されます。

- HGCD を最大有効$N/2$ リムに対して再帰的に呼び出す。入力値全体に変換行列T_1 を適用し，最大サイズ$3N/2$ まで減次。
- $3N/2$ リムまで減次された値に対して除算と減算を数回行う。ここが大きな商の場合と異なる点です。
- HGCD を再帰的に，減次された数の最大有効$N/2$ リムに対して呼び出す。導出された変換行列T_2 をこの減次された数全体に適用し，更にそのサイズを最大$N/2$ まで縮める。
- $T = T_1T_2$ を求める。
- 小さい数の除算と減算を実行して要求を満足するまで行い，結果を返す。

この結果，GCD は Lehmer のアルゴリズムに似た HGCD を繰り返して実装されていることが分かります。Lehmer のアルゴリズムは，mpn_hgcd2関数の中で繰り返し大きい方の 2 リム分を切り捨て，変換行列を入力値全体に適用し，準 2 次 GCD はその最大有効リムの 3 番目を切り捨て(この切り捨てるリムの位置はチューニングパラメータで決定しますが，1/2 より大きく，1/3 ぐらいが適当のようです)，mpn_hgcdを呼び出し，その結果得られる変換行列を適用します。入力値がGCD_DC_THRESHOLD以下の長さになれば，Lehmer のアルゴリズムは残り 1 回だけ実行すれば良いことになります。

HGCD と GCD の両方の漸近的な計算時間は$O(M(N) \log N)$ になります。ここで$M(N)$ は二つのN リム長の乗算の計算時間です。

15.3.4 拡張 GCD

拡張 GCD 関数，GCDEXT はgcd (a, b) に加えて，$ax + by = \gcd(a, b)$ を満足する余因子x とy も導出します。通常の GCD アルゴリズムは全て拡張 GCD 計算に適用できます。2 進 GCD アルゴリズムは単一リム用の GCDEXT にのみ適用されます。Lehmer のアルゴリズムは，GCDEXT_DC_THRESHOLDのサイズまで適用され，それを超えるサイズに対しては GCDEXT は HGCD のループ適用を行い，余因子の導出用にテーブルを作って保存するようにします。こうすることで，GCD や HGCD の計算時間は漸近的に$O(M(N) \log N)$ となります。

通常の GCD との違いは，入力値a とb に対して減次プロセスを回す一方，余因子x とy の桁数がだんだん伸びていく所にあります。これによって計算を打ち切るポイントをどこに設定するのかが難しくなります。現在の実装では，最初の HGCD の呼び出しに対しては入力値の半分程度の有効桁数のところで打ち切るようにし，以降の呼び出しでは有効桁数の2/3 程度に対して実行するようにしています。このやり方はまだ改良の余地があります。また，停止則についても，Lehmer のアルゴリズムを一度実行して入力値をGCDEXT_DC_THRESHOLD以下になるようにしたところで止める，というだけでなく，その時点の余因子のサイズも考慮すると改善できるのではないかなぁと思っています。

15.3.5 Jacobi 記号の計算

[この節は廃止予定です。現在の Jacobi 記号のコードはもっと効率的なものに置き換わっています。]

`mpz_jacobi`関数と`mpz_kronecker`関数の現在の実装は，前述の GCD アルゴリズム(see Section 15.3.1 [Binary GCD], p. 105)のところで述べた 2 進 GCD アルゴリズムを簡易化したものを使っており，入力値が大きい値の時はさほど高速ではありません。Lehmer のアルゴリズムや 2 進アルゴリズムを多段階にしたものを使えばマシになるかもしれません。

入力値が 1 リムで収まっている場合は，`mpz_kronecker_ui`関数やその周辺の関数でも同様ですが，最初のリダクションは`mpn_mod_1`関数か`mpn_modexact_1_odd`関数を使って行われます。これらは 1 リム対象の 2 進アルゴリズムを用いているもので，この場合については大変に効率的に実行できます。

テーブル参照や条件分岐を防ぐため，本ルーチン内では計算結果の符号変化については数ビット内に収まるように操作しています。

15.4 べき乗の計算

15.4.1 通常のべき乗

通常の`mpz` や`mpf`関数におけるべき乗は，2 進べき乗アルゴリズムを使っており，指数の 2 進表現の"1"ビットに対して 2 のべき乗と乗算を繰り返して行っています。詳細は Knuth 本の 4.6.3 を見て下さい。ここではアルゴリズム A ではなく，この Knuth 本にある「左から右へ」という方法を使っています。というのも，幾分か一時メモリ領域を減らして実行できるからです。

15.4.2 べき剰余

べき剰余は，2^k 項のスライディングウィンドウアルゴリズム(“Handbook of Applied Cryptography” algorithm 14.85 (see 付録 B [References], p. 128)参照）で実装されています。k は指数の大きさに従って決めます。大きな指数に対しては大きなk を使用して，2 乗を求めるための平均乗算回数を最小化します。

剰余乗算と剰余平方は，単純除算か，Montgomery アルゴリズム(see 付録 B [References], p. 128)による REDC によって求めます。REDC の方は多少高速にはなりますが，本質的には N 回の単一リム除算を，完全剰余計算(see Section 15.2.6 [Exact Remainder], p. 104)に似たやり方で減らしています。

15.5 べき乗根のアルゴリズム

15.5.1 平方根のアルゴリズム

平方根は，Paul Zimmermann (see 付録 B [References], p. 128)が提唱した“Karatsuba 平方根” アルゴリズムを使用します。入力値n をk bit 単位で 4 つに分割し，$b = 2^k$ を用いて，$n = a_3b^3 + a_2b^2 + a_1b + a_0$ とします。a_3 の部分は，最大 bit もしくはその次の bit が 1 になるように“正規化”されていなければなりません。GMP では, k はリムの境界上に置かれ，入力値は偶数ビット数分左シフトして正規化します。

上位から二つ分の平方根は下記の反復を適用することで得られます(1 リム用の Newton 法で収束させる）。

$$s', r' = \text{sqrtrem}\,(a_3b + a_2)$$

これによって平方根の近似値が得られるので，s,r を求めるための除算を行って平方根の精度を上げていきます。

$$q, u = \text{divrem}\,(r'b + a_1, 2s')$$
$$s = s'b + q$$
$$r = ub + a_0 - q^2$$

a_3 を正規化する理由は，この時点においては，s は正しい値か 1 だけ大きい値になっているからです。後者の場合はr は負数になっているので，下記のように補正します。

$$\begin{aligned} &\text{if } r < 0 \text{ then} \\ &\quad r \leftarrow r + 2s - 1 \\ &\quad s \leftarrow s - 1 \end{aligned}$$

このアルゴリズムは，分割統治法のスタイルで表現されていますが，論文で指摘しているように，Newon 法の離散バージョンとしてみることも，一度に 2 桁平方根を出していくスクールボーイ法(いちいち説明しねぇよ）の変種と見ることもできます。

剰余r が不要の場合は，r とu の上の数リムだけ求めて，s の調整量が必要かどうかを見るだけに使用できますが，このような最適化は今のところ実装していません。

Karatsuba 乗算の範囲内では，このアルゴリズムは$O(\frac{3}{2}M(N/2))$ となります。ここで$M(n)$ はn リム長の 2 数の乗算の計算時間です。FFT 乗算の範囲内では，$O(6M(N/2))$ で抑えられます。実際には，Karatsuba 乗算と 3 方向 Toom-Cook では 1.5 〜 1.8 倍ぐらいですが，FFT 乗算の範囲ではこれが 2 〜 3 倍に増えます。

このアルゴリズムは全て整数演算で行います。`mpn_sqrtrem`関数の結果は`mpz_sqrt`関数や`mpf_sqrt`関数で利用されています。`mpf_sqrt_ui`関数では，ゼロリムをパディングすることで長い桁の平方根を導出しています。

15.5.2 N 乗根

整数の N 乗根は下記のような Newton 法による反復を行うことで得られます。ここでA は入力値，n は指数を表わし，$A^{1/n}$ を導出します。

$$a_{i+1} = \frac{1}{n}\left(\frac{A}{a_i^{n-1}} + (n-1)a_i\right)$$

初期値a_1 は，2 進表現の‘‘1"ビットごとにべき乗を行ってビット単位で生成し，真の根より大きくなるようにします。初期値が良ければ 2 乗オーダーで収束していきます。n が大きい時には，初期値の長さを大きく取ると収束状態が良くなります。現在の実装では特段最適化は行っていません。

15.5.3 完全平方

入力値が小さい整数に対する剰余平方根になっているかどうかをチェックすると，完全平方数ではない有効小数部を素早く確定することができます。

`mpz_perfect_square_p`関数はまず最初に入力値の 256 に対する剰余を求めて，低いバイト値の方を確認します。256 の剰余平方根に対しては 44 種類しかないので，入力値の 82.8% は即座に完全平方数ではないということが確定します。

32-bit システム上では，同様のテストを 9, 5, 7, 13，17 の剰余に対して行い，結果として入力値の 99.25% が完全平方数ではないことを確定しています。64-bit システム上では更に 97 の剰余テストを加えてこの確率を 99.62% まで高めています。

これらの数は$2^{24}-1$ (64bit システムでは$2^{48}-1$)の因数になっているものを使っています。さすれば，この剰余は加算を行うだけで高速に導出できます(`mpn_mod_34lsub1`関数参照)。

ネイルが利用できる場合は，`gen-psqr.c`プログラムを使って剰余演算の代わりができ，`mpn_mod_1`関数が使えるようになります。$2^{24}-1$ や$2^{48}-1$ でも余分なビットシフトを使ったネイルを用いて同様のことができますが，今のところ実装していません。

どんな場合でも，剰余計算は`mpn_mod_34lsub1`関数，もしくは`mpn_mod_1`関数で実行し，テーブル参照で完全平方数でないことを確定します。‘‘modexact’’スタイルの計算を行って，ふさわし

い置換テーブルを使うと，乗算は 1 回だけで済むようになります（詳細はソースコードを参照）。参照テーブルが特別大きいものでなければ，剰余演算もこの組み合わせを使うことでコスト削減できます。gen-psqr.cではこの手の前処理を全て行っています。

平方根はこのテストを通過した値全てに対して導出できなければなりません。その数は当然，真の平方数でなくてはならず，テストに通過してしまうわずかな小数が出てくるものでもダメです。(即ち，準平方数になる場合もダメ)。

となれば，当然更なる剰余テストが必要になりますが，mpz_perfect_square_p関数ではそのためのコンパクトで効率的なテスト集合だけを利用します。入力値が大きい時には確率的にこの手の追加剰余テストが有効ですが，小さい場合はあまり効果が出ません。入力値における完全平方数 vs. 非平方数の分布を考えてこの手のテストの効率を考える必要があります。

15.5.4 完全べき乗

完全べき乗を見つけるには，いくつかの因数分解アルゴリズムを組み合わせて使う必要があります。現状の実装ではmpz_perfect_power_p関数は，素数べき乗根についての考慮は必要になりますが，N べき乗根を繰り返すようになっています(See Section 15.5.2 [Nth Root Algorithm], p. 109)。

素数因子pがe個見つかったとすると，eの因数となるべき乗根だけ考慮すればよく，計算の節約ができます。そのためには，小さい素数で割り切れるかどうかをチェックすることになります。

15.6 基数変換

基数変換の重要度はさほど高くありません。もし変換に時間を取られるプログラムになっているのであれば，別のデータ形式を使うべき，ということを意味しますので。

15.6.1 2 進数からの変換

2 進表現から，2 のべき乗の基数表現への変換は簡単で，$O(N)$ のビット抽出アルゴリズムが使えます。

2 進表現からそれ以外の基数表現に変換するには，これから述べる二つのアルゴリズムのどちらかを利用します。GET_STR_PRECOMPUTE_THRESHOLD以下のサイズの時には基本的な$O(N^2)$ の方法を使います。これはb^n で繰り返し除算していきます。ここでb は基数，n は 1 リム分に収まる最大のべき乗の指数です。単に剰余r を使う代わりに，追加的に除算を行うことで，r/b^n を表現する小数リム部分が導出できます。この時，r の桁数はb を乗じて得ることができます。10 進の場合は，特別にコードを記述してあり，10 を乗じる処理をシフトと加算だけ使用して最適実装してあります。

GET_STR_PRECOMPUTE_THRESHOLDを超えるサイズの時には，準 2 次アルゴリズムを使います。入力値をt とすると，基数のべき乗b^{n2^i} が計算され，べき乗が$t \sim \sqrt{t}$ になるまで繰り返されます。さすれば，t は最大のべき乗で割られるので，べき乗数以上の桁数となる商と，それより小さい剰余が得られます。この 2 数は，2 番目に大きいべき乗で更に割られて，同様の結果を得ます。これを繰り返して返還を行います。最終的にはGET_STR_DC_THRESHOLDリムまで値が小さくなった時に，前述の基本アルゴリズムが適用されて終了します。

このアルゴリズムの利点は，デカい数の除算を準 2 次の分割統治法アルゴリズムで行っていることです(see Section 15.2.3 [Divide and Conquer Division], p. 103)。その結果，この除算のオーバヘッドが単一リムの除算を繰り返すよりも相対的に小さく抑えられます。どんな場合でも，べき乗b^{n2^i} の計算が一番コスト高なので改善が必要です。

GET_STR_PRECOMPUTE_THRESHOLD とGET_STR_DC_THRESHOLDは基本同じものを表現しており，デカい数の除算を行って入力値を半分にカットするメリットが出る地点を表わしています。GET_STR_PRECOMPUTE_THRESHOLDは基数のべき乗の計算コストも考慮しており，GET_STR_DC_THRESHOLDはそれを含めた上で，更に再帰呼び出しのコストも考えて設定します。

ベースケースの場合は小さい桁から大きい桁の方に桁導出を行っていますが，やりたいのは大きい桁から小さい桁への格納（表示）になりますので，格納に必要な桁数，もしくは最低でも過小評価にはならないようにすることが求められます。GMP では入力ビット数を`chars_per_bit_exactly`を`mp_bases`から乗じて切り上げることで得ています。その結果はピッタリ以上の桁数になります。

入力値の大きいビット数のいくつかを調べれば，何桁表示になるかを知ることができるようになりますが，毎回そうそうピッタリ分かる訳ではなく，大体，100...になるか，99...になるかは末尾の数ビット分が効くわけで，こればかりは実際に 99...を出してみるまでは分かりっこありません。

以上述べてきたr/b^n スキームは乗算を使って桁を出しており，1 リム以上の数には有用です。何度か実験をしてみましたが，実装が悪いのか，再帰呼び出ししてもあまり劇的な改善は望めません。準 2 乗除算を使っても同じようなものですので，デカい基数べき乗を計算するコストが影響しているのでしょう。

準 2 乗部分の改善案としては，他にも基数べき乗部分を，商と剰余のサイズのバランスを取るように決めるというものがあります。これは最大のべき乗がb^{nk} となり，大体$\sqrt{t}$ に等しいことから，2^i には拘らなくても良いということです。計算時間とサイズのグラフを描いてみるとうまく速度向上しているはずですが，実際のところは何とも言えません。

15.6.2 2 進数への変換

ここに書いてあるのは GMP 4.3 以前のアルゴリズムなので，書き直す必要があります。

2 のべき乗を基数とする表現から 2 進数への変換は，簡単で高速な$O(N)$ のビットごとに導出するアルゴリズムが使えます。

それ以外の基数表現から変換するにはこれから述べる 2 つのアルゴリズムのどちらかを使います。サイズが`SET_STR_PRECOMPUTE_THRESHOLD`以下の場合は，基本的な$O(N^2)$ 法を使い，n 桁ごとにリムに変換します。n は 1 リムに収まる基数b の最大べき乗です。このグループをb^n を乗じた結果として保存しておき，足し込んでいきます。Knuth 本の 4.4 part E (see 付録 B [References], p. 128)によれば，これで多倍長演算を節約できます。10 進からの変換については特別にコード化しており，コンパイラで 10 の乗算を最適化できるようにしてあります。

`SET_STR_PRECOMPUTE_THRESHOLD`を超えるサイズの場合は，準 2 次アルゴリズムを使い，n 桁に分割した最初のグループをリムに変換します。さすれば，前のリムとくっついて$xb^n + y$ というリムのペアになります。ここでx とy はリムを表わします。この隣り合うリムペアは更に 4 つのリムに統合され$xb^{2n} + y$ となります。これを最後の 1 ブロックになるまで続け，最終結果が返されます。

この方法の利点は，各x に対する乗算がデカいブロックになっていることで，ここに Karatsuba アルゴリズムや高次の乗算アルゴリズムが適用できるようになります。とはいえ，べき乗b^{n2^i} を計算するコストは改善すべきでしょう。`SET_STR_PRECOMPUTE_THRESHOLD`は大概大きめにとってあり，大体 5000 桁ぐらいですが，CPU によっては大きすぎるかもしれません。

`SET_STR_PRECOMPUTE_THRESHOLD`は入力値を基準に決まっています(10 進表現用には最適化してある）がリム数を基準とした方が，基数依存にならず良いかもしれません。このカウントはベースケースでは使用されませんので，最初に計算しておくか評価しておく必要が出てくるでしょう。

`SET_STR_PRECOMPUTE_THRESHOLD`を使う理由は，対応する`GET_STR_PRECOMPUTE_THRESHOLD` よりずっと大きい値になるからで，`mpn_mul_1`関数は，`mpn_divrem_1`関数よりずっと高速になります(大体 5 倍以上）。

15.7 その他のアルゴリズム

15.7.1 素数テスト

`mpz_probab_prime_p` (see Section 5.9 [Number Theoretic Functions], p. 39)で行っている素数テストは，最初に小さい因数でお試し的に除算を行い，しかる後に Miller-Rabin 確率的素数テストアルゴリズムを行います。これは Knuth 本の 4.5.4 algorithm P (see 付録 B [References], p. 128)に解説されている方法です。

入力値n が奇数で，奇数n を用いて$n = q2^k + 1$ と表現できるならば，このアルゴリズムは乱数x を選び，$x^q \bmod n$ が 1 か-1, もしくは，$x^{q2^j} \bmod n$ が1 かどうかを$1 \le j \le k$ に対して確認します。このどれかに当て嵌まるのであれば，n は素数である確率が高く，あてはまらないのであれば，n は素数に非ずと断言できます。

任意の素数n はこのテストを必ずパスしますが，素数でないものも幾つかはパスしてしまいます。このような非素数をベースx の強い偽素数（strong pseudoprimes）と呼びます。強い偽素数が存在する確率が1/4 以上になるn は存在していません(Knuth exercise 22 参照)ので，x をランダムに選ぶと，「確率的素数」が非素数であるような確率は1/4 未満に抑えられます。

実際，強い偽素数は滅多に存在しておらず，解析的に知られている以上にこのテストは強力です。1/4 は任意のn に対して調べられたうち最大の確率です。

15.7.2 階乗の計算

階乗は二つのアルゴリズムを組み合わせて計算しています。考え方としては: 階乗の基数部分を計算する；最終段階で 2 のべき乗の乗法をシフト演算で実行する，というものになります。

n が小さい時には，$n!$ の奇数因子の計算は次のように行えることが分かります。即ち，n より小さい全ての正の奇数因子の積と，$\lfloor n/2 \rfloor!$ の奇数因子を乗じたものが等しくなります。ここで$\lfloor x \rfloor$ はx の整数部を意味します。この関係を繰り返して使用して階乗の計算を行います。下記にその例を示します。

$$23! = (23.21.19.17.15.13.11.9.7.5.3)(11.9.7.5.3)(5.3)2^{19}$$

現在の実装では，全ての因数を一本のリストに集め，再帰的ではないループでその積をバンバン計算していきます。

n が大きい時には，素数櫛を通して計算を行います。この時には Peter Luschny が提唱したヘルパー関数を使っています。

$$\mathrm{msf}(n) = \frac{n!}{\lfloor n/2 \rfloor!^2 \cdot 2^k} = \prod_{p=3}^{n} p^{\mathrm{L}(p,n)}$$

ここでp は素数です。指数k は奇数になるようにしておきます。というのも，k は$\lfloor n/2 \rfloor$ の 2 進表現における“1"ビットの数にしておきたいからです。関数 $\mathrm{L}(p,n)$ は，p が合成数である時にはゼロとします。任意の素数p に対しては次のように計算を行います。

$$\mathrm{L}(p,n) = \sum_{i>0} \left\lfloor \frac{n}{p^i} \right\rfloor \bmod 2 \le \log_p(n)$$

このヘルパー関数を使うと，$n! = \lfloor n/2 \rfloor!^2 \cdot \mathrm{msf}(n) \cdot 2^k$ という漸化式を使って，$n!$ の奇数部分の計算ができるようになります。この漸化式は，$\lfloor n/2^i \rfloor$ 上の小さいn に対するアルゴリズムを用いて停止させます。

上記の二つのアルゴリズムはどちらも 2 進分割法を使ってたくさんの小さい因数の積を求めています。最初にできる限りたくさんの数の積を一つのレジスタに求めておき，マシンワードに収

まる因数のリストを生成していきます。このリストは 2 分割して，その積を再帰的に求めていきます。

このような分割法を使用することで，繰り返し N×1 の積を行うよりは効率的に実行できます。大きな数の積を作っていくと，Karatsuba アルゴリズムや高次のアルゴリズムを使うことで効率が良くなるからです。Karatsuba 法の閾値より小さい場合は筆算乗算アルゴリズムを使うことになりますが，効率アップを狙って大きなブロック単位で実行するようにできます。

15.7.3 二項係数

二項係数$\binom{n}{k}$ は，まず最初に$k \le n/2$ の場合は$\binom{n}{k} = \binom{n}{n-k}$ と変換し，しかる後に下記の積を$i = 2$ から$i = k$ 方向に取って簡略化していきます。

$$\binom{n}{k} = (n-k+1)\prod_{i=2}^{k}\frac{n-k+i}{i}$$

各分母i は積に分割できることは簡単に分かるので，完全除算アルゴリズムが利用できます(see Section 15.2.5 [Exact Division], p. 103)。

分子$n-k+i$ と分母i はまず最初にリムサイズに収まる程度まで積を取っておき，なるべく多倍長演算量を抑えるようにします。`mpz_bin_ui`関数は除算に対してのみ使うようにします。n は`mpz_t`型で，$n-k+i$ は一般に 1 リムに収まるとは限りません。

15.7.4 Fibonacci 数

Fibonacci 関数`mpz_fib_ui`と`mpz_fib2_ui`は，F_n もしくはF_n とF_{n-1} を個別に効率的に計算するためのものです。

小さいn の場合，`__gmp_fib_table`テーブルに入っている 1 リム長分の値が使われます。32-bit リムの場合はF_{47} まで，64-bit リムの場合はF_{93} までが格納されているものです。利便性を上げるため，この表はF_{-1} から始まっています。

このテーブルを超える範囲の値の場合は，2 進べき乗アルゴリズムを用いてn の 2 進表現を高い方から低い方へ，F_n とF_{n-1} をペアで求めます。漸化式は下記の通りです。

$$\begin{aligned} F_{2k+1} &= 4F_k^2 - F_{k-1}^2 + 2(-1)^k \\ F_{2k-1} &= F_k^2 + F_{k-1}^2 \\ F_{2k} &= F_{2k+1} - F_{2k-1} \end{aligned}$$

各ステップにおいて，k はn の高い方のb ビット分にあたります。n の次のビットがゼロの時は，F_{2k} とF_{2k-1} が使用され，1 の時はF_{2k+1} とF_{2k} が使用されます。このプロセスが繰り返され，n の全てのビットに対して行われます。これらの漸化式は，n の 1 ビットに対して 2 回の 2 乗が実施されます。

最初の 2，3 のn を単一リムテーブル上で加算を使うだけで扱うことができます。この時には Fibonacci 数列の漸化式$F_{k+1} = F_k + F_{k-1}$ を使います。しかし，n は 10 〜 20 個程度まとめて扱う方が高速なので，このためのコードは意味がないと思われます。もしそうしたければ，データテーブルを拡大した方がいいでしょう。

テーブルを使うことで，小さい数に対する計算を大幅にカットできますし，小さいn に対しては高速にできます。テーブルを大きくしてやれば更に小さいn に対しては高速にできますが，サイズと計算速度のバランスをどうとるかが問題になってきます。GMP ではなるべくコードはコンパクトにしたいので，べき乗アルゴリズムを基本的には使うようにしています。

`mpz_fib2_ui`関数はF_n とF_{n-1} の両方を返しますが，`mpz_fib_ui`関数はF_n だけを計算します。この場合はこのアルゴリズムの最後のステップで，2 回の 2 乗の代わりに乗算を 1 回だけ実行す

るようにしています。下記の 2 式のうち一つがそれで，n の偶奇によって使い分けます。

$$F_{2k} = F_k(F_k + 2F_{k-1})$$
$$F_{2k+1} = (2F_k + F_{k-1})(2F_k - F_{k-1}) + 2(-1)^k$$

ここで，F_{2k+1} は上記の場合は同じもので，乗算のためにアレンジしているだけです。$2(-1)^k$ の項は両方，桁の借り上げ・桁上がりが発生しないので，低いリムの方向に計算を進めることができるようになっており，コード量の削減ができています。どうやっているかは，`mpz_fib_ui`関数や基盤関数`mpn_fib2_ui`関数のコードのコメントをご覧下さい。

15.7.5 Lucas 数

`mpz_lucnum2_ui`関数は，下記の式に従って Fibonacci 数のペアから Lucas 数のペアを導出します。

$$L_k = F_k + 2F_{k-1}$$
$$L_{k-1} = 2F_k - F_{k-1}$$

`mpz_lucnum_ui`関数はL_n のみ求めるもので，計算が効率化されています。n にある連続したゼロビットがあると，2 乗の計算が効率化できます。

$$L_{2k} = L_k^2 - 2(-1)^k$$

更に，n の末尾 1bit 分で，`mpz_fib_ui`関数と同様に，Fibonacci 数のペアの掛算を効率化しています。

$$L_{2k+1} = 5F_{k-1}(2F_k + F_{k-1}) - 4(-1)^k$$

15.7.6 乱数

`urandomb`関数は，乱数生成器から出てくる鎖状のビット列を用いて乱数を生成しています。この生成器が良いいランダム性を保持している限り，生成される乱数もうまくばらけたN bit の値になっているはずです。

`urandomm`関数は，$0 \le R < N$ の範囲の乱数を生成します。方法としては，$R < N$ を満足するようになるまで，$\lceil \log_2 N \rceil$ bit のR を抽出します。大概，1 ，2 回計算すれば出てきますが，生成器がおかしくなって 1bit 程度しか出力しなくなったりすると，繰り返し計算する必要が出てきます。

Mersenne Twister 乱数生成器は，松本 & 西村(see 付録 B [References], p. 128)が提唱したものです。これは繰り返しのない$2^{19937} - 1$ 周期の生成器です。$2^{19937} - 1$ が Mersenne 素数なのでこの名がついています。乱数状態保存変数は 624 ワード(32bit /word)長で，32bit word ごとに XOR とシフトを用いて与えられますので非常に高速です。ランダム性に優れており，GMP ではデフォルトのアルゴリズムになっています。

線型合同乱数生成器は，色々なテキストに解説があり，例えば Knuth 本 volume 2 (see 付録 B [References], p. 128)があります。法をM，パラメータをA とC で与え，状態保存はS に行い，ここに$S \leftarrow AS + C \bmod M$ の計算を繰り返して生成を行います。各ステップでは新しい状態が，M を法として前の状態の線型関数として計算されるので，この名前がついています。

GMP では，2^N のみ法としてサポートしており，現在の実装では特に最適化も行っていません。N が小さい時にはオーバーヘッドも大きくなり，N が大きいと各ステップにおける乗算が重くなっていきます。あらゆる点で Mersenne Twister 乱数生成器が優れているので，通常のアプリケーションに利用することはお勧めしません。

どちらの乱数生成器でも，現在の乱数生成状態は出力値をじっくり観察して，いくつかの線型計算(Mersenne Twister の場合は体 GF(2)上）を試してみると予測できてしまう恐れがあります。一般的には，長いハッシュ値などを利用せずに暗号化を行うようなアプリケーションに，乱数生成器の生の出力を使うことは望ましいことではありません。

15.8 アセンブラコード

GMP のアセンブラサブルーチンは，小〜中サイズの多倍長演算を高速に実行するための中核です。大サイズ用の演算にはアルゴリズムの選択の方が重要となりますが，基盤ルーチンが高速化できれば当然，計算のサイズに関わらずパフォーマンスは良くなります。

GMP で重要なのは桁上がり・桁下がり(キャリー操作）や乗算数のサイズ広域化で，これは C では簡単に実現できるものではありません。GCC の`asm`ブロックを`longlong.h`に記述してこれらの高速化を行っていますが，それだけでは足りず，職人芸的にアセンブラコードを書くことで，汎用 C コードに比べて 2 倍から 10 倍程度の高速化を達成しています。

15.8.1 アセンブラコードの構成

`mpn`下のディレクトリには CPU 用のプログラムが入っており，C かアセンブラで記述されています。`mpn/generic`ディレクトリにはデフォルトの汎用 C コードが入っており，特に CPU 指定のない時にはこのコードが使われます。

`mpn`の各サブディレクトリは ISA ファミリー毎に分かれています。各ファミリーの一般的な 32bit, 64bit 用の部分は同じようには書けませんので，別のディレクトリに分かれています。同じファミリーであっても全然別のサブディレクトリに存在している CPU 種別も存在します。

`nails`ディレクトリ内のサブディレクトリごとに，CPU タイプ用にネイルをサポートしたコードが入っています。各ファイルに書いてある`NAILS_SUPPORT`ディレクティブ部分はそこで使用するネイル値を示しています。ネイル用のコードが書いてあるのは，通常のコードより高速化できる場合に限られており，それができそうもない時にはネイルサポートはありません。

15.8.2 アセンブラコードの基本

`mpn_addmul_1`関数と`mpn_submul_1`関数が最も重要なルーチンで，これが GMP のパフォーマンス全般に効いてきます。すべての乗算の除算はこの二つの関数を繰り返し使用して実装されています。`mpn_add_n`関数，`mpn_sub_n`関数，`mpn_lshift`関数，`mpn_rshift`関数は 2 番目に重要な関数です。

いくつかの CPU では，基盤的内部関数である`mpn_mul_basecase`関数と`mpn_sqr_basecase`関数は，主として関数呼び出しによるオーバーヘッドを避けることで結構なスピードアップを図ることができます。つまり，スーパースカラープロセッサをうまく使うことで，`mpn_addmul_2`関数や`mpn_addmul_4`関数のようにより幅の広い数を扱う基盤関数を使うのと同じぐらいの有効利用ができる可能性が出てきます。

計算元の値と，計算結果の格納先が同じメモリ領域を使用しないように制限することで(see Chapter 8 [Low-level Functions], p. 59)，実装をより良いものにすることができるようになります。例えば，`mpn_add_n`関数にそのような制限をかけることで，スーパースカラープロセッサ上で書き込むより先に読み取りを行うことができ，桁上がり・桁下がり処理の有無にもよりますが，ループ展開してベクトルプロセッサ対応させることができるようになります。

15.8.3 桁上がり・桁借り上げ処理

GMP における最もチャレンジングな課題は，リムからリムへの桁上がり・桁借り下げ処理です。`mpn_addmul_1`関数や`mpn_add_n`関数では，桁上がり処理はリム間の処理にお任せしています。

キャリーフラグを持っているプロセッサ上では，直線的な CISC スタイルの`adc`を使うのが一番効果的です。AMD K6 の`mpn_addmul_1`は，例外的に分岐処理がうまく働く環境の一例です。

RISC プロセッサ上では，通常，オーバーフローした値に対しての加算と比較が使用可能です。この種のものは`mpn/generic/aors_n.c`で見ることができます。桁上がり・桁借り上げ処理は，環境によって異なりますが，4 命令使う必要があるケースがあり，これは少なくとも 1 リムあたり 4 サイクル必要になるということです。.別の環境では 1 ，2 命令で済むこともあります。広

いスーパースカラープロセッサのパフォーマンスは完全に，リムごとの桁借り上げ(carry-in)，桁下がり(carry-out) 処理に要する命令数で決まってきます。

ベクトルプロセッサ上では，1 ビットの桁上がりは 1 リムを超えることはめったにない，という事実を踏まえて効率化します。つまり，リムに 1 ビット加える際には，桁上がりによるハミダシはリムが0xFF...FFである時に限られ，この結果，2^mp_bits_per_limb になります。mpn/cray/add_n.c はこの処理を行っている具体例で，この中では全てのリムを並列に加算し，キャリービットもまとめて並列に加えており，そこからさらにハミダシ桁上がりが発生することは滅多にないことが分かります。

x86 プロセッサ上では，GCC (例えば version 2.95.2)は RISC スタイルのイデオムで，良いキャリー処理のコードを生成することはありません。本来ならadc やsbbを使うのが望ましいところで，条件付きジャンプを生成してしまいます。キャリービットを含むループは，どうもアセンブラコードで書かないといい結果が得られないようです。

15.8.4 キャッシュ制御

GMP は，オペランド（被演算数）が L1 キャッシュに入り切るかどうかに関わらず，高速に演算を行うように設計されています。

mpn_add_n やmpn_lshiftのような基盤関数は大きな桁の演算に利用されることが多く，L2 キャッシュやメインメモリのパフォーマンスがこれらの関数に大きな影響を与えます。mpn_mul_1関数やmpn_addmul_1関数は筆算乗算や筆算二乗では頻繁に利用されるので，mpn_mul_basecase関数やmpn_sqr_basecase関数のアセンブラバージョンの有無に関わらず，L1 キャッシュのパフォーマンスが効いてきます。この二つの関数は大きな桁の演算にも良く利用されます。

L2 キャッシュやメインメモリ関数に対しては，演算回数よりも，メモリアクセスの回数が確実に多くなります。従って，メモリスループットを最大化することを目標に，次のキャッシュラインから呼び出すようにするとともに，一度呼びだしたデータを再使用するようにします。CPU がロックアップフリーキャッシュや先読み命令を持っているのであれば，スループットを向上させることができるでしょう。現在の CPU は大概この両方を持っています。

先読み命令はループ展開と相性が良く，展開されたループ 1 回ごとに先読みが初期化されます（ループがキャッシュラインを複数カバーしていれば 2 回以上初期化される）。

書き込み用に確保されたキャッシュがない CPU の場合，先読みされる先は個別に書きこまれますので，キャッシュ階層の下に下がってくることはなく，（メインメモリの）バンド幅の制約を受けます。当然mpn_divrem_1のような関数の計算は遅くなりますので，書き込みスループットが重要になってきます。

先読み命令の間隔は，メモリレイテンシとスループットの兼ね合いで決まってきます。目的は当然，データを連続的に受け付けることなので，ピーク性能はスループットになります。ある種の CPU はフェッチ数と先読み数に実行中の制約が存在します。

特別な先読み命令が存在していない場合は，生のロード命令が使われますので，過去に読み込まれたオペランドを最後から読み取ろうとはしません。従って，セグメンテーション違反が発生してしまう可能性があります。

ある種の CPU やシステムにはシーケンシャルメモリアクセスを検知するハードウェアを持っており，これによって自動的にキャッシュの動作を初期化することができ，利便性が高まります。

15.8.5 関数ユニット

アセンブラルーチンでループをする方法を選択するにあたっては，どんな演算を同時に実行するのか，スループットはどの程度になるのか，ということを考えることになります。

ループの制御は普通，カウンタとポインタの更新が必要となり，5 命令程度でコストを考え，分岐による遅れが加わってきます。CPU のアドレスモードによってはポインタの更新回数を減ら

す効果が見込めます。一例としては，1 つだけポインタの更新を行うと，他のポインタはそれを基準とするオフセットとして表現できる場合が挙げられます。また，CISC CPU では，全てのアドレス計算は，スケールインデックスを用いてループカウンタと連動できたりします。

最終的なループ制御のコストは，ループ 1 回のなかで複数リムを処理することで償却できます(see Section 15.8.9 [Assembly Loop Unrolling], p. 119)。ループ制御がそれほどコスト高でない場合は，これを保証できます。

メモリのスループットは常に障壁になります。1 サイクルで 1 回のみロード，もしくはストアすることができる場合は，3 サイクル/リムが，2 進演算，例えば`mpn_add_n`関数のようなものの最高速になり，これが全てのコードにおける最高値になります。

整数リソースは，ループカウンタを浮動小数点レジスタに持つようにするか，必ず 2 番目のリムは浮動小数点数として持つようにして，浮動小数点演算ユニットを乗算用に使用するようにすれば，開放することができるようになります(see Section 15.8.6 [Assembly Floating Point], p. 117)。

浮動小数点リソースは，整数演算として桁上がり・桁借り上げ処理を行うようするか，整数としてビット操作して整数を浮動小数点数に変換すれば開放することができるようになります。

15.8.6 浮動小数点演算

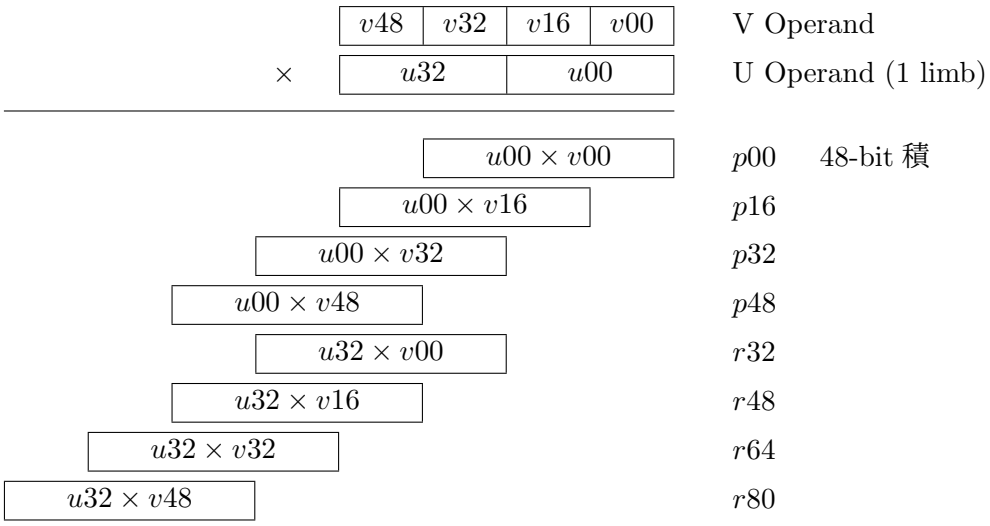
GMP における浮動小数点数の乗算は，性能の良くない整数乗算を使って実装されています。この際役立つのが 64bit マシンにおける`mpn_mul_1`, `mpn_addmul_1`，`mpn_submul_1`関数です。`mpn_mul_basecase`関数は 32bit, 64bit マシンの両方で活用されています。

IEEE 53bit の倍精度演算を使うと，整数乗算は高々53bit までしか正しい結果が得られません。64×64 乗算を，8 個の 16×32 → 48 bit に分割すると簡便になります。6 個の 21×32 → 53 bit 乗算に分割することもできますが，この場合は最小の 21bit 分については符号ビットだけ使用することになります。

64bit マシンにおける`mpn_mul_1`ファミリーは，処理開始時に一つのリムを 3 または 4 つに分割します。ループの内部では，多倍長オペランドを 32bit 単位で分割します。これらの符号なし 32bit 整数に分割されたものを浮動小数点数に効率よく変換できるかどうかはマシンによってかなり違いが出ます。読み込んだデータを整数に分割し，ゼロでパディングして 64bit 長に変換したり，あるいはメモリを経由して浮動小数点数に変換したりすることも可能になっています。

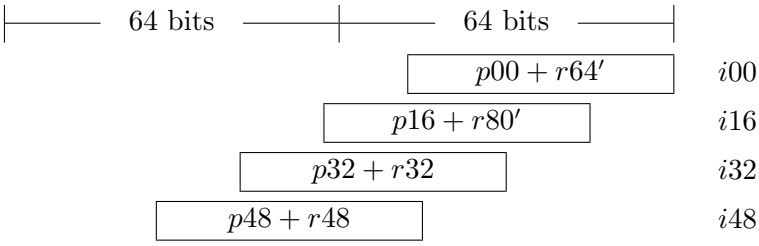
積の一部を 64bit リムに変換するのが，符号付きの変換としては最良です。全ての値が2^{53} より小さいのであれば，符号付きでも符号なしでも同じ処理になりますが，多くのプロセッサでは符号なしの変換機能は持っていません。

以下に示す図は，64bit リムを用いて`mpn_mul_1`や`mpn_addmul_1`関数を使って 16×32 bit 積を行う時のものです。ひとまとまりのリムオペランド V は 4 つの 16bit 長に分割されます。こうしてつくられる複数リムオペランド U はループの中で，二つの 32bit 部分を形成します。



$p32$ と$r32$ も，$p48$ and $r48$ も浮動小数点加算可能です。$p00$ と$p16$ は，前の繰り返しで生成された$r64$ と$r80$ に加えられます。

それぞれのループで，4 つの 49bit 長の結果を整数に変換し，下記のように並べられます。



次に行うべきことは，これらを効率的に加算しつつ，桁上がりリムも加え，低位の 64bit リムの結果と，高位の 33bit 桁上がりリムを生成していくかということです($i48$ は 33bit 分，高い桁数の方向に拡張されます)

15.8.7 SIMD 命令

複数データに対する単一命令(SIMD, Single-Instruction Multiple-Data)は現在の CPU では普通に使用できるもので，元々は信号処理アルゴリズムで，各データポイントを複数独立に扱うためのものです。GMP で発生するキャリー処理にはあまり向いていません。

所謂，4 つの 16×16 bit の乗算を行う SIMD 乗算は，GMP の側から見れば，いくつかのシフトと加算が伴うことを除けば，32×32 乗算を行う手間と同じになります。とはいえもちろん，SIMD 形式がフルでパイプライン化されていて，命令デコードを減らすことができるのであれば，十分価値は出てきます。

x86 CPU 上では，MMX 命令が`mpn_rshift`関数と`mpn_lshift`関数で使われており，更に P55 に対する`mpn_mul_1`関数における 16 ビット乗算の特別な場合にも使用されています。SSE2 は Pentium 4 用の`mpn_mul_1`, `mpn_addmul_1`, `mpn_submul_1`関数で使用されています。

15.8.8 ソフトウェアパイプライン

ソフトウェアパイプラインは，ループ内の分岐点周りのスケジューリングのための命令から構成されています。例えば，ループ内で，今の反復ではなく，次の反復でロード命令を発行する予定がある場合，メモリからデータを届けるための余分なサイクルを発生させたりできます。

当然，これが望まれるのは，終了までに数サイクル要するロード命令や乗算を実行したり，CPU が複数の関数ユニットを持ち，他の仕事をついでに実行できるような環境であったりした場合に限られます。

複数ステージを持つパイプラインは，実行中も各ステージでデータ値を持ち，ループの反復中もステージに沿ってデータを移動します。まるでお手玉のように。

命令のレイテンシがループの時間より長いのであれば，ループを展開する必要が出てくるでしょう。その結果，他の（多数の場合もあり）レジスタが実行中であっても，一つのレジスタがいつでも使える結果を持つことができるようになります(see Section 15.8.9 [Assembly Loop Unrolling], p. 119).

15.8.9 ループ展開

ループ展開は，複数のリムを各ループで処理するコードを複製することでできるようになります。ループのオーバーヘッドを展開した分だけ減らすことができるは当然として，他にも例えばレジスタを他と組み合わせて使うことで，もっと有効活用できるようになることも期待できます。

展開する分量は，1 回回るごとにN 減らされるループカウンタを用いて制御します。停止するのは，残りのループカウンタがN より小さくなった時です。もしくは，開始時にN を差し引き，停止条件はループカウンタC がゼロより小さくなった時とします（残っているリムの数は$C + N$ になります）。

別のやり方としては，2 のべき乗の場合は，ループカウンタや残り数はシフトとマスクの処理だけで計算することもできます。大規模なループの真ん中にジャンプする場合にも便利に使えます。

リムは多重ループ展開ではないので，様々なやり方で制御できます。例えば下記のようにします。

- 単純ループの最後（もしくは最初）で，多めに処理してしまう。展開部分の処理より遅くならないように心がけてやってください。
- 2 進テストの集合，例えば 8 リム展開の後，4 以上のリムをテストとして，更に 2 回以上行い，最後に 1 回以上行う。これは単純ループよりコード量が必要になります。
- `switch`文を使用して，起こりうるケースに対して別々のコードを書く。例えば 8 リムのループ展開に対しては，あまりなし，1 あまり，・・・，7 あまり，まで対応したコードを書く。大量のコードが必要になりますが，深くパイプライン化したループを組み合わせたものに対しては，全てのケースに対して最適化を行える最良の方法です。
- ループの中ほどに計画的にジャンプを入れておくと，最初の反復であまり余計な仕事をしないようにできます。これを行うのであれば，徐々にサイズを大きくしていきます。魅力的な方法ですが，ジャンプの準備と，ポインタの位置合わせをトリッキーに行うことになり，深いパイプラインの組み合わせに対しては難しくなります。

15.8.10 アセンブラコードの書き方

ここではアセンブラで，リム配列の処理を行うソフトウェアパイプラインループの書き方を案内します。

まずは，アルゴリズムと必要となる命令を決め，展開もスケジューリングもしないコードを書き，きちんと動作することを確認します。3 オペランド命令が使える CPU では，新しい値一つ一つに対して新しいレジスタに書き込むようにしておくと，後に続くステップが簡単にできます。

この時，各命令に対して関数ユニットが対応したり，I/O ポート要求を発行したりすることを覚えておいて下さい。ある命令が 2 つのユニット，例えば U0 もしくは U1 の片方を使うことができるときは“U0/U1”というカテゴリーを作ります。それぞれのユニット（あるいか統合したユニット）を使ってそのトータルを数え上げ，しかる後，全ての命令を数え上げます。

これらのトータル数からベストなループ時間を導きます。最終的には，完全に命令レイテンシを隠せる完ぺきなスケジューリングを目指すことになります。トータルの命令数はその限界を示すものとなるか，あるいは特定の関数ユニットになります。命令をいじくることで，その限界を軽くすることができるかもしれません。

ループ回数をN とすると，命令の束をN 回発行し，最終ループに終了のためのループ分岐を持たせます。この命令の束として，望ましい関数ユニットを用いてダミーの命令で埋めてみましょう。これを動かすことで，到達可能な速度が確認できるようになります。

さて，高速だが意味のないダミーの命令を実際の命令に置き換え，低速だが正しいループにして動かしてみましょう。最初はまずロード命令から始まるのが普通で，次にロードした値を使った命令に置き換えていきます。このループをもう一度動かし，目標とするスピードに達しているかどうかをチェックします。

置き換えた命令はそのままにして，ループの計測を頻繁に行うようにします。何度目か繰り返した後，ループ内の最後から最初まで書き換えを完了させます。新しい値には新しいレジスタを使うという戦略を使うと，レジスタの衝突は起こらなくなります。これをしないのであれば，一度使用したレジスタをぶち壊すことがないよう注意して下さい。一度使用したレジスタを書き換えるのはエラーの元になりかねません。

この書き変えたループは，もとのループを 1，2 回重ねて実行し，ベクトルの一要素の計算を新しいループの最初の 1 回目開始し，更に数回以上繰り返して完結させて下さい。

最終段階では，次々に実行されて終了する（feed-in and wind-down）コードをループ用に生成します。そのためには，スタート時にループのコピーを行い，有効な先行命令を持たない命令群を一切削除します。そして，最後では，望ましくない結果（一切のロード命令も含む）をもたらす命令群を削除します。

ループはロードして処理する最小のリム数を保持するようにしますので，次々に実行される(feed-in)コードは要求サイズを小さくし，終了処理(wind-down)のための部分，もしくは小さいサイズのための特別なコードは飛ばして，テストしなければなりません。

16 GMP の内部構造

この章で扱っている GMP の内部構造についての記述はあくまで情報提供のためのもので，将来変更される可能性があります。互換性を重視するのであれば，これ以前に解説してある関数やマクロを使うようにして下さい。

16.1 整数型の内部構造

mpz_t型の変数は，符号付き整数を表現するためのもので，動的にメモリ割り当て，再割り当てができます。この変数は次のような構成になっています。

_mp_size　リムの数を表わします。負の整数の場合はこの値もマイナスになります。ゼロの場合は，_mp_sizeもゼロになり，_mp_dの値は参照されません。

_mp_d　割り当て済みのリムの配列へのポインタです。リム配列は，mpn型同様に「リトルエンディアン」形式で格納されていますので，_mp_d[0]に最小桁にあたるリムが入り，_mp_d[ABS(_mp_size)-1]に最大桁が格納されます。_mp_sizeが非ゼロであれば，格納してある整数の最大桁も非ゼロになります。

現時点では常に 1 リムは確保するように実装されていますので，例えばmpz_set_ui関数が再割り当てをするようなことはありませんし，mpz_get_ui関数は必ず_mp_d[0]を参照(fetch)します(_mp_sizeが非ゼロであれば必要になるので)。

_mp_alloc

_mp_allocは現時点で_mp_dに割り当てられているリムの数を意味します。従って，当然_mp_alloc >= ABS(_mp_size)となります。mpzルーチンが_mp_sizeを増やそうとしている（もしくは増やす可能性のある）時には_mp_allocをチェックして，十分な領域があるかどうかを確認し，足りなければ再割り当てを行います。この時にはMPZ_REALLOC関数が使用されます。

様々な論理ビット演算を行う関数，例えばmpz_andなどは，負の値は 2 の補数であるとして扱います。しかし，符号も絶対値も常に参照され，必要な値の変更は計算中に実行されます。不都合なこともままありますが，符号と絶対値は他の関数にとっては最高に便利なツールなのです。

MPZ_TMP_INIT関数と共にいくつかの内部一時変数が確保されます。これはメモリ割り当て関数ではなくTMP_ALLOC関数を使って確保され，_mp_d分の領域が確保されます。再割り当てが不要な程度に十分な領域が確保されます(予測不能な事態が起きないようにするため)。

_mp_sizeと_mp_allocはint型，対して，mp_size_tは常にlong型になります。ある種の 64bit システム上では 32bit 長となるので，十分な幅を確保しつつ，数バイトはデータ領域を節約できます。

16.2 有理数型の内部構造

mpq_t型は，mpz_t型の整数を分子と分母に持つ有理数を表現するためのものです(see Section 16.1 [Integer Internals], p. 121)。

標準形としては，分母が正かつ非ゼロの整数の既約分数となります。ゼロは 0/1 という形で表現します。

計算過程の各ステージごとに公約数を除する方法が，一般的には最良であるとされています。GCD(最大公約数)の計算は$O(N^2)$ の演算が必要になるため，小さい公約数から削っていき，大きい公約数は後から除する方法を取っています。分母と分子に公約数がないということが分かれば，例えばmpq_mul関数においては，4 組ではなく，2 組の GCD だけ分かればよいことになります。

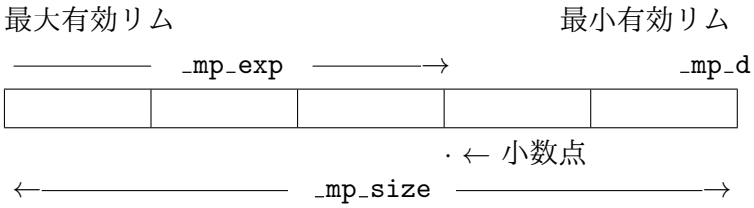
このような公約数の扱い方をすると，単純な因数分解においては必ずしも最適なものではなく，約せるかどうかの見通しも立ちづらいことになりますが，GMP はそれ以外の手段が取れませ

ん。Section 3.11 [Efficiency], p. 23でも述べましたが，この点はアプリケーションソフトの方でうまく対処して下さい。mpq_t演算のフレームワークは，mpq_numref関数とmpq_denref関数を使って直接分子と分母にアクセスするか，mpz_t変数を直接処理することを推奨しています。

16.3 浮動小数点型の内部構造

GMP の浮動小数点演算は，全てのリムを活用し，丸め処理を簡素にすることで高速な計算を実現することを目的としたものです。

mpf_t型の浮動小数点数は，可変桁数の仮数部と，符号付きの単一マシンワードの指数部を持ち，仮数部は符号と絶対値で構成されています。



各フィールドは以下の通りです。

_mp_size 現時点で使用しているリム数。これが負の時は保持している値が負であることを表わしています。値がゼロの場合は_mp_sizeと_mp_expもゼロが代入され，_mp_dの値は使用されません(未来のバージョンではゼロの場合，_mp_expの値も未定義になるかもしれません)。

_mp_prec 仮数部のリム単位の精度桁数。計算の種類に寄らず，計算結果の_mp_prec個のリムを生成するためのものです（最大桁を格納するリムは必ず非ゼロにする)。

_mp_d 仮数部の絶対値を格納しているリム配列へのポインタ。mpn関数が処理している通り，「リトルエンディアン」形式になっていますので，_mp_d[0]に最小桁を，_mp_d[ABS(_mp_size)-1]に最大桁を格納しています。

最大桁リムは常に非ゼロとなりますが，それ以外の縛りはないので，最大桁に当たる 1bit がこのリム内のどの位置に置かれるかどうかは常に変化します。

_mp_prec+1個のリムは_mp_dが示す位置に置かれており，余分な 1 リムは便利さのために置かれています（詳細は下記参照)。計算中にメモリの再確保が起こらないよう，精度桁数の変更はmpf_set_prec関数を使わないとできないようになっています。

_mp_exp リム配列内に置かれる指数部で，小数点の位置を規定しています。この値がゼロの時は，小数点が最大桁のすぐ上にあることを意味し，正の値の時はその値の分，最小桁寄りに位置していることを意味します。従って，例えばこの値が≥ 1であれば，上記の図に示したような小数点の位置になります。負数の時は，最大桁リムよりずっと上に小数点が位置していることになります。

指数部はどんな値にもなりうるわけで，必ずしも上記の図のようにリム配列内に小数点が位置するとは限りません。上の方にも下の方も外れた位置に来ることもあり得ます。{_mp_d,_mp_size}データに含まれるもの以外のリムはゼロとして扱われます。

_mp_size と_mp_prec はint型ですが，mp_size_t や_mp_expは常にlong型です。こうすることである種の 64bit システム上では 32bits フィールドを抱え込むことになりますが，結果として数バイト分のメモリが節約でき，大きな精度桁数と極めて広範囲の浮動小数点数を扱うことができるようになります。

下記の事項についても留意しておいて下さい。

下位桁のゼロ
: 下位桁リムである_mp_d[0]はゼロになることがあり，大概無視されるものとして扱われます。計算効率アップのために下位桁ゼロを確認したりするルーチンもありますが，大概はスルーします。

仮数部のサイズ範囲
: _mp_sizeはリムの数を表わしますが，その値を格納するためにより少ない桁数で済むのであれば，_mp_precより小さくなることもあります。つまり，低精度の値や桁数の小さい整数を高精度のmpf_tに格納すると，計算効率がアップします。

  逆に，_mp_sizeが_mp_precより大きくなることもあり得ます。_mp_d用に用意された_mp_prec+1個のリム全てを使う値であり，mpf_set_prec_raw関数が_mp_precの値を小さく抑えてしまうと，_mp_sizeの値は変化せず，_mp_precよりいくらでも大きくなり得ます。

丸め処理
: 丸め処理は必ずリムの境界で行われます。非ゼロの_mp_prec個のリムを使って計算しても，最小精度しか要求しないアプリケーションであれば問題ありません。

  単純なゼロ方向への「打ち切り(trunc)」しか行いませんので効率的です。余計なリムも，繰り上げも繰り下げも必要ありませんから。

ビットシフト
: 指数部はリムの中にありますので，mpf_add関数やmpf_mul関数のような基本演算ではビットシフトは起こりません。互いの指数部が異なっている場合は，小数点をずらす処理を行うだけで済みます。

  当然，mpf_mul_2exp関数やmpf_div_2exp関数を使うときにはビットシフトが必要になりますが，それは指数部や仮数部でシフトが必要になった時に限られます。

_mp_prec+1個のリムの使用
: _mp_dを格納する余分なリム(_mp_prec+1個目のリムのこと)は，mpfルーチンが桁上がりを必要とする時に使用されます。例えばmpf_add関数が_mp_prec個のリムに対してmpn_add関数を実行する時などです。桁上がりが起きなければその結果はそのまま格納されますが，起きてしまえば余分なリムにも格納され，_mp_sizeの値は_mp_prec+1になります。

  _mp_prec+1個のリムが使われる時には，その下位桁リムは拡張された精度分を補てんするためには使われませんので，上位の_mp_prec個のリムだけが使用されます。しかし，その下位桁リムをゼロにしたり，繰り下げたりする必要はありません。後に続く処理では，必要な上位桁分だけが使われ，桁数が同じであれば_mp_prec桁の結果を得ることになります。そうでなければ，元々の値と精度が異なってしまいます。

  値をコピーするmpf_set関数等は_mp_prec+1リム全部をコピーします。こうすることで，_mp_sizeが_mp_prec+1であるような値でも完全に正確な値を保持することができます。アプリケーションが要求する精度が_mp_precであればそこまでする必要はないのですが，mpf_set関数は正確な値のコピーを作るべきものであると考えているからです。

アプリケーションの精度
: __GMPF_BITS_TO_PRECは精度が必要なアプリケーションを_mp_prec桁にコンバートします。ビットで表わされた値は丸められたリム全体に格納されると，余分なリムが加えられます。というのも_mp_dの最上位リムは非ゼロであり，たとえ 1 ビット分であっても残る可能性があるからです。

  __GMPF_PREC_TO_BITSはリバースコンバージョンを行い，コンバートする前に余分なリムをカットします。mpf_get_prec関数で読み取ると，その影響で，桁数がmp_bits_per_limbの倍数に丸められてしまいます。

  非ゼロの余分なリムを付け加えるということは，_mp_dに余分のリムを確保して追加する，ということです。例えば，32 ビット長のリムを使う場合，アプリケーションが 250 ビット要求すると，それは 8 リムに丸められますので，余分なリムが

付加され，_mp_precは 9 リムになります。従って_mp_dは 10 リム割り当てられます。mpf_get_prec関数で読み取ると，_mp_precマイナス 1 リムとなり，これに 32 をかけて 256 ビットという値になります。

厳密に言うと，1 ビットでも 1 リム必要になるということは，32 ビット長の 3 リムはせいぜい 65 ビットしか精度を保持しないということになります。しかし，mpf_t型の目的を鑑みると，64bit で十分と考えられます。

16.4 単純出力形式の内部構造

mpz_out_rawは次のようなフォーマットを使います。

サイズ	データバイト列

最初の「サイズ」部は 4 バイトの大きさで，その後ろに「データバイト列」が続きます。サイズ部が 2 の補数で表現された負数の時は，負の整数という意味になります。データバイト列の部分は整数の絶対値で，最大桁にあたる部分がデータバイト列の先頭に来ることになります。

最大桁にあたるデータバイト列は常に非ゼロとなりますので，リムサイズと無関係に，どんな計算機システム上でも同じ出力結果となります。

GMP Version 1 では，リムサイズ分だけゼロとなるバイト値を詰め込んでいました。mpz_inp_rawでも互換性のため，そのようにしています。

サイズ部分とデータバイト列を「ビッグエンディアン」として表現するのは慎重にお願いします。というのも，16 進ダンプファイルとして読み取る分にはいいのですが，リムの並びは逆から読み書きしなければなりません。ということで，ビッグエンディアン環境でもリトルエンディアン環境でも，_mp_dだけを読み書きすることはできません。

16.5 C++インターフェースの内部構造

演算子定義のテンプレートシステムは，例えばa=b+cという式を使うと自動的にmpz_add関数を呼び出す，ということを保証するためのものです。mpf_classクラスでは，例えばf=w*x+y*zという式を計算するための内部一時変数に対しても，最終的な演算結果と同じ桁数の計算を行うようにしており，単純な実装では提供していない重要な機能と言えるでしょう。

実装は式とは異なるデータタイプを返してきたりしてややこしいので，以下ではこの仕組みを簡単に説明します。

加算などの演算を実行するには，まず関数オブジェクト(function object)を定義します。

```
struct __gmp_binary_plus
{
  static void eval(mpf_t f, const mpf_t g, const mpf_t h)
  {
    mpf_add(f, g, h);
  }
};
```

そして下記のように，加算表現オブジェクト(additive expression object)を生成します。

```
__gmp_expr<__gmp_binary_expr<mpf_class, mpf_class, __gmp_binary_plus> >
operator+(const mpf_class &f, const mpf_class &g)
{
  return __gmp_expr
    <__gmp_binary_expr<mpf_class, mpf_class, __gmp_binary_plus> >(f, g);
}
```

一見冗長な__gmp_expr<__gmp_binary_expr<...>>という表現ですが，一つのテンプレート型に全ての可能な式表現をカプセル化することができます。実際，他のmpf_class等のクラスでも，__gmp_exprテンプレートのデータ型独自の特殊化を行っています。

次に，__gmp_exprに対してmpf_classが使えるように代入を定義します。

```
template <class T>
mpf_class & mpf_class::operator=(const __gmp_expr<T> &expr)
{
  expr.eval(this->get_mpf_t(), this->precision());
  return *this;
}

template <class Op>
void __gmp_expr<__gmp_binary_expr<mpf_class, mpf_class, Op> >::eval
(mpf_t f, mp_bitcnt_t precision)
{
  Op::eval(f, expr.val1.get_mpf_t(), expr.val2.get_mpf_t());
}
```

ここでexpr.val1とexpr.val2は式表現中の演算子に対して参照（リファレンス)を提供しています(ここで，exprは__gmp_exprに格納されている__gmp_binary_exprのことです)。

このようにして，式表現は割当てられた時にだけ実際に計算され，そこで必要となる精度(fに定義されている)も分かります。更に，計算ターゲットのmpf_tデータも使えるようになります。これでようやくmpf_add関数を直接fを使って実行できるようになります。

より複雑な式の場合は，細かい式に分割して扱うようにします。例えば下記の式は次のようになります。

```
template <class T, class U>
__gmp_expr
<__gmp_binary_expr<__gmp_expr<T>, __gmp_expr<U>, __gmp_binary_plus> >
operator+(const __gmp_expr<T> &expr1, const __gmp_expr<U> &expr2)
{
  return __gmp_expr
    <__gmp_binary_expr<__gmp_expr<T>, __gmp_expr<U>, __gmp_binary_plus> >
    (expr1, expr2);
}
```

対応するfの特殊化は次のようになります。

```
template <class T, class U, class Op>
void __gmp_expr
<__gmp_binary_expr<__gmp_expr<T>, __gmp_expr<U>, Op> >::eval
(mpf_t f, mp_bitcnt_t precision)
{
  // declare two temporaries
  mpf_class temp1(expr.val1, precision), temp2(expr.val2, precision);
  Op::eval(f, temp1.get_mpf_t(), temp2.get_mpf_t());
}
```

従って，式表現は再帰的に評価され，全ての分割された式表現はfの桁数と同じ精度で計算されます。

付録 A GMP ライブラリ制作者たち

Torbjörn Granlund は最初の GMP ライブラリを作り上げ，現在もメイン開発者として頑張っています。特に開発者を明記していないコードは全て Torbjörn が書いたものです。GMP 開発にあたっては個人開発者や複数の組織が貢献を行ってきましたので，以下，時系列順に記しておきます。

Gunnar Sjödin と Hans Riesel は GMP の初期バージョンの開発に際して直面した数学的諸問題解決に助力を行ってくれました。

Richard Stallman は GMP のインターフェースデザインに関して助言を行い，本マニュアルの初期バージョンを改良してくれました。

Brian Beuning と Doug Lea は GMP の初期バージョンのテストを手伝い，クリエイティブな助言をしてくれました。

カナダにあるヨーク大学の John Amanatides は`mpz_probab_prime_p`関数を提供してくれました。

Paul Zimmermann は REDC ベースの mpz_powm コード, Schönhage-Strassen FFT 乗算コード，Karatsuba 平方根コードを開発しました。また，GMP 4.2 の Toom3 コードを改良してくれました。また，GMP 2 の開発を後押しし，多倍長パッケージとの比較を行ってくれました。ECMNET project を組織し，GMP 3 の最適化に関して協力にバックアップしてくれました。加えて，Torbjörn と共同で GMP 4.3 の n 乗根コードを新しく開発しました。

Ken Weber (ケント州立大学, Universidade Federal do Rio Grande do Sul) は，現在では廃止されたバージョンの`mpz_gcd`, `mpz_divexact`, `mpn_gcd`, `mpn_bdivmod`の開発に貢献しました。これらは一部，CNPq (Brazil) grant 301314194-2 の援助によってなされたものです。

Cygnus Support の Per Bothner は GMP の Cygnus 用の設定を手伝ってくれました。また，価値ある助言と，中間リリースに際しては大量のテストを行ってくれました。

Joachim Hollman は GMP Version 2 の`mpf`関数群と，`mpz`関数群の再設計を行いました。

Bennet Yee は`mpz_jacobi` と`mpz_legendre`の初期バージョン開発に際して貢献を行いました。

Andreas Schwab は`mpn/m68k/lshift.S` と`mpn/m68k/rshift.S` (現在は`.asm`ファイルに変更)の開発に際して貢献を行いました。

Robert Harley(Inria, France と David Seal of ARM, England)は 2 進数内のビット数カウント方法の改良法を示唆してくれました。また，GMP 3 における Karatsuba 乗算や 3-way Toom-Cook 乗算関数の最適化も行ってくれました。ARM 用アセンブラコードの開発にも貢献しています。

ストックホルム大学数学科の Torsten Ekedahl は，GMP の開発の節目節目で重要な示唆を行ってくれました。彼の数学的専門性に基づく助言によっていくつかのアルゴリズムが改良できました。

Linus Nordberg は，autoconf を用いた新しい設定システムを構築し，新しい乱数関数を開発しました。

Kevin Ryde の働きは多岐に渡っています。列挙すると，x86 コードの最適化, m4 アセンブラマクロ，パラメータのチューニング, 速度測定, 設定システム, インライン関数化，除算可能性テスト，ビットスキャン方法，Jacobi 記号計算，Fibonacci 数列関数と Lucas 数関数, printf 関数と scanf 関数の開発，perl インターフェース，デモ用数式パーサ，本マニュアルにおけるアルゴリズムの章執筆, `gmpasm-mode.el`の開発，そしてそれ以外の雑多な改良一切合切，です。

Kent Boortz は Mac OS 9 に移植してくれました。

Steve Root は，最適化した Alpha 21264 のアセンブラコードの開発を手伝ってくれました。

Gerardo Ballabio は`gmpxx.h` C++ クラスインターフェースと，C++ `istream` 入力ルーチンを開発しました。

Jason Moxham は`mpz_fac_ui`を書き直してくれました。

Pedro Gimeno は Mersenne Twister を実装し，その他の乱数生成に関しても改良を行ってくれました。

Niels Möller は準 2 乗 GCD, 拡張 GCD と jacobi 記号関数コード, 2 乗 Hensel 除算コード, GMP 4.3 の新しい分割統治除算コード(Torbjörn と共同)の開発を行いました。更に，GMP 4.3 における新しい Toom-Cook 乗算コードの開発に助力し，GMP 5.0 における Toom-Cook アルゴリズムにおける値評価を簡略化するヘルパー関数を実装しました。mpn_mulmod_bnm1 関数も書き起こしています。また，GMP 簡易利用のための mini-gmp パッケージのメイン開発者でもあります。

Alberto Zanoni と Marco Bodrato は不均衡乗算の方法について助言を行ってくれました。また，Toom-Cook 乗算における最適な値評価と補間方法を発見してくれました。

Marco Bodrato は GMP 4.3 における新しい Toom-Cook 乗算のコード開発に助力を行い，GMP 5.0 の新しい Toom-Cook 乗算や 2 乗コードの開発を行ってくれました。現在でも使用されている mpn_mulmod_bnm1 関数，mpn_mullo_n 関数，mpn_sqrlo 関数のメイン作成者でもあります。また，mpn_invert 関数や mpn_invertappr 関数も開発し，整数平方根の性能を向上させました。現在でも使用されている組み合わせ関数である，2 項係数，階数，多段階数，素数階乗計算の著者でもあります。

David Harvey は内部関数である`mpn_bdiv_dbm1`の開発に助言を行い，Toom-Cook 乗算に関連する除算を実装してくれました。また，アセンブラコードを高速化し，特に AMD64 用の`mpn_mul_basecase` 関数を高速化しました。内部乗算関数である`mpn_mulmid_basecase`, `mpn_toom42_mulmid`, `mpn_mulmid_n`や関連するヘルパールーチンも開発しました。

Martin Boij は`mpn_perfect_power_p`を開発しました。

Marc Glisse は`gmpxx.h`を改良してくれました。列挙すると，一時利用メモリ領域の削減（高速化に寄与), `numeric_limits` と`common_type`の特殊化, C++11 の利用(移動コンストラクタ，明示的ブール型変換, UDL), `mpq_class`から`mpz_class`への変換の明示化，引数が小さいコンパイル時定数である時の演算子の最適化, ヒープ割り当てをスタック割り当てに置き換え，です。また，C++ストリームにおける eof ビット操作のバグ取りや，`mpq/aors.c`における不要な除算除去を行ってくれました。

David S Miller は SPARC T3 と T4 のアセンブラコードを開発しました。

Mark Sofroniou は mul_fft.c のデータ型を整理し，超巨大なオペランドにも対応できるようにしてくれました。

Ulrich Weigand は GMP を powerpc64le ABI に移植してくれました。

(このリストは時系列順になっており，貢献の重要度によるものではありません。リストに掲載されていない貢献者がいた場合は，その旨`gmp-devel@gmplib.org`にメールして下さい。)

GNU MP 2 の浮動小数点演算関数の開発に当たっては，ESPRIT-BRA (Basic Research Activities) 6846 project POSSO (POlynomial System SOlving)の援助を受けました。

GMP 2, 3, 4.0 の開発に当たっては，IDA Center for Computing Sciences の援助を受けました。

GMP 4.3, 5.0, 5.1 の開発に当たっては，スウェーデン戦略研究財団の援助を受けました。

Hans Thorsen は GMP のテスト環境として SGI システムを提供してくれました。感謝致します。

付録 B 参考文献

B.1 書籍

- Jonathan M. Borwein and Peter B. Borwein, “Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity”, Wiley, 1998.
- Richard Crandall and Carl Pomerance, “Prime Numbers: A Computational Perspective”, 2nd edition, Springer-Verlag, 2005.
  http://www.math.dartmouth.edu/~carlp/
- Henri Cohen, “A Course in Computational Algebraic Number Theory”, Graduate Texts in Mathematics number 138, Springer-Verlag, 1993.
  http://www.math.u-bordeaux.fr/~cohen/
- Donald E. Knuth, “The Art of Computer Programming”, volume 2, “Seminumerical Algorithms”, 3rd edition, Addison-Wesley, 1998.
  http://www-cs-faculty.stanford.edu/~knuth/taocp.html
- John D. Lipson, “Elements of Algebra and Algebraic Computing”, The Benjamin Cummings Publishing Company Inc, 1981.
- Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, “Handbook of Applied Cryptography”, http://www.cacr.math.uwaterloo.ca/hac/
- Richard M. Stallman and the GCC Developer Community, “Using the GNU Compiler Collection”, Free Software Foundation, 2008, available online https://gcc.gnu.org/onlinedocs/, and in the GCC package https://ftp.gnu.org/gnu/gcc/

B.2 論文

- Yves Bertot, Nicolas Magaud and Paul Zimmermann, “A Proof of GMP Square Root”, Journal of Automated Reasoning, volume 29, 2002, pp. 225-252. Also available online as INRIA Research Report 4475, June 2002, http://hal.inria.fr/docs/00/07/21/13/PDF/RR-4475.pdf
- Christoph Burnikel and Joachim Ziegler, “Fast Recursive Division”, Max-Planck-Institut fuer Informatik Research Report MPI-I-98-1-022,
  http://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1998-1-022
- Torbjörn Granlund and Peter L. Montgomery, “Division by Invariant Integers using Multiplication”, in Proceedings of the SIGPLAN PLDI’94 Conference, June 1994. Also available https://gmplib.org/~tege/divcnst-pldi94.pdf.
- Niels Möller and Torbjörn Granlund, “Improved division by invariant integers”, IEEE Transactions on Computers, 11 June 2010. https://gmplib.org/~tege/division-paper.pdf
- Torbjörn Granlund and Niels Möller, “Division of integers large and small”, to appear.
- Tudor Jebelean, “An algorithm for exact division”, Journal of Symbolic Computation, volume 15, 1993, pp. 169-180. Research report version available
  ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-35.ps.gz
- Tudor Jebelean, “Exact Division with Karatsuba Complexity - Extended Abstract”, RISC-Linz technical report 96-31,
  ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1996/96-31.ps.gz
- Tudor Jebelean, “Practical Integer Division with Karatsuba Complexity”, ISSAC 97, pp. 339-341. Technical report available
  ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1996/96-29.ps.gz

- Tudor Jebelean, ‘‘A Generalization of the Binary GCD Algorithm”, ISSAC 93, pp. 111-116. Technical report version available
  `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1993/93-01.ps.gz`
- Tudor Jebelean, ‘‘A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers”, Journal of Symbolic Computation, volume 19, 1995, pp. 145-157. Technical report version also available
  `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-69.ps.gz`
- Werner Krandick and Tudor Jebelean, ‘‘Bidirectional Exact Integer Division”, Journal of Symbolic Computation, volume 21, 1996, pp. 441-455. Early technical report version also available `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1994/94-50.ps.gz`
- Makoto Matsumoto and Takuji Nishimura, ‘‘Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”, ACM Transactions on Modelling and Computer Simulation, volume 8, January 1998, pp. 3-30. Available online
  `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.ps.gz` (or .pdf)
- R. Moenck and A. Borodin, ‘‘Fast Modular Transforms via Division”, Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory, October 1972, pp. 90-96. Reprinted as ‘‘Fast Modular Transforms”, Journal of Computer and System Sciences, volume 8, number 3, June 1974, pp. 366-386.
- Niels Möller, ‘‘On Schönhage’s algorithm and subquadratic integer GCD computation”, in Mathematics of Computation, volume 77, January 2008, pp. 589-607.
- Peter L. Montgomery, ‘‘Modular Multiplication Without Trial Division”, in Mathematics of Computation, volume 44, number 170, April 1985.
- Arnold Schönhage and Volker Strassen, ‘‘Schnelle Multiplikation grosser Zahlen”, Computing 7, 1971, pp. 281-292.
- Kenneth Weber, ‘‘The accelerated integer GCD algorithm”, ACM Transactions on Mathematical Software, volume 21, number 1, March 1995, pp. 111-122.
- Paul Zimmermann, ‘‘Karatsuba Square Root”, INRIA Research Report 3805, November 1999, `http://hal.inria.fr/inria-00072854/PDF/RR-3805.pdf`
- Paul Zimmermann, ‘‘A Proof of GMP Fast Division and Square Root Implementations”,
  `http://www.loria.fr/~zimmerma/papers/proof-div-sqrt.ps.gz`
- Dan Zuras, ‘‘On Squaring and Multiplying Large Integers”, ARITH-11: IEEE Symposium on Computer Arithmetic, 1993, pp. 260 to 271. Reprinted as ‘‘More on Multiplying and Squaring Large Integers”, IEEE Transactions on Computers, volume 43, number 8, August 1994, pp. 899-908.

付録 C GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both

covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled ‘‘History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled ‘‘History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its

Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the ‘‘History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled ‘‘Acknowledgements” or ‘‘Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled ‘‘Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled ‘‘Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled ‘‘Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties---for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled ‘‘History” in the various original documents, forming one section Entitled ‘‘History”; likewise combine any sections Entitled

‘‘Acknowledgements”, and any sections Entitled ‘‘Dedications”. You must delete all sections Entitled ‘‘Endorsements.”

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ‘‘aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled ‘‘Acknowledgements”, ‘‘Dedications”, or ‘‘History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been

terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ‘‘or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    ‘‘Massive Multiauthor Collaboration Site” (or ‘‘MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A ‘‘Massive Multiauthor Collaboration” (or ‘‘MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

    ‘‘CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    ‘‘Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is ‘‘eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Function and Type Index

O

S

T