GNU MPFR

MPFR：高品質多倍長浮動小数点演算ライブラリ
Edition 4.0.2
January 2019

MPFR チーム
mpfr@inria.fr

このマニュアルは，MPFR(Multiple Precision Floating-point Reliable) ライブラリ Version 4.0.2 のインストール方法，及び，使用方法を記述したものです。

目次

MPFRの著作権について

GNU MPFR ライブラリ (MPFR と略す) はフリーです。フリーとは，万人が自由に使用でき，自由な基盤の上で，再配布も自由にできるという意味です。本ライブラリはパブリックドメインではありません。つまり，著作権で守られており，再配布には制限があります。しかしその制限は，良き協力者たる市民が行おうとする全ての行為を許容するように設計されています。許されないのは，あなたから入手したあらゆるバージョンの本ライブラリを，更に広く共有する行為を妨害することだけです。

特に，我々は次のことを確認したいと考えています。あなたは本ライブラリのコピーを手放す権利があり，欲しい時にソースコードなどを受け取り，新しいフリーなプログラムを作るために本ライブラリの改変を行ったり，一部を利用したりすることができる，ということを知っているのです。

万人が上記の権利を持っているということを知らしめるため，我々は，あなたが，他の人達からこれらの権利を剥奪することを禁止します。例えば，あなたが GNU MPFR ライブラリのコピーを配布するのであれば，それを受け取った人達にも，あなたが有する全ての権利を与えなければなりません。そして，受け取った人たちもまた，元のソースコードを入手できるということを確認しなければなりません。更に，あなたは彼らにそのことを周知しなくてはなりません。

また，我々自身を保護するため，GNU MPFR ライブラリは無保証であることを周知させなくてはなりません。本ライブラリを改良して放出した場合，我々は，それがもとの配布物とは別物であることを知って頂きたいと思います。そうすることで，改良の結果生じた諸問題によって我々の評判が貶められることはなくなります。

GNU MPFR ライブラリのライセンスの正確な著作権の条項については，ソースコードに付随する劣等 GPL(Lesser General Public License) に書いてあります。COPYING.LESSER ファイルを参照して下さい。

1 MPFRについて

MPFR はポータブルな C ライブラリで，任意精度の浮動小数点演算を行います。その基盤ライブラリとして GNU MP(GMP) を使用しています。MPFR の目的は，緻密な方針に基づいた浮動小数点数クラスを提供することにあり，下記に示すように，他の任意精度浮動小数点ソフトウェアとは異なる特徴があります。

- MPFR のコードはポータブルです。「ポータブル (portable)」とは，どんな演算でもその結果は，その計算環境のワードサイズ `mp_bits_per_limb` (現在のプロセッサ上では普通は 64 ビット) に依存せずに確定する，ということです。但し，忠実丸め方式（効率的かつ高精度な丸め方式，faithful rounding) については計算環境に影響される可能性があります。計算環境における丸めモードや丸める精度桁数には影響されません。
- ビット数で表現される精度桁数 (precision) は,「厳格に」各変数ごとに設定されます。低精度桁数の設定も可能です。
- MPFR は IEEE 754-1985 規格で定められている 4 つの丸めモードと，ゼロ離反 (away-from-zero) 丸めをサポートしつつ，基本演算，数学関数の機能を提供します。（一部の環境で使用できる）忠実丸め方式もサポートしていますが，その場合の演算結果はポータブルではありません。

MPFR を使用して 53 ビットの精度桁数を設定し，4 つの丸めモードのいずれかを使用した場合，四則演算と平方根を用いたハードウェアの倍精度 (double precision) 浮動小数点演算 (例えば C の `double`型を使い，ISO C99 標準規格の Annex F を厳守して実装した C コンパイラを使用し，`FP_CONTRACT`プラグマを `OFF`にして実行) による計算結果と全く同じものを，MPFR は再生できます。 但し，MPFR のデフォルトの指数部は倍精度に比して大幅に広い，非正規化数はサポートしていない（エミュレートは可能），という違いはあります。

本バージョンの MPFR は，GNU Lesser General Public License Version 3.0 以降の条項の元にリリースされています。MPFR にフリーでないプログラムをリンクすることも可能ですが，この非フリーのプログラムを配布する際には，リンク元のプログラムに MPFR のソースコードと，改良された MPFR ライブラリとリンクするための方法も一緒に配布する必要があります。

1.1 本マニュアルの使い方

まず Chapter 4 [MPFR Basics], 頁 7 は読んで下さい。MPFR のインストールを自力で行う必要がある場合は，Chapter 2 [Installing MPFR], 頁 3 にも目を通して下さい。MPFR ライブラリの使用時には Chapter 5 [MPFR Interface], 頁 14 が参考になるでしょう。

本章以降の記述は後々の参照用ですが，一通り目を通しておくと良いでしょう。

2 MPFRのインストール

MPFR ライブラリは，GNU/Linux ディストリビューションには予めインストールされていることが多くなりました。しかし，開発用に必要な mpfr.h等のファイルが入っていないことはままあります。MPFR がフルでインストールされているかどうかを確認するには，/usr/include に mpfr.h があるかどうかをチェックしたり，#include <mpfr.h> (mpfr.hの置き場所は色々です）と書いてある小さいプログラムをコンパイルしたりします。例えば，下記のプログラムをコンパイルしてみましょう。

```
#include <stdio.h>
#include <mpfr.h>
int main (void)
{
  printf ("MPFR library: %-12s\nMPFR header:  %s (based on %d.%d.%d)\n",
          mpfr_get_version (), MPFR_VERSION_STRING, MPFR_VERSION_MAJOR,
          MPFR_VERSION_MINOR, MPFR_VERSION_PATCHLEVEL);
  return 0;
}
```

上記のプログラムは，下記のようにしてコンパイルします。

```
cc -o version version.c -lmpfr -lgmp
```

さすれば，次のような記述から始まるエラーが出るかもしれません。

```
version.c:2:19: error: mpfr.h: No such file or directory
```

このエラーが出たら，MPFR は多分ちゃんとインストールされていません。もしこのプログラムが実行できれば，インストールされている MPFR のバージョン番号が表示される筈です。

MPFR がインストールされていなかったり，別バージョンの MPFR をインストールしたい時には，次の節に示す手順でやってみて下さい。

2.1 インストール方法

この節では，UNIX システムに MPFR をインストールするための手続きを説明します。詳細は INSTALLファイルをご覧下さい。

1. MPFR のビルドには，まず GNU MP(Version 5.0.0 以降) のインストールが必要になります。C コンパイラは不可欠で，GCC をお勧めしますが，普通に使えるコンパイラであれば問題ないでしょう。また UNIX 標準の ‘make’コマンドも必要ですし， 他にも UNIX 標準のユーティリティコマンド類が必要になります。
   以上の準備ができたら，MPFR のビルド用ディレクトリで，下記のコマンドを実行しましょう。
2. ‘./configure’
   これによって，ビルドの準備が完了し，貴方のシステムにふさわしいオプションがセットアップされます。インストールディレクトリ (デフォルトは/usr/local) や，スレッドのサポートなど，デフォルトとは異なるオプションを指定することも可能です。詳細については INSTALL ファイルや，‘./configure --help’ の出力結果をご覧下さい。エラーメッセージが出た時もこの辺りの記述を参照して下さい。
3. ‘make’
   このコマンドで MPFR をコンパイルし，ライブラリアーカイブファイルである libmpfr.aを生成します。大方の環境では，動的ライブラリ (dynamic library) も一緒に生成されます。
4. ‘make check’
   このコマンドは，MPFR のビルドが正常に行われたかどうかを確認します。確認に失敗した時は，どこでどのようにコケているかを tests/test-suite.logファイルで調べることがで

きます。このファイルの中身を自動的に出力したい時には，予め ‘VERBOSE’環境変数を 1 にセットしてから ‘make check’ を実行します。例えば次のように打ち込んで下さい。

‘VERBOSE=1 make check’

コケた時にはその要因が既知のものかどうかを知りたいと思うのが人情ですよね？(つか，調べて下さいな)。未知の要因によるものでしたら，MPFR メーリングリスト ‘mpfr@inria.fr’ に報告をお願いします。詳細は Chapter 3 [Reporting Bugs], 頁 6 をご覧下さい。

5. ‘make install’

これを実行することで，mpfr.hや mpf2mpfr.hといったヘッダファイルを /usr/local/includeディレクトリへ，ライブラリファイル (libmpfr.aや動的ライブラリファイル等) を/usr/local/libディレクトリへ， mpfr.infoファイルは/usr/local/share/infoディレクトリへ，その他のドキュメントファイルを /usr/local/share/doc/mpfrディレクトリへそれぞれコピーします。‘--prefix’オプションが configure実行時に指定されていれば，この prefix ディレクトリ (‘--prefix’に続く引数として与える) を/usr/localの代わりに使用します。

2.2 その他の ‘make’ オプション

以上で示したもの以外にも有用な make オプションがあります。

- ‘mpfr.info’もしくは ‘info’
  このMPFR マニュアルの info 形式ファイルの生成，もしくは更新。mpfr.infoができます。このファイルは MPFR アーカイブに含まれています。
- ‘mpfr.pdf’ もしくは ‘pdf’
  このマニュアルの PDF バージョンが生成されます。ファイルは mpfr.pdfです。
- ‘mpfr.dvi’ もしくは ‘dvi’
  このマニュアルの DVI バージョンの生成。ファイルは mpfr.dviです。
- ‘mpfr.ps’ もしくは ‘ps’
  このマニュアルの Postscript バージョンの生成。ファイルは mpfr.psです。
- ‘mpfr.html’ もしくは ‘html’
  このマニュアルの HTML バージョンの生成。複数ページが生成されて doc/mpfr.htmlディレクトリに格納されます。一つの HTML ファイルにまとめたい場合は，‘doc’ディレクトリに移動し，make コマンドではなく，‘makeinfo --html --no-split mpfr.texi’と打ち込んで下さい。
- ‘clean’
  全てのオブジェクトファイルとアーカイブファイルを消去します。設定用のファイル類はそのまま残ります。
- ‘distclean’
  配布ファイル以外，すべての生成ファイルを消去します。
- ‘uninstall’
  ‘make install’でインストールされたすべてのファイルを消去します。

2.3 ビルド時のトラブル

何かしらトラブった時には，バグだと騒ぎ立てる前に INSTALLファイル，特に,「問題が起こった時には」の節を熟読して下さい。MPFR に限らず，ユーザー側でヘンな設定を行っているためにトラブルが発生してしまうことがあります。MPFR サイトの FAQ https://www.mpfr.org/faq.htmlにも，トラブルについての記載があります。

トラブルの報告は MPFR のメーリングリスト ‘mpfr@inria.fr’宛にお願いします (Chapter 3 [Reporting Bugs], 頁 6 参照)。修正済みのバグは，MPFR 4.0.2 の Web ページ https://www.mpfr.org/mpfr-4.0.2/に掲載されています。

2.4 MPFR 最新バージョンの入手先

MPFR の最新バージョンは `https://ftp.gnu.org/gnu/mpfr/` もしくは `https://www.mpfr.org/`から入手できます。

3 バグ報告

MPFR ライブラリにバグを発見したと思ったら，まずは MPFR 4.0.2 の Web ページ `https://www.mpfr.org/mpfr-4.0.2/` と，FAQ `https://www.mpfr.org/faq.html`を見て下さい。多分，それは既知のバグで，それに関係したものを見つけたということかもしれません。バグ情報については，MPFR のメーリングリストのアーカイブ `https://sympa.inria.fr/sympa/arc/mpfr` にもあります。その上で，既知のものではないと判断したら，調査の上，報告して下さい。我々開発陣は，この MPFR ライブラリを作り上げ，貴方に提供している訳です。貴方が見つけたバグの報告をお願いしても罰は当たりませんよね？報告の際には，以下の事柄を守って下さい。

まず，我々にも再現できるようなバグの事例をきちんと報告して下さい。例えば，MPFR だけを使用した単独の小さいプログラムと，その実行方法を示して下さい。

どこがおかしいのかを説明することも必要です。クラッシュする，結果が正しくないということでしたら，どんな場合にどのようにすればそれが起きるのか，ということを示して下さい。

バグ報告には，コンパイラのバージョン番号も記載して下さい。これは‘`cc -V`’や，GCC の場合は‘`gcc -v`’で取得できます。また，‘`uname -a`’の出力結果と MPFR のバージョン番号 (GMP のバージョンが分かるのであればそれも) をお知らせ下さい。‘`make`’や‘`make check`’の実行時にエラーが出るのであれば，`config.log`ファイルもバグ報告に添付して下さい。テスト時にエラーが出る場合は，`tests/test-suite.log`ファイルも添付して下さい。

貴方のバグ報告が良いものであれば，我々開発陣は全力でその解決に努力しますが，そうでなければ,「もう少しまともな報告をしてよね」ぐらいの文句は言ってオシマイになるかもしれません。

バグ報告は MPFR のメーリングリスト‘`mpfr@inria.fr`’に送って下さい。

このマニュアルの記述が良く分からない，不正確な記述がある，言語不明瞭故に改良求む，ということでしたら，バグ報告同様，MPFR のメーリングリストにその旨伝えて下さい。

(訳注：日本語訳についてのご意見は‘`kouya.tomonori@sist.ac.jp`’にお寄せ下さい。)

4 MPFRの基本

4.1 ヘッダファイルとライブラリファイル

MPFR を使用するにあたり，必要となる変数や宣言は全て mpfr.hに含まれており，C でも C++でもそのままインクルードできるようになっています。MPFR ライブラリを使用するプログラムは例外なく下記のようにこのファイルをインクルードして下さい。

```
#include <mpfr.h>
```

MPFR の関数では FILE *型の引数を利用しており，このプロトタイプ宣言は<stdio.h> で行われているので，mpfr.hの前で，一緒にインクルードしておきましょう。

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

同様に，mpfr_vprintf関数等で va_list型引数を利用しているため，<stdarg.h> (あるいは <varargs.h>) もこのデータ型のプロトタイプ宣言を行うためにインクルードしておく必要があります。

また，intmax_tを使用する関数に対しては必ず<stdint.h> もしくは <inttypes.h> を，mpfr.h より先にインクルードして下さい。これによって，mpfr.hでこれらの関数のプロトタイプ宣言が有効になります。更に，いくつかの環境下では (特に C++コンパイラで使用する場合)，ユーザー側で MPFR_USE_INTMAX_Tを (ポータブルにしたいなら特に)，mpfr.hより先に定義しておかなければいけません。あるいは，コマンドラインで-DMPFR_USE_INTMAX_Tとして定義しておくことも可能です。

[注記] mpfr.hファイル，もしくは gmp.h (mpfr.h内部で使用) が，他のヘッダファイルでも重複してインクルードされるようであれば，<stdio.h> や <stdarg.h> (もしくは <varargs.h>) は，mpfr.h や gmp.hより前でインクルードすべきです。あるいは，MPFR_USE_FILE (MPFR I/O 関数で必要) や MPFR_USE_VA_LIST (va_listパラメータを使用する MPFR 関数で必要) を mpfr.hより前で定義しておくという手もあります。つまり，mpfr.hをインクルードするような独自のヘッダファイルを作るのであれば，後者の方法を使う必要があるわけです。

MPFRのマクロを呼び出す際には，使用してはいけないキーワード (現時点ではdo，while，sizeof など) と同じ名前のマクロを定義しないようにしましょう。

mpfr.hをインクルードする前に，MPFR_USE_NO_MACROマクロを被せた関数を使った MPFR マクロの使用は控えましょう。このマクロはデバッグ用で，普通は使いません。あるマクロを使うことで，警告オプション指定時にはコンパイラがニセの警告を出し，プロトタイプ宣言のチェックを妨げる可能性が出てくるからです。

MPFR を利用するプログラムは，必ず libmpfrライブラリファイルと libgmpライブラリファイルをリンクしなければなりません。UNIX コマンドラインでリンクする際には '-lmpfr -lgmp' (この順序でリンクすること) と指定します。GCC の場合は下記のように指定してリンクします。

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

MPFR のビルドには Libtool を使用しているため，それを利用することも可能です。詳細は *GNU Libtool* 参照のこと。

MPFR が標準的ではない所にインストールされている場合は，'C_INCLUDE_PATH'や 'LIBRARY_PATH' といった環境変数にそのパスを設定したり，コンパイラの '-I' や '-L' オプションで適切なディレクトリを指定することで対応できます。共有ライブラリの場合は，ランタイムライブラリパス ('LD_LIBRARY_PATH'等) の設定も行っておく必要があります。その他のインストールに関する情報については INSTALLファイルをご覧下さい。

その他，下記のように，‘pkg-config’ (MPFR 4.0 から用意されている ‘mpfr.pc’ファイル) を使用することもできます。

```
cc myprogram.c $(pkg-config --cflags --libs mpfr)
```

‘MPFR_’や ‘mpfr_’といった語句から始まる関数，マクロ，定数等は MPFR で予め定義されているものになります。従って，MPFR を利用するソフトウェアは，通常はこれらの文字列の使用は避けるようにして下さい。

4.2 用語とデータ型

浮動小数点数 *(floating-point number)*（短く *float* とも称します）は 2 を基数とする浮動小数点数を表現するオブジェクトであり，符号部，任意長の正規化された有効小数部（仮数部とも呼ぶ），指数部（固定長の整数）で構成されているものを，正規化数 (regular number) と呼びます。IEEE754 規格と同様，MPFR の浮動小数点数も正規化数以外に，3 種類存在します。符号付きゼロ，符号付き無限大，非数 (NaN) がそれにあたります。NaN は浮動小数点オブジェクトのデフォルト値で，ゼロ割るゼロ，+無限大引く+無限大といった演算の結果としても使用されます。特に明記されていなければ，NaN の符号部は不定です。IEEE754 規格とは異なり，MPFR には一種類の NaN しかありませんし，非正規化数も存在しません。それ以外はほぼ IEEE754 規格に準拠していますが，多少異なるところがあるやもしれません。MPFR 浮動小数点数オブジェクトの C データ型は `mpfr_t`で，要素を一つだけ持つ構造体の配列として定義されています。こうすることで，配列の引数として渡される時にはこの配列へのポインタとして扱われるからです。このポインタは `mpfr_ptr`という C データ型として定義されています。

精度桁数 *(precision)* は，浮動小数点数の有効小数部のビット数を意味します。C のデータ型としては `mpfr_prec_t`を使用します。精度桁数は `MPFR_PREC_MIN`から `MPFR_PREC_MAX`までの範囲の任意の整数値に設定できます。現在の実装では `MPFR_PREC_MIN`は 1 となっています。

[警告] MPFR は内部で精度桁数を増やして計算する必要があります。こうすることで，正確な演算結果（特に，正しく丸めたもの）を返すことができる訳です。従って，精度桁数を `MPFR_PREC_MAX` 付近に設定しないようにして下さい。さもないと，アサーションエラーを誘発してしまいます。また，使用環境のメモリ限度目一杯まで使用することは避けて下さい。プログラムが強制終了されたり，クラッシュしたり，C プログラムによっては訳の分からない挙動を引き起こすことになりかねません。

指数部 *(exponent)* は正規化されている通常の浮動小数点数のパーツの一つです。C のデータ型は `mpfr_exp_t`となります。有効な指数部の範囲は元となるデータ型が扱える範囲に制限されます。また，Section 5.13 [Exception Related Functions], 頁 42 で説明している通り，指数部の範囲をグローバルに設定することも可能です。3 種類の特殊な浮動小数点値は指数部を保持しません。

丸めモード *(rounding mode)* は，浮動小数点演算結果を丸める方式を定めるものです。丸めは，演算結果がそのまま格納先の変数に収まり切れない時に実行されます。丸めモードを表わす C のデータ型は `mpfr_rnd_t`です。

MPFR はグローバルに，もしくはスレッドごとにフラグを保持しており，サポートしている例外をここで判別します (Section 4.6 [Exceptions], 頁 11)。 フラグの C データ型は，複数のフラグやビットマスクを一括して表現できるようになっています。

4.3 MPFR 変数のデータ変換

MPFR 変数を割り当てる前に，初期化のための関数を呼び出す必要があります。変数が用済みになった際には，当該変数を消去する関数を呼び出す必要があります。変数の初期化は一回だけにしておきましょう。何度も初期化しようとすると，その間に消去関数を呼ぶ必要が出てきます。一度初期化された変数に対しては何度でも値を代入することができます。

動作が遅くならないよう，変数の初期化と消去をループ内で繰り返さず，ループに入る前に一度だけ初期化を行い，ループを抜けてから変数を消去するようにしましょう。

MPFR 変数用に追加的なメモリスペースを確保する必要はありません。どんな精度桁数の変数でも，仮数部は固定サイズで確保されます。従って，精度桁数の変更，初期化，再初期化等をしない限り，変数が有効である間は使用メモリ量は変化しません。

一般に，全ての MPFR 関数の引数の並び順は，前の方が出力用，その後ろが入力用となります (訳注:LAPACK や BLAS とは真逆)。これは，代入演算子に倣った作法です。MPFR の場合，同一の関数の引数として入力用・出力用に同じ変数を指定することも可能です。例えば，浮動小数点数の乗算を行う代表的な関数は `mpfr_mul`ですが，これは `mpfr_mul (x, x, x, rnd)`という指定を行っても全く問題ありません。この場合は，xの二乗を丸めモード `rnd`の下で計算し，その演算結果を xに書き戻します。

4.4 丸めモード

MPFR がサポートする丸めモードは以下の通りです。

- `MPFR_RNDN`: 最近接値への丸め (IEEE 754-2008 規格の roundTiesToEven に相当),
- `MPFR_RNDZ`: ゼロ方向への丸め (切り捨て) (IEEE 754-2008 規格の roundTowardZero に相当),
- `MPFR_RNDU`: +∞方向への丸め (IEEE 754-2008 規格の roundTowardPositive に相当),
- `MPFR_RNDD`: -∞方向への丸め (IEEE 754-2008 規格の roundTowardNegative に相当),
- `MPFR_RNDA`: ゼロから遠ざかる丸め
- `MPFR_RNDF`: 忠実丸め (faithful rounding)。現在まだ試験的なもので，使用できるのは基本演算（加減乗除算，2 乗，平方根）や，本文書で言及されている MPFR の一部の演算だけです。他の演算でもこの丸めモードを使用できるかもしれませんが，正しい動作は保証致しかねます。テストコードで問題が起これば対処はしますが，目視でのチェックは今のところしていません。もし問題があればバグとして報告して頂ければ，対処することは可能でしょう。

‘最近接値への丸め’(RN) モードの挙動は IEEE 754 規格に定められている通りです。浮動小数点数として表現できる数のちょうど中間地点の値の場合は，最小有効ビット (LSB, Least Significant Bit) がゼロになる方の値に丸められます。例えば，2.5 という値の場合は，2 進数では (10.1) と表現されますが，2 ビットに収めようとする場合，(11.0) = 3 ではなく，(10.0) = 2 の方を選択します。このルールは，Knuth の"The Art of Computer Programming" Vol.2 の 4.2.2 節 (訳注: アスキー・メディアワークスの日本語訳では P.223) で解説されている ドリフト現象を防ぐために設けられています。

`MPFR_RNDF`モードの場合，`MPFR_RNDD`モード時の値になるか，`MPFR_RNDU`モード時の値になります。計算結果がピッタリ収まる場合，つまり，演算結果が丸めなしで正しく表現できる場合は，その値がそのまま返されます。従って，丸めの結果は 2 種類存在し，アンダーフローやオーバーフローが起こらなければ，丸められた値の誤差は厳格に 1ULP(Unit in Last Place) 未満で収まります。`MPFR_RNDF`モードの場合, 返り値の三種値 (ternary value, 定義は後述) と不正確フラグ (inexact flag, 定義はフラグの解説時にまとめて述べる) は不定，ゼロ除算フラグ (divide-by-zero flag) は他の丸めモードと同様に設定，アンダーフローフラグとオーバーフローフラグは，丸めた値が `MPFR_RNDD`モード時のものになるか，`MPFR_RNDU` モード時のものになるかにより，相応するものが設定されます。この丸めの結果は再現できないことがあり得ます。

MPFR が提供する関数の大部分は，最初の引数に演算結果を格納し，2 番目の引数以降に入力値を与え，最後の引数で丸めモードの指定を行い，関数の返り値は `int`型，という形式になっています。この返り値を三種値 *(tenary value)* と称します。演算結果として格納される値は正しく丸められた (correctly rounded) ものとなっています。つまり，MPFR は，無限桁計算で得られる結果を，この変数の指定精度ビット数に収まるように丸めて返す，という処理を行う訳です。入力値には誤差はない（指定精度ビット数に収めても変化しない）ものとして扱います。

結果として，ゼロではない実数を丸めた場合の誤差は，最近接値丸めモードの場合は 1/2 ulp(unit in the last place) 以下となり，その他の丸めモードでは 1ULP 未満となります (この ULP は，丸めた後の値の有効ビットに対するものとなる)。

特に記述がなければ，`int`型の返り値を取る関数は三種値を返します。この返り値がゼロの時は，先頭の変数に格納される演算結果は丸めの必要のない正確な値であることを示しています。返り値が正の値の時（負の値の時）は，格納される値は正確な値よりも大きくなっている（小さくなっている）ことを表わしています。例えば，`MPFR_RNDU`モードの時は，返り値の三種値は常に正の値となり，例外として，丸めの必要のない場合のみ，ゼロが返ってきます。無限大（∞）になる場合はオーバーフローを起こしているので不正確な値と判断し， それ以外の場合は正確な値と見なします。非数 (NaN, Not-a-Number) は常に正しい値とします。返り値の三種値の逆符号の値は必ず `int`型として表現できるようになっています。

特に記述がなければ，特別な場合に限り (`acos(0)`など)，返り値として `1`(もしくは本マニュアル記載の指定値）となる関数は，その値が現時点の指数部の許容範囲内に収まらない場合はオーバーフロー，もしくはアンダーフローを発生させることになります。

4.5 特殊な値を表現する浮動小数点数

この節では，MPFR 関数が返す特殊な浮動小数点数値 (`mpfr_t`型として表現される) について解説します。ここで,「返す」とは，出力用の引数オブジェクトに格納される演算結果の値のことを意味します。関数の返り値そのものは int 型の三種値 (-1, 0, 1) なので，混同しないようにして下さい。複数の値を返す関数（例えば `mpfr_sin_cos`関数など) はこの規則が複数の値の格納先ごとに適用されます。

MPFR の関数には複数の入力引数を必要とするものがあります。この入力引数一つ分は，MPFR 数値からなる入力値集合への対応付けを表わしています。入力値集合から非数を除くと，この写像が指し示す先は拡張された実数，つまり両端の無限大 (±∞) を含む拡張実数集合と見なせます。

入力データが目的の数学関数の定義域内に収まっていれば，関数の返り値は “丸めモード” の節で解説している方式で丸められます (符号付きゼロの場合は後述)。定義域外の場合は，各 MPFR 関数ごとに本マニュアルの関数の解説 (Chapter 5 [MPFR Interface], 頁 14) 通りに処理し，特に言及がなければ本節に述べた一般的なルールで処理されます。

入力データが数学関数の定義域外であっても，正負無限大も含む拡張実数集合に収まっていて，かつ，連続的に拡張できる場合，関数の演算結果は極限値となります。例えば，定義域が (+Inf, 0) の `mpfr_hypot`関数は+Inf を返します。但し，このルールを使っても，`mpfr_pow` 関数は (1,+Inf) では定義できません。この理由は，1 に収束する x_n と，+Inf に発散する y_n を組み合わせた x_n, y_n という数列を考えてみれば分かります。この時，$(x_n)^{y_n}$ となりますが，nを無限大に飛ばすと，任意の正の値に収束させることができるからです。

入力値が数学関数の定義域の端っこだったり，+0 (もしくは −0) だったりすると, 数学的には入力値の 0 への右極限（もしくは左極限）を考える必要が出てきます。極限値が存在しない場合，例えば，−0 に対する `mpfr_sqrt`関数や `mpfr_log`関数の極限値などは，MPFR 関数の実装次第で演算結果が変わってきますが，例えば入力値 ±0 に対する `mpfr_log`関数は常に −Inf を返す，というように，前述のような規則に則った決め方をしなければなりません。

結果が 0 になる時は，ゼロの符号は，入力値が定義域に存在しない状況での極限値を考慮して決定します。0 より大きい（小さい) ところからの極限値は+0 (−0) になります。例えば，−0 を引数として与えた `mpfr_sin`関数の値は −0 となり，引数が 1 の時の `mpfr_acos`関数の値は，丸めモードに関わらず+0 になります。その他の場合は，MPFR 関数の実装によって符号が決定されます。例えば，−0 と +0 を入力値とする `mpfr_max`関数の値は+0 になります。

入力値が定義域外にある場合の値は NaN になります。例えば，−17 を入力値とする `mpfr_sqrt`関数は NaN を返します。

入力値が NaN の時は，定数関数である場合を除いて，結果も NaN になります。その辺りのことは Chapter 5 [MPFR Interface], 頁 14 に明示してあります。

例）(NaN,0) という入力に対して `mpfr_hypot`関数は NaN を返しますが，入力が (NaN,+Inf) であれば+Inf を返します。このことは Section 5.7 [Special Functions], 頁 26 のところで解説しています。入力値 xが有限値であれ無限大であれ，`mpfr_hypot`関数は (x,+Inf) という入力に対しては +Inf を返します。

4.6 例外

MPFR は，サポートする例外ごとにグローバルフラグ（もしくはスレッドごとのフラグ）を定義しています。フラグは 2 のべき乗を計算するマクロで，それぞれのフラグと例外を関係づけています。複数のフラグやマスクを一つにまとめても特定できるよう，個別のマクロとの OR を取ることで，例外を特定できるようになります。

フラグは，クリアしたり (lowered), 立てたり (raised), Section 5.13 [Exception Related Functions], 頁 42 に解説してある関数で状態を確認することができます。

サポートする例外の一覧を下記に示します。カッコ内は各例外に関係するマクロを意味します。

- アンダーフロー (`MPFR_FLAGS_UNDERFLOW`): アンダーフローは，計算結果の真値がゼロではない実数で，指数部長を制限せずに丸めた結果，現時点での最小指数部値より小さい指数部になった時に発生します。最近接値への丸めの場合，ちょうど中間地点の値の場合は，ゼロ方向に丸められます。

  [注記] アンダーフローの定義はこれだけに限りません。MPFR は丸めの後でアンダーフローを関知しますが，丸めの前のアンダーフローというものも定義可能です。例えば，$7 \times 2^{e-4}$ を求める関数を考えましょう。ここで eは最小の指数部とし，仮数部は 1/2 と 1 の間にあるものとします。格納先は 2 ビットの精度桁数しかないものとし，正の無限大方向への丸めを考えます。真値の指数部は $e-1$ となります。丸め処理の前にアンダーフローが起きると，この関数はアンダーフローを生成します。$e-1$ は現在の指数部範囲の外側になります。しかしながら，MPFR は最初に指数部の範囲を限定せずに丸めを行って考えます。正確に計算結果を 2 ビット精度桁数では表現できませんので，ここでは 0.5×2^e は現在の指数部の範囲で表現できます。従って，MPFR におけるアンダーフローは発生しません。
- オーバーフロー (`MPFR_FLAGS_OVERFLOW`): オーバーフローは，計算結果の真値が非ゼロの実数で，指数部長を制限せずに丸めた結果，現在の指数部の最大値より大きい指数部になる時に発生します。最近接値への丸めの場合は，無限大を返します。

  [注記] アンダーフローとは違い， オーバーフローの定義はこれ一つです。
- ゼロ除算 (`MPFR_FLAGS_DIVBY0`): 有限値の入力に対して，演算結果が無限大になった時に発生します。
- 非数 (NaN)(`MPFR_FLAGS_NAN`): NaN 例外は，計算の結果が NaN になる場合に発生します。
- 近似化 (`MPFR_FLAGS_INEXACT`): 不正確例外 (inexact exception) は，丸め誤差が生じる時に発生します
- 範囲エラー (`MPFR_FLAGS_ERANGE`): 範囲エラーは，MPFR 数を返さない関数（例えば比較関数，整数への変換関数など）が不正な結果を返す場合（`mpfr_cmp`関数の引数が NaN だったり，整数変換関数において，変換先のデータ型に収まらない場合など）に発生します。

更に言うと，これら全てのフラグから成るグループは `MPFR_FLAGS_ALL`マクロで表現することができます。MPFR の将来のバージョンで新しいフラグが追加されても，このマクロにはそれが追加される予定です。

ISO C99 規格との相違点は下記の通りです。

- C では qNaN(quiet NaN) のみが定義されており，NaN が伝播して不正な例外を発生することはありません。全ての NaN が sNaN(signaling NaN) のように見えますが，明記してある場

合を除き，MPFR では NaN フラグが立つのは，NaN が生成されたり，NaN が NaN を呼んだり (例えば NaN + NaN の場合など) する時のみです。

- C における不正な例外に相当するのは，MPFR の NaN 例外，もしくは範囲エラー です。

4.7 メモリ制御

MPFR 関数の中にはキャッシュを生成するものもあります。例えば π などの定数を計算する時には，ユーザは `mpfr_const_pi`関数を直接呼び出せばいいですし，内部的にこのような関数が呼び出されて計算されることもあるので，定数をキャッシュしておくわけです。もし，キャッシュしてある値より多くの精度桁数が必要となれば，自動的に再計算されます。この事態を避けるため，10% 精度桁数が増えても対処できるようにしています。

MPFR 関数の中には，スレッド単位のメモリプールを生成するものもあります。このメモリプールは `mpfr_free_pool`関数で解放できます。但し，デフォルトのまま MPFR ビルドを行うとメモリ割り当てサイズの制限ができてしまうので，あまり大量のメモリを確保しないようにしましょう。

ユーザはいつ何時でも `mpfr_free_cache`関数や `mpfr_free_cache2`関数を使ってキャッシュやメモリプールを解放できます。スレッドを終了する際には必ずスレッドローカルのキャッシュを解放するようにして下さい。また，‘`valgrind`’ (メモリリークを防止するデバッグソフト）のようなツールを使う際には，全てのキャッシュを消去して下さい。

MPFR は，一時メモリ領域をスタック内にも確保します。このメモリ割り当て処理は，GMP ビルド時の設定で指定された関数と同じものを使用します。詳細は *GNU MP* マニュアルの “Custom Allocation” を参照して下さい。つまり，メモリ処理関数を実行中に変更すると，変更後のメモリ処理関数を使ってのメモリの再割り当てや解放は行われなくなるということです。従って，実際には，`mp_set_memory_functions`関数を使ってメモリ処理関数を変更するのであれば，現在のメモリ処理関数で割り当てた全てのメモリを最初に解放しておく必要があります。独自のデータに対しては `mpfr_clear`関数を, キャッシュやメモリプールは `mpfr_mp_memory_cleanup`関数を，MPFR が使われる可能性のあるすべてのスレッドで実行しておきます。`mpfr_free_cache`関数を使ってもできますが，先々メモリ割り当て関数を変更する予定があるのなら，`mpfr_mp_memory_cleanup`関数を使うことをお勧めします。例えば，メモリ割り当て関数が変更される際にキャッシュが解放されないよう，浮動小数点定数を `malloc`関数で確保しておく，ということも可能です。MPFR は間接的に使用されることもあるライブラリなので，そのようなライブラリでは，`mpfr_mp_memory_cleanup` 関数を呼び出して処理するメモリ解放関数を提供するようにし，ユーザにこの点周知しておいて下さい。

[注意] マルチスレッド動作のアプリケーションでは，MPFR が利用できる全てのスレッドでメモリ割り当てができるようにしなければなりません。こうしておくことで，あるスレッドで割り当てられたデータは，他のスレッドにおいても割り当てや解放ができるようになります。

フラグ，指数部の有効範囲, デフォルト精度桁数，デフォルト丸めモード，キャッシュなど，パラメータ経由でアクセスされることがない MPFR の内部データは，(スレッドセーフでない MPFR をビルドした場合には) グローバル変数であり，スレッドセーフの場合は，スレッド単位のローカル変数 (スレッドローカルストレージ (Thread Local Storage)，TLS) となります。スレッド生成後の TLS の初期値は，コンパイラやスレッド実装によって決まります。MPFR は単純に変数の初期化を行い， 実装で定義された TLS 識別子を変数に付加します。

MPFR を使ってライブラリを書くのであれば，それが利用されるアプリケーションや他のライブラリでも MPFR を使っているものとお考え下さい。つまり，指数部の有効範囲，デフォルトの精度桁数や丸めモードは自分のライブラリ以外からも変更される可能性があり，異なるスレッドで保存されていない限り，それらの値は残りません。従って，ライブラリ内で使用するこれらの MPFR 内部データは，関数の処理が終わる前に保存しておく必要があります（その関数が内部データを変更するものでない限り）。MPFR を使うソフトウェアを制作する場合は，この手の内部データの変更に非互換性が発生しないよう注意して実装を行って下さい。

4.8 MPFR からパフォーマンスを引き出す術

ここでは，MPFR のパフォーマンスを生かすヒントを記しておきます。

- 変数の割り当てと解放はなるべく避け，一度確保した変数は使い回すようにしましょう。やむを得ない時にも，ループの外で変数の割り当てと解放は行うようにし，一時変数はサブルーチンの内部で確保するのではなく，その外で確保してからサブルーチンに渡すようにしましょう。
- できれば `mpfr_set`関数ではなく，`mpfr_swap`関数を使って下さい。これで仮数部のコピーを避けることができます。
- MPFR を C++で使用することは避けましょう。止むを得ない時も，不要なメモリ割り当てやコピーを行わない C++インターフェースを使って下さい。
- MPFR 関数は in-place 動作を保証しています。`a = a + b`という計算では，余計な変数は不要なので，ドーンと `mpfr_add (a, a, b, ...)`と書いちゃって下さい。

5 MPFRの関数

MPFR の多倍長精度浮動小数点関数は，mpfr_t型の引数を取ります。

MPFR の浮動小数点演算関数のインターフェースは GNU MP の関数とよく似た作りになっており，関数名は mpfr_という文字列から始まります。

ユーザは変数ごとに精度桁数を指定する必要があります。計算は格納先の変数の指定精度桁数で行われますので，入力用の変数の精度桁数で計算に要するコストが決まるわけではありません。

MPFR における計算の戦略について説明しましょう。まず，要求された演算を正確に（つまり「無限桁で」）実行し，しかる後に，その結果を出力先の変数の精度桁数に，指定された丸め方式で丸めて収めます。MPFR の浮動小数点演算関数は IEEE754 規格の演算の自然な拡張になっており，異なるワードサイズ，異なるコンパイラ，異なる OS 環境であっても，同一の結果を返します。

MPFR には計算結果の有効精度を保証する機能はありません。つまり，ユーザ自身が，より高度なレイヤーの機能を活用するなどして (例えば MPFI(多倍長区間演算ライブラリ) 等）自分で計算結果の有効性を確認しなければなりません。演算の結果，使用する 2 変数が数桁程度の有効桁数しかなく，より大きな精度桁数の変数にその結果を格納するとなれば，MPFR は馬鹿正直に格納先の桁数の精度で計算を行います。

C の標準マクロである errnoの値は，エラーの有無に関わらず，MPFR 関数やマクロではゼロにセットされています。本マニュアルに記述してある場合を除き，MPFR は，利用している他の関数 (libc 提供の関数群，メモリ割り当て関数など) と同様，errnoをセットすることはありません。

5.1 初期化関数

mpfr_t型の変数オブジェクトは，値を代入する前に，mpfr_init関数や mpfr_init2関数で初期化しておく必要があります。

void mpfr_init2 (*mpfr_t x*, *mpfr_prec_t prec*) [関数]
変数 *x*を初期化します。その際，精度桁数 (仮数部の桁数) を正確に *prec*ビットに設定し，値を NaN とします。

[注記] この関数とは違い，GNU MP の MPF 初期化関数はゼロを代入します。

通常，変数の初期化は 1 回だけ行い，mpfr_clear関数を使って消去してから再初期化するようにしましょう。初期化後の変数に対する精度桁数の変更は，mpfr_set_prec関数を使って行います。精度桁数 *prec*は MPFR_PREC_MIN以上，MPFR_PREC_MAX以下の整数として指定します。この指定に従わない場合の挙動は規定されていませんので注意して下さい。

void mpfr_inits2 (*mpfr_prec_t prec*, *mpfr_t x*, ...) [関数]
va_listで指定された全ての mpfr_t型変数を初期化します。精度桁数は *prec*ビットに，値は NaN が代入されます。詳細は mpfr_init2関数の説明を参照して下さい。mpfr_t型の変数，もしくは mpft_ptrポインタだけから構成されている va_listが指定できます。この va_listは変数 *x*から始まり，mpfr_ptr型の NULL ポインタで終わります。

void mpfr_clear (*mpfr_t x*) [関数]
変数 *x*の仮数部が確保している記憶領域を解放します。この関数を使用する際には，該当 mpfr_t変数への処理がすべて完了していることを確認するようにして下さい。

void mpfr_clears (*mpfr_t x*, ...) [関数]
va_listで指定された全ての mpfr_t変数の記憶領域を消去します。 詳細は mpfr_clear関数の解説を参照して下さい。va_listで指定された変数は全て mpfr_t型（もしくは同等の mpfr_ptr型) であることを想定しています。変数リストは変数 *x*から始まり，最後は mpfr_ptr型の NULL ポインタが来る形式で指定します。

以下，まとめて初期化する関数の使い方を示します。既に解説した通り，NULLポインタが変数リストの終端に来る必要がありますが，下記の例は (mpfr_ptr) 0と記述しています。この場合，(mpfr_ptr) NULLと書いても O.K. です。

```
{
  mpfr_t x, y, z, t;
  mpfr_inits2 (256, x, y, z, t, (mpfr_ptr) 0);
  ...
  mpfr_clears (x, y, z, t, (mpfr_ptr) 0);
}
```

void mpfr_init (*mpfr_t* x) [関数]
変数 *x*を初期化し，精度桁数をデフォルト値にセットし，値として NaN を代入します。デフォルトの精度桁数は mpfr_set_default_prec関数で変更できます。

[注意] プログラムの実行中，使用している他のライブラリからデフォルト精度桁数を変更され，初期化済みの変数の精度桁数は変更前のままになっているケースが見受けられます。確実に精度桁数を指定したければ mpfr_init2関数を使って下さい。

void mpfr_inits (*mpfr_t* x, ...) [関数]
va_listリストにあるすべての mpfr_t変数を初期化し，デフォルトの精度桁数をセットし，値として NaN を代入します。詳細は mpfr_init関数の説明を参照して下さい。va_listリストの変数は全て mpfr_t型（もしくは同等の mpfr_ptrポインタ) であると仮定しています。このリストは変数 *x*から始まり, 最後は mpfr_ptr型の NULL ポインタが終端となります。

[注意] プログラムの中で，リンクした他のライブラリからデフォルト精度桁数を変更され，元の変数の精度桁数はそのままにされるということも起こり得ます。確実に精度桁数を設定したければ mpfr_inits2関数を使用して下さい。

MPFR_DECL_INIT (*name*, *prec*) [マクロ]
このマクロは *name*という名前で自動的に mpfr_t型の変数を生成し，初期化を行い，確実に精度桁数を *prec*ビットに設定し，値として NaN を代入します。*name*は有効な変数名でなければなりません。このマクロは変数宣言部分で使用して下さい。mpfr_init2関数より高速に動作しますが，次のような欠点もあります。

- このマクロで宣言した変数に対しては絶対に mpfr_clear関数で消去してはいけません。メモリ領域はポインタとして割り当てられており，マクロが有効なブロック (カッコ内部) を出る時に消去されます。
- このマクロで設定された精度桁数を変更することはできません。
- このマクロでバカでかい精度桁数を設定した変数を生成することは避けた方がいいでしょう。
- このマクロを使えるのは，‘Non-Constant Initializers’ (C++と ISO C99 で規定) と ‘Token Pasting’ (ISO C89 で規定) が有効なコンパイラだけです。定数表現以外の精度桁数 *prec*の指定をするのであれば，‘variable-length automatic arrays’ (ISO C99 で規定) も有効でなければいけません。GCC 2.95.3 以上であればこれらの機能はすべて有効です。C89 の GCC で ‘-pedantic’オプションをつけてコンパイルしたいのであれば，MPFR_USE_EXTENSIONマクロを有効化することで，この MPFR_DECL_INITマクロ定義部分に起因して現れる警告を抑制することができます。

void mpfr_set_default_prec (*mpfr_prec_t* prec) [関数]
デフォルトの精度桁数を *prec*ビットに設定します。ここで *prec*には MPFR_PREC_MIN以上，MPFR_PREC_MAX以下の整数が指定できます。ここで言う変数の精度桁数は，仮数部のビット長を意味します。この関数の使用以降に呼び出される mpfr_init関数や mpfr_inits関数は設定後の精

度桁数を変数にセットしますが，それ以前に初期化された変数の精度桁数は変更されません。精度桁数の初期値は 53 ビットです。

[注意] MPFR を '`--enable-thread-safe`'オプション設定してビルドすると，デフォルトの精度桁数は各スレッドごとに設定されたものになります。詳細は Section 4.7 [Memory Handling], 頁 12 を参照して下さい。

`mpfr_prec_t mpfr_get_default_prec (`*void*`)` [関数]
現在の MPFR のデフォルト精度桁 (ビット数) を返します。詳細は `mpfr_set_default_prec`関数の説明を参照して下さい。

以下，多倍長浮動小数点変数の初期化方法の例を示します。

```
{
  mpfr_t x, y;
  mpfr_init (x);                /* デフォルトの精度桁数で初期化 */
  mpfr_init2 (y, 256);          /* 精度桁数$2256 ビットに設定して初期化 */
  ...
  /* プログラムが終了する間際に実行する。 */
  mpfr_clear (x);
  mpfr_clear (y);
  mpfr_free_cache ();           /* pi のような定数のキャッシュを消去 */
}
```

下記の 2 つの関数は，計算の途中で精度桁数を変更したい場合に有用です。例えば，Newton-Raphson 法のような反復法において，解の近似度に応じて精度桁数を適宜調整する際に役立ちます。

`void mpfr_set_prec (`*mpfr_t* `x`, *mpfr_prec_t* `prec``)` [関数]
変数 *x*の精度桁数を *prec*ビットに設定し直し，NaN を代入します。変数 *x*に入っていた値は NaN に上書きされます。この関数の機能としては，`mpfr_clear(x)`で変数を消去した後に`mpfr_init2(x, prec)`で初期化した場合と同じですが，*x*の仮数部長が，設定後の精度桁数が十分格納できるものであれば，メモリ領域の再確保を行わない分，高速に実行できます。精度桁数 *prec* は `MPFR_PREC_MIN`以上，`MPFR_PREC_MAX`以下の任意の整数を設定できます。変数 *x*に格納されている値を保存しておきたい場合は，`mpfr_prec_round`関数を利用して下さい。

[注意] 変数 *x*が`MPFR_DECL_INIT`マクロや，`mpfr_custom_init_set`(see Section 5.15 [Custom Interface], 頁 47) 関数で初期化されている場合は，この関数は使用できません。

`mpfr_prec_t mpfr_get_prec (`*mpfr_t* `x`)` [関数]
変数 *x*の精度桁数，つまり，仮数部のビット長を返します。

5.2 代入関数

この節で解説する関数は，初期化済み (Section 5.1 [Initialization Functions], 頁 14 参照) の浮動小数点変数に新しい値を代入するためのものです。

`int mpfr_set (`*mpfr_t* `rop`, *mpfr_t* `op`, *mpfr_rnd_t* `rnd``)` [関数]
`int mpfr_set_ui (`*mpfr_t* `rop`, *unsigned long int* `op`, *mpfr_rnd_t* `rnd``)` [関数]
`int mpfr_set_si (`*mpfr_t* `rop`, *long int* `op`, *mpfr_rnd_t* `rnd``)` [関数]
`int mpfr_set_uj (`*mpfr_t* `rop`, *uintmax_t* `op`, *mpfr_rnd_t* `rnd``)` [関数]
`int mpfr_set_sj (`*mpfr_t* `rop`, *intmax_t* `op`, *mpfr_rnd_t* `rnd``)` [関数]
`int mpfr_set_flt (`*mpfr_t* `rop`, *float* `op`, *mpfr_rnd_t* `rnd``)` [関数]
`int mpfr_set_d (`*mpfr_t* `rop`, *double* `op`, *mpfr_rnd_t* `rnd``)` [関数]
`int mpfr_set_ld (`*mpfr_t* `rop`, *long double* `op`, *mpfr_rnd_t* `rnd``)` [関数]

int mpfr_set_float128 (*mpfr_t rop, __float128 op, mpfr_rnd_t rnd*) [関数]
int mpfr_set_decimal64 (*mpfr_t rop, _Decimal64 op, mpfr_rnd_t rnd*) [関数]
int mpfr_set_z (*mpfr_t rop, mpz_t op, mpfr_rnd_t rnd*) [関数]
int mpfr_set_q (*mpfr_t rop, mpq_t op, mpfr_rnd_t rnd*) [関数]
int mpfr_set_f (*mpfr_t rop, mpf_t op, mpfr_rnd_t rnd*) [関数]

これらの関数群は，*op*の値を，丸め方式 *rnd*で丸め，*rop*に代入します。それぞれ下記のような違いがあるので，留意して下さい。

mpfr_set_ui関数，mpfr_set_si関数，mpfr_set_uj関数，mpfr_set_sj関数は，入力値が 0 の時，+0 に変換して代入します。

mpfr_set_float128関数は，'--enable-float128'オプション付きで設定されたときにのみ有効な関数で，当然，'__float128'型が定義されているコンパイラ (GCC 4.3 以降でサポート) が必要となります。mpfr_set_float128関数を利用したければ，mpfr.hをインクルードする前に MPFR_WANT_FLOAT128マクロを定義しておく必要があります。

mpfr_set_z関数，mpfr_set_q関数，mpfr_set_f関数では，丸め方式の指定は無効になります。

実行環境が IEEE 754 標準規格をサポートしていない場合は，mpfr_set_flt関数，mpfr_set_d関数，mpfr_set_ld関数，mpfr_set_decimal64関数は，符号付きゼロを正しく扱えません。

mpfr_set_decimal64関数は '--enable-decimal-float'を設定してビルドし，かつ，コンパイラやシステムが '_Decimal64'型をサポートしている場合 (最近の GCC はサポートしています) のみ，利用できるようになります。この mpfr_set_decimal64関数を使う場合は，mpfr.h をインクルードする前に MPFR_WANT_DECIMAL_FLOATSマクロを定義しておく必要があります。

mpfr_set_q関数は，分子もしくは分母が mpfr_t型で表現できない場合はエラーとなります。

mpfr_set関数では，IEEE 754 の copy動作に準じて，NaN の符号もそのまま渡します。但し，IEEE 754 とは異なり，NaN フラグを立てます。

[注意] mpfr_t型の浮動小数点定数を代入する際には，mpfr_set_str関数か，他の定数関数，例えば π は mpfr_const_pi関数を使うようにし，mpfr_set_flt関数，mpfr_set_d関数，mpfr_set_ld関数，mpfr_set_decimal64関数は使用しないで下さい。これらの関数を使うと，浮動小数点数は一旦，精度の低い浮動小数点数 (53 ビット倍精度や，mpfr_set_decimal64関数の場合は 10 進精度) に変換されてから，MPFR に渡されてしまいます。

int mpfr_set_ui_2exp (*mpfr_t rop, unsigned long int op, mpfr_exp_t e, mpfr_rnd_t rnd*) [関数]
int mpfr_set_si_2exp (*mpfr_t rop, long int op, mpfr_exp_t e, mpfr_rnd_t rnd*) [関数]
int mpfr_set_uj_2exp (*mpfr_t rop, uintmax_t op, intmax_t e, mpfr_rnd_t rnd*) [関数]
int mpfr_set_sj_2exp (*mpfr_t rop, intmax_t op, intmax_t e, mpfr_rnd_t rnd*) [関数]
int mpfr_set_z_2exp (*mpfr_t rop, mpz_t op, mpfr_exp_t e, mpfr_rnd_t rnd*) [関数]

$op \times 2^e$ を丸め方式 *rnd*で丸めて *rop*に代入します。入力値が 0 の場合は+0 に変換されます。

int mpfr_set_str (*mpfr_t rop, const char *s, int base, mpfr_rnd_t rnd*) [関数]

*base*進数表現として文字列 *s*を解釈し，丸め方式 *rnd*で丸めた値を *rop*に代入します。有効な文字列形式の詳細については mpfr_strtofr関数の説明を読んで下さい。mpfr_strtofr関数とは異なり，mpfr_set_str関数は，完全に浮動小数点数として解釈できる文字列だけを扱います。

返り値の意味は，他の MPFR 関数とは異なりますので注意して下さい。最後の NULL 終端まで完璧な *base*進表現の浮動小数点数になっていれば 0 を返し，そうでない場合は，*rop*の値は

書き換えられ，−1 を返します。(返り値として三種値 [ternary value], 頁 9 が必要であれば，mpfr_strtofr関数を使って下さい。)

[注意] *rop*が無限大になった時，それが無限大の *s*を入力したせいなのか，オーバーフローしたせいなのかを見分けたい場合は mpfr_strtofr関数を使って下さい。

int mpfr_strtofr (*mpfr_t* ***rop***, *const char* ***nptr***, *char* ****endptr***, *int* ***base***, *mpfr_rnd_t* ***rnd***) [関数]

*base*進表現の文字列 *nptr*を読み取り，丸め方式 *rnd*で丸めて返します。*base*は 0 (これも下記に示すように有効な進数です）か，2 以上 62 以下の整数でなければなりません。これ以外の指定をした場合の挙動は未確定です。*nptr*が有効なデータ形式から始まっていれば，値は *rop*に代入され，***endptr***は，これが NULL ポインタでなければ，有効な文字列データの終端文字を指します。もし NULL ポインタであれば，*rop*にはゼロが代入されます (strtod関数の処理に準じています)。*nptr*の値は，*endptr*が NULL ポインタでなければ，これが指しているメモリ領域に格納されます返り値は三種値 (-1, 0, +1) になります。

文字列の解釈処理は，標準 C の strtod関数に多少の拡張を加えたものに準じます。文字列先頭のホワイトスペースは読み飛ばされ，符号 (‘+’ or ‘-’), 数字，特殊文字から成る文字列のみ対象文字列として解釈されます。ホワイトスペースを除いた，最大限長い有効文字列が解釈の対象となります。

数値データは，小数点 (なくても良い) を含む空白のない仮数部，指数部を表わす先頭文字，そして，符号を含む 10 進表記の空白のない指数部列（なくても良い）から成る形式でなければなりません。仮数部の数値は 10 進数字 (0〜9) か， アルファベット (最大 62 文字）で表記されます。後者については，‘A’ = 10, ‘B’ = 11, . . . , ‘Z’ = 35 となり，36 以下の進数の場合は大文字・小文字の区別はせずに扱い，37 以上の進数表記の場合は，‘a’ = 36, ‘b’ = 37, . . . , ‘z’ = 61 となり，この場合は大文字・小文字の区別を行います。仮数部の数値は進数未満の数でなければなりません。小数点は，現時点におけるロケールに基づくもの (C 標準仕様）か，ピリオド (ロケールとは無関係に MPFR 側で規定）が使用できます。指数部開始を示す文字は ‘e’や ‘E’が 10 以下の進数までは使用でき，‘@’は進数に寄らず使用できます。この区切り文字以降が乗じられる進数のべき乗を表わします。2 進や 16 進表現の場合は，この区切り文字として ‘p’ や ‘P’が使用できますが，この場合は，*2*のべき乗の意味になります。つまり，16 進表現の場合，‘1@2’と書けば 256 を意味しますが，‘1p2’と書くと 4 を意味することになるわけです。 指数部の表記は必ず 10 進表現でなければなりません。

引数 *base*が 0 の場合，基数は次のように自動的に決定されます。仮数部が ‘0b’や ‘0B’で始まる場合は基数は 2 となります。仮数部が ‘0x’や ‘0X’で始まる場合は基数 16，それ以外は全て基数は 10 と決定します。

[注意] 指数部の指定をするのであれば，少なくとも 1 桁以上の表記が必要です。1 桁以上の数値の指定がないと，指数部開始場所の文字やそれに続く符号は指数部として解釈されず，仮数部の終端扱いとなります。同様に，‘0b’, ‘0B’, ‘0x’，‘0X’の表記のない 2 進表現や 16 進表現は，‘0’という文字で読み込みがストップしますので，00 と読み込まれます。

特殊データ (無限大や非数) は ‘@inf@’もしくは ‘@nan@(n-char-sequence-opt)’という文字列表記となり，*base* ≤ 16 であれば，‘inf’, ‘nan’，‘nan(n-char-sequence-opt)’は大文字・小文字の区別なく認識できます。‘n-char-sequence-opt’は，数字，アルファベット，アンダースコア (0, 1, 2, . . . , 9, a, b, . . . , z, A, B, . . . , Z, _) のみ含む文字列で，空でも構いません。

[注意] NaN も含めて，符号も指定できます。例えば，‘-@nAn@(This_Is_Not_17)’は有効な 17 進数の NaN を表現しています。

void mpfr_set_nan (*mpfr_t* ***x***) [関数]

void mpfr_set_inf (*mpfr_t* ***x***, *int* ***sign***) [関数]

`void mpfr_set_zero` (*mpfr_t* **x**, *int* **sign**) [関数]
それぞれ，変数 *x* に NaN (非数), 無限大，ゼロを代入します。`mpfr_set_inf`関数や `mpfr_set_zero`関数は，*sign*が非負であれば，*x*にプラス無限大とプラスゼロを代入します。`mpfr_set_nan`関数の場合は，符号ビットは不確定となります。

`void mpfr_swap` (*mpfr_t* **x**, *mpfr_t* **y**) [関数]
変数 *x* と変数 *y* が指している構造体を入れ替えます。値が丸められることはありません。この点，3 番目の引数で丸めモードを指定する 3 つの`mpfr_set`関数グループとは挙動が異なります。

[注意] 精度桁数が入れ替わると，その後の代入処理に影響が出てくる恐れがあります。また，仮数部のポインタも入れ替わりますので，*x* や *y* にそれを許容しない割り当て方をした場合は，この関数を使わないようにして下さい。*x* や *y* が，`MPFR_DECL_INIT`マクロや `mpfr_custom_init_set` (see Section 5.15 [Custom Interface], 頁 47) で確保されたものである場合が相当します。

5.3 初期化代入関数

`int mpfr_init_set` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [Macro]
`int mpfr_init_set_ui` (*mpfr_t* **rop**, *unsigned long int* **op**, *mpfr_rnd_t* **rnd**) [Macro]
`int mpfr_init_set_si` (*mpfr_t* **rop**, *long int* **op**, *mpfr_rnd_t* **rnd**) [Macro]
`int mpfr_init_set_d` (*mpfr_t* **rop**, *double* **op**, *mpfr_rnd_t* **rnd**) [Macro]
`int mpfr_init_set_ld` (*mpfr_t* **rop**, *long double* **op**, *mpfr_rnd_t* **rnd**) [Macro]
`int mpfr_init_set_z` (*mpfr_t* **rop**, *mpz_t* **op**, *mpfr_rnd_t* **rnd**) [Macro]
`int mpfr_init_set_q` (*mpfr_t* **rop**, *mpq_t* **op**, *mpfr_rnd_t* **rnd**) [Macro]
`int mpfr_init_set_f` (*mpfr_t* **rop**, *mpf_t* **op**, *mpfr_rnd_t* **rnd**) [Macro]
変数 *rop* を初期化し，丸め方式 *rnd* で丸めた *op* を代入します。*rop* の精度桁数は，`mpfr_set_default_prec`関数で設定した現状のデフォルト値が設定されます。

`int mpfr_init_set_str` (*mpfr_t* **x**, *const char* ***s**, *int* **base**, *mpfr_rnd_t* **rnd**) [関数]
x を初期化し，*base*進表現の文字列 *s* を丸め方式 *rnd* で丸めて代入します。詳細は `mpfr_set_str`関数の説明を参照して下さい。

5.4 データ型変換関数

`float mpfr_get_flt` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`double mpfr_get_d` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`long double mpfr_get_ld` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`__float128 mpfr_get_float128` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`_Decimal64 mpfr_get_decimal64` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
変数 *op* の値を丸め方式 *rnd* で丸めて，`float`，`double`, `long double`，`_Decimal64`にそれぞれ変換します。*op* が NaN の場合は, fixed NaN (qNaN もしくはシグナル) か，0.0/0.0 の結果を返します。*op* が± Inf の場合は，同じ符号を持つ無限大か，± 1.0/0.0 の結果を返します。*op* がゼロの場合は，同じ符号を持つゼロを返します。`mpfr_get_float128`関数と `mpfr_get_decimal64`関数は，それぞれの関数を有効化するオプション付きでビルドされた時のみ使用できます。詳細は `mpfr_set_float128` 関数と `mpfr_set_decimal64`関数の説明文を参照して下さい。

`long mpfr_get_si` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`unsigned long mpfr_get_ui` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`intmax_t mpfr_get_sj` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`uintmax_t mpfr_get_uj` (*mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
op の値を丸めモード *rnd* で整数に変換し，`long`型，`unsigned long`型，`intmax_t`型，`uintmax_t`型にそれぞれ変換します。*op* が NaN の時は 0 を返し，範囲エラー (erange) フラグを立てます。*op* が変換後のデータ型としては大きすぎる場合は，オーバーフローの方向に応じて，C データ型として定義されている最大値，もしくは最小値を返し，範囲エラーフラグも立てます。変換

後ののデータ型に収まるようであれば，*op*とは異なる値が返る場合，つまり，*op*が整数ではない場合，不正確 (inexact) フラグを立てます。`mpfr_fits_slong_p`関数, `mpfr_fits_ulong_p`関数, `mpfr_fits_intmax_p`関数，`mpfr_fits_uintmax_p`関数の説明も参照して下さい。

`double mpfr_get_d_2exp` (*long *exp, mpfr_t op, mpfr_rnd_t rnd*) [関数]
`long double mpfr_get_ld_2exp` (*long *exp, mpfr_t op, mpfr_rnd_t rnd*) [関数]
返り値として dを返し，*exp*には，形式的には *exp*へのポインタを代入します。ここで，*exp*は，$0.5 \leq |d| < 1$ ，かつ，$d \times 2^{exp}$ が，丸めモード *rnd* 方向に *op*を丸めて double 型 (long double 型) に変換したときに等しくなるように決定されます。*op*がゼロの時は，同じ符号になります。符号なしゼロを扱うように実装されていれば，符号なしのままとなり，*exp*も 0 が代入されます。*op*が NaN もしくは無限大の時は，対応する double 型（long double 型) の NaN もしくは無限大が返され，*exp*は不定値となります。

`int mpfr_frexp` (*mpfr_exp_t *exp, mpfr_t y, mpfr_t x, mpfr_rnd_t rnd*) [関数]
exp(形式的には *exp*へのポインタ) と yを，$0.5 \leq |y| < 1$，かつ，$y \times 2^{exp}$ が，xを yの精度桁数に *rnd*方向に丸めたものと等しくなるように値を代入します。xがゼロの時は，yにも同じ符号が付加されて，*exp*はゼロになります。xが NaN もしくは無限大の場合は同じ値が yに代入され，*exp*の値は不定になります。

`mpfr_exp_t mpfr_get_z_2exp` (*mpz_t rop, mpfr_t op*) [関数]
*op*の仮数部をスケールし，*op*の精度桁数の整数として *rop*に代入し，指数部 *exp*を返り値とします。この際，現状の指数部の範囲を超えたものになる可能性があります。つまり，*op*は $rop \times 2^{\exp}$. と等しい値になります。*op*がゼロであれば, 指数部の最小値 `emin`が返されます。*op*が NaN もしくは無限大であれば，範囲エラーフラグがセットされ，*rop*には 0 が入り, 指数部の最小値 `emin`が返されます。返り値の指数部は，実行時における MPFR の最小指数部より小さい値になる可能性があります。指数部が`mpfr_exp_t`型として表現できない値になる場合は，範囲エラーフラグがセットされ，`mpfr_exp_t`型で表現できる最小値が返されます。

`int mpfr_get_z` (*mpz_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
*op*を，*rnd*方向に丸めた後，`mpz_t`型に変換します。*op*が NaN もしくは無限大の時は，範囲エラーフラグを立て，*rop*に 0 を代入し, 0 を返します。それ以外の時は，*rop*が *op*と等しい時 (つまり，*op* が整数の時）はゼロを返し，*op*より大きくなる時は正の値を，*op*より小さくなる時は負の値を返します。また，*rop*が *op*と異なる時，つまり，*op*が整数でない時は，不正確フラグを立てます。

`void mpfr_get_q` (*mpq_t rop, mpfr_t op*) [関数]
*op*を変換し `mpq_t`に格納します。*op*が NaN，もしくは無限大の時は，範囲エラーフラグがセットされ，*rop*にはゼロが代入されます。それ以外の場合は，常に正確な変換が行われます。

`int mpfr_get_f` (*mpf_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
*op*を *rnd*方向に丸めて `mpf_t`型に変換します。*op*が NaN もしくは無限大の時は，これに相当するものは MPF には存在しないので，範囲エラーフラグを立てます。*op*が NaN の時は *rop*の値は不定になります。*op*が+Inf (−Inf) の時は，*rop*は MPF 型変数の精度桁数における最大値 (最小値) になります。将来 MPF が無限大をサポートするようになれば，このような動作は正しくないので，手直しされることでしょう (ポータブルなプログラムを目指すのであれば，*rop* には有限の値か，無限大が代入されるようにすべきです)。現状，MPFR の指数部は MPF と同じデータ型なので（基数は異なりますが)，指数部の範囲は MPF の方が MPFR より大きくなります。従って，オーバーフローやアンダーフローの対応付けはできません。

`char * mpfr_get_str` (*char *str, mpfr_exp_t *expptr, int b, size_t n,*
mpfr_t op, mpfr_rnd_t rnd) [関数]
*op*を基数 $|b|$ の文字列として，丸めモード *rnd*で丸めて変換します。ここで nはゼロ（下記参照)，もしくは文字列として出力される有効桁数を意味します。後者の意味の場合は，nは 2 以

上でなければなりません。基数として使用できるのは 2 以上 62 以下の自然数，もしくは −2 以下 −36 以上の整数です。これ以外の値を基数として指定すると，この関数は何もせず即座に NULL ポインタを返します。

*b*が 2 以上 36 以下である時，数字と小文字を使って文字列を生成します。−2 以下 −36 以上の場合は，数字と大文字を使います。37 以上 62 以下の場合は，数字，大文字，小文字がこの順に使用されます。注意して頂きたいのは，$b > 10$ の時で，9 に続く文字は基数 *b*によって異なります。これは GMP の `mpf_get_str`関数と互換性を持たせるための仕様です。もっとマシな動作をさせたいとお思いでしたら，シンプルなラッパー関数を実装してみて下さい。

入力値が NaN の時は，‘`@NaN@`’という文字列を返し，NaN フラグを立てます。入力値が+Inf (−Inf) の時は，‘`@Inf@`’ (‘`-@Inf@`’) を返します。

入力値が有限値の場合は，指数部は *expptr*ポインタを通じて渡されます。入力値が 0 の時は現時点における最小の指数部の値が書き込まれます。有限値でさえあれば，`mpfr_exp_t`型は任意の指数部の値も格納できます。

生成される文字列は小数形式ですが，小数点は明示せず，文字列の先頭（最大桁の左側) に置かれているものとしています。例えば，−3.1416 という数に対しては，`"−31416"`という文字列が生成され，*expptr*が指す先には 1 が格納されます。丸めモード *rnd*が最近接丸めで，*op*がちょうど出力すべき同じ指数部を持つ近似値の左右候補のど真ん中に位置している場合は，偶数になる近似値を採用します。基数が奇数の場合は，最小桁が偶数にできない場合があることに留意して下さい。例えば，基数が 7 で有効桁数が 2 桁の場合，7 進数 (14) と (7 進表現の) 半分は丸められて (15)(10 進数で 12) になりますし 7 進数で (16) と半分となり，丸められて (20)(10 進数で 14) になります。同様に，7 進数の (26) と半分は丸められて (26)(10 進数の 20) になります。

*n*がゼロの時は，仮数部が完全に収まるだけの桁数を確保しますので，出力された文字列をすべて再読み込みすれば，入出力ともに最近接丸めモードの場合は，元の *op*と完全に同じ数を表現できます。正確に言うと，ほとんどの場合，*str*の桁数は，上記の性質を満足する $p = \mathrm{PREC}(op)$ と *b*にのみ依存して決まる最小の桁数 *m* になります。つまり，$m = 1 + \left\lceil p \frac{\log 2}{\log b} \right\rceil$, ということになります。ここで *b*が 2 のべき乗の場合は *p*は $p-1$ に置き換えて下さい。但し，レアケースですが，$m + 1$ 桁になる場合もあります。基数が 62 以下の最小の事例としては，基数が 7 と 49 で，*p*が 186564318007 の時がそれにあたります。

*str*が NULL ポインタの時も，仮数部用の文字列メモリ領域はメモリ割り当て関数 (Section 4.7 [Memory Handling], 頁 12 参照) で確保され，基数が無効な値でなければ，その文字列へのポインタが返り値となります。この場合，返されたポインタが指す文字列を解放するためには必ず `mpfr_free_str`関数を使って下さい。

*str*が NULL ポインタでない限り， ポインタが指すメモリ領域は仮数部を格納できる十分な大きさが確保されていなければなりません。どんな値に対しても安全なサイズは，*n*がゼロでない場合は `max(n + 2, 7)`，*n*がゼロの場合は，前述の説明の通り $m + 1$ 確保しておく必要があります。2 バイト分余計に確保するのは，終端子用の NULL 文字に加えて，マイナス符号が付加される可能性があるからです。最低でも 7 バイト確保する理由は，‘`-@Inf@`’を NULL 終端子含めて表現するためです。基数が無効な値でない限り，文字列 *str*へのポインタが返り値となります。

他の関数同様，変換された文字列が丸められて誤差を含む時には，不正確フラグを立てます。

`void mpfr_free_str (`*char *str*`)` [関数]

`mpfr_get_str`関数で割り当てられた文字列を，メモリ解放関数 (Section 4.7 [Memory Handling], 頁 12 参照) で解放します。このメモリブロックは `strlen(str)+1`バイト確保されているものと想定されています。

int mpfr_fits_ulong_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
int mpfr_fits_slong_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
int mpfr_fits_uint_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
int mpfr_fits_sint_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
int mpfr_fits_ushort_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
int mpfr_fits_sshort_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
int mpfr_fits_uintmax_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
int mpfr_fits_intmax_p (*mpfr_t op*, *mpfr_rnd_t rnd*) [関数]

*op*を *rnd*方向に丸めた結果が，それぞれunsigned long型，long型，unsigned int型，int型，unsigned short型，short型，uintmax_t型，intmax_t型といった C の基本データ型に正確に変換できる時には非ゼロを返します。例えば，−0.5 を MPFR_RNDUモードで丸めた場合，上記のどの関数でも非ゼロを返します。MPFR_RNDFモードで丸めた場合は，上記の関数と対応する変換関数 (例えば，mpfr_fits_ulong_p関数に対しては mpfr_get_ui関数）が忠実丸めモードで対応するデータ型で表現できる数になるなら，非ゼロ値を返します。従って，MPFR_RNDFモードの場合は，mpfr_fits_ulong_p関数は， ULONG_MAX以下の非負値に対しては，非ゼロを返します。

5.5 基本演算関数

int mpfr_add (*mpfr_t rop*, *mpfr_t op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_add_ui (*mpfr_t rop*, *mpfr_t op1*, *unsigned long int op2*, [関数]
mpfr_rnd_t rnd)
int mpfr_add_si (*mpfr_t rop*, *mpfr_t op1*, *long int op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_add_d (*mpfr_t rop*, *mpfr_t op1*, *double op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_add_z (*mpfr_t rop*, *mpfr_t op1*, *mpz_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_add_q (*mpfr_t rop*, *mpfr_t op1*, *mpq_t op2*, *mpfr_rnd_t rnd*) [関数]

op1 + *op2* を求めて *rnd*方向に丸め，*rop*に代入します。符号付きゼロの場合は IEEE 754 のルールが適用されます。符号なしゼロの場合も，0 は符号付きとして処理されます。つまり (+0) + 0 = (+0) や，(−0) + 0 = (−0)) となります。mpfr_add_d関数においては，double型は基数が 2 のべき乗，精度桁数は C の実装値 (IEEE_DBL_MANT_DIGマクロ定義値，なければ 53 ビット) として処理を行います。

int mpfr_sub (*mpfr_t rop*, *mpfr_t op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_ui_sub (*mpfr_t rop*, *unsigned long int op1*, *mpfr_t op2*, [関数]
mpfr_rnd_t rnd)
int mpfr_sub_ui (*mpfr_t rop*, *mpfr_t op1*, *unsigned long int op2*, [関数]
mpfr_rnd_t rnd)
int mpfr_si_sub (*mpfr_t rop*, *long int op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_sub_si (*mpfr_t rop*, *mpfr_t op1*, *long int op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_d_sub (*mpfr_t rop*, *double op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_sub_d (*mpfr_t rop*, *mpfr_t op1*, *double op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_z_sub (*mpfr_t rop*, *mpz_t op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_sub_z (*mpfr_t rop*, *mpfr_t op1*, *mpz_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_sub_q (*mpfr_t rop*, *mpfr_t op1*, *mpq_t op2*, *mpfr_rnd_t rnd*) [関数]

op1 − *op2* を計算して *rnd*方向に丸め，*rop*に代入します。IEEE 754 規格のルールに従い，符号付きゼロも正しく扱います。符号なしゼロの場合は，(+0) − 0 = (+0), (−0) − 0 = (−0), 0 − (+0) = (−0)，0 − (−0) = (+0)) というように扱います。mpfr_add_d関数に対する制限は，mpfr_d_sub関数に対しても，mpfr_sub_d関数に対しても，より厳格に適用されています。

int mpfr_mul (*mpfr_t rop*, *mpfr_t op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
int mpfr_mul_ui (*mpfr_t rop*, *mpfr_t op1*, *unsigned long int op2*, [関数]
mpfr_rnd_t rnd)
int mpfr_mul_si (*mpfr_t rop*, *mpfr_t op1*, *long int op2*, *mpfr_rnd_t rnd*) [関数]

`int mpfr_mul_d` (*mpfr_t* **rop**, *mpfr_t* **op1**, *double* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_mul_z` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpz_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_mul_q` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpq_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
op1 × *op2* を計算し，丸めモード *rnd*で丸めて *rop*に格納します。計算結果がゼロになる時の符号は，二数の符号の積で決定されます。符号なしゼロはプラス符号と判断します。`mpfr_add_d` 関数と同様の制限が`mpfr_mul_d`にも課されます。

`int mpfr_sqr` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
op^2 を計算し，*rnd*方式で丸めて *rop*に代入します。

`int mpfr_div` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_ui_div` (*mpfr_t* **rop**, *unsigned long int* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_div_ui` (*mpfr_t* **rop**, *mpfr_t* **op1**, *unsigned long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_si_div` (*mpfr_t* **rop**, *long int* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_div_si` (*mpfr_t* **rop**, *mpfr_t* **op1**, *long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_d_div` (*mpfr_t* **rop**, *double* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_div_d` (*mpfr_t* **rop**, *mpfr_t* **op1**, *double* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_div_z` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpz_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_div_q` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpq_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
op1/*op2* を求め，丸め方式 *rnd*で丸めて *rop*に代入します。演算結果がゼロの時は，二数の符号の積で符号が決定されます。符号なしゼロの時は，プラス 0 として扱います。*op1*が非ゼロで，*op2*がゼロの時，演算結果は今のところ ±Inf ですが，将来的には NaN になるかもしれません。IEEE 754 で正反対の決定をしなければ，ですが。`mpfr_add_d`関数における制限事項は，`mpfr_d_div`関数にも，`mpfr_div_d`関数にも適用されます。

`int mpfr_sqrt` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_sqrt_ui` (*mpfr_t* **rop**, *unsigned long int* **op**, *mpfr_rnd_t* **rnd**) [関数]
$\sqrt{op}$ を求め，*rnd*方式で丸めて *rop*に代入します。*op*が −0 の時は，IEEE754 規格の定めに従って処理します。*op*が負の時は，*rop*には NaN が代入されます。

`int mpfr_rec_sqrt` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
$1/\sqrt{op}$ を計算し，*rnd*方向に丸めて *rop*に代入します。*op*が ±0 の時は *rop*には+Inf が，*op*が+0 の時は+Inf が, *op*が負数の時は NaN が，それぞれ代入されます。

[注意] −0 に対する値は+Inf で，IEEE 754-2008 規格 (9.2.1 節) で推奨されている rSqrt 関数の値である-Inf ではありません。

`int mpfr_cbrt` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_rootn_ui` (*mpfr_t* **rop**, *mpfr_t* **op**, *unsigned long int* **k**, *mpfr_rnd_t* **rnd**) [関数]
*op*の立方根 (*k*乗根) を計算し，*rnd*方式で丸めて *rop*に代入します。$k = 0$ の場合は，*rop*に NaN を代入します。*k*が奇数 (偶数) で，*op*が (−Inf も含む) 負数である時，*rop*には，*k*のゼロではない) 値に関わらず，負数 (NaN) が代入されます。*op*がゼロの時は，*rop*には通常の極限値ルールに則った符号付けがなされたゼロが代入されます。つまり， *k*が奇数の時は *op*と同じ符号になり，偶数の時はプラス符号が付きます。

これらの関数は，IEEE 754-2008 規格 (9.2 節) の rootn 関数の処理に則った動作を行います。

`int mpfr_root` (*mpfr_t* **rop**, *mpfr_t* **op**, *unsigned long int* **k**, *mpfr_rnd_t* **rnd**) [関数]
この関数は`mpfr_rootn_ui`関数と同じ機能を持ちます。但し，*op* が −0 で *k*が偶数の時は，+0 ではなく −0 を返すところが異なります。これは`mpfr_sqrt`関数と齟齬が出ないようにするための措置です。*op*がゼロの時は，*rop*に *op*の値をそのまま代入します。

この関数はIEEE 754-2008規格が固まる前に作られたものなので，規格が定めたrootn関数とは動作が多少異なります。将来のバージョンで廃止予定の関数です。

`int mpfr_pow` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_pow_ui` (*mpfr_t* **rop**, *mpfr_t* **op1**, *unsigned long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_pow_si` (*mpfr_t* **rop**, *mpfr_t* **op1**, *long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_pow_z` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpz_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_ui_pow_ui` (*mpfr_t* **rop**, *unsigned long int* **op1**, *unsigned long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_ui_pow` (*mpfr_t* **rop**, *unsigned long int* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]

op1^{op2} を計算し，*rnd*方式で丸めて *rop*に代入します。特殊値に対しては，ISO C99 及びIEEE 754-2008規格が定めるpow関数に準じた振る舞いをします。

- pow(± 0, *y*)は，*y*が負の奇数である時，正の無限大，もしくは負の無限大を返します。
- pow(± 0, *y*)は，*y*が負の偶数である時，正の無限大を返します。
- pow(± 0, *y*)は，*y*が正の奇数である時，正のゼロ，もしくは負のゼロを返します。
- pow(± 0, *y*)は，*y*が正の偶数である時，正のゼロを返します。
- pow(-1, ± Inf)は 1 を返します。
- pow(+1, *y*)は， NaN を含む任意の *y*に対して，1 を返します。
- pow(*x*, ± 0)は，NaN を含む任意の *y*に対して，1 を返します。
- pow(*x*, *y*)は，有限の負数 *x*と有限の非整数 *y*に対して，NaN を返します。
- pow(*x*, -Inf)は，$0 < |x| < 1$ の時は正の無限大を，$|x| > 1$ の時は正のゼロを返します。
- pow(*x*, +Inf)は，$0 < |x| < 1$ の時は正のゼロを，$|x| > 1$ の時は正の無限大を返します。
- pow(-Inf, *y*)は，*y*が負の奇数の時，負のゼロを返します。
- pow(-Inf, *y*)は，*y*が負の偶数の時，正のゼロを返します。
- pow(-Inf, *y*)は，*y*が正の奇数の時，負の無限大を返します。
- pow(-Inf, *y*)は，*y*が正の偶数の時，正の無限大を返します。
- pow(+Inf, *y*)は，*y*が負数の時は正のゼロを，*y*が正数の時は正の無限大を返します。

[注記] 整数型の 0 は，上記の関数では+0 と扱います。この場合，pow関数と同様，通常の極限値としては考えません。

`int mpfr_neg` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_abs` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]

それぞれ，$-op$ と *op*の絶対値を，*rnd*方式で丸めて *rop*に代入します。*rop*と *op*が同じ変数である時には，符号を必要に応じて変更し，異なる変数であれば，*rop* の精度桁数が *op*より小さい時には丸め処理が実施されます。

このような符号処理の方式は，IEEE 754 の negate関数と abs関数を真似して，NaN に対しても行われます。つまり，mpfr_neg関数は符号を反転し，mpfr_abs関数は符号を正にします。但し，IEEE 754 とは違って，通常 NaN フラグは立てます。

`int mpfr_dim` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]

*op1*と *op2*の正定値 (positive difference) を計算し，*rnd*方式で丸めて *rop*に代入します。つまり，$op1 > op2$ の場合は $op1 - op2$ を求め，$op1 \leq op2$ の時は+0 を代入し，*op1*もしくは *op2* が NaN の時は NaN を代入します。

`int mpfr_mul_2ui` (*mpfr_t* **rop**, *mpfr_t* **op1**, *unsigned long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_mul_2si` (*mpfr_t* **rop**, *mpfr_t* **op1**, *long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
op1 × 2^{op2} を計算し，*rnd*方式で丸めて *rop*に代入します。*rop*と *op1*が同一の変数である場合は，2 の *op2*乗分増えていきます。

`int mpfr_div_2ui` (*mpfr_t* **rop**, *mpfr_t* **op1**, *unsigned long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_div_2si` (*mpfr_t* **rop**, *mpfr_t* **op1**, *long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
op1/2^{op2} を求め，*rnd*方式で丸めて *rop*に代入します。*rop*と *op1*が同一の変数である場合は，2 の *op2*乗分減っていきます。

5.6 比較関数

`int mpfr_cmp` (*mpfr_t* **op1**, *mpfr_t* **op2**) [関数]
`int mpfr_cmp_ui` (*mpfr_t* **op1**, *unsigned long int* **op2**) [関数]
`int mpfr_cmp_si` (*mpfr_t* **op1**, *long int* **op2**) [関数]
`int mpfr_cmp_d` (*mpfr_t* **op1**, *double* **op2**) [関数]
`int mpfr_cmp_ld` (*mpfr_t* **op1**, *long double* **op2**) [関数]
`int mpfr_cmp_z` (*mpfr_t* **op1**, *mpz_t* **op2**) [関数]
`int mpfr_cmp_q` (*mpfr_t* **op1**, *mpq_t* **op2**) [関数]
`int mpfr_cmp_f` (*mpfr_t* **op1**, *mpf_t* **op2**) [関数]
*op1*と *op2*を比較します。*op1* > *op2* の時は正の値を，*op1* = *op2* の時はゼロを，*op1* < *op2* の時は負数を返します。*op1*と *op2*がどちらも同じ精度桁数フルで値が入っている場合は，差を取って判断します。どちらかの値が NaN の時は，範囲エラーフラグが立てられて，ゼロを返します。

[注記] この比較関数は 3 種類のケースを見分けることができます。2 種類だけを見分けたい場合は，後述するように，先読み関数 (例えば`mpfr_equal_p`関数は二数が等しいかどうか判断できます) を使って下さい。比較される値に NaN がある場合は，IEEE 754 規格の比較と同じ振る舞いをします。比較は浮動小数点数のみ対象なので，必要に応じてデータ変換も行われます。

`int mpfr_cmp_ui_2exp` (*mpfr_t* **op1**, *unsigned long int* **op2**, *mpfr_exp_t* **e**) [関数]
`int mpfr_cmp_si_2exp` (*mpfr_t* **op1**, *long int* **op2**, *mpfr_exp_t* **e**) [関数]
*op1*と *op2* × 2^e を比較します。返り値は上記の関数と同じです。

`int mpfr_cmpabs` (*mpfr_t* **op1**, *mpfr_t* **op2**) [関数]
|*op1*| と |*op2*| を比較します。|*op1*| > |*op2*| の時は正の値を，|*op1*| = |*op2*| の時はゼロを，|*op1*| < |*op2*| の時は負の値を返します。比較対象の値に NaN がある場合は，範囲エラーフラグを立ててゼロを返します。

`int mpfr_nan_p` (*mpfr_t* **op**) [関数]
`int mpfr_inf_p` (*mpfr_t* **op**) [関数]
`int mpfr_number_p` (*mpfr_t* **op**) [関数]
`int mpfr_zero_p` (*mpfr_t* **op**) [関数]
`int mpfr_regular_p` (*mpfr_t* **op**) [関数]
*op*がそれぞれ NaN，無限大，NaN でも無限大でもない浮動小数点数，ゼロ，NaN でも無限大でもゼロでもない浮動小数点数である場合は，非ゼロ数を返します。それ以外の場合はゼロを返します。

`int mpfr_sgn` (*mpfr_t op*) [Macro]
op > 0 の時は正の値を，*op* = 0 の時はゼロを，*op* < 0 の時は負の値を返します。引数が NaN の場合は，範囲エラーフラグを立ててゼロを返します。`mpfr_cmp_ui (op, 0)`と同じ働きをしますが，このマクロの方が高速に実行できます。

`int mpfr_greater_p` (*mpfr_t op1*, *mpfr_t op2*) [関数]
`int mpfr_greaterequal_p` (*mpfr_t op1*, *mpfr_t op2*) [関数]
`int mpfr_less_p` (*mpfr_t op1*, *mpfr_t op2*) [関数]
`int mpfr_lessequal_p` (*mpfr_t op1*, *mpfr_t op2*) [関数]
`int mpfr_equal_p` (*mpfr_t op1*, *mpfr_t op2*) [関数]
それぞれ *op1* > *op2*, *op1* ≥ *op2*, *op1* < *op2*, *op1* ≤ *op2*, *op1* = *op2* の場合は非ゼロ数を，それ以外の場合はゼロを返します。*op1*と *op2*の一つ以上が NaN であれば，必ずゼロを返します。

`int mpfr_lessgreater_p` (*mpfr_t op1*, *mpfr_t op2*) [関数]
op1 < *op2* もしくは *op1* > *op2* であれば非ゼロを返します。これは *op1*も *op2*も NaN ではなく，*op1* ≠ *op2* である場合と同等です。それ以外の場合，つまり *op1* と *op2*に NaN がある場合，もしくは *op1* = *op2* であれば，ゼロを返します。

`int mpfr_unordered_p` (*mpfr_t op1*, *mpfr_t op2*) [関数]
*op1*もしくは *op2*が NaN であれば，つまり両者の比較ができない場合は非ゼロを返します。それ以外の場合はゼロを返します。

5.7 初等関数・特殊関数

一部の例外 (`mpfr_sin_cos`関数など) を除き，ここで述べている全ての関数の返り値は三種値 [ternary value], 頁 9 を返します。丸めなしの真値を返す場合は 0 を，真値より大きい値を返す場合は正の値を，それ以外の場合は負の値を返します。

[重要] 引数がある領域にある場合，要求精度桁数が小さくても，初等関数や特殊関数の計算 (ことに，正確な丸め処理を行うため）に時間がかかることがあります。例えば三角関数やベッセル関数の引数が大きい場合がそれにあたります。他にも，関数によっては，メモリの使用量が必ずしも出力する値の精度桁数に依存しないこともあります。`mpfr_rootn_ui`関数は引数 *k*の大きさに比例し，不完全ガンマ関数は引数 *op*に比例してメモリ使用量が増えます。

`int mpfr_log` (*mpfr_t rop*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
`int mpfr_log_ui` (*mpfr_t rop*, *unsigned long op*, *mpfr_rnd_t rnd*) [関数]
`int mpfr_log2` (*mpfr_t rop*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
`int mpfr_log10` (*mpfr_t rop*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
上から順に，*op*の自然対数，$\log_2 op$，$\log_{10} op$ を計算し，丸め方式 *rnd*で丸めて *rop*に代入します。丸め方式に関わらず，*op*が 1 の時は，*rop*は+0 になります。これは ISO C99 規格と IEEE 754-2008 標準規格に基づいたものです。*op*が ±0 の時，即ち，ゼロの符号が結果に何ら影響がない場合は，*rop*には −Inf が代入されます。

`int mpfr_log1p` (*mpfr_t rop*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
op+1 の自然対数の計算を行い，丸め方式 *rnd*で丸めて *rop*に格納します。*op*が −1 の時は，*rop*には −Inf が代入されます。

`int mpfr_exp` (*mpfr_t rop*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
`int mpfr_exp2` (*mpfr_t rop*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
`int mpfr_exp10` (*mpfr_t rop*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]
この二つの関数は，それぞれ *op*の 2 のべき乗 (2^{op})，もしくは 10 のべき乗 (10^{op}) を計算し，丸めモード *rnd*で丸め，*rop*に代入します。

`int mpfr_expm1` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
$e^{op} - 1$ を求め，丸めモード *rnd*で丸め，*rop*に代入します。

`int mpfr_cos` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_sin` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_tan` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
それぞれ三角関数 $\sin(op)$, $\cos(op)$, $\tan(op)$ の値を求め，*rnd*方向に丸め，*rop*に代入します。

`int mpfr_sin_cos` (*mpfr_t* **sop**, *mpfr_t* **cop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*sop*には $\sin(op)$ の値を，*cop*には $\cos(op)$ の値を，*sop*, *cop*の精度桁数に収まるよう，それぞれ *rnd*方向に丸めて同時に代入します。*sop*と *cop*は異なる変数でなければなりません。両方の値が丸めなしで収まる時は 0 を返します。より正確に言うと，返り値は $s + 4c$ という式で表現され，*sop*が丸めなしの場合は $s = 0$ となり，丸めた近似値が大きくなる時は $s = 1$，小さくなる時は $s = 2$ となります。c の値も，*cop*の丸めた結果に応じて s 同様に決まります。

`int mpfr_sec` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_csc` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_cot` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
$\sec(op)$, $\mathrm{cosec}(op)$, $\cot(op)$ の値を求め，*rnd*方式で丸めて *rop*に代入します。

`int mpfr_acos` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_asin` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_atan` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の逆三角関数の値，$\arccos(op)$，$\arcsin(op)$，$\arctan(op)$ を計算し，*rnd*方式で丸めて *rop*に代入します。`acos(-1)`は，与えられた丸めモードに従って π に近い浮動小数点数を求めるので，必ずしも arccos の定義による $0 \le rop < \pi$ という範囲に収まるわけではありません。求めた結果は，丸め方式によって出力範囲が決まります。同様に，`asin(-1)`, `asin(1)`, `atan(-Inf)`, `atan(+Inf)`，大きい *op*に対して `atan(op)`の代入先の *rop*の精度桁数が少ない時，等についても同様のことが言えます。

`int mpfr_atan2` (*mpfr_t* **rop**, *mpfr_t* **y**, *mpfr_t* **x**, *mpfr_rnd_t* **rnd**) [関数]
アークタンジェント $2(\mathrm{atan2}(y,\ x))$ の値を求め，*rnd*方式で丸めて *rop*に格納します。`x > 0`の時は `atan2(y, x) = atan(y/x)`となります。`x < 0`の時は `atan2(y, x) = sign(y)*(Pi - atan(|y/x|))`となりますので，計算結果は，$-\pi$ 以上 π 以下の範囲に収まります。`atan`関数同様，数学的な定義では $+\pi$ もしくは $-\pi$ となることがありますが，丸めの結果，数学的な値域に収まらないこともあり得ます。

`atan2(y, 0)`は浮動小数点演算例外を発生させません。特殊な入力値に対しては ISO C99 及び IEEE 754-2008 規格の `atan2`関数同様，次のように値が決まります。

- `atan2(+0, -0)`は $+\pi$ を返す。
- `atan2(-0, -0)`は $-\pi$ を返す。
- `atan2(+0, +0)`は+0 を返す。
- `atan2(-0, +0)`は -0 を返す。
- `atan2(+0, x)`は，$x < 0$ の場合は $+\pi$ を返す。
- `atan2(-0, x)`は，$x < 0$ の場合は $-\pi$ を返す。
- `atan2(+0, x)`は，$x > 0$ の場合は+0 を返す。
- `atan2(-0, x)`は，$x > 0$ の場合は -0 を返す。
- `atan2(y, 0)`は，$y < 0$ の場合は $-\pi/2$ を返す。
- `atan2(y, 0)`は，$y > 0$ の場合は $+\pi/2$ を返す。
- `atan2(+Inf, -Inf)`は $+3\pi/4$ を返す。

- `atan2(-Inf, -Inf)`は $-3\pi/4$ を返す。
- `atan2(+Inf, +Inf)`は $+\pi/4$ を返す。
- `atan2(-Inf, +Inf)`は $-\pi/4$ を返す。
- `atan2(+Inf, x)`は，有限の x の場合は $+\pi/2$ を返す。
- `atan2(-Inf, x)`は，有限の x の場合は $-\pi/2$ を返す。
- `atan2(y, -Inf)`は，有限の $y > 0$ の場合は $+\pi$ を返す。
- `atan2(y, -Inf)`は，有限の $y < 0$ の場合は $-\pi$ を返す。
- `atan2(y, +Inf)`は，有限の $y > 0$ の場合は+0 を返す。
- `atan2(y, +Inf)`は，有限の $y < 0$ の場合は -0 を返す。

`int mpfr_cosh` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_sinh` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_tanh` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
双曲線関数の値，cosh(*op*), sinh(*op*)，tanh(*op*) の値を求め，*rnd*方式で丸めて *rop*に代入します。

`int mpfr_sinh_cosh` (*mpfr_t* **sop**, *mpfr_t* **cop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
双曲線関数 sinh(*op*) と cosh(*op*) の値を同時に求め，それぞれ *sop*と *cop*の精度桁数に収まるように *rnd*方式で丸めて代入します。この時，*sop*と *cop*は必ずそれぞれ異なる変数を指定して下さい。両方の出力値が丸めなしで正しい値になる場合のみ 0 を返します。返り値の詳細については`mpfr_sin_cos`関数の説明を参照して下さい。

`int mpfr_sech` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_csch` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_coth` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
双曲線関数，sech(*op*)，cosech(*op*)，coth(*op*) の値を求め，*rnd*方式で丸めて *rop*に代入します。

`int mpfr_acosh` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_asinh` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_atanh` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
逆双曲線関数，arccosh(*op*)，arcsinh(*op*)，arctanh(*op*) の値をそれぞれ求めて，*rnd*方式で丸めて *rop*に代入します。

`int mpfr_fac_ui` (*mpfr_t* **rop**, *unsigned long int* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の階乗を求め，*rnd*方式で丸めて *rop*に代入します。

`int mpfr_eint` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
指数積分値 を求め，*rnd*方式で丸めて *rop*に代入します。数学的な定義は，オイラー定数，*op* の絶対値の自然対数 ($\log(|op|)$，k に関する無限和 $op^k/(k \cdot k!)$ の和となります。*op*が正の時は Ei(*op*) の値を返し (Abramowitz and Stegun:"Handbook of Mathematical Functions"の 5.1.10 式参照) ，*op*が負の時は E1(-*op*) の値 (eint1(-*op*) とも書く) を返します (同著 5.1.1 式を参照)。

`int mpfr_li2` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
多重対数関数 Li2(*op*) の値を求め，*rnd*方式で丸めて *rop*に代入します。MPFR では $-\int_{t=0}^{op} \log(1-t)/t \; dt$ という定義を多重対数関数として採用しています。

`int mpfr_gamma` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_gamma_inc` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
*op*のガンマ関数と *op*と *op2*の不完全ガンマ関数の値を計算し，*rnd*方式で丸めて *rop*に代入します。このマニュアルでは`mpfr_gamma_inc`を不完全ガンマ関数 (incomplete Gamma function)

と呼びますが，補不完全ガンマ関数 (complementary incomplete Gamma function) と呼ぶこともあります。mpfr_gamma関数と，*op2*がゼロの時の mpfr_gamma_inc関数は，*op*が負の整数の時，*rop*に NaN を代入します。

[注記] 現状の mpfr_gamma_inc関数は，*rop*や *op*が大きい場合は低速になってしまう問題があります。また，時によっては内部の計算でオーバーフローが発生することがあります。

int mpfr_lngamma (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の対数ガンマ関数の値を求め，*rnd*方式で丸めて *rop*に代入します。*op*が 1 ないし 2 の時は，丸めモードに関わらず *rop*は+0 になります。*op*が無限大，もしくは正でない整数の時は，*rop*は+Inf になります。このあたりの特殊な引数に対する対応は一般的なルールに従っています。$-2k-1 < op < -2k$ の時は，kが非負の整数であれば，*rop*は NaN になります。mpfr_lgamma 関数の解説も参照して下さい。

int mpfr_lgamma (*mpfr_t* **rop**, *int* ***signp**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*のガンマ関数の値の絶対値の対数を求め，*rnd*方式に丸めて *rop*に代入します。*op*のガンマ関数 ($\Gamma(op)$) の値の符号は 1 か -1 として表現され，*signp*ポインタが指す先に格納されています。*op*が 1 ないし 2 の時は，丸めモードに関わらず *rop*は+0 になります。*op*が無限大，もしくは非正の整数であれば，*rop*は+Inf になります。*op*が NaN, −Inf，負の整数のいずれかであれば，**signp*の指す値は不定値となります。*op*が ± 0 の時は，**signp*の指す値はゼロの符号を意味するものになります。

int mpfr_digamma (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*のディガンマ関数 (プサイ関数とも呼ばれる) の値を求め，*rnd*方式で丸め，*rop*に代入します。*op*が負の整数の場合は，*rop*は NaN になります。

int mpfr_beta (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
引数が *op1*と *op2*のベータ関数の値を求め，*rop*に代入します。

[注記] 現時点の実装では内部でオーバーフローやアンダーフローが起きるケースについて何ら対処を行っていませんので，超高精度な値を内部で使うとトラブるかもしれません。

int mpfr_zeta (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
int mpfr_zeta_ui (*mpfr_t* **rop**, *unsigned long* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*のリーマン・ゼータ関数の値を求め，*rnd*方式で丸め，*rop*に代入します。

int mpfr_erf (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
int mpfr_erfc (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の誤差関数および相補誤差関数の値を求め，*rnd*方向に丸めて，*rop*に代入します。

int mpfr_j0 (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
int mpfr_j1 (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
int mpfr_jn (*mpfr_t* **rop**, *long* **n**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の 0, 1, *n*次の第 1 種ベッセル関数の値を求め，*rnd*方式で丸めて *rop*に代入します。*op*が NaN 場合は，*rop*は常に NaN になります。*op*が+Inf もしくは-Inf の時は *rop*は+0 になります。*rop*が+0 でかつ *n*が非ゼロの時は，*n*の偶奇や符号によって *rop*の値は+0 もしくは −0 になります。

int mpfr_y0 (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
int mpfr_y1 (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
int mpfr_yn (*mpfr_t* **rop**, *long* **n**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の 0, 1, *n*次の第 2 種ベッセル関数の値を求め，*rnd*方式で丸めて *rop*に代入します。*op*が NaN もしくは負の場合は，*rop*は常に NaN になります。*op*が+Inf の時は *rop*は+0 になります。，*op*がゼロの時は，*n*の偶奇や符号によって *rop*の値は+Inf もしくは −Inf になります。

int mpfr_fma (*mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_t op3, mpfr_rnd_t rnd*) [関数]
int mpfr_fms (*mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_t op3, mpfr_rnd_t rnd*) [関数]
$(op1 \times op2) + op3$ や，$(op1 \times op2) - op3$ の値を求め，*rnd*方式で丸め，*rop*に代入します。特別な値 (符号付きゼロ，無限大，NaN) がこの中に入っている場合，加減算の後に行う乗算の流儀に従って値が決定します。つまり，複合演算としての意味は丸め処理だけになります。

int mpfr_fmma (*mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_t op3, mpfr_t op4, mpfr_rnd_t rnd*) [関数]
int mpfr_fmms (*mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_t op3, mpfr_t op4, mpfr_rnd_t rnd*) [関数]
$(op1 \times op2) + (op3 \times op4)$ や $(op1 \times op2) - (op3 \times op4)$ の値を求め，*rnd*方式で丸めて *rop*に代入します。$op1 \times op2$ や $op3 \times op4$ の計算でオーバーフローやアンダーフローが発生した場合は，この両者を繋ぐ乗算によって *rop*の値が決まります。

int mpfr_agm (*mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd*) [関数]
*op1*と *op2*の算術幾何平均を求め，*rnd*方式で丸め，*rop*に代入します。算術幾何平均とは，数列 u_n と v_n の共通の極限値で，u_0=*op1*, v_0=*op2*という初期値から出発し，u_{n+1} は u_n と v_n の算術平均，v_{n+1} は u_n と v_n の幾何平均として計算したものです。*op1*，*op2*のどちらも負か，片方が非負である時は，*rop*は NaN になります。*op1*，*op2*のどちらもゼロか，片方が有限値 (無限大) の時は，*rop*は+0 (NaN) になります。

int mpfr_hypot (*mpfr_t rop, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd*) [関数]
*x*と *y*のユークリッドノルム，即ち $\sqrt{x^2 + y^2}$ を求め，*rnd*方式で丸めて *rop*に代入します。特殊値の場合は ISO C99 (F.9.4.3 節) および IEEE 754-2008 (9.2.1 節) に述べられている通りに処理されます。 つまり，*x*もしくは *y*が無限大の時は，+Inf が *rop*に代入され，それ以外の特殊値の場合は NaN が代入されます。

int mpfr_ai (*mpfr_t rop, mpfr_t x, mpfr_rnd_t rnd*) [関数]
アーリー関数 Ai(x) の値を求め，*rnd*方式で丸め，*rop*に代入します。*x*が NaN の時は，*rop*は常に NaN になります。*x*が+Inf もしくは −Inf の時は，*rop*は+0 になります。現状の実装では，引数が大きな値になることが想定されておらず，$|x|$ は 500 より十分小さい値でないとうまく計算できません。大きな引数になる場合は他の方法を使用するか，未来のバージョンの登場をお待ち下さい。

int mpfr_const_log2 (*mpfr_t rop, mpfr_rnd_t rnd*) [関数]
int mpfr_const_pi (*mpfr_t rop, mpfr_rnd_t rnd*) [関数]
int mpfr_const_euler (*mpfr_t rop, mpfr_rnd_t rnd*) [関数]
int mpfr_const_catalan (*mpfr_t rop, mpfr_rnd_t rnd*) [関数]
*rop*に，2 の自然対数，π，オイラー定数 0.577...，カタラン定数 0.915... をそれぞれ *rnd*方式で丸めて代入します。一度これらの値がキャッシュされると，これと同じか，より低い精度桁数の値の要求があっても再計算を行いません。キャッシュをクリアするには，mpfr_free_cache 関数か mpfr_free_cache2関数を使って下さい。

void mpfr_free_cache (*void*) [関数]
MPFR の内部で使用される，キャッシュやプールを全て消去します。スレッドローカルのものや，全てのスレッドで共有されているものが対象となります。明示的に mpfr_const_*関数を呼び出していなくても，MPFR の内部で利用していることがあるので，スレッドを停止させる前にはこの関数を呼び出すようにして下さい。

`void mpfr_free_cache2` (*mpfr_free_cache_t* `way`) [関数]
MPFR の内部で使用しているキャッシュやプールを *way*のフラグ値で指定した方法で消去します。このフラグの設定には以下のものが使えます。

- `MPFR_FREE_LOCAL_CACHE`フラグが立っている場合は，現在のスレッドにおけるローカルキャッシュやプールを消去
- `MPFR_FREE_GLOBAL_CACHE`フラグが立っている場合は，全てのスレッドから共有されているキャッシュやプールを消去

上記以外の *way*に立っているフラグは現状では無視されます。将来は使用するかもしれませんので，その他のフラグは立てずにゼロにしておくようにしましょう。

[注記] `mpfr_free_cache2(MPFR_FREE_LOCAL_CACHE|MPFR_FREE_GLOBAL_CACHE)`は今のところ，`mpfr_free_cache()`と同じ働きをします。

`void mpfr_free_pool` (*void*) [関数]
MPFR の内部で使用されているプールを消去します。

[注記] この関数は，`mpfr_free_cache`関数や `mpfr_free_cache2`関数でスレッドローカルなキャッシュを消去した後には自動的に呼ばれます。

`int mpfr_mp_memory_cleanup` (*void*) [関数]
`mp_set_memory_functions`関数を呼ぶ前にこの関数を呼ぶようにして下さい。詳細については Section 4.7 [Memory Handling], 頁 12 をご覧下さい。処理が成功した時はゼロを，エラーが発生した時には非ゼロを返します。エラーが発生することは現状ない筈ですが，将来もきちんと動作させたいのであれば，返り値をチェックすることをお勧めします。

`int mpfr_sum` (*mpfr_t* `rop`, *const mpfr_ptr* `tab`[], *unsigned long int* `n`, *mpfr_rnd_t* `rnd`) [関数]
*n*個の *tab*要素の全ての和を求め，*rnd*方式で丸めて *rop*に代入します。

[警告] 動作を高速化するために *tab*は `mpfr_t`を要素とする配列へのポインタとなっており，`mpfr_t`の配列そのものではありません。$n = 0$ の時は+0 となり，$n = 1$ の時は `mpfr_set`関数と同じ動作を行います。誤差なしで和が求められる特殊ケースの場合のみ，通常の加算処理 (`mpfr_add`関数) を無限桁で逐次行った時と同じ値を得ることができます。特に，$n \geq 1$ で，正確な和がゼロになる場合は次のようになります。

- 全ての入力値が同じ符号を持つ，つまり全て+0 もしくは -0 になる時などは，演算結果は入力値と同じ符号を持ちます。
- 上記以外の時，入力値はゼロだが+0 と -0 の組み合わせが一つ以上あったり，幾つかの入力値が非ゼロで，計算の結果打ち消されたりする時は，演算結果は+0 になります。但し，`MPFR_RNDD`方式が設定されている場合のみ，-0 になります。

5.8 入出力関数

この節では，入出力ストリームを使用する入出力関数について解説します。ここで述べる関数の `stream`に NULL ポインタを与えると，入力は `stdin`から，出力は `stdout`に行うという意味になります。

`FILE *`を引数とする関数を使う時には，必ず`<stdio.h>`ヘッダファイルを `mpfr.h`より前の位置でインクルードして下さい。`mpfr.h`の中でこの型を用いた関数のプロトタイプを宣言しているからです。

`size_t mpfr_out_str` (*FILE *stream*, *int base*, *size_t n*, *mpfr_t op*, *mpfr_rnd_t rnd*) [関数]

出力ストリーム *stream*に，*rnd*方式で丸めて得た *op*の *base*進表現 *n*桁の値を出力します。基数は 2 以上 62 以下の自然数が指定できます。*n*桁の有効桁数は正確に出力されます。*n*が 0 の時は，*op*を読み戻した時にも同じ出力結果が出せる程度に十分な桁数分出力します。`mpfr_get_str`関数の説明も参照して下さい。

有効桁だけでなく，小数点も現時点のロケールに従って最初の桁の右側に出力されます。その後に続く指数部は，'`eNNN`'という形式の 10 進表現となります。基数 *base*が 10 より大きい時には，指数部の区切り文字として '`e`'ではなく，'`@`'が使用されます。

関数の返り値は出力した文字数で，エラー発生時にはゼロが返ります。

`size_t mpfr_inp_str` (*mpfr_t rop*, *FILE *stream*, *int base*, *mpfr_rnd_t rnd*) [関数]

入力ストリーム *stream*から *base*進表現の文字列を入力し，*rnd*方式で丸め，浮動小数点型変数 *rop*に代入します。

この関数は単語単位，つまりホワイトスペースの間の文字列を読み取り，`mpfr_set_str`関数を使って処理ます。有効な文字列形式については`mpfr_strtofr`関数の説明を参照して下さい。

返り値は読み取ったバイト数で，エラーが発生した場合はゼロが返ります。

`int mpfr_fpif_export` (*FILE *stream*, *mpfr_t op*) [関数]

*op*をファイルストリーム *stream*に，浮動小数点交換形式でエクスポートします。特に，32 ビットコンピュータとでエクスポートして，64 ビットコンピュータでインポートしたり，リトルエンディアン環境でエクスポートしてビッグエンディアン環境でインポートしたりする場合に有用です。*op*の精度桁数と NaN の符号もストアされます。エクスポートが正常に行われた時のみゼロを返します。

[注記] この関数の実装は試験的なものなので，インターフェースが将来変わる可能性があります。

`int mpfr_fpif_import` (*mpfr_t op*, *FILE *stream*) [関数]

ファイルストリーム *stream*から浮動小数点交換形式 (`mpfr_fpif_export`関数参照) で読み取りを行い，*op*にインポートします。*op*の精度桁数はストリームから読み取ったもので，NaN であっても符号も常に読み取ります。インポートされた精度桁数がゼロであったり，`MPFR_PREC_MAX`を越えたものである時は，インポートに失敗し，非ゼロを返して *op*には手を加えません。他の理由でインポートに失敗した場合は *op*には NaN がセットされます。また，*op*の精度桁数についても，読み取りされたとしても不定になります。

インポートが正常に行われた時のみゼロを返します。

[注記] この関数の実装は試験的なものなので，インターフェースが将来変わる可能性があります。

`void mpfr_dump` (*mpfr_t op*) [関数]

形式は特に指定せずに，*op*を標準出力 `stdout`に改行付きで表示します。主としてデバッグ用に使用する関数で，正常でないデータに対しても適用できます。特に定めていない事項については ABI を破壊しないよう，環境依存で決まります。

現状では次のような出力形式になっています。符号ビットが立っていれば NaN であってもマイナスを出力し，NaN, 無限大，ゼロに対してはそれぞれ '`@NaN@`', '`@Inf@`', '`0`'と出力します。それ以外の値は，符号に続いて次のように出力されます。'`0.`'に続いて *p*ビットの 2 進仮数部表現 (*p*は精度桁数)，その後にゼロが続くようなら（通常のデータではあり得ないケース)，大かっこを付けて出力します。大文字 '`E`'の後に 10 進表現の指数部が表示され，不正なデータ形

式や指数部範囲を超えている場合は，3 つの感嘆詞 (‘!!!’) を出力し，フラグに続いて，もう一度 3 つの感嘆詞 (‘!!!’) を出力します。フラグの意味は次の通りです。‘N’は仮数部の最大ビット (MSB) がゼロ，つまり正規化されていないことを，‘T’は非ゼロなビットが続いていることを，‘U’は UBF 数 (内部使用のみ) を，‘<’は指数部が現状の最小指数部値より小さいことを，‘>’は指数部が現状の最大指数部値より大きいことを，それぞれ意味します。

5.9 書式指定出力関数

5.9.1 利用条件

mpfr_printf関数は書式指定された出力を，標準 C の printf関数と同様に行うことができます。ビルド対象のシステムで ISO C の可変長引数の関数と，可変長引数にアクセスするためのマクロがサポートされている場合のみ，この関数が使用できます。

この関数を使う際には，mpfr.hの前に，必ず<stdio.h>をインクルードしておく必要があります。mpfr.hにおけるこの関数のプロトタイプ宣言で使用しているからです。

5.9.2 書式指定文字列

mpfr_printf関数で指定できる書式指定は，printf関数の書式指定を拡張したものになっています。書式指定の形式は次のようになっています。

```
% [フラグ] [文字数] [.[精度桁数]] [データ型] [丸め方式] 変換形式
```

‘フラグ (flag)’, ‘文字数 (width)’, ‘精度桁数 (precision)’は printf関数と同じ意味を持ちます。‘精度桁数 (precision)’は変換形式で指定した基数 (base) に基づいて表示される桁数となります。しかし，例えば‘Re’という書式指定を行うと，デフォルトの表示精度桁数は，‘e’という書式指定で設定した場合とは違ったものが設定されます。mpfr_printf関数は GMP が提供するデータ型に対する書式指定と同じものが利用できます。但し，廃止予定の書式指定である‘q’ではなく，‘ll’と指定して下さい。

‘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
‘t’	ptrdiff_t
‘z’	size_t

上記は標準データ型にする書式指定で，これ加えて，GMP が定義する‘データ型 (type)’と，MPFR のデータ型に対する‘R’指定と‘P’指定が利用できます。下記の表の 2 列目が，‘データ型 (type)’に続く，‘変換形式 (conv)’の書式指定となります。

‘F’	mpf_t, 浮動小数点形式
‘Q’	mpq_t, 整数形式
‘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
‘P’	mpfr_prec_t, 整数形式
‘R’	mpfr_t, 浮動小数点形式

‘データ型 (type)’の指定は，GMP のマニュアルに記してある制限をそのまま受け継ぎます。詳細は *GNU MP* の “書式指定形式” を参照。特に，‘データ型 (type)’指定は (‘R’と‘P’を除き)，GMP

ビルド時に gmp_printf関数が有効になっていないと使用できません。当然，‘t’のような標準のデータ型指定も，使用する環境で C ライブラリがサポートしていないと使用できません。

‘丸め方式 (rounding)’フィールドは mpfr_t型の場合のみ指定でき，他のデータ型に対しては使用しないで下さい。

‘P’指定や ‘R’指定がない場合は，mpfr_printf関数は gmp_printf関数と同じ書式で出力します。

‘P’指定は，その次に ‘d’, ‘i’, ‘o’, ‘u’, ‘x’, ‘X’が続き，mpfr_prec_t型の値に対して適用されます。この指定が必要になる理由は，mpfr_prec_t型が，必ずしも int型などの固定サイズの標準データ型であるとは限らないからです。‘精度桁数 (precision)’フィールドは表示される最小の桁数の指定で，デフォルトは 1 です。

プログラム例:

```
mpfr_t x;
mpfr_prec_t p;
mpfr_init (x);
...
p = mpfr_get_prec (x);
mpfr_printf ("%Pu ビットの変数 x", p);
```

‘R’指定の後は ‘a’, ‘A’, ‘b’, ‘e’, ‘E’, ‘f’, ‘F’, ‘g’, ‘G’, ‘n’が続き，mpfr_t型データの出力指定が行われます。‘R’指定の後に ‘丸め方式 (rounding)’の指定も，下の表のように可能となります。

‘U’	正の無限大方向の丸め
‘D’	負の無限大方向の丸め
‘Y’	ゼロから遠ざかる方向の丸め（切り上げ）
‘Z’	ゼロ方向への丸め（切り捨て）
‘N’	最近接値への丸め (なるべく偶数になるように)
‘*’	mpfr_t型データの前にある mpfr_rnd_t指定の方式による丸め

デフォルトの丸めモードは最近接値 (‘N’) です。下記の例では同じ出力形式になる 3 種類の書式指定を使っています。

```
mpfr_t x;
mpfr_init (x);
...
mpfr_printf ("%.128Rf", x);
mpfr_printf ("%.128RNf", x);
mpfr_printf ("%.128R*f", MPFR_RNDN, x);
```

ゼロから遠ざかる方向の丸めが ‘Y’で指定されるのは，ISO C 規格で，‘A’が 16 進出力の指定として予約されているからです (下記の表参照)。

‘変換書式 (conv)’は mpfr_t用の書式指定と共に，下記のようなものが使用できます。

‘a’ ‘A’	16 進浮動小数点表示, C99 形式
‘b’	2 進出力
‘e’ ‘E’	指数形式の浮動小数点表示
‘f’ ‘F’	固定小数点形式
‘g’ ‘G’	固定小数点形式，または　指数形式表示

‘b’という書式変換指定を使うと，対象の mpfr_t型データを 2 進表現します。他のデータ型には使用しないで下さい。それ以外の書式変換指定は，double型に対する指定と同じ意味です。

10 進以外の出力の場合，仮数部は指定された基数の進数表現となりますが，指数部は常に 10 進表現となります。非数や無限大は常に特定の文字列で出力されます。‘a’, ‘b’, ‘e’, ‘f’, ‘g’の場合は nan, -inf, infと出力され，‘A’, ‘E’, ‘F’, ‘G’の場合は NAN, -INF, INFと出力されます。

‘精度桁数 (precision)’フィールドに指定がある場合，mpfr_t型の値は，指定丸め方式で，指定精度桁数になるように丸められます。精度桁数指定がゼロで，最近接値への丸めが指定されおり，‘変換書式 (conv)’の指定が ‘a’, ‘A’, ‘b’, ‘e’, ‘E’, のどれかである場合，丸めた結果が同じ指数部の値で，二つの値のちょうど中間にあるなら偶数になる方に丸め，それ以外の場合は，ゼロから遠ざかる方向の値に丸められます。例えば"%.ORNe"という書式指定に対しては，85 は"8e+1"と出力され，95 は"1e+2"と出力されます。この方式は，‘g’ (‘G’の場合も同様) 指定の時，‘e’ (‘E’も同様) 形式で出力される際にも適用されます。精度桁数を int型の最大値より大きい値に指定した場合は，特に警告も出さず，INT_MAXの値を精度桁数として使用します。

‘精度桁数 (precision)’フィールドが空で，‘変換書式（conv)’に ‘e’や ‘E’を指定した場合 (例えば %Reや%.REという指定) の場合は，出力値を正確に読み戻すことができる桁数で出力されます。つまり，入力値と出力値が同じ精度，かつ，どちらも最近接値へ丸められているものとして扱います。これは mpfr_get_str関数に対する場合と同様です。‘f’, ‘F’, ‘g’, ‘G’が変換書式に指定されている時，‘精度桁数 (precision)’フィールドが空の時のデフォルトの精度桁数は 6 です。

5.9.3 書式指定入出力関数

下記の関数に対して，int型の最大値 INT_MAXを超えるパラメータの設定がされている場合は，何も出力しません (出力先が stdoutでも *buf*でも *str*であっても同じです)。この場合，関数は −1 を返し，範囲エラーフラグを立て，POSIX 環境など EOVERFLOWマクロが定義されていれば errnoに EOVERFLOWを代入します。注意してほしいのは，これ以外のエラー発生時（今のところ，メモリ解放関数で起こり得る）にも，内部ライブラリ呼び出しの結果， errnoの値は変化してしまう可能性があるということです。

`int mpfr_fprintf (`*FILE *stream, const char *template, ...*`)` [関数]
`int mpfr_vfprintf (`*FILE *stream, const char *template, va_list ap*`)` [関数]
出力先 *stream*に対して，書式指定文字列 *template*に従った出力を行います。返り値は出力した文字数で，エラー発生時は負数が返されます。

`int mpfr_printf (`*const char *template, ...*`)` [関数]
`int mpfr_vprintf (`*const char *template, va_list ap*`)` [関数]
標準出力 *stdout*に対して，書式指定文字列 *template*に従った出力を行います。返り値は出力した文字数で，エラー発生時は負数が返されます。

`int mpfr_sprintf (`*char *buf, const char *template, ...*`)` [関数]
`int mpfr_vsprintf (`*char *buf, const char *template, va_list ap*`)` [関数]
NULL 終端子を持つ文字列を，書式指定文字列 *template*に従って生成し，*buf*に格納します。*buf*と他の引数はメモリ内で重複してはいけません。返り値は NULL 終端子を除いて *buf*に書き込まれた文字数で，エラーが発生した場合は負数が返されます。

`int mpfr_snprintf (`*char *buf, size_t n, const char *template, ...*`)` [関数]
`int mpfr_vsnprintf (`*char *buf, size_t n, const char *template, va_list ap*`)` [関数]
NULL 終端子を持つ文字列を，書式指定文字列 *template*に従って生成し，*buf*に格納します。*n*がゼロの時は何も書き込まず，*buf*は NULL ポインタとなります。*n*に正数が指定される場合は，最初の $n-1$ 文字が *buf*に書き込まれ，*n*番目の文字が NULL になります。*n*が十分大きい場合は，最後の NULL 文字を除き，書き込まれた文字数が返り値となり，エラーが発生した場合は負数が返り値となります。

`int mpfr_asprintf` (*char **str, const char *template, ...*) [関数]
`int mpfr_vasprintf` (*char **str, const char *template, va_list ap*) [関数]
現在のメモリ割り当て関数 (Section 4.7 [Memory Handling], 頁 12 参照) を使って確保したメモリブロックに，NULL 終端子を持つ文字列を書き込みます。メモリブロックへのポインタは *str*に与えます。メモリブロックを解放するときには必ず `mpfr_free_str`を使って下さい。返り値は NULL 終端子を除いて書き込まれた文字数で，エラーが発生した場合は負数が返され，*str*の値は不定となります。

5.10 整数関数，剰余関数

`int mpfr_rint` (*mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
`int mpfr_ceil` (*mpfr_t rop, mpfr_t op*) [関数]
`int mpfr_floor` (*mpfr_t rop, mpfr_t op*) [関数]
`int mpfr_round` (*mpfr_t rop, mpfr_t op*) [関数]
`int mpfr_roundeven` (*mpfr_t rop, mpfr_t op*) [関数]
`int mpfr_trunc` (*mpfr_t rop, mpfr_t op*) [関数]
*op*を丸めて整数にして *rop*に代入します。`mpfr_rint`関数は，*rnd*方式を用いて最も近い整数に丸めます。他の 5 つの関数も，丸め方式を固定して整数に丸めます。

- `mpfr_ceil`関数: *op*以上となる隣接整数に丸める (`mpfr_rint`関数で `MPFR_RNDU`を指定した時と同等)
- `mpfr_floor`関数: *op*以下となる隣接整数に丸める (`mpfr_rint`関数で `MPFR_RNDD`を指定した時と同等)
- `mpfr_round`関数: *op*をゼロから遠ざかる方向に丸めて，最も近い整数に (IEEE 754-2008 規格の roundTiesToAway モードに相当)
- `mpfr_roundeven`関数: *op*に偶数丸めを行って，最も近い整数に (`mpfr_rint`関数で `MPFR_RNDN`を指定した時と同等)
- `mpfr_trunc`関数: *op*をゼロ方向に丸めて，最も近い整数に (`mpfr_rint`関数で `MPFR_RNDZ`を指定した時と同等)

*op*がゼロ，もしくは無限大の時には，同符号の同じ値を *rop*に代入します。

返り値については，丸め誤差なしで正確に変換できた時はゼロ，*op*より大きくなる場合は正数，*op*より小さくなる場合は負数となります。正確に言うと，*op*が正確に整数として *rop*に代入できればゼロ，*op*が，整数ではあるが，*rop*では正確に表現できない場合は 1 もしくは -1，*op*が整数でない時には，2 ないし -2 を返します。

*op*が NaN の時は，NaN フラグが立ちます。他の例外処理も同様で，*rop*に代入される値が *op* とは異なる場合は，ISO C99 の `rint`関数の処理方法に則って，不正確フラグを立てます。より IEEE 754 や ISO TS 18661-1 に従った振る舞いをしたい時，つまり，これらの整数化関数を数学関数として扱いたい時には，`mpfr_rint_*`関数を使った方がいいでしょう。

これらの関数では 2 重に丸めが発生することはありません。例えば 10.5 (2 進では 1010.1) を最近接丸めで `mpfr_rint`関数を使うと 12 (2 進 1100) という 2 ビット整数になります。この場合，2 つの候補となる 2 ビット整数 8 と 12 がありますが，最も近いのは 12 だからです。2 重丸めを行ったとすると，最初に偶数丸めを行って 10 となり，次にまた偶数丸めを行って 8 となります。

`int mpfr_rint_ceil` (*mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
`int mpfr_rint_floor` (*mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
`int mpfr_rint_round` (*mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
`int mpfr_rint_roundeven` (*mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
`int mpfr_rint_trunc` (*mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd*) [関数]
*op*を丸めて整数化して *rop*に代入します。

- `mpfr_rint_ceil`関数: *op*以上の整数に丸める
- `mpfr_rint_floor`関数: *op*以下の整数に丸める
- `mpfr_rint_round`関数: ゼロから遠ざかる方式で最も近い整数に丸める
- `mpfr_rint_roundeven`関数: 偶数丸め方式で最も近い整数に丸める
- `mpfr_rint_trunc`関数: ゼロ方向へ丸めて整数に

素直に整数化できない時には *rnd*方式で丸めます。*op*がゼロもしくは無限大の時には *rop*には符号も含めて同じ値を代入します。返り値は三種値で，他の数学関数同様に扱われ，整数化関数 (round-to-integer funcion) と同じものになります。

`mpfr_rint`関数とは対照的に，これらの関数は 2 重に丸めを行います。最初に *op*を関数ごとに指定された方向に丸めて最近接の整数に丸め，この整数が *rop*で正確に表現できない時には，*rnd*で指定した丸め方式でもう一度丸めを行います。従って，これらの整数化関数 (round-to-integer function) はより数学関数らしく振舞います。つまり，返される結果は，実数に対して正確に整数化したものを，正しく丸めたものになる訳です。

例えば，`mpfr_rint_round`関数に，最近接丸めと 2 ビットの精度桁指定を行ったとすると，6.5 は 7 となり (ゼロから遠ざかる丸め), 7 は最近偶数丸めの結果，8 になります。6 も 2 ビット整数表現でき，8 よりも 6.5 に近いのですが。

`int mpfr_frac` (*mpfr_t* **rop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の小数部を *rnd*方向に丸めて取り出し，符号も等しくなるように *rop*に代入します。`mpfr_rint`関数と同様に，*rnd*は正確な小数部を丸める時にのみ影響し，小数部の取り出しには影響を与えません。*op*が整数，もしくは無限大の時は *rop*には *op*と同符号のゼロを代入します。

`int mpfr_modf` (*mpfr_t* **iop**, *mpfr_t* **fop**, *mpfr_t* **op**, *mpfr_rnd_t* **rnd**) [関数]
*op*の整数部を *iop*に，小数部を *fop*に同時に代入します。それぞれ *rnd*方向に丸めて *iop*と *fop*の精度桁数に納めます。これは `mpfr_trunc(iop, op, rnd)`と `mpfr_frac(fop, op, rnd)`) と同じ処理になります。変数 *iop*と *fop*は異なる変数でなければなりません。どちらも正確な値を返すことができた時のみゼロを返します。`mpfr_sin_cos`関数の返り値についての記述も参照して下さい。

`int mpfr_fmod` (*mpfr_t* **r**, *mpfr_t* **x**, *mpfr_t* **y**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_fmodquo` (*mpfr_t* **r**, *long** **q**, *mpfr_t* **x**, *mpfr_t* **y**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_remainder` (*mpfr_t* **r**, *mpfr_t* **x**, *mpfr_t* **y**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_remquo` (*mpfr_t* **r**, *long** **q**, *mpfr_t* **x**, *mpfr_t* **y**, *mpfr_rnd_t* **rnd**) [関数]
$x - ny$ の値を計算し，*rnd*方式で丸めて *r*に代入します。ここで *n*は，*x*を *y*で割った時の整数の商で，次のように定義されます。*n*は `mpfr_fmod`関数と `mpfr_fmodquo`関数を用いてゼロ方向に丸められ，`mpfr_remainder`関数と `mpfr_remquo`関数に対しては偶数丸めで最近接の整数になります。

特殊値については，ISO C99 規格の F.9.7.1 節に述べられているように扱います。*x*が無限大，もしくは *y*がゼロの時は，*r*は NaN になります。*y*が無限大で *x*が有限値の時は，*r*は *x*を *r*の精度桁数に丸められた値になります。*r*がゼロの時は，*x*と同符号になります。返り値は *r*に応じた三種値になります。

加えて，`mpfr_fmodquo`関数と `mpfr_remquo`関数は**q*における商 *n*の低位のビットを，*x*を *y*で割った時の符号と共に格納します。正確に言うと，`long`型のビット数から 1 少ないビット数分ということになります。但し，全てのビットがゼロの時は除きます。この時はゼロを返します。*x*は，正しい商が実用的なものではない *y*に比して絶対値が大きくなる可能性があります。`mpfr_remainder`関数と `mpfr_remquo`関数は引数のリダクション用に使用されています。

`int mpfr_integer_p` (*mpfr_t* *op*) [関数]
*op*が整数の時のみ，ゼロ以外の数を返します。

5.11 丸め処理関数

void mpfr_set_default_rounding_mode (*mpfr_rnd_t* **rnd**) [関数]
デフォルトの丸めモードを *rnd*に設定します。初期設定では最近偶数丸めがデフォルトです。

mpfr_rnd_t mpfr_get_default_rounding_mode (*void*) [関数]
現在のデフォルトの丸めモードが返されます。

int mpfr_prec_round (*mpfr_t* **x**, *mpfr_prec_t* **prec**, *mpfr_rnd_t* **rnd**) [関数]
*x*を *rnd*方式で丸め，精度桁数 *prec*に納めます。精度桁数は MPFR_PREC_MIN以上，MPFR_PREC_MAX以下でなければなりません。そうでない時の動作は定義されていません。*prec*が *x*の精度桁数以上であれば，新たに仮数部を格納するために必要なメモリ領域を確保し，下の桁にはゼロが詰め込まれます。*prec*が *x*の精度桁数未満であれば，指定された丸めモードで仮数部は *prec*桁に丸められます。どちらの場合でも，*x*の精度桁数は *prec*に置き換えられます。

ここで，mpfr_prec_round関数を使って，*a*の逆数を求めるニュートン法を実装した例を示します。*x*は既に頭から *n*ビット正しい近似値であると仮定しています。

```
mpfr_set_prec (t, 2 * n);
mpfr_set (t, a, MPFR_RNDN);          /* a を 2n ビットに丸める */
mpfr_mul (t, t, x, MPFR_RNDN);       /* t は 2n ビットに正確に丸められる */
mpfr_ui_sub (t, 1, t, MPFR_RNDN);    /* 大きい方の n ビット分桁落ち */
mpfr_prec_round (t, n, MPFR_RNDN);   /* t は n ビットに正確に丸められる */
mpfr_mul (t, t, x, MPFR_RNDN);       /* t は n ビットに正確に丸められる */
mpfr_prec_round (x, 2 * n, MPFR_RNDN); /* 丸めなしで 2n ビットに */
mpfr_add (x, x, t, MPFR_RNDN);       /* x は正確に 2n ビットに丸められる */
```

[警告] この関数で使用する *x*は MPFR_DECL_INITマクロや，mpfr_custom_init_set関数 (see Section 5.15 [Custom Interface], 頁 47) で絶対に初期化しないで下さい。

int mpfr_can_round (*mpfr_t* **b**, *mpfr_exp_t* **err**, *mpfr_rnd_t* **rnd1**, [関数]
mpfr_rnd_t **rnd2**, *mpfr_prec_t* **prec**)
*b*を未知数 *x*を *rnd1*方向に丸め，2 の E(b)-*err*乗の誤差を持つ近似値と仮定します。ここで E(b) は *b*の指数部を意味します。この時，指数部の範囲制限はないものとして，*x*を正しく *rnd2*方向に丸めて *prec*桁にできるのであれば，この関数は非ゼロを返します。それ以外の場合は，NaN や無限大の場合も含めて 0 を返します。言い換えると，*b*の誤差が 2 の *k*乗 ulp 以下で抑えられ，*b*が *prec*桁の精度を持っているならば，*err*=*prec*−*k*という評価式を得られるということです。この関数は，**引数を変更しません。**

*rnd1*が MPFR_RNDNもしくは MPFR_RNDFである時には，誤差は正もしくは負になると想定されますので，誤差範囲は *rnd1*が行う丸めの 2 倍，つまり *err*と同じ値になります。

*rnd2*が MPFR_RNDFの時，*rnd3*には *rnd1*とは反対方向の丸め方式を設定するものとします。*rnd1*が MPFR_RNDNもしくは MPFR_RNDFの時は，*rnd3*には MPFR_RNDNを設定します。さすれば，mpfr_can_round (b, err, rnd1, MPFR_RNDF, prec)の返り値は，mpfr_set (y, b, rnd3)を *prec*桁の *y*で呼び出した後に，*y*が *x*の忠実丸め結果と等しくなると保証できる時のみ，非ゼロを返します。

[注記] この関数については，返り値が三種値 [ternary value], 頁 9 のどれになるかは一般的には決められません。しかし，真値が正しく *prec*桁で表現できないと分かっているのであれば，下記のような方法を使って, 非ゼロな三種値のどちらになるのか，丸め方式 *rnd2*に関係なく決めることができます。以下の例では MPFR_RNDZはいずれの方向付き丸めに置き換え可能です。

```
if (mpfr_can_round (b, err, MPFR_RNDN, MPFR_RNDZ,
                    prec + (rnd2 == MPFR_RNDN)))
```

```
{
  /* 'b' を，丸め方式'rnd2' で丸めて'prec' ビットにしたものを'r' に代入し
     返り値の三種値を'inex' に格納する */
  inex = mpfr_set (r, b, rnd2);
}
```

実際，*rnd2*が MPFR_RNDNであれば，方向付き丸めで *prec*+1 ビットに丸められるかどうかを確認できます。これができるなら，最近接丸めで確実に *prec* ビットになりますし，加えて，非ゼロ三種値のどれになるかも正しく決められます。ただし，*b*が *prec* ビットで表現できる数に近接しているときはその限りではありません。

この関数についての詳細な事例は examplesサブディレクトリにある can_round.cを参照して下さい。

mpfr_prec_t mpfr_min_prec (*mpfr_t x*) [関数]
*x*の仮数部を格納するために必要となる最小のビット数を返します。*x*がゼロや非数など特別な数の場合は，0 を返します。

const char * mpfr_print_rnd_mode (*mpfr_rnd_t rnd*) [関数]
丸め方式 *rnd*に対して，その丸めモードを表わす文字列 ("MPFR_RNDD", "MPFR_RNDU", "MPFR_RNDN", "MPFR_RNDZ","MPFR_RNDA") を返します。*rnd*が丸めモードとして不適切な値の時は，NULL ポインタを返します。

int mpfr_round_nearest_away (*int* (*foo*)(*mpfr_t*, *type1_t*, ..., *mpfr_rnd_t*), [Macro]
mpfr_t rop, *type1_t op*, ...)
1 ないし複数の引数 *op*(データ型は mpfr_t, long, doubleなど) を取る関数 *foo*を与え，*rop*に *foo*(*op*,...)の結果を，最近接値から離れる丸め (round-nearest-away) 方式で丸めた値を代入します。この丸め方式は，同じ値になる場合を除いて，偶数最近接値への丸めと同じ手法で定義されており，ゼロから離れる方向の値を返します。関数 *foo*は入力値で， 次の引数から最後から 2 番目の引数については，*rop*以降の引数が，丸めモードの指定が最後となるように与えられ，これらが最初の引数である関数の引数として *foo*(*op*,...)に渡されて実行され，丸めモード指定に従って値が丸められます。返り値は三種値になります。但し，処理が正しくできたと期待できる時のみで，期待できない時には mpfr_round_nearest_awayマクロはうまく働かないでしょう。実装上の制約により，このマクロは，最小指数部値 eminに到達してしまう可能性のある時には使用しないで下さい。また，このマクロは，コンパイラが *foo*のプロトタイプ宣言と，引数リスト *op*の不整合を検出できるようになっています。C99 コンパイラでのみ，*op*の多重指定が可能になっており，C99 コンパイラでは一つだけを受け付けます。

[注記] このマクロは試験的な実装になっており，インターフェースは将来変更される可能性があります。

```
unsigned long ul;
mpfr_t f, r;
/* r, f, ul を初期化して値をセットする。必要があれば emin も設定 */
int i = mpfr_round_nearest_away (mpfr_add_ui, r, f, ul);
```

5.12 その他の関数

void mpfr_nexttoward (*mpfr_t x*, *mpfr_t y*) [関数]
*x*と *y*のどちらかが NaN であれば，*x*には NaN が代入され，他の場合と同様，NaN フラグも立ちます。*x*と *y*が等しい時は *x*は変更されません。それ以外の場合は，*x*が *y*と異なる場合は，*x*は，*x*の精度桁数と現状の指数部の範囲で，*y*方向に隣の浮動小数点数が代入されます。無限大の場合は，最小あるいは最大の浮動小数点数として機能します。結果がゼロの時は，符号は

同じものが保持されます。アンダーフロー，オーバーフロー，不正確例外が発生することはありません。

`void mpfr_nextabove` (*mpfr_t x*) [関数]
`void mpfr_nextbelow` (*mpfr_t x*) [関数]
*y*が正の無限大 (負の無限大) の時の `mpfr_nexttoward`関数と同じ処理を行います。

`int mpfr_min` (*mpfr_t rop*, *mpfr_t op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
`int mpfr_max` (*mpfr_t rop*, *mpfr_t op1*, *mpfr_t op2*, *mpfr_rnd_t rnd*) [関数]
*rop*に *op1*と *op2*の最小値（最大値）を代入します。*op1*と *op2*が共に NaN の時は，*rop*にも NaN が代入されます。*op1*と *op2*のどちらかが NaN の時は，*rop*には有限数値の方が代入されます。*op1*と *op2*が互いに符号の異なるゼロの時は，*rop*には −0 (+0) が代入されます。

`int mpfr_urandomb` (*mpfr_t rop*, *gmp_randstate_t state*) [関数]
$0 \leq rop < 1$ 区間の一様乱数を浮動小数点数として与えます。正確に言うと，この乱数は正規化されていない仮数部と指数部がゼロの浮動小数点数として表現されます。当然，最終的には正規化されて指数部が eとなりますから，仮数部の最後の e ビットは常にゼロとなります。

指数部が現状の指数部範囲に収まっていれば 0 を返し，指数部範囲を超えていれば *rop*には NaN を代入し，非ゼロを返します。とはいえ指数部範囲を超えるということは普通は起こりません。2 番目の引数は，`gmp_randstate_t`構造体で，GMP の `gmp_randinit`関数を使って生成しておいて下さい (GMP マニュアル参照)。

[注記] MPFR では，*rop*に代入される値や，以降の乱数を制御する *state*の新しい値は，マシンのワードサイズには依存せずに決まります。

`int mpfr_urandom` (*mpfr_t rop*, *gmp_randstate_t state*, *mpfr_rnd_t rnd*) [関数]
一様分布に従う浮動小数点数を生成します。浮動小数点数 *rop*は，[0, 1] 区間の連続一様分布に従う実数乱数が *rnd*方向に丸められた値と見ることができます。

2 番目の引数である `gmp_randstate_t`構造体は GMP の `gmp_randinit`関数で生成したものを指定して下さい (GMP マニュアル参照)。

[注記] `mpfr_urandomb`の注記はこの関数でも有効です。さらに言うと，丸められる前の正しい乱数値と次の乱数状態は，現状の指数部範囲と丸めモードはに依存しませんが，代入される変数の精度桁数には依存します。つまり，乱数生成器の状態が同じものから出発したとしても，代入される変数の精度桁数が変更されると，その値や乱数生成器の状態は完全に違ったものに変化します。

`int mpfr_nrandom` (*mpfr_t rop1*, *gmp_randstate_t state*, *mpfr_rnd_t rnd*) [関数]
`int mpfr_grandom` (*mpfr_t rop1*, *mpfr_t rop2*, *gmp_randstate_t state*, *mpfr_rnd_t rnd*) [関数]
平均 0，分散が 1 のガウス分布に従う乱数を一つ (`mpfr_grandom`では最大 2 つ) 生成し，浮動小数点数として与えます。`mpfr_grandom`関数は，*rop2*が NULL ポインタの場合は，乱数を一つだけ生成して *rop1*に代入します。

浮動小数点数 *rop1* (と *rop2*) は，標準正規ガウス分布に従った実数乱数を *rnd*方向に丸めたものと解釈できます。

`gmp_randstate_t`引数は GMP の `gmp_randinit`関数で生成したものを指定して下さい。(GMP マニュアル参照)。

`mpfr_grandom`関数の返り値は三種値で，その組み合わせは `mpfr_sin_cos`関数と同じです。*rop2*が NULL ポインタであれば，2 番目の三種値はゼロになります。三種値を一つだけ返す

ということは，結果を一つだけ返す関数の三種値とは異なるものになる，ということに留意して下さい。それ以外の時は，三種値のうち非ゼロの値が返ってきます。

[注記] mpfr_urandomb関数の注記はここでも有効です。加えて，指数部の範囲と丸めモードは次の乱数生成器の状態に影響を与える可能性があります。

[注記] mpfr_nrandom関数は，精度桁数が大きくなるとmpfr_grandom関数よりずっと高速に動作します。従って，mpfr_grandom関数は廃止予定で，将来消去される予定です。

`int mpfr_erandom (`*mpfr_t* ***rop1***, *gmp_randstate_t* ***state***, *mpfr_rnd_t* ***rnd***`)` [関数]
平均が 1 の指数分布に従う乱数を浮動小数点数として与えます。その他の性質は mpfr_nrandom 関数と同じです。

`mpfr_exp_t mpfr_get_exp (`*mpfr_t* ***x***`)` [関数]
*x*は通常の非ゼロな浮動小数点数で，仮数部は [1/2,1) に存在しているという前提で，*x*の指数部を返します。この関数では *x*が現状の指数部範囲を超えていても正しく処理を行います。*x*が NaN, 無限大，ゼロの時の処理は定めていません。

`int mpfr_set_exp (`*mpfr_t* ***x***, *mpfr_exp_t* ***e***`)` [関数]
*x*が通常の非ゼロの浮動小数点数である時，*x*の指数部に *e*を代入します。*e*が現状の指数部範囲に収まっていればゼロを返し，それ以外の時は，非ゼロを返し，*x*は変更しません。

`int mpfr_signbit (`*mpfr_t* ***op***`)` [関数]
*op*の符号部がセット，つまり負数や，−0, 符号部の設定された NaN の時のみ非ゼロを返します。

`int mpfr_setsign (`*mpfr_t* ***rop***, *mpfr_t* ***op***, *int* ***s***, *mpfr_rnd_t* ***rnd***`)` [関数]
*op*を丸め方式 *rnd*で丸め，*rop*に代入し，しかる後に *s*の値が非ゼロ（ゼロ）の時は，符号も代入（クリア）します。これは *op*が NaN の場合も同様に処理されます。

`int mpfr_copysign (`*mpfr_t* ***rop***, *mpfr_t* ***op1***, *mpfr_t* ***op2***, *mpfr_rnd_t* ***rnd***`)` [関数]
*op1*を丸め方式 *rnd*で丸めて *rop*に代入し，その符号ビットを *op2*に代入します。*op1*や *op2*が NaN であっても同様に処理されます。この関数は mpfr_setsign (*rop*, *op1*, mpfr_signbit (*op2*), *rnd*)と同じ働きを行います。

`const char * mpfr_get_version (`*void*`)` [関数]
NULL 終端子付きの文字列として MPFR のバージョンを返します。

MPFR_VERSION [マクロ]
MPFR_VERSION_MAJOR [マクロ]
MPFR_VERSION_MINOR [マクロ]
MPFR_VERSION_PATCHLEVEL [マクロ]
MPFR_VERSION_STRING [マクロ]
MPFR_VERSIONは定数として与えられる MPFR のバージョン番号です。MPFR_VERSION_MAJOR, MPFR_VERSION_MINOR, MPFR_VERSION_PATCHLEVELはそれぞれ，MPFR のメジャー番号，マイナー番号，パッチレベルを表わす定数です。MPFR_VERSION_STRINGは，文字列定数として与えられるバージョン定数で，開発バージョンやプレリリース用のバージョンに使用されるサフィックスも含まれています。この文字列定数は mpfr_get_versionと比較でき，ヘッダファイルやライブラリをチェックする際に役立ちます。

```
if (strcmp (mpfr_get_version (), MPFR_VERSION_STRING))
  fprintf (stderr, "警告: ヘッダファイルとライブラリのバージョンが一致していません。\n");
```

[注記] 上記のように比較して文字列が一致しなくてもエラーとは言えません。古い MPFR バージョン用に作ったプログラムは，ライブラリのバージョン管理システムが許可していれば，新しいバージョンの MPFR と動的リンクすることもできます。

`long MPFR_VERSION_NUM (`*major*, *minor*, *patchlevel*`)` [Macro]
major, *minor*，*patchlevel*から，`MPFR_VERSION`と同じ形式の整数値を生成します。下記は，コンパイル時に MPFR のバージョンをチェックするプログラム例です。

```
#if (!defined(MPFR_VERSION) || (MPFR_VERSION<MPFR_VERSION_NUM(3,0,0)))
# error "MPFR のバージョンが正しくありません。"
#endif
```

`const char * mpfr_get_patches (`*void*`)` [関数]
MPFR ライブラリに適用されたパッチ（内容は`PATCHES`ファイルに書いてあります）の id を含む，NULL 終端子文字列を返します。文字列はスペースで区切られています。

[注記] 古い MPFR とコンパイルしたプログラムを新しい MPFR と動的リンクするのであれば，コンパイル時の古い MPFR の識別子は無効になります。まぁ大して重要な情報ではありませんので。

`int mpfr_buildopt_tls_p (`*void*`)` [関数]
MPFR がスレッドセーフになるよう，スレッドローカルなストレージを有効にしてコンパイルされた時には（‘`--enable-thread-safe`’オプションつきで設定します。`INSTALL`ファイルを参照のこと），非ゼロを返します。スレッドセーフでない時にはゼロを返します。

`int mpfr_buildopt_float128_p (`*void*`)` [関数]
MPFR が ‘`__float128`’をサポートしている時には非ゼロを返します。つまり，MPFR が ‘`--enable-float128`’オプション付きで設定するとそうなります。サポートしていない時はゼロを返します。

`int mpfr_buildopt_decimal_p (`*void*`)` [関数]
MPFR を ‘`--enable-decimal-float`’オプション付きで設定すると MPFR は 10 進浮動小数点数をサポートしますが，この時にはこの関数は非ゼロを返します。サポートしていない時はゼロを返します。

`int mpfr_buildopt_gmpinternals_p (`*void*`)` [関数]
MPFR が ‘`--with-gmp-build`’オプション，もしくは ‘`--enable-gmp-internals`’オプション付きでビルドされていると，MPFR は GMP の内部関数を利用します。この時この関数は非ゼロを返します。そうでない時にはゼロを返します。

`int mpfr_buildopt_sharedcache_p (`*void*`)` [関数]
MPFR が全てのスレッドで`mpfr_const_pi`や`mpfr_const_log2`などの MPFR 定数を保持するキャッシュを共有するようにコンパイルされていると (‘`--enable-shared-cache`’オプション付きでビルドした場合)，非ゼロを返します。 キャッシュ共有ができない場合はゼロを返します。この関数が非ゼロを返す時には，MPFR を使用するアプリケーションは，‘`-pthread`’オプションを付けてコンパイルする必要があります。

`const char * mpfr_buildopt_tune_case (`*void*`)` [関数]
コンパイル時に使用した閾値ファイルを文字列で返します。このファイルは通常，プロセッサの種類ごとに決まっています。

5.13 例外処理関数

`mpfr_exp_t mpfr_get_emin (`*void*`)` [関数]

`mpfr_exp_t mpfr_get_emax` (*void*) [関数]
関数実行時点における，MPFR 浮動小数点型の指数部の最小値と最大値をそれぞれ返します。浮動小数点型の正の最小値は $1/2 \times 2^{\mathrm{emin}}$，正の最大値は $(1-\varepsilon) \times 2^{\mathrm{emax}}$ です。ここで，ε は浮動小数点型の精度桁数によって決まる定数です。

`int mpfr_set_emin` (*mpfr_exp_t* **exp**) [関数]
`int mpfr_set_emax` (*mpfr_exp_t* **exp**) [関数]
浮動小数点型の指数部の最小値と最大値をそれぞれ設定します。*exp*を，実行環境に依存して決まる指数部の範囲内で最小値，あるいは最大値として設定できない場合は非ゼロ数を返します。この場合は，現状の指数部最小値ないし指数部最大値は変化しません。指数部の最小値・最大値を設定できれば，ゼロを返します。

これらの関数の実行後は，ユーザの責任で入力される浮動小数点数が，`mpfr_check_range`関数を使用するなどして，この新しい指数部の範囲内に収まっていることをチェックする義務が生じます。この範囲からはみ出している値が入力された時のデフォルトの動作は，ISO C 規格でも決まっていません。`mpfr_check_range`関数にあるように，その際の挙動については明文化されています。

[注記] 定数値のキャッシュは，これらの関数で指数部の範囲が変更された後もそのままです。これは，API 経由でキャッシュ値を直接ユーザが使うことができない，ということではありません。MPFR は，内部的には必要に応じて指数部の範囲を超えることを許容しています。

`emin` > `emax` かつ，浮動小数点値を出力しなければならない場合は，その挙動は不定です。`mpfr_set_emin`関数も`mpfr_set_emax`関数もこの条件はスルーしますし，いつでも起こり得る事象ではあります。

`mpfr_exp_t mpfr_get_emin_min` (*void*) [関数]
`mpfr_exp_t mpfr_get_emin_max` (*void*) [関数]
`mpfr_exp_t mpfr_get_emax_min` (*void*) [関数]
`mpfr_exp_t mpfr_get_emax_max` (*void*) [関数]
`mpfr_set_emin`関数や`mpfr_set_emax`関数で設定できる指数部の最小値と最大値をそれぞれ返します。これらの値は事項環境に依存しますので，`mpfr_set_emax(mpfr_get_emax_max())`や`mpfr_set_emin(mpfr_get_emin_min())`といったことを行うと，ポータビリティを損ねます。

`int mpfr_check_range` (*mpfr_t* **x**, *int* **t**, *mpfr_rnd_t* ***rnd***) [関数]
この関数は，*x*が *y*を *rnd*方式で，指数部の範囲も拡張して正しく丸めた値になっており，*t*には三種値（[ternary value], 頁 9）が入っていることを仮定しています。例えば，`t = mpfr_log (x, u, rnd)`で，*y*は *u*の自然対数の正しい値が入っているものとします。さすれば，*t*は，*x*が *y*より小さい時には負数，*x*が *y*より大きければ正数，*x*と *y*が等しければゼロになります。この関数は，*x*が現状受け入れ可能な値の範囲に入っていれば，値を入れなおします *x*の指数部が現状の許容範囲から外れていれば，オーバーフローもしくはアンダーフローが発生します。*t*の値は 2 重に丸めが発生することを防止するために使用されます。新しい *x*の値が真値である *y*と等しい時にはゼロを，*y*より大きくなる場合は正数を，*y*より小さくなる場合は負数を返します。他の関数とは違い，新しい *x*の値は（未知の）真値である *y*と比較し，入力時の *x*とは比較しません。つまり，三種値は伝達されます。

[注記] *x*が無限大で，*t*が非ゼロである時（つまり，丸めた結果，不正確な無限大になった時）オーバーフラグが立てられます。これが役に立つのは，`mpfr_check_range`関数が，MPFR の関数内で呼ばれ，内部処理でフラグがセットされる場合です。

`int mpfr_subnormalize` (*mpfr_t* **x**, *int* **t**, *mpfr_rnd_t* ***rnd***) [関数]
この関数は，非正規数演算 (subnormal) をエミュレートしつつ，*x*を丸めます。*x*が非正規化エリアの外にある場合は，三種値 [ternary value], 頁 9, *t*だけを伝えます。*x*が非正規エリアに入っている時には，丸めモード *rnd*と，引数の三種値 *t*に従って，二重に丸めを行わないようにしつ

つ，xを EXP(x)-emin+1精度桁数に丸めます。正確に言うと，非正規化エリアの中では，e は eminの値になり，xは丸められて固定小数点演算用として 2^{e-1} を掛けられて整数になります。結果として $1.5 \times 2^{e-1}$，tがゼロの時は 2^e を最近接値に丸めます。

PREC(x)はこの関数では変更されません。rndは丸めモード，tは xが計算される際に使われる三種値でなければなりません。これは mpfr_check_range関数と同じです。非正規化エリアとなる指数部の範囲は，eminから emin+PREC(x)-1までとなります。emaxの設定値が小さすぎる場合など，現状の MPFR の許容指数部の範囲内では結果が表現できない時のこの関数の挙動は定義されていません。他の関数とは異なり，演算結果は，入力時の xではなく，真値と比較されます。つまり，三種値は引数の値が引き継がれます。

通常は，返り値の三種値がゼロの場合，不正確フラグが立ちます。更に，入力値 xが最初から非正規化エリアの値だったりして，2 回目の丸め処理が起きた場合はアンダーフローフラグがセットされます。

[警告] mpfr_subnormalize関数を呼び出す前に eminを mpfr_set_emin関数で変更してしまうと，その値が MPFR の現状の指数部範囲に入っているかどうかを確認しなければなりません。しかし計算の前に eminを変更することができることは悪いことではありません。

以下のプログラムは，2 進倍精度 IEEE 754 演算 (IEEE 754-2008 の binary64) を MPFR でエミュレートしたものです。

```
{
  mpfr_t xa, xb; int i; volatile double a, b;

  mpfr_set_default_prec (53);
  mpfr_set_emin (-1073); mpfr_set_emax (1024);

  mpfr_init (xa); mpfr_init (xb);

  b = 34.3; mpfr_set_d (xb, b, MPFR_RNDN);
  a = 0x1.1235P-1021; mpfr_set_d (xa, a, MPFR_RNDN);

  a /= b;
  i = mpfr_div (xa, xa, xb, MPFR_RNDN);
  i = mpfr_subnormalize (xa, i, MPFR_RNDN); /* new ternary value */

  mpfr_clear (xa); mpfr_clear (xb);
}
```

mpfr_set_emin関数と mpfr_set_emax関数は前もって呼び出しておき，全ての計算すべき値が，現状の指数部範囲に収まっていることを確認しておきましょう。

[警告] 上記のプログラムは非正規化エリアでも正しく丸めを行って倍精度 IEEE 754 演算をエミュレートしています。このような動作はハードウェアでは行われません。

下記の例は，特別なケースで固定点小数点演算をエミュレートする方法を示しています。ここでは，2^{-42} で丸める固定小数点演算を行って整数の 1 から 17 までのサイン (sine) 関数の値を求めています。絶対値としてはほぼ 1 になってしまうということを利用しています。

```
{
  mpfr_t x; int i, inex;

  mpfr_set_emin (-41);
  mpfr_init2 (x, 42);
```

```
      for (i = 1; i <= 17; i++)
        {
          mpfr_set_ui (x, i, MPFR_RNDN);
          inex = mpfr_sin (x, x, MPFR_RNDZ);
          mpfr_subnormalize (x, inex, MPFR_RNDZ);
          mpfr_dump (x);
        }
      mpfr_clear (x);
    }
```

`void mpfr_clear_underflow (void)` [関数]
`void mpfr_clear_overflow (void)` [関数]
`void mpfr_clear_divby0 (void)` [関数]
`void mpfr_clear_nanflag (void)` [関数]
`void mpfr_clear_inexflag (void)` [関数]
`void mpfr_clear_erangeflag (void)` [関数]
それぞれアンダーフロー，オーバーフロー，ゼロ除算，不正な計算，不正確演算，範囲エラーフラグをクリアします。

`void mpfr_clear_flags (void)` [関数]
全てのグローバルフラグ (アンダーフロー，オーバーフロー，ゼロ除算，不正な計算，不正確演算，範囲エラー) をクリアします。

[注記] フラグのグループをまとめてクリアするための`mpfr_flags_clear`関数も使用可能です。

`void mpfr_set_underflow (void)` [関数]
`void mpfr_set_overflow (void)` [関数]
`void mpfr_set_divby0 (void)` [関数]
`void mpfr_set_nanflag (void)` [関数]
`void mpfr_set_inexflag (void)` [関数]
`void mpfr_set_erangeflag (void)` [関数]
アンダーフロー，オーバーフロー，ゼロ除算，不正な計算，不正確演算，範囲エラーフラグを立てます。

`int mpfr_underflow_p (void)` [関数]
`int mpfr_overflow_p (void)` [関数]
`int mpfr_divby0_p (void)` [関数]
`int mpfr_nanflag_p (void)` [関数]
`int mpfr_inexflag_p (void)` [関数]
`int mpfr_erangeflag_p (void)` [関数]
それぞれ，対応するフラグ (アンダーフロー，オーバーフロー，ゼロ除算，不正な計算，不正確演算，範囲エラー) を返します。フラグが立っている時のみ，非ゼロ数を返します。

以下の`mpfr_flags_`関数群は，*mask*という引数を取りますが，これは例外フラグを自在に操るためのものです。一つのフラグは，対応する*mask*の当該ビットがセットされている時のみ，例外フラグの集合の一部となります。`MPFR_FLAGS_`マクロは，この*mask*を設定するために使用されます。Section 4.6 [Exceptions], 頁 11 もご参照下さい。

`void mpfr_flags_clear (mpfr_flags_t mask)` [関数]
*mask*で指定されたフラグのグループをクリアします。

`void mpfr_flags_set (mpfr_flags_t mask)` [関数]
*mask*で指定されたフラグのグループを立てます。

mpfr_flags_t mpfr_flags_test (*mpfr_flags_t mask*) [関数]
*mask*で指定されたフラグを返します。*mask*にセットしたフラグが立っているかどうかは，返り値がゼロかどうかで判断できます。個別のフラグが立っているかどうかの判断は，MPFR_FLAGS_マクロとの AND を取ることで可能になります。

例)

```
mpfr_flags_t t = mpfr_flags_test (MPFR_FLAGS_UNDERFLOW|
                                  MPFR_FLAGS_OVERFLOW)
...
if (t)  /* アンダーフローかオーバーフロー */
  {
    if (t & MPFR_FLAGS_UNDERFLOW)  { /* アンダーフローを制御 */ }
    if (t & MPFR_FLAGS_OVERFLOW)   { /* オーバーフローを制御 */ }
  }
```

mpfr_flags_t mpfr_flags_save (*void*) [関数]
全てのフラグを返します。これは mpfr_flags_test(MPFR_FLAGS_ALL)と同じ意味になります。

void mpfr_flags_restore (*mpfr_flags_t flags*, *mpfr_flags_t mask*) [関数]
*mask*で特定されたフラグの状態を *flags*に格納します。

5.14 MPF との互換性

MPFR パッケージに同梱されている mpf2mpfr.hヘッダファイルは，GNU MP が提供する MPF 型との互換性を維持するためのものです。下記のように 2 行分を#include <gmp.h>の後に追加します。

```
#include <mpfr.h>
#include <mpf2mpfr.h>
```

このように使うことで，MPF 型を前提として作られたプログラムを，一切変更することなく MPFR でコンパイルすることができるようになります。

全ての計算はデフォルトの MPFR 丸めモードの下で行われますので，mpfr_set_default_rounding_mode関数を使って丸めモードの変更ができます。

[警告] MPF 型と MPFR ではいくつか異なる点があります。特に下記のようなものが挙げられます。

- 精度桁数が変わってきます。MPFR はきっちりビット単位で丸めます。(内部的には余りビットには 0 が詰め込まれています)。ユーザーは変数の精度桁数を増やしておく必要があります。
- 指数部の範囲も変わってきます。
- 書式指定付き出力関数 (gmp_printf関数など) は，任意精度の浮動小数点数型に対しては動作しません。mpf_t型は mpf2mpfr.hの中で mpfr_t型として再定義されます。

void mpfr_set_prec_raw (*mpfr_t x*, *mpfr_prec_t prec*) [関数]
*x*の精度桁数を正確に *prec*ビットに設定し直します。mpfr_set_prec関数との違いは，*prec*は仮数部が現状の変数 *x*のメモリ領域に収まる長さになっていると仮定していることです。そうでなければ，この関数の振る舞いは不定となります。

int mpfr_eq (*mpfr_t op1*, *mpfr_t op2*, *unsigned long int op3*) [関数]
*op1*と *op2*とが，ともに通常の浮動小数点数で，等しい指数部を持ち，仮数部の冒頭 *op3*ビットが等しいか，両方ともゼロか，同じ符号を持つ無限大であるか，これらのいずれかであれば，非ゼロ数を返します。この関数は MPF の対応する関数との互換性のために定義されたもので，それ以外の目的には使用しないことをお勧めします。この関数は，二数が近接しているかどう

かを確認したい時には使用しないで下さい。例えば，1.011111 と 1.100000 は，*op3*が 1 を超える時には等しくないものとして扱われます。

`void mpfr_reldiff` (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpfr_rnd_t* **rnd**) [関数]
*op1*と *op2*の相対差を計算し，その結果を *rop*に代入します。この関数は，相対差に関しては正しい丸めを保証しません。単純に $|op1 - op2|/op1$ を *rop*の精度桁数と *rnd*方式で計算して丸めます。

`int mpfr_mul_2exp` (*mpfr_t* **rop**, *mpfr_t* **op1**, *unsigned long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
`int mpfr_div_2exp` (*mpfr_t* **rop**, *mpfr_t* **op1**, *unsigned long int* **op2**, *mpfr_rnd_t* **rnd**) [関数]
これらの関数は`mpfr_mul_2ui`関数や`mpfr_div_2ui`関数と同一のものです。MPF の互換性のために残された関数なので，その必要がなければ`mpfr_mul_2ui`関数や`mpfr_div_2ui`関数の利用をお勧めします。

5.15 カスタムインターフェース

幾つかのアプリケーションでは，スタックを使ってメモリやオブジェクトの操作を行います。しかし，MPFR のメモリデザインは，そういう目的には向いていません。それ故，その手のアプリケーションで MPFR が使用できるようにするのであれば，補助的なメモリインターフェースがなくてはなりません。それがカスタムインターフェースです。

これらの機能を MPFR で使用できるようにするには，下記の二つの方法があります。

- `mpfr_t`型の変数を直接スタックに積む。
- 独自の表現形式でスタックに積み，必要に応じてその都度一時的な`mpfr_t`型変数を作る。

浮動小数点数を削除するには，使用したメモリをガベージコレクションに為すがままにさせておくしかありません。全てのメモリ管理機能（割り当て，破壊，解除) はアプリケーションにお任せするのです。

このインターフェースのための MPFR の機能は，高速性のためにマクロとして実装されています。例えば，`mpfr_custom_init (s, p)`はマクロを使って実行されますが，`(mpfr_custom_init) (s, p)` は関数を使用しています。

[注記 1] MPFR の関数は，一時的な浮動小数点数用のために`mpfr_init`関数や同様の関数を使って初期化を行います。詳細は GNU MP のカスタムアロケーションの解説を読んで下さい。

[注記 2] MPFR の関数は，内部的にキャッシュ用の関数 (`mpfr_const_pi`関数など）を利用します。従って，`mpfr_init`関数が GMP のカスタムアロケーション機能経由で呼ばれ，アプリケーションのスタック上にメモリが確保されるようであれば，メモリを廃棄する際には毎回`mpfr_free_cache`関数を呼ばなくてはなりません。

`size_t mpfr_custom_get_size` (*mpfr_prec_t* **prec**) [関数]
精度桁数 *prec*ビットの仮数部を格納するのに必要なバイト数を返します。

`void mpfr_custom_init` (*void* ***significand**, *mpfr_prec_t* **prec**) [関数]
精度桁数 *prec*ビットの仮数部を初期化します。ここで *significand*は`mpfr_custom_get_size (prec)`バイトのメモリ領域が最低でも必要で，`mp_limb_t`型 (GMP のデータ型，see Section 5.16 [Internals], 頁 48 参照) の配列になっていなければなりません。

`void mpfr_custom_init_set` (*mpfr_t* **x**, *int* **kind**, *mpfr_exp_t* **exp**, *mpfr_prec_t* **prec**, *void* ***significand**) [関数]
`mpfr_t`型のダミー初期化を実行し，次のように値をセットします。

- $|kind| =$ MPFR_NAN_KINDの時は，xには NaN がセットされる。
- $|kind| =$ MPFR_INF_KINDの時は，xには $kind$と同じ符号の無限大がセットされる。
- $|kind| =$ MPFR_ZERO_KINDの時は，xには $kind$と同じ符号のゼロがセットされる。
- $|kind| =$ MPFR_REGULAR_KINDの時は，xには $kind$と同じ符号，指数部と仮数部にはそれぞれ expと $significand$がセットされる。

全ての場合に共通するのは，$significand$が xを使う以降の計算に直接利用されるということです。この関数はメモリ領域の確保は行いません。この関数で初期化される浮動小数点数はmpfr_set_prec関数やmpfr_prec_round関数を使ったリサイズや，mpfr_clear関数を使ったメモリ解放は出来ません。$significand$は，同じ精度桁 $prec$を引数に渡したmpfr_custom_init関数を使って初期化しなくてはいけません。

int mpfr_custom_get_kind (*mpfr_t x*) [関数]
mpfr_custom_init_set関数を使って生成されたmpfr_t型データの種類を返します。mpfr_custom_init_set関数を使用せずに初期化されたmpfr_t型データが渡された時の動作は定義されていません。

void * mpfr_custom_get_significand (*mpfr_t x*) [関数]
mpfr_custom_init_set関数を使って初期化されたmpfr_t型データの仮数部へのポインタを返します。mpfr_custom_init_set関数を使用せずに初期化されたmpfr_t型データが渡された時の動作は定義されていません。

mpfr_exp_t mpfr_custom_get_exp (*mpfr_t x*) [関数]
xが非ゼロの通常の浮動小数点数で，仮数部が [1/2,1) にあると想定される時，xの指数部を返します。xが NaN，無限大，ゼロのように，mpfr_get_exp関数の挙動が定義されていない値である時，返り値は不定となりますが，mpfr_exp_t型の有効な数になります。mpfr_custom_init_set関数を使用せずに初期化されたmpfr_t型データが渡された時の動作は定義されていません。

void mpfr_custom_move (*mpfr_t x, void *new_position*) [関数]
MPFR に，xの仮数部がガベージコレクションによって移動され，新しい位置に収まっていると知らせます。とはいえ，アプリケーション自体が仮数部とmpfr_t型データを移動するようにしないといけません。mpfr_custom_init_set関数を使用せずに初期化されたmpfr_t型データが渡された時の動作は定義されていません。

5.16 MPFR の内部構造

リム (*limb*) は，1 ワードを単位とする多倍長精度浮動小数点数の構成要素を意味します。1 リムは通常，32 ビットか 64 ビットです。リムを表わす C データ型は mp_limb_tです。

mpfr_tデータ型は，内部的には構造体の配列の 1 つ分として定義されており，mpfr_ptr型は，この構造体へのポインタを表現しているデータ型です。mpfr_tデータ型は，次の 4 つのフィールドから構成されています。

- _mpfr_precフィールドは，変数の仮数部の精度桁数 (ビット数) を保持します。最小値はMPFR_PREC_MINです。
- _mpfr_signフィールドは，変数の符号を保持します。
- _mpfr_expフィールドには仮数部を格納します。指数部がゼロの時は，小数点はちょうど MSB(most significant digits) のすぐ上にあることになります。 n が非ゼロの時は，2^n を乗じたところに小数点が移動します。NaN，無限大，ゼロは，この指数部に特別な値を設定して識別します。
- 最後の_mpfr_dフィールドには，LSB(lease significant bits) が先頭に来るリム配列へのポインタが格納されます。使用するリム数は_mpfr_precで決定され，具体的には ceil(_mpfr_

`prec`/`mp_bits_per_limb`) となります。非数以外の通常の数は MSL(Most significant limb) の MSB(Most significant bits) が 1 になります。精度桁数がリム数とは一致しない時は，余った下のビットは全てゼロにします。

6 APIの互換性

本節では，MPFR のバージョンアップに伴って変化した API について解説し，古い MPFR でもコンパイルできるプログラムの書き方について述べます。但し，MPFR 2.2.0 (2005 年 9 月 20 日リリース）以降のものについてのみフォローします。

API の変更は，バージョン番号の最上位，もしくはその下の桁が変わった時に行われ，パッチレベルの変更（MPFR バージョン番号の第 3 桁目）では行われません。MPFR の内部構造を利用するようなプログラムでなければ，バグフィックスや意図しない動作による影響以上の変更は行われません。

一般的なルールとして，MPFR を利用するプログラムは，メジャーバージョンアップでもしない限り，多少のアップデートした MPFR でも動くように書くべきです。廃止予定と予告されている機能はそのうち使えなくなるので，そのような関数を利用していると，そのうちコンパイルやリンク時にエラーを起こすことになりかねません。アップデートに伴う変更が原因で結果がおかしくなるようなら，これ以降に記述してある，使用バージョンに関する FAQ や MPFR の Web ページにある変更点（最小限度にとどめていますので，ほとんどのソフトウェアでは影響はないでしょう）を確認してみて下さい。バグ混入や，既に修正されているものもあります。特に記述がない場合は，バグ報告を送って下さい (Chapter 3 [Reporting Bugs], 頁 6 参照)。

しなしながら，現在の MPFR を利用したプログラムは，このマニュアルに書いてある通り，以前のバージョンの MPFR で動作させる必要はないはずです。この節では，ポータブルなプログラムを書くための情報を提供します。

[注記] ここに記した情報は網羅的なものではありませんので，API の変更情報については MPFR のそれぞれのバージョンごとに提供される NEWS ファイルを参照して下さい。諸々の更新情報も併せて掲載されています。

6.1 データ型とマクロの変更

指数部の公式なデータ型を `mp_exp_t`から `mpfr_exp_t`に変更したのは MPFR 3.0 からです。`mp_exp_t`データ型は GMP ではこれからも違った意味で使用されると思われます。この二つのデータ型は現状同じものです (`mpfr_exp_t`は `typedef`で `mp_exp_t`が指定されている) ので，`mp_exp_t`を指数部のデータ型として使用することは今でも可能でが，将来は変更されるかもしれません。代わりに，`mpfr.h`をインクルードした後で `mpfr_exp_t`型が MPFR 2.x で使用されることがないよう，次の指定を行うと良いでしょう。

```
#if MPFR_VERSION_MAJOR < 3
typedef mp_exp_t mpfr_exp_t;
#endif
```

公式の精度桁数と丸めモードデータ型はそれぞれ `mp_prec_t`と `mp_rnd_t`から，`mpfr_prec_t`と `mpfr_rnd_t`に，MPFR 3.0 で変更されました。この変更は MPFR では大分以前から行わており，少なくとも MPFR 2.2.0 では下記のようなコードが `mpfr.h`:に入っています。

```
#ifndef mp_rnd_t
# define mp_rnd_t  mpfr_rnd_t
#endif
#ifndef mp_prec_t
# define mp_prec_t mpfr_prec_t
#endif
```

上記コードの意味は，新しい公式データ型である `mpfr_prec_t`型と `mpfr_rnd_t`型をあなたのプログラムで安全に使用することができる，というものです。データ型 `mp_prec_t`と `mp_rnd_t` (MPFR でのみ使用可能) は将来的に削除されるかもしれません。`mp_`という名前は GMP で予約されているものですから。

精度桁数のデータ型 mpfr_prec_t (mp_prec_t) は，MPFR 3.0 より前は符号なし (unsigned) でしたが，現在は符号付き (signed) です。MPFR_PREC_MAXは変更されていません。実際，MPFR のソースコードは MPFR_PREC_MAXが指数部のデータ型で表現可能であることを要求しており，これは mpfr_prec_tと同じデータサイズですが，常に符号付きでした。従って，新旧の MPFR バージョンを通じて使用できるプログラムを書くのであれば，mpfr_prec_tの符号あるなしに左右されないようにしましょう。通常の演算における型変換を考えた場合，符号なしデータ型を符号付きデータ型に変更するのは有効で，負数を通常の方法で型変換した時に，正しくない結果を招くことが防止できます。

[警告] 内部的にあろうが表面的にであろうが，mpfr_prec_tが符号付きであるという前提でプログラムを書くと，MPFR 2.x でコンパイルして実行した時に影響が出る可能性があります。

丸めモード名 GMP_RNDxは，MPFR 3.0 で MPFR_RNDxに変更されました。しかしながら旧名 GMP_RNDxも互換性を保つために有効になっています（が，将来変更するかも)。実際下記のような定義がされています。

```
#define GMP_RNDN MPFR_RNDN
#define GMP_RNDZ MPFR_RNDZ
#define GMP_RNDU MPFR_RNDU
#define GMP_RNDD MPFR_RNDD
```

丸めモード「ゼロから遠ざかる丸め」(MPFR_RNDA) は MPFR 3.0 で追加されました。ですが，GMP_RNDAという定義はありません。忠実丸め (MPFR_RNDF) は MPFR 4.0 で追加されましたが，現状では部分的なサポートにとどまっています。

MPFR_FLAGS_という名前で始まるフラグ関連のマクロは MPFR 4.0 で追加されました。新しい関数 mpfr_flags_clear，mpfr_flags_restore, mpfr_flags_set，mpfr_flags_testも同様です。

6.2 追加された関数

ここでは，アルファベット順に，MPFR 2.2 以降に追加されてきた関数と関数ライクなマクロを，追加時の MPFR バージョンと共に列挙します。

- mpfr_add_d関数は MPFR 2.4 で追加。
- mpfr_ai関数は MPFR 3.0 で追加 (不完全な実験的実装)。
- mpfr_asprintf関数は MPFR 2.4 で追加
- mpfr_beta関数は MPFR 4.0 で追加 (不完全な実験的実装)
- mpfr_buildopt_decimal_p関数は MPFR 3.0 で追加。
- mpfr_buildopt_float128_p関数は MPFR 4.0 で追加。
- mpfr_buildopt_gmpinternals_p関数は MPFR 3.1 で追加。
- mpfr_buildopt_sharedcache_p関数は MPFR 4.0 で追加。
- mpfr_buildopt_tls_p関数は MPFR 3.0 で追加。
- mpfr_buildopt_tune_case関数は MPFR 3.1 で追加。
- mpfr_clear_divby0関数は MPFR 3.1 で追加 (新しいゼロ除算例外追加も)。
- mpfr_copysign関数は MPFR 2.3 で追加。
  [注記] MPFR 2.2 で既に mpfr_copysign関数は使用可能になっていましたが，マニュアルには明記しておらず，動作も少し異なっていました（2 番目の引数が NaN の場合)。
- mpfr_custom_get_significand関数は MPFR 3.0 で追加。この関数は以前のバージョンで mpfr_custom_get_mantissa関数と命名されました。現在でも mpfr.hのマクロとして，この名前で使用可能です。

  ```
  #define mpfr_custom_get_mantissa mpfr_custom_get_significand
  ```

  上記の定義から分かるように，MPFR 2.x と MPFR 3.x では mpfr_custom_get_mantissa を使って下さい。

- mpfr_d_div関数と mpfr_d_sub関数は MPFR 2.4 で追加。
- mpfr_digamma関数は MPFR 3.0 で追加。
- mpfr_divby0_p関数は MPFR 3.1 で，新たにゼロ除算例外機能付きで追加。
- mpfr_div_d関数は MPFR 2.4 で追加。
- mpfr_erandom関数は MPFR 4.0 で追加。
- mpfr_flags_clear関数，mpfr_flags_restore関数，mpfr_flags_save関数, mpfr_flags_set関数，mpfr_flags_test関数は MPFR 4.0 で追加。
- mpfr_fmma関数と mpfr_fmms関数は MPFR 4.0 で追加。
- mpfr_fmod関数は MPFR 2.4 で追加。
- mpfr_fmodquo関数は MPFR 4.0 で追加。
- mpfr_fms関数は MPFR 2.3 関数で追加。
- mpfr_fpif_export関数と mpfr_fpif_import関数は MPFR 4.0 で追加。
- mpfr_fprintf関数は MPFR 2.4 で追加。
- mpfr_free_cache2関数は MPFR 4.0 で追加。
- mpfr_free_pool関数は MPFR 4.0 で追加。
- mpfr_frexp関数は MPFR 3.1 で追加。
- mpfr_gamma_inc関数は MPFR 4.0 で追加。
- mpfr_get_float128関数は MPFR 4.0 で追加。オプション '--enable-float128'付きで設定してビルドすると使用できます。
- mpfr_get_flt関数は MPFR 3.0 で追加。
- mpfr_get_patches関数は MPFR 2.3 で追加。
- mpfr_get_q関数は MPFR 4.0 で追加。
- mpfr_get_z_2exp関数は MPFR 3.0 で追加。この関数は，以前のバージョンでは mpfr_get_z_expという名前でした。mpfr_get_z_exp関数も，下記のように mpfr.hでマクロとして定義されていますので，今でも使用可能です。

  ```
  #define mpfr_get_z_exp mpfr_get_z_2exp
  ```

  従って，MPFR 2.x と MPFR 3.x の両方を使って動作するプログラムは mpfr_get_z_exp 関数の方を使うべきです。
- mpfr_grandom関数は MPFR 3.1 で追加。
- mpfr_j0関数, mpfr_j1関数，mpfr_jn関数は MPFR 2.3 で追加。
- mpfr_lgamma関数は MPFR 2.3 で追加。
- mpfr_li2関数は MPFR 2.4 で追加。
- mpfr_log_ui関数は MPFR 4.0 で追加。
- mpfr_min_prec関数は MPFR 3.0 で追加。
- mpfr_modf関数は MPFR 2.4 で追加。
- mpfr_mp_memory_cleanup関数は MPFR 4.0 で追加。
- mpfr_mul_d関数は MPFR 2.4 で追加。
- mpfr_nrandom関数は MPFR 4.0 で追加。
- mpfr_printf関数は MPFR 2.4 で追加。
- mpfr_rec_sqrt関数は MPFR 2.4 で追加。
- mpfr_regular_p関数は MPFR 3.0 で追加。
- mpfr_remainder関数と mpfr_remquo関数は MPFR 2.3 で追加。
- mpfr_rint_roundeven関数と mpfr_roundeven関数は MPFR 4.0 で追加。
- mpfr_round_nearest_away関数は MPFR 4.0 で追加。

- mpfr_rootn_ui関数は MPFR 4.0 で追加。
- mpfr_set_divby0関数は，新たにゼロ除算例外機能付きで MPFR 3.1 で追加。
- mpfr_set_float128関数は MPFR 4.0 で追加。‘--enable-float128’オプション付きでビルドすると使用できます。
- mpfr_set_flt関数は MPFR 3.0 で追加。
- mpfr_set_z_2exp関数は MPFR 3.0 で追加。
- mpfr_set_zero関数は MPFR 3.0 で追加。
- mpfr_setsign関数は MPFR 2.3 で追加。
- mpfr_signbit関数は MPFR 2.3 で追加。
- mpfr_sinh_cosh関数は MPFR 2.4 で追加。
- mpfr_snprintf関数と mpfr_sprintf関数は MPFR 2.4 で追加。
- mpfr_sub_d関数は MPFR 2.4 で追加。
- mpfr_urandom関数は MPFR 3.0 で追加。
- mpfr_vasprintf関数， mpfr_vfprintf関数， mpfr_vprintf関数， mpfr_vsprintf関数，mpfr_vsnprintf関数は MPFR 2.4 で追加。
- mpfr_y0関数, mpfr_y1関数，mpfr_yn関数は MPFR 2.3 で追加。
- mpfr_z_sub関数は MPFR 3.1 で追加。

6.3 変更された関数

ここに記した関数は MPFR 2.2 以降に変更がなされています。これらの変更により，使用する MPFR のバージョンとの組み合わせ次第でプログラムの挙動が変わるかもしれません。

- mpfr_abs関数，mpfr_neg関数，mpfr_set関数は，MPFR 4.0 で変更されました。以前の MPFR バージョンでは，NaN の符号ビットは不定でしたが，実用的になるよう，マニュアルに記述したように規定されるようになりました。但し，mpfr_neg関数に対して引数を mpfr_neg(x,x,rnd)のように再利用した場合は除きます。
- mpfr_check_range関数は MPFR 2.3.2 と MPFR 2.4 で変更されました。値が不正確な無限大の場合，現在はオーバーフローフラグが立っていない場合はセットされますが，以前は変更なしのままでした。これは実用上期待されているもの（かつ，MPFR のソースコードで必要としていたもの）ですが，かつての動作がバグであると見なされてるようになったため，MPFR 2.3.2 で変更されました。
- mpfr_eint関数は MPFR 4.0 で変更されました。現在は E1 の値とと，負の引数に対しては eint1 関数の値を返します。MPFR 4.0 より前のバージョンでは NaN を返していました。
- mpfr_get_f関数は MPFR 3.0 で変更されました。以前は，ゼロを返していましたが，NaN や無限大のように MPF 型には存在しない特殊値には対応していませんでした。現在では特殊値に対しては範囲エラーフラグが立てられ，mpfr_get_f関数は通常の三種値を返します。
- mpfr_get_si関数, mpfr_get_sj関数, mpfr_get_ui関数，mpfr_get_uj関数は MPFR 3.0 で変更されました。以前のバージョンでは，範囲エラーフラグが立つケースでは決まっていませんでした。
- mpfr_get_str関数は MPFR 4.0 で変更されました。現在では，NaN が入力値の時は NaN が返されます。これは NaN に関する MPFR のルールと IEEE 754-2008 の推奨する文字列変換 (5.12.1 項) に基づいた規則です。変換に際して丸め誤差が入ると，不正確フラグが立てられます。
- mpfr_get_z関数は MPFR 3.0 で変更されました。以前の返り値は void型でしたが，現在は int型になり，通常の三種値が返されます。従って，MPFR 2.x と 3.x の両方で動作するプログラムは，この関数の返り値を利用してはいけません。この場合でも，mpfr_get_z関数を使って，条件演算子の 2 番目，もしくは 3 番目の項指定するような C プログラムは影響を受

けます。例えば，下記のプログラムは MPFR 3.0 では正常に動きますが，MPFR 2.x ではうまく動きません。

```
bool ? mpfr_get_z(...) : mpfr_add(...);
```

一方，下記の例は MPFR 2.x では正常に動きますが，MPFR 3.0 ではうまくいきません。

```
bool ? mpfr_get_z(...) : (void) mpfr_add(...);
```

ポータブルなプログラムを書きたければ，`mpfr_get_z(...)`を `void`型にキャストして，下記のように条件演算子の両方の項を `void`型にします。

```
bool ? (void) mpfr_get_z(...) : (void) mpfr_add(...);
```

条件演算子の代わりに，`if ... else`も使用可能です。

さらに，範囲エラーフラグが立てられる場合は，MPFR 2.x では特に定められていませんでした。

- `mpfr_get_z_exp`関数は MPFR 3.0 で変更されました。以前の MPFR のバージョンでは，範囲エラーフラグが立つケースは特に定められていませんでした。

  [注記] この関数は MPFR 3.0 で `mpfr_get_z_2exp`という名前に変更されましたが，`mpfr_get_z_exp`関数も互換性維持のためにまだ使用できます。

- `mpfr_set_exp`関数は MPFR 4.0 で変更されました。MPFR 4.0 より前のバージョンでは，指数部は，引数に指定した MPFR オブジェクトの内容が何であれ，設定されていました。実用的には，MPFR 数を，内部構造の各フィールドごとに作り上げていくときには，低レベル関数としては便利に使えますが，API としては，内部構造を利用する場合を除いてそのような機能は提供しません。従って，API に対してはこの関数のような機能は無用の長物で，NaN，無限大，ゼロのような特殊数を使用する MPFR では，不正なフォーマットの値を作ってしまうかもしれません。

- `mpfr_strtofr`関数は MPFR 2.3.1 と MPFR 2.4 で変更されました。行われたのはバグフィックスで，ソースコードとマニュアルの記述が食い違っていましたが，整合性を取り，有用な動作になるように，両方を修正しました。ソースコードの主要な変更点は次の通りです。2 進表現の指数部を，`0b`や `0x`のようなプレフィックスなしでも受け付けるようにしました。また，以前は不正としていた符号付き NaN に対応するデータにも対応しました。

- `mpfr_strtofr`関数は MPFR 3.0 で変更されました。今では基数として 37 から 62 も受け付けるようになっています（これ以外の基数の場合については変更ありません）。

  [注記] サポート外の基数が与えられた時のこの関数の挙動は不定です。正確に言うと，MPFR 2.3.1 以降ではアサーションによる失敗を発生させます。この挙動については将来変更される可能性があります。

- `mpfr_subnormalize`関数は MPFR 3.1 で変更されました。行われたのはバグフィックスです。`mpfr_subnormalize`関数は，MPFR 3.0.0 まではフラグは立てませんでした。特に，不正確フラグについては一般的なルールに従っていませんでしたし，特に挙動も指定されていませんでした。アンダーフローフラグについてはもっと何も定まっていませんでした。

- `mpfr_sum`関数は MPFR 4.0 で変更されました。`mpfr_sum`関数は MPFR 4.0 で完全に書き直され，仕様もアップデートされました。演算結果が完全にゼロになる時の符号が規定され，返り値も通常の三種値になっています。旧 `mpfr_sum`関数の実装も，全てのメモリ上の値を扱うようになっていましたが，大きさのばらつきが大きい入力に対してはクラッシュする可能性がありました。

- `mpfr_urandom`関数と `mpfr_urandomb`関数は MPFR 3.1 で変更されました。この二つの関数の振る舞いは環境に依存しないようになっています。但し，GMP の乱数生成器も環境依存ではないという前提です。GMP 4.1 から 4.2 までのバージョンでは，`gmp_randinit_default`が使われているので環境依存になっています。結果として， 返り値は，MPFR 3.1 と，それ以前のバージョンの MPFR とでは違うものになることがあります。

  [注記] MPFR 3.1 より前では，これらの関数における値の再現性は特別指定していませんでした。従って，MPFR 3.1 での動作も以前のバージョンとの互換性は考えていません。

- mpfr_urandom関数は MPFR 4.0 で変更されました。次の乱数の状態は，現在の指数部の範囲と丸めモードとには依存せずに決まります。乱数の丸めに伴う例外は，一様分布に従って正しく生成されるようになりました。結果として，返される乱数は，MPFR 4.0 とそれ以前のバージョンとでは異なる可能性があります。

6.4 削除された関数

mpfr_random関数と mpfr_random2関数は MPFR 3.0 で削除されました。この変更は MPFR 3.0 より前のバージョン用に作った古いプログラムだけに影響します。mpfr_random関数は MPFR 2.2.0 以降から，mpfr_random2関数は MPFR 2.4.0 以降から，使用しないよう警告されています。

mpfr_add_one_ulp関数や mpfr_sub_one_ulp関数といったマクロは MPFR 4.0 で削除されました。これらの関数は，既に MPFR 2.1.0 でマニュアルに掲載しておりませんし，MPFR 3.1.0 以降は無効になった旨，明記されています。

mpfr_grandom関数は MPFR 4.0 で廃止予定の警告が出ています。以降のバージョンアップで廃止される予定です。

6.5 その他の変更点

C++コンパイラを使うのであれば，intmax_tを検出する方法が MPFR 3.0 で変更されたことに留意して下さい。MPFR 2.x では，INTMAX_Cマクロ，もしくは，UINTMAX_Cマクロが定義されている時，つまり，<stdint.h> もしくは <inttypes.h>をインクルードする前に__STDC_CONSTANT_MACROSマクロが定義されている時は，intmax_tは既に定義されているものと仮定します。しかし，必ずしもこの仮定は正しくありません。正確に言うと，intmax_tは，Boost ライブラリのように，名前空間 stdでのみ定義されていますので，コンパイルに失敗します，従って，C++コンパイラで INTMAX_C もしくは UINTMAX_Cのチェックするのは止め，次の方法を試して下さい。

- intmax_t型が必要な MPFR 2.x を使ったプログラムは，MPFR 3.0 以降ではコンパイルできないものと考えて下さい。mpfr.hをインクルードする前に，#define MPFR_USE_INTMAX_T が必要になります。
- 上記理由で，MPFR 3.0 で作ったプログラムを，MPFR 2.x でコンパイルすると失敗します。回避策としては，intmax_tと uintmax_tの定義をグローバルな名前空間で行うと良いかもしれません。

ゼロ除算例外は MPFR 3.1 で新たに追加されたものですが，この例外を使うのはこのバージョン以降に追加された新しい関数だけなので，この導入によって互換性に問題が生じることはありません。

MPFR 3.1 以降では，mpfr.hヘッダファイルを複数回インクルードできます。また追加された関数もあります (Section 4.1 [Headers and Libraries], 頁 7 参照)。

MPFR のメモリ割り当て方法については，MPFR 4.0 で整理されたものをそのまま順守すべきです。

7 MPFRとIEEE 754浮動小数点標準規格

この節では，MPFR と IEEE 754 規格の相違点と，IEEE 754 ではまだ規定がない挙動について解説します。

MPFR では原則として非正規化数を使用しません。IEEE 754 より広い指数部が扱えるので，IEEE 754 より非正規化数のメリットがなく，実装が面倒だからです。但し， `mpfr_subnormalize`関数を使って非正規化数のエミュレートはできるようになっています。

MPFR には NaN が一種類しかありません。基本的には，状態によって，シグナル NaN(sNaN) か沈黙 NaN(qNaN) と機能を持ちます。NaN を返す全ての関数 (NaN を生成，もしくはコピーしたりする) は NaN フラグを立てます。IEEE 754 の場合は，たとえシグナル NaN であっても，何も実行しません。

`mpfr_rec_sqrt`関数は IEEE 754 とは異なり，入力値が −0 である場合は，+0 が入力されたときと同じく+Inf を返します。IEEE 754 では通常の極限値計算結果に従って −Inf を返します。

`mpfr_root`関数は IEEE 754-2008 規格が固まる前に作ったものなので，n 乗根の扱いが異なります。なるべく `mpfr_rootn_ui`関数を使って下さい。

符号なしのゼロを用いた演算: 引数として整数型や有理数型を取る関数においては，浮動小数点数のゼロと異なり，符号なしのゼロしかありません。`unsigned long` 型でも同じく，数学的にはゼロはゼロ。対して，浮動小数点数のゼロは，アンダーフローの結果そうなることもあり，非ゼロ数の符号と同じものがついている訳です。従って，本マニュアルに明記してなくても，最初に `mpfr_set_ui`関数や `mpfr_set_si` 関数で変換された結果，+0 として扱われます。これは数学的な意味での極限値とは異なるものになっています。但し，加減算 (`mpfr_add_ui`等々) に対しては当てはまりませんので，+0 も −0 も同一のものとして扱います。このような演算結果をもたらす MPFR の現時点における仕様は，IEEE754 規格の枠外のもので，IEEE 754 改定を行う浮動小数点演算ワーキンググループでは議論したくない代物だったようです。

MPFR では，変数ごとに自身の精度桁数を持ち，この点 IEEE754 規格とは全く異なっているという事実も思い出して下さい。例えば，同一符号を持つ二数の減算を行うと，オーバーフローが発生する可能性があります。同様に，`mpfr_set`関数，`mpfr_neg`関数，`mpfr_abs`関数でも，精度桁数が少ないとオーバーフローが起きるかもしれません。

MPFR 貢献者一覧

MPFR のメイン開発陣は，Guillaume Hanrot，Vincent Lefèvre，Patrick Pélissier，Philippe Théveny，そして Paul Zimmermann です。

Sylvie Boldo（仏，ENS-Lyon）は `mpfr_agm`関数と `mpfr_log`関数の開発を行いました。Sylvain Chevillard は `mpfr_ai`関数の開発を担当しました。David Daney は双曲線関数，逆双曲線関数，2 のべき乗関数，階乗関数の開発を行いました。Alain Delplanque は新たな `mpfr_get_str`関数を開発しました。Mathieu Dutour は `mpfr_acos`関数, `mpfr_asin`関数，`mpfr_atan`関数，以前の `mpfr_gamma`関数を開発しました。Laurent Fousse は `mpfr_sum`関数のオリジナルバージョン（MPFR 3.1 以前) の開発を行いました。Emmanuel Jeandel（ENS-Lyon）は汎用の幾何級数コード，`mpfr_exp3` 内部関数，三角関数の初期バージョンの開発，`mpfr_const_log2`関数や `mpfr_const_pi`関数の改良を行いました, Ludovic Meunier は `mpfr_erf`関数プログラムの設計の助力を行いました。Jean-Luc Rémy は `mpfr_zeta`関数を開発しました。Fabrice Rouillier は `mpfr_xxx_z`関数，`mpfr_xxx_q`関数の開発を行いました。また，Microsoft Windows への移植の助力も行いました。Damien Stehlé は `mpfr_get_ld_2exp`関数を開発しました。Charles Karney は `mpfr_nrandom`関数と `mpfr_erandom` 関数を開発しました。

Jean-Michel Muller と Joris van der Hoeven には，本プロジェクトの初期段階で実り多き議論にお付き合い頂き，感謝しています。Torbjörn Granlund と Kevin Ryde は本ライブラリの設計に関して助力を惜しみませんでした。Nathalie Revol は本ドキュメントの草稿を丁寧に読んでいただきました。Kevin Ryde は，こと，2002 年から 2004 年にかけて，MPFR のポータビリティに関して多大なる貢献を惜しみませんでした。

MPFR ライブラリの開発は，INRIA, LORIA (Nancy, 仏)，LIP (Lyon, 仏) 研究所からの継続的なサポートなしでは不可能なプロジェクトでした。主要開発陣は，LORIA の PolKA, Spaces, Cacao, Caramel，Caramba プロジェクトチームや，LIP の Arénaire，AriC プロジェクトチームのメンバーなどから構成されています。本プロジェクトは，INRIA の Fiable (reliable の仏語) 活動の一つとして開始され，AOC 活動の一つとして継続されてきたものです。MPFR の開発は，様々な資金の支援を受けて行われてきました。以下列挙しますと，2002 年，Conseil Régional de Lorraine の 202F0659 00 MPN 121 支援金，2003-2005 年，INRIA より"associate engineer"支援金，2007-2009 年，"opération de développement logiciel"支援金，2009-2010 年， Sylvain Chevillard のポスドク支援金です。

2012 年 6 月の MPFR-MPC ワークショップは，Andreas Enge が受けた ERC grant ANTICS より一部補助されて行われました。

2013 年 1 月の MPFR-MPC ワークショップは， ERC grant ANTICS, GDR IM と Caramel プロジェクトチームからの補助を受けて行われましたが，この間，Mickael Gastineau は MPFRbench プログラムを開発し，Fredrik Johannsson はより高速な `mpfr_const_euler`関数を開発しました。

参考文献

- Richard Brent and Paul Zimmermann, "Modern Computer Arithmetic", Cambridge University Press, Cambridge Monographs on Applied and Computational Mathematics, Number 18, 2010. Electronic version freely available at `https://members.loria.fr/PZimmermann/mca/pub226.html`.
- Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier and Paul Zimmermann, "MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding", ACM Transactions on Mathematical Software, volume 33, issue 2, article 13, 15 pages, 2007, `http://doi.acm.org/10.1145/1236463.1236468`.
- Torbjörn Granlund, "GNU MP: The GNU Multiple Precision Arithmetic Library", version 6.1.2, 2016, `https://gmplib.org`.
- IEEE standard for binary floating-point arithmetic, Technical Report ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985: IEEE Standards Board; approved July 26, 1985: American National Standards Institute, 18 pages.
- IEEE Standard for Floating-Point Arithmetic, ANSI-IEEE Standard 754-2008, 2008. Revision of ANSI-IEEE Standard 754-1985, approved June 12, 2008: IEEE Standards Board, 70 pages.
- Donald E. Knuth, "The Art of Computer Programming", vol 2, "Seminumerical Algorithms", 2nd edition, Addison-Wesley, 1981.
- Jean-Michel Muller, "Elementary Functions, Algorithms and Implementation", Birkhäuser, Boston, 3nd edition, 2016.
- Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé and Serge Torrès, "Handbook of Floating-Point Arithmetic", Birkhäuser, Boston, 2009.

付記 A GNU Free Documentation License

Version 1.2, November 2002

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.

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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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 `http://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.

A.1 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 1.2
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

Function and Type Index