High-Precision Floating Point Numbers in LEDA *

Christoph Burnikel        Jochen Könemann

January 26, 1996

Contents

1. Introduction ............................................................... 2
2. The Manual Page of data type bigfloat .................................. 3
3. Representation of bigfloats ............................................ 7
5. The Header-File ....................................................... 8
10. File bigfloat.c .................................................... 12
11. General Functions .................................................. 13
17. Rounding .............................................................. 15
29. Arithmetical functions .............................................. 19
48. Comparison Operators .............................................. 27
52. Conversion between the data types bigfloat and double ........... 29
74. Functions for input and output ...................................... 36
90. References .......................................................... 42

*This work was supported in part by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141(ALCOM II) and the BMFT(Förderungskennzeichen ITS 9103).
1. Introduction

The data type `bigfloat` is the high-precision floating point type of LEDA. A bigfloat is a number of the form $s \cdot 2^e$ where $s$ and $e$ are integers. $s$ is called the significant or mantissa and $e$ is called the exponent. Arithmetic on bigfloats is governed by two parameters: the mantissa length and the rounding mode. Both parameters can either be set globally or for a single operation. The arithmetic on bigfloats works as follows: first the exact result of an operation is computed and then the mantissa is rounded to the prescribed number of digits as dictated by the rounding mode. The available rounding modes are `TO_NEAREST` (round to the nearest representable number), `TO_P_INF` (round upwards), `TO_N_INF` (round downwards), `TO_ZERO` (round towards zero), `TO_INF` (round away from zero) and `EXACT`. The latter mode only applies to addition, subtraction and multiplication. In this mode the precision parameter is ignored and no rounding takes place. Since the exponents of bigfloats are arbitrary integers (type integer) arithmetic operations never underflow or overflow. However, exceptions (division by zero, square root of a negative number) may occur. They are handled according to the IEEE floating point standard, e.g. $5/0$ evaluates to $\infty$, $-5/0$ evaluates to $-\infty$, $+\infty + 5$ evaluates to $+\infty$ and $0/0$ evaluates to $NaN$ (=not a number). This report is structured as follows. Section 2 defines the bigfloat through its manual page and the remaining sections contain the implementation. The implementation is split into files `bigfloat.h` and `bigfloat.c`. 
2. The Manual Page of data type bigfloat

1. Definition

In general a bigfloat is given by two integers $s$ and $e$ where $s$ is the significant and $e$ is the exponent. The tuple $(s, e)$ represents the real number

$$s \cdot 2^e.$$ 

In addition, a bigfloat can be in a special state like $\text{NAN}$ ($\equiv$ not a number), $\text{PZERO}$, $\text{NZERO}$ ($\equiv +0, -0$), and $\text{PINF}$, $\text{NINF}$ ($\equiv +\infty, -\infty$). bigfloats in special states behave as defined by the IEEE floating point standard. In particular, $\frac{5}{\infty} = \infty$, $-\frac{5}{\infty} = -\infty$, $\infty + 1 = \infty$, $\frac{5}{\infty} = +0$, $+\infty + (-\infty) = \text{NaN}$ and $0 \cdot \infty = \text{NaN}$ ($\text{NaN}$ is a bigfloat in special state $\text{NAN}$). Arithmetic on bigfloats uses two parameters: The precision $\text{prec}$ of the result (in number of binary digits) and the rounding mode $\text{mode}$. Possible rounding modes are:

- $\text{TO\_NEAREST}$: round to the closest representable value
- $\text{TO\_ZERO}$: round towards zero
- $\text{TO\_INF}$: round away from zero
- $\text{TO\_P\_INF}$: round towards $+\infty$
- $\text{TO\_N\_INF}$: round towards $-\infty$
- $\text{EXACT}$: do not round at all for $+, -, \ast$ and round to nearest otherwise

Operations $+, -, \ast$ work as follows. First, the exact result $z$ is computed. If the rounding mode is EXACT then $z$ is the result of the operation. Otherwise, let $s$ be the significant of the result; $s$ is rounded to $\text{prec}$ binary places as dictated by $\text{mode}$. Operations $/$ and $\sqrt{}$ work accordingly except that EXACT is treated as TO\_NEAREST.

The parameters $\text{prec}$ and $\text{mode}$ are either set directly for a single operation or else they are set globally for every operation to follow. The default values are 53 for $\text{prec}$ and TO\_NEAREST for $\text{mode}$.

2. Creation

A bigfloat may be constructed from data types double, long, int and integer, without loss of accuracy. In addition, an instance of type bigfloat can be created as follows.

$\text{bigfloat } x(\text{integer } s, \text{integer } e);$ 

introduces a variable $x$ of type bigfloat and initializes it $s \cdot 2^e$

$\text{bigfloat } x(\text{special\_values } sp);$ 

creates an instance of a special value of type bigfloat
3. Operations

The arithmetical operators +, -, *, /, sqrt, the comparison operators <, ≤, >, ≥, =, ≠ and the stream operators ≪ and ≫ are available.

Addition, subtraction, multiplication, division and square root are implemented by the functions add, sub, mul, div and sqrt, respectively. For example, the call

\[ \text{add}(x, y, \text{prec}, \text{mode}, \text{is\_exact}) \]

computes the sum of bigfloats \( x \) and \( y \) with \( \text{prec} \) binary digits, in rounding mode \( \text{mode} \), and returns it. The optional last parameter \( \text{is\_exact} \) is a boolean variable that is set to \( \text{true} \) if and only if the returned bigfloat exactly equals the sum of \( x \) and \( y \).

The parameters \( \text{prec} \) and \( \text{mode} \) are also optional and have the global default values \( \text{global\_prec} \) and \( \text{round\_mode} \) respectively; that is, the three calls \( \text{add}(x, y, \text{glob\_prec}, \text{round\_mode}) \), \( \text{add}(x, y, \text{glob\_prec}) \), and \( \text{add}(x, y) \) are all equivalent. The syntax for functions \( \text{sub}, \text{mul}, \text{div}, \) and \( \text{sqrt} \) is analogous.

The operators +, -, *, and / are implemented by their counterparts among the functions \( \text{add}, \text{sub}, \text{mul} \) and \( \text{div} \). For example, the call \( x + y \) is equivalent to \( \text{add}(x, y) \).

The rounded value of a bigfloat \( x \) can be obtained by

\[ \text{round}(x, \text{prec}, \text{mode}, \text{is\_exact}, \text{bias}) \]

Here \( \text{bias} \) is an optional long variable taking values in \( \{-1, 0, +1\} \), with default value 0. The function \( \text{round} \) rounds \( x + \text{bias} \cdot \varepsilon \), where \( \varepsilon \) is a positive infinitesimal, with \( \text{prec} \) binary digits in rounding mode \( \text{mode} \). The optional boolean variable \( \text{is\_exact} \) is set to \( \text{true} \) iff the rounding operation did not change the value of \( x \) and \( \text{bias} == 0 \).

bigfloats offer a small set of mathematical functions (e.g. abs, log2, ceil, floor, sign), functions to test for special values, conversions to doubles and integers, functions to access significant and exponent, and functions to set the global precision, the rounding mode and the output mode.

\[
\begin{align*}
\text{bool} & \quad \text{isNaN(bigfloat } x) & \quad \text{returns } \text{true} \text{ iff } x \text{ is in special state } \text{NAN} \\
\text{bool} & \quad \text{isnInf(bigfloat } x) & \quad \text{returns } \text{true} \text{ iff } x \text{ is in special state } \text{NINF} \\
\text{bool} & \quad \text{ispInf(bigfloat } x) & \quad \text{returns } \text{true} \text{ iff } x \text{ is in special state } \text{PINF} \\
\text{bool} & \quad \text{isnZero(bigfloat } x) & \quad \text{returns } \text{true} \text{ iff } x \text{ is in special state } \text{NZERO} \\
\text{bool} & \quad \text{ispZero(bigfloat } x) & \quad \text{returns } \text{true} \text{ iff } x \text{ is in special state } \text{PZERO}
\end{align*}
\]
bool isZero(bigfloat x) returns true iff isnZero(x) or isnZer o

bool isInf(bigfloat x) returns true iff isnInf(x) or isnInf(x)

bool isSpecial(bigfloat x)
returns true iff x is in a special state

long sign(bigfloat x) computes the sign of x.

long sign_of_special_value(bigfloat x)
computes the sign of special values. For example:
sign_of_special_value (bigfloat(PZERO)) == 1

bigfloat abs(bigfloat x) the absolute value of x

bigfloat pow2(integer p) returns \(2^p\)

integer log2(bigfloat x) returns the binary logarithm of x, rounded up to the next integer. Precondition: \(x > 0\)

integer ceil(bigfloat x) rounds x up to the next integer

integer floor(bigfloat x) rounds x down to the next integer

doubletodouble(bigfloat x)
returns the double value next to x (in rounding mode TO_NEAREST)

integer tointeger(bigfloat x, rounding_modes rmode = TO_ZERO)
returns the integer value next to x (in the given rounding mode)

os tream& os << x
writes x to output stream os

is tream& is >> bigfloat& x
reads x from input stream is

long x.get_precision(void)
returns the precision, i.e. the length of the significant of x.

integer x.get_exponent(void)
returns the exponent of x

integer x.get_significant(void)
returns the significant of x.
\begin{verbatim}
void bigfloat::set_glob_prec(long p)
    sets the global precision value to p

void bigfloat::set_round_mode(rounding_modes m = TO_NEAREST)
    sets the global rounding mode

void bigfloat::set_output_mode(output_modes o_mode = DEC_OUT)
    sets the output mode
\end{verbatim}
3. Representation of bigfloats

A bigfloat is stored as four quantities: integers *significant* and *exponent*, a flag *special* of enumeration type *special values* with elements *NOT*, *PZERO*, *NZERO*, *PINF*, *NINF*, and *NAN*, and a long *precision*. We maintain the following invariants:

1. if *special* ≡ *NOT* then the number represented by the bigfloat is
   
   \[
   \text{significant} \cdot 2^{\text{exponent}},
   \]

   and *precision* is the number of binary digits in the *significant*

2. if *special* ≠ *NOT* then *special* is the value of the bigfloat

4. A bigfloat number does not necessarily have a unique representation because there may be zeros at the end of the *significant*. However, some functions require a unique representation. We call a bigfloat *normalized* if its *significant* ends in a one. To guarantee uniqueness some functions normalize their input before working on it. To do this we have an internal procedure *normalize*. Warning: The value 0 can have many different representations. After calling the function *normalize* a bigfloat is special with value *PZERO* or *NZERO* iff its value is 0.

```
functions for internal use 4) ≡

void bigfloat :: normalize(void)
{
    if (special ≠ NOT) return;
    int signum = ::sign(significant);
    if ((signum ≡ 0) ∧ (special ≡ NOT)) special = PZERO;
    long z = significant.zeros();
    /* z is the number of final zeros in the significant */
    if (z > 0) significant = significant >> z;
    precision = significant.length();
    exponent += z;
}
```

See also chunks 19, 54, 57, 75, 76, 77, and 84.

This code is used in chunk 10.
5. The Header-File

The header file contains the prototypes of all the functions mentioned in the manual and some additional material: the definition of the rounding modes, the output modes, and the special values, the definition of some private functions, and the definition of the functions required by any LEDA type.

```c
#include <LEDA/integer.h>
#include <math.h>
enum rounding_modes { TO_NEAREST, TO_ZERO, TO_P_INF, TO_N_INF, TO_INF, EXACT };
enum output_modes { BIN_OUT, DEC_OUT };
enum special_values { NOT, PZERO, NZERO, PINF, NINF, NAN };
class bigfloat {
private:
    static long global_prec;
    static rounding_modes round_mode;
    static bool dool;
    static output_modes output_mode;
    // data members of class bigfloat
    // private functions
public:
    ~bigfloat() { }
    bigfloat();
    bigfloat(double);
    bigfloat(long);
    bigfloat(int);
    bigfloat(const integer &);
    bigfloat(const integer &, const integer &);
    bigfloat(special_values sp);    /* the default copy constructor and assignment
    operator are used (element-wise copy) */
    /* arithmetical functions and operators */
    friend bigfloat add(const bigfloat &, const bigfloat &, long prec = global_prec,
                       rounding_modes mode = round_mode, bool &is_exact = dool);
    friend bigfloat sub(const bigfloat &, const bigfloat &, long prec = global_prec,
                       rounding_modes mode = round_mode, bool &is_exact = dool);
    friend bigfloat mul(const bigfloat &, const bigfloat &, long prec = global_prec,
                       rounding_modes mode = round_mode, bool &is_exact = dool);
    friend bigfloat div(const bigfloat &, const bigfloat &, long prec = global_prec,
                       rounding_modes mode = round_mode, bool &is_exact = dool);
    friend bigfloat sqrt(const bigfloat &, long prec = global_prec, rounding_modes
                       mode = round_mode, bool &is_exact = dool);
    friend bigfloat operator+(const bigfloat &a, const bigfloat &b) { 
        return add(a, b);  
    }
```
friend bigfloat operator-(const bigfloat &a, const bigfloat &b) {
    return sub(a, b); }
friend bigfloat operator *(const bigfloat &a, const bigfloat &b) {
    return mul(a, b); }
friend bigfloat operator/(const bigfloat &a, const bigfloat &b) {
    return div(a, b); }
friend bigfloat operator-(const bigfloat &); /* comparison operators */
friend bool operator==(const bigfloat &, const bigfloat &);
friend bool operator>(const bigfloat &, const bigfloat &);
friend bool operator!=(const bigfloat &a, const bigfloat &b) {
    return !((a == b) || (a > b)); }
friend bool operator>=(const bigfloat &a, const bigfloat &b) {
    return !((a < b) && (a == b)); }
friend bool operator<=(const bigfloat &a, const bigfloat &b) {
    return !((a > b) && (a == b)); }
friend bool operator<=(const bigfloat &a, const bigfloat &b) {
    return !((a < b) && (a == b)); } /* rounding */
friend bigfloat round(bifloat x, long digits = 0, rounding_modes
mode = round_mode, bool &is_exact = dbool, long bias = 0);
/* tests for special values */
inline friend bool isnNaN(const bigfloat &x) {
    return (x specials == NAN); }
inline friend bool isnInf(const bigfloat &x) {
    return (x specials == NINF); }
inline friend bool isnInf(const bigfloat &x) {
    return (x specials == PINF); }
inline friend bool isnZero(const bigfloat &x) {
    return (x specials == NZERO); }
inline friend bool isnZero(const bigfloat &x) {
    return (x specials == NZERO); }
inline friend bool isZer0(const bigfloat &x) {
    return (x specials == PZERO); }
inline friend bool isZer0(const bigfloat &x) {
    return (x specials == PZERO); }
inline friend bool isInf(const bigfloat &x) {
    return (x specials == PINF) || (x specials == NINF); }
inline friend bool isInf(const bigfloat &x) {
    return (x specials == PINF) || (x specials == NINF); }
inline friend bool isSpecial(const bigfloat &x) {
    return (x specials != NOT); } /* mathematical functions */
friend long sign(const bigfloat &x);
friend long sign_of_special_value(const bigfloat &x);
inline friend bigfloat abs(const bigfloat &x) {
    return (x < bigfloat (PZERO)) ? -x : x; }
inline friend bigfloat pow2(const integer &p) {
    return bigfloat (1, p); }
inline friend integer log2(const bigfloat &x) {
    return x.get_precision() + x.get_exponent(); }
inline friend integer ceil(const bigfloat &x)  
{ return tointeger(x, TO_P_INF); }  
inline friend integer floor(const bigfloat &x)  
{ return tointeger(x, TO_N_INF); }  
/ * conversion functions */  
double friend todouble(const bigfloat &x);  
integer friend tointeger(bigfloat x, rounding_modes rmode = TO_ZERO);  
/ * input/output operations */  
friend ostream &operator<<(ostream &os, const bigfloat &x);  
friend istream &operator>>(istream &is, bigfloat &x);  
/* access functions */  
long get_precision(void) const  
{ return precision; }  
integer get_exponent(void) const  
{ return exponent; }  
integer get_significant(void) const  
{ return significant; }  
/ * functions to set global constants */  
static void set_glob_prec(long p)  
{ global_prec = p; }  
static void set_round_mode(rounding_modes m = TO_NEAR)  
{ round_mode = m; }  
static void set_output_mode(output_modes o_mode = DEC_OUT)  
{ output_mode = o_mode; }  
};  
< LEDA functions 6 >
# endif

6. The following functions have to be defined for every LEDA type.
< LEDA functions 6 > ≡
inline void Print(const bigfloat &x, ostream &out) { out << x; }  
iline void Read(bigfloat &x, istream &in) { in >> x; }  
iline int compare(const bigfloat &x, const bigfloat &y) { return sign(x - y); }  
iline char * TypeName(const bigfloat *) { return "bigfloat"; }  
This code is used in chunk 5.

7. We need to define some additional functions for internal use: normalize removes trailing zeroes in the significant, error_in_rounding returns the error made in rounding.
< private functions 7 > ≡
void normalize();  
bigfloat error_in_rounding(long digits = 0, rounding_modes mode = round_mode,  
bool &is_exact = dbool, long bias = 0);  
This code is used in chunk 5.
8. The static elements have to be initialized.

\[
\text{Initialization of static members 8) } \equiv \\
\quad \text{long bigfloat} :: \text{global_prec} = 53; \\
\quad \text{rounding\_modes bigfloat} :: \text{round\_mode} = \text{TO\_NEAREST}; \\
\quad \text{bool bigfloat} :: \text{dbool} = \text{true}; \\
\quad \text{output\_modes bigfloat} :: \text{output\_mode} = \text{DEC\_OUT};
\]

This code is used in chunk 10.

9. The following section contains the data members of data type bigfloat.

\[
\text{data members of class bigfloat 9) } \equiv \\
\quad \text{integer significant, exponent; } \\
\quad \text{special\_values special; } \\
\quad \text{long precision;}
\]

This code is used in chunk 5.
10. File bigfloat.c

The file bigfloat.c has a simple structure. At first the static members of class bigfloat are initialized. Afterwards we define some global identifiers needed by some later defined functions immediately followed by some auxiliary functions, e.g. the *sign* function for the data types integer and long. Then the definition of functions for internal use like *normalize* and the definition of public member functions are given. The member functions come in five big chunks: constructors, general functions (*round*, *conversion*, *sign*, ...), arithmetical functions, comparison operators and input/output operators.

```c
#include "bigfloat.h"
#include <iostream.h>
#include <sstream.h>
#include <stdio.h>
#include <unistd.h>
#include <cstring.h>

#include <string.h>

// Initialization of static members 8
// global identifiers 58
// auxiliary functions 16
// functions for internal use 4 /* member functions */
// constructors 12
// general functions 14
// arithmetical functions 30
// comparison operators 48
// input/output operators 82
```
11. General Functions

12. Simple Constructors.

A bigfloat can be constructed from an int, a long, a special value, a pair of integers and a double. All but the last one are trivial and given now. The bodies of the constructors from data types int, long and integer are all the same and hence are collected in a special refinement. The default constructor constructs a bigfloat with value PZERO.

\[
\begin{align*}
\langle \text{constructors } 12 \rangle & \equiv \\
\text{bigfloat} \colon& \text{bigfloat}() \\
& \{ \\
& \hspace{1em} \text{special} = \text{PZERO}; \\
& \hspace{1em} \text{exponent} = \text{significant} = \text{precision} = 0; \\
& \} \\
\text{bigfloat} \colon& \text{bigfloat}(\text{const integer } \& s, \text{const integer } \& e) \\
& \{ \hspace{1em} \text{if} \ (s \neq 0) \ {\{} \\
& \hspace{2em} \text{significant} = s; \\
& \hspace{2em} \text{exponent} = e; \\
& \hspace{2em} \text{special} = \text{NOT}; \\
& \hspace{2em} \text{precision} = s.\text{length}(); \\
& \hspace{1em} \} \\
& \text{else} \ {\{} \\
& \hspace{2em} \text{special} = \text{PZERO}; \\
& \hspace{2em} \text{exponent} = \text{significant} = \text{precision} = 0; \\
& \hspace{1em} \} \\
\text{bigfloat} \colon& \text{bigfloat}(\text{special\_values } sp) \ {\{} \ \text{special} = sp; \ {\}} \\
\text{bigfloat} \colon& \text{bigfloat}(\text{const integer } \& a) \\
& \{ \langle \text{constructor body for integer data type 13} \rangle \} \\
\text{bigfloat} \colon& \text{bigfloat}(\text{long } a) \\
& \{ \langle \text{constructor body for integer data type 13} \rangle \} \\
\text{bigfloat} \colon& \text{bigfloat}(\text{int } a) \\
& \{ \langle \text{constructor body for integer data type 13} \rangle \}
\end{align*}
\]

See also chunk 59.

This code is used in chunk 10.

13. (constructor body for integer data type 13) \(\equiv\)

\[
\begin{align*}
\text{special} &= \text{NOT}; \\
\text{significant} &= a; \\
\text{exponent} &= 0; \\
\text{precision} &= \text{significant}.\text{length}(); \\
\text{if} \ (\text{significant} \equiv 0) \ \text{special} &= \text{PZERO};
\end{align*}
\]

This code is used in chunk 12.

14. The function \textit{sign} computes the sign of an instance of data type bigfloat. It returns a -1 if the bigfloat represents an negative value, a zero if it is zero and otherwise a 1.
15. The function `sign_of_special_value` returns a nonzero long value. The function enables the user to determine the sign of 0 or \( \infty \). If this function is called with argument `NAN` an error message will be thrown out. The call of function `normalize` at the beginning guarantees that a bigfloat with value 0 is represented by a special value. On non special values this function performs the same actions as the `sign` function for data type `bigfloat`.

16. Before we come to the core of the bigfloat implementation, we list some useful functions to compute the maximum of two longs, the square of two integers, the binary logarithm of a double and the sign for types `long` and `int`, respectively.
17. Rounding

The round function is the central tool for the implementation of bigfloat arithmetic. Recall that \( \text{round}(x, \text{digits}, \text{mode}, \text{is_exact}, \text{bias}) \) rounds the number

\[
(significant + bias \cdot \epsilon) \cdot 2^{\text{exponent}}
\]

to \text{digits} binary digits in rounding mode \text{mode}, where \( \epsilon \) is a positive infinitesimal and bias is out of \([-1, 0, 1]\). \text{is_exact} is set to \text{true} if the rounded number equals to \( x \).

18. Now we outline the implementation of function \text{round}. At first the bigfloat is normalized in order to avoid final zeros of the significant. Then we distinguish cases. It might be that the significant has more places than specified by parameter \text{digits}. In this case the bias can be neglected, with the one exception of the \text{TO_NEAREST} mode (details see below). In all cases except for \text{EXACT} the significant is rounded to \text{digits} places and the exponent is adapted accordingly. On the other hand, it might be that the length of the significant is less than \text{digits}. In that case the return value depends on the bias (again with the exception of mode \text{TO_NEAREST}).

\[
\text{bigfloat \text{round}(\bigfloat x, \text{long \space digits}, \text{rounding\_modes \space mode}, \text{bool &is\_exact, \text{long \space bias})}
\]

\[
\{\text{x.normalize();}\text{if \( (\text{isSpecial(x)} \lor (\text{mode} \equiv \text{EXACT})) \{\text{is\_exact} = \text{true};\text{return \( x;\)}\}}\text{int test} = 0;\text{long shift;}\text{if \( (\text{digits} < \text{x.precision}) \{\text{switch (mode) \{\text{case \text{TO\_NEAREST}: \{\text{round to nearest 20}\text{break;}\text{case \text{TO\_ZERO}: \{\text{round to zero 22}\text{break;}\text{case \text{TO\_P\_INF}: \{\text{round to plus infinity 24}\text{break;}\text{case \text{TO\_N\_INF}: \{\text{round to minus infinity 25}\text{break;}\text{case \text{TO\_INF}: \{\text{round to infinity 23}\text{break;\}}\}}\}}\}}\}
\]
```c
x.exp = (x.precision - digits);

else if (bias != 0) {
  bias_rounding 26
  is_exact = (digits >= x.precision) && (bias == 0);
  x.normalize();
  return x;
}
```

19. We need a function that cuts an integer to an arbitrary amount of digits.

```c
{functions for internal use 4} +=
  void cut(integer &b, long prec) {
    b = (b >> (b.length() - prec));
  }
```

20. First we implement rounding in the case that the significant of the normalized bigfloat has more than digits places and hence the rounded value is not equal to the exact value.

We start with the rounding case **TO_NEAREST**. Write the significant as \( x_1 \cdot 2^{\text{precision}-\text{digits}} + x_2 \) and recall that \( x_2 \) is odd since \( x \) is normalized. If \( x_2 \) starts with a zero then \( x_1 \) is the rounded significant. If \( x_2 \) starts with a one and has more than one digit, then \( x_1 + \text{sign}(x_1) \) is the rounded significant. If \( x_1 \) has only one digit of value one the bias decides the rounding.

```c
{round to nearest 20} +=
  cut(x.significant, digits + 1);
  test = (significant is even 21);
  cut(x.significant, digits);
  if (!test) {
    /* \( x_2 \) starts with a one */
    if ((x.precision > digits + 1) || ::sign(bias) == ::sign(x.significant)) {
      /* \( x_1 + \text{sign}(x_1) \) is the rounded value */
      if (::sign(x.significant) > 0) x.significant++;
      else x.significant--;
    }
  }
```

This code is used in chunk 18.

21. It is simple to test whether the significant of \( x \) is even.

```c
{significant is even 21} +=
  (x.significant & integer(1)) == 0
```

This code is used in chunk 20.

22. In the **TO_ZERO** case we cut the significant to the wanted amount of digits.

```c
{round to zero 22} +=
  cut(x.significant, digits);
```

This code is used in chunk 18.
23. In the TO_INF case we always add ±1 because we already know that the rounded value is not exact.

\[
\{ \text{round to infinity 23} \} \equiv \\
\text{cut}(x.\text{significant},\text{digits}); \\
\text{if} \ (::\text{sign}(x.\text{significant}) > 0) \ x.\text{significant}++; \ \textbf{else} \ x.\text{significant}--; \\
\text{This code is used in chunk 18.}
\]

24. The next case is TO_P_INF. Here the significant is rounded up. If the number is negative this is done by cutting significant to length digits. Otherwise we increment the significant of x after the cutting.

\[
\{ \text{round to plus infinity 24} \} \equiv \\
\text{cut}(x.\text{significant},\text{digits}); \\
\text{if} \ (::\text{sign}(x.\text{significant}) > 0) \ x.\text{significant}++; \\
\text{This code is used in chunk 18.}
\]

25. The TO_N_INF-mode is the opposite case to TO_P_INF.

\[
\{ \text{round to minus infinity 25} \} \equiv \\
\text{cut}(x.\text{significant},\text{digits}); \\
\text{if} \ (::\text{sign}(x.\text{significant}) < 0) \ x.\text{significant}--; \\
\text{This code is used in chunk 18.}
\]

26. Now we come to the cases where the wanted precision is less or equal to the original precision and bias is crucial. Again we consider the rounding modes separately. For mode TO_NEAREST there is nothing to do. In the other cases, it depends on the sign of the bias whether we have to do nothing, or else we have to add a 1 or −1 at the digits th place.

\[
\{ \text{bias rounding 26} \} \equiv \\
\text{shift} = \text{digits} - x.\text{precision}; \\
\text{long} \ bf_\text{sign} = ::\text{sign}(x.\text{significant}); \\
\text{switch} \ (\text{mode}) \ { \\
\text{case} \ TO\_ZERO: \\
\text{if} \ (::\text{sign}(\text{bias}) \neq bf_\text{sign}) \ { \\
\{ \text{shift significant 27} \} \\
\text{if} \ (bf_\text{sign} \equiv 1) \ x.\text{significant}--; \ \textbf{else} \ x.\text{significant}++; \\
\} \\
\text{break;} \\
\text{case} \ TO\_INF: \\
\text{if} \ (::\text{sign}(\text{bias}) \equiv bf_\text{sign}) \ { \\
\{ \text{shift significant 27} \} \\
\text{if} \ (bf_\text{sign} \equiv 1) \ x.\text{significant}++; \ \textbf{else} \ x.\text{significant}--; \\
\} \\
\text{break;} \\
\text{case} \ TO\_P\_INF: \\
\text{if} \ (::\text{sign}(\text{bias}) \equiv 1) \ { \\
\{ \text{shift significant 27} \}
\text{break;} \\
\}
\]
\[
x.\text{significant}++; \\
\} \\
\text{break; case } \text{TO.N.INF:} \\
\text{if } ((\text{sign(bias)} = -1) \{ \\
\text{shift significant } 2\text{7} \\
x.\text{significant}--; \\
\} \\
\text{break; /* in the cases } \text{TO.NEAREST} \text{ and } \text{EXACT} \text{ we have to do nothing */} \\
\text{case } \text{TO.NEAREST: case } \text{EXACT: ;} \\
\}
\]

This code is used in chunk 18.

27. When shifting the \textit{significant} left by \textit{shift} digits we have to lower the bigfloat's exponent by the same amount.

\[
\text{\texttt{\{shift significant 27\}}} = \\
\text{\texttt{if} } \text{\textit{shift} > 0} \{ \\
x.\text{significant} = x.\text{significant} \ll \text{shift}; \\
x.\text{exponent} -= \text{shift}; \\
\} \\
\]

This code is used in chunk 26.

28. A bigfloat \(x\) can also be rounded to integer format by the function \texttt{to_integer}. If \(x\) represents a nonzero special value, we throw out an error message. Now let us assume that \(x\) is non special. We round \(x\) to \(\text{length} = \max(x.\text{precision} + x.\text{exponent}, 1)\) places. The choice of \(\text{length}\) guarantees that the rounding returns an integer value nearest to the original fractional value of \(x\). In particular, the \textit{exponent} of \(x\) is nonnegative after rounding. Due to the choice of \(\text{length}\), \(x\) is representable with an \textit{exponent} of value zero. If the final \texttt{normalize} in function \texttt{round} produced an \textit{exponent} \(\geq 0\) the \textit{significant} of \(x\) has to be shifted to the left by this amount of binary places. Afterwards \(x.\text{significant}\) might be returned as the result.

\[
\text{\texttt{(general functions 14) +\texttt{=} \{}} \\
\text{integer tointeger(bigfloat } x, \text{rounding_modes } rmode) \\
\text{\{}} \\
\text{if } ((\text{isNaN}(x)) \lor (\text{isInf}(x))) \text{ error\_handler}(1, \\
\text{"tointeger\_special\_values\_cannot\_be\_converted\_to\_integer"}); \\
\text{if } (-x.\text{exponent} + (\text{integer}) x.\text{significant}.\text{length(}) ).\text{islong}( ) \\
\text{error\_handler}(1, "{\text{tointeger\_\((exp+sig\_len\)_has\_to\_be\_in\_long\_range"}); \\
\text{long length} = \max(1, x.\text{precision} + x.\text{exponent}.\text{tolong}()); \\
x = \text{round}(x, \text{length}, rmode); \\
\text{if } \text{\texttt{(isZero}}(x)) \text{ \texttt{return}} \text{integer}(0); \\
\text{else return } (x.\text{significant} \ll x.\text{exponent}.\text{tolong}()); \\
\text{\}} \\
\text{\}} \\
\text{\}}
\]
29. Arithmetical functions

30. The Add-Function.

In the following sections we explain the arithmetical functions \textit{add}, \textit{sub}, \textit{mul}, \textit{div}, and \textit{sqrt}. We start right off with the \textit{add} function.

Let \(a\) and \(b\) be the operands of the addition. At first we make sure that the binary logarithm of \(a\) (rounded up to the next integer) is not less than the binary logarithm of \(b\). Then we compute bigfloats \(\textit{sum}\) and \(\textit{error}\) such that we have the exact equality

\[
a + b = \text{sum} + \text{error}.
\]

Here \(\textit{error}\) is bounded by half the least significant digit of \(\text{sum}\) and \(\text{error} \equiv 0\) for \(\textit{mode} \equiv \text{EXACT}\). Finally we give back the rounded value of \(\text{sum}\). Procedure \textit{round} only needs to know the sign of \(\textit{error}\).

\[
\langle\text{arithmetical functions 30} \rangle \equiv
\]

\[
\text{bigfloat add}(\text{const bigfloat} \ & x, \text{const bigfloat} \ & y, \text{long prec}, \text{rounding_modes} \ & \text{mode}, \text{bool} \ & \text{is}._{\text{exact}})
\]

\{

\[
\text{bigfloat } a, b;
\]

\{ handle special cases 35 \}

\{ find bigger operand 31 \} /* now \lceil \log_2 a \rceil \geq \lceil \log_2 b \rceil */

\[
\text{bigfloat \ text{sum}, \ error};
\]

\{ compute sum and error 32 \}

\[
\text{return round(sum, prec, mode, is}_{\text{exact}}, \text{sign(error))};
\}

See also chunks 36, 37, 39, 43, and 47.

This code is used in chunk 10.

31. It is helpful to know the \textit{bigger} operand. To decide which operand is bigger we compute for \(x\) and \(y\) the sums of exponent and precision, called \(\log_x\) and \(\log_y\). The difference \(\text{diff}\) of these quantities must not be confused with the difference of the exponents, \(\text{exp}_\text{diff}\).

\[
\langle\text{find bigger operand 31} \rangle \equiv
\]

\[
\text{bigfloat *a.ptr, *b.ptr;}
\]

\[
\text{integer \ log}_x = x._{\text{exponent}} + (\text{integer} \ x._{\text{precision}});
\]

\[
\text{integer \ log}_y = y._{\text{exponent}} + (\text{integer} \ y._{\text{precision}});
\]

\[
\text{integer \ diff = log}_x - log_y;
\]

\[
\text{if (diff} \geq 0) \{ \ a.ptr = (\text{bigfloat} \ *) \ & x; \ b.ptr = (\text{bigfloat} \ *) \ & y; \}
\]

\[
\text{else} \{ \ a.ptr = (\text{bigfloat} \ *) \ & y; \ b.ptr = (\text{bigfloat} \ *) \ & x; \}
\]

\[
a = *a.ptr;
b = *b.ptr;
\]

\[
\text{diff._{absolute}();}
\]

\[
\text{integer \ exp}_\text{diff} = a._{\text{exponent}} - b._{\text{exponent}};
\]

This code is used in chunk 30.
32. Often it is unnecessary to compute the exact sum. If \( b \) has no influence on the addition’s result it suffices to set \( \text{sum} \) to \( a \) and \( \text{error} \) to \( b \). This is the case if \( b \) is less than the least significant digit of \( a \) and also less than the \((\text{prec} + 1)\)th digit of \( a \). (Here we can assume that \( a \) has at least \( \text{prec} + 1 \) digits by conceptually shifting left, if it has less digits.) These restrictions can be formulated as \( \text{diff} > \text{a.precision} \) and \( \text{diff} > (\text{prec} + 1) \). Furthermore the rounding mode must be different from \textsc{Exact}. If one of the conditions is violated we compute the addition exactly.

\[
\begin{align*}
\{ \text{compute sum and error 32} \} & \equiv \\
\text{if } ((\text{mode} \neq \textsc{Exact}) \land (\text{diff} > \text{max}(\text{prec} + 1, \text{a.precision}))) \\
& \{ \\
& \text{sum} = a; \\
& \text{error} = b; \\
& \} \\
\text{else } \{ \\
& \{ \text{exact addition 34} \} \\
& \}
\end{align*}
\]
This code is used in chunk 30.

33. The exact addition of two bigfloat numbers can easily be attributed to integer addition. The precondition for the use of integer arithmetic is that the two bigfloat operands have equal exponents. In this case the result can be calculated as follows:

\[
\begin{align*}
\{ \text{calculate sum 33} \} & \equiv \\
\text{sum} = \text{bigfloat}(a.\text{significant} + b.\text{significant}, a.\text{exponent});
\end{align*}
\]
This code is used in chunk 34.

34. If \( a.\text{exponent} > b.\text{exponent} \) we shift \( a.\text{significant} \) leftwards by \( \text{exp.dif} \equiv a.\text{exponent} - b.\text{exponent} \) binary places. Similarly, if \( b.\text{exponent} > a.\text{exponent} \) we shift \( b.\text{significant} \) leftwards by \( -\text{exp.dif} \) places. In both cases the exponents have to be set to the smallest exponent of \( a \) and \( b \).

\[
\{ \text{exact addition 34} \} \equiv \\
\text{if } (\neg \text{exp.dif}.\text{islong}()) \\
\text{error.handler}(1,"\text{bigfloat::add()}\|\text{exponential}\|\text{difference}\\
\text{ex}^\text{has to be in \text{long\_range}}"); \\
\text{if } (\text{exp.dif} > 0) \{ \\
\text{a}\_\text{significant} = a.\text{significant} \ll \text{exp.dif}.\text{tolong}(); \\
\text{a}\_\text{exponent} = b.\text{exponent}; \\
\} \\
\text{else } \{ \\
\text{b}\_\text{significant} = b.\text{significant} \ll (\neg \text{exp.dif}).\text{tolong}(); \\
\text{b}\_\text{exponent} = a.\text{exponent}; \\
\}
\}
\{ \text{calculate sum 33} \}
\]
This code is used in chunk 32.
35. The rules to handle special cases in addition are simple. The result is NaN if one of the
operands is NaN or for the sums $\infty + (-\infty)$ and $(-\infty) + \infty$. In the other cases we
return the sum of $x$ and $y$.

\[
\begin{align*}
\langle \text{handle special cases 35} \rangle & \equiv \\
((\text{bigfloat} \& \ x)\text{.normalize}()) & ; \\
((\text{bigfloat} \& \ y)\text{.normalize}()) & ; \\
\text{if } (\text{isSpecial}(x) \lor \text{isSpecial}(y)) & \{ \\
\text{if } (\text{isNaN}(x) \lor \text{isNaN}(y)) & \text{ return bigfloat}(\text{NAN})\; ; \\
\text{if } (\text{isZero}(x)) & \text{ return y} \; ; \\
\text{if } (\text{isZero}(y)) & \text{ return x} \; ; \\
\text{if } (\text{isInf}(x) \wedge \text{isInf}(y)) & \{ \\
\text{if } (\text{sign_of_special_value}(x) = \text{sign_of_special_value}(y)) & \text{ return x} \; ; \\
\text{else } & \text{ return bigfloat}(\text{NAN})\; ; \\
\text{if } (\text{isInf}(x)) & \text{ return x} \; ; /* it is obvious that $y$ has to be $\infty */ \\
\text{return y} & \}
\end{align*}
\]
This code is used in chunk 30.

36. The Sub-Function.
This function is quite easy because it can be reduced to the \textit{add} function.

\[
\langle \text{arithmetical functions 30} \rangle +\equiv
\begin{align*}
\text{bigfloat} \; & \text{sub}(\text{const bigfloat} \& a, \text{const bigfloat} \& b, \text{long prec}, \text{rounding_modes} \\
& \text{mode}, \text{bool} \& \text{is_exact}) \\
& \{ \\
& \text{ return add}(a, -b, \text{prec}, \text{mode}, \text{is_exact})\; ; \\
& \}
\end{align*}
\]

37. The Mul-Function.
The multiplication is always done by first computing the exact result and rounding
afterwards, in contrast to the procedure for addition. Computing the exact result is done
by simply multiplying the significants of the operands and adding their exponents.

\[
\langle \text{arithmetical functions 30} \rangle +\equiv
\begin{align*}
\text{bigfloat} \; & \text{mul}(\text{const bigfloat} \& a, \text{const bigfloat} \& b, \text{long prec}, \text{rounding_modes} \\
& \text{mode}, \text{bool} \& \text{is_exact}) \\
& \{ \\
& \langle \text{special cases for mul 38} \rangle \\
& \text{bigfloat} \; \text{result}(a.\text{significant} \ast b.\text{significant}, a.\text{exponent} + b.\text{exponent})\; ; \\
& \text{return round}\text{result}(\text{prec}, \text{mode}, \text{is_exact})\; ; \\
& \}
\end{align*}
\]

38. We come to the special case treatment of \textit{mul}. The result is $NaN$ if one of the
operands is $NaN$ or else, if one of the operands is zero and the other one infinity. Other-
wise, if one operand is zero, we return zero, and if one of the operands is infinity, we
return infinity. Here the sign of the return value is the product of the operand signs.
\begin{verbatim}
{special cases for mul 38} \equiv
long sign_result;
((bigfloat & a).normalize();
(bigfloat & b).normalize();
if ((isSpecial(a) \lor (isSpecial(b))) {
  if ((isNaN(a)) \lor (isNaN(b))) return bigfloat(NAN);
  if ((isZero(a) \land isInf(b)) \lor (isInf(a) \land isZero(b))) return bigfloat(NAN);
  sign_result = sign_of_special_value(a) * sign_of_special_value(b);
  if (isZero(a) \lor isZero(b)) {
    if (sign_result \equiv 1) return bigfloat(PZERO);
    else return bigfloat(NZERO);
  }
  if (isInf(a) \lor isInf(b)) {
    if (sign_result \equiv 1) return bigfloat(PINF);
    else return bigfloat(NINF);
  }
}

This code is used in chunk 37.

39. The Div-Function.

One important difference between the division and other arithmetical functions is that
the exact calculation of a division in bigfloat format is impossible. Instead we use inexact
integer division of the significant to approximate the result up to \((\text{prec} + 1)\) digits\(^1\). Here
it may be necessary to shift the significant of the dividend to the left. We also compute
the sign of the division remainder in the variable \(\text{bias}\). Then the correctly rounded division
result can be obtained by calling the \textit{round} function with this bias.

\begin{verbatim}
{arithmetical functions 30} \equiv
bigfloat div(const bigfloat &a, const bigfloat &b, long prec, rounding_modes
mode, bool &is_exact)
{
  bigfloat result;
  long bias;
  \{special cases for div 42\}
  \{shift dividend’s significant 40\}
  \{compute approximative result 41\}
  return round(result, prec, mode, is_exact, bias);
}

40. We first calculate if and by how many digits the dividend \(a\) has to be shifted. Let
\(S_a\) and \(S_b\) be the significant of the operands with binary lengths \(l_a\) and \(l_b\). We suppose
that \(S_a\) and \(S_b\) are nonzero. Then we have \(2^{l_a-1} \leq S_a < 2^{l_a}\) and \(2^{l_b-1} \leq S_b < 2^{l_b}\)
which implies
\[
2^{l_a-l_b-1} < S_a/S_b < 2^{l_a-l_b+1}.
\]
\(^1\)We need one extra digit to guarantee exact rounding
\end{verbatim}

22
Hence the precision of $S_a/S_b$ is at least $l_a - l_b$. Therefore we need $l_a - l_b \geq \text{prec} + 1$ and we have to shift the significant of $a$ by $d$ digits, if $d = l_b - l_a + \text{prec} + 1 > 0$.

\[
\text{shift dividend's significant } 40 \} \equiv \\
\text{bigfloat } aa = a; \\
\text{long } d = \text{prec} + b.\text{significant}.\text{length}() - a.\text{significant}.\text{length}() + 1; \\
\text{if } (d > 0) \{ \\
\quad a.\text{significant} = a.\text{significant} \ll d; \\
\quad a.\text{exponent} = a.\text{exponent} - d; \\
\} \\
\text{This code is used in chunk 39.}
\]

41. Computing the approximation of the result is now easy. We simply do a Euclidean division of $S_a$ and $S_b$, that is, $S_a = S_b \cdot S_r + R$ where $R$ is the division remainder. This is equivalent to

\[
S_a/S_b = S_r + R/S_b
\]

and hence the bias is given by the sign of $R/S_b$.

\[
\text{compute approximative result } 41 \} \equiv \\
\text{result}.\text{special} = \text{NOT}; \\
\text{result}.\text{significant} = aa.\text{significant} / b.\text{significant}; \\
\text{result}.\text{exponent} = aa.\text{exponent} - b.\text{exponent}; \\
\text{result}.\text{precision} = \text{result}.\text{significant}.\text{length}(); \\
\text{integer } R = aa.\text{significant} - b.\text{significant} \times \text{result}.\text{significant}; \\
\text{if } (R \neq 0) \text{ is_exact} = \text{false}; \\
\text{if } (\text{mode} \equiv \text{EXACT}) \text{ mode} = \text{TO_NEAREST}; \\
\quad \text{bias} = \text{sign}(R) \times \text{sign}(b.\text{significant}); \\
\text{This code is used in chunk 39.}
\]

42. The special case handling for division is very similar to that for multiplication. We omit the details.

\[
\text{special cases for div } 42 \} \equiv \\
\text{((bigfloat } \&\& \ a).\text{normalize}(); \\
\text{((bigfloat } \&\& \ b).\text{normalize}(); \\
\text{if } ((\text{isSpecial}(a)) \lor (\text{isSpecial}(b))) \{ \\
\quad \text{long sign_result}; \\
\quad \text{if } ((\text{isNaN}(a)) \lor (\text{isNaN}(b))) \text{ return bigfloat}(\text{NAN}); \\
\quad \text{if } (((\text{isZero}(a)) \land (\text{isZero}(b))) \lor ((\text{isInf}(a)) \land (\text{isInf}(b)))) \text{ return bigfloat}(\text{NAN}); \\
\quad \text{sign_result} = \text{sign_of_special_value}(a) \times \text{sign_of_special_value}(b); \\
\quad \text{if } ((\text{isInf}(a)) \lor (\text{isZero}(b))) \{ \\
\quad\quad \text{if } (\text{sign_result} \equiv 1) \text{ return bigfloat}(\text{PINF}); \\
\quad\quad \text{else return bigfloat}(\text{NINF}); \\
\quad\} \\
\quad \text{// it is clear that isZero(a) } \lor ( \text{isInf}(b) \lor \\
\quad \text{if } (\text{sign_result} \equiv 1) \text{ return bigfloat}(\text{PZERO}); \\
\quad \text{else return bigfloat}(\text{NZERO}); \\
\} \\
\text{This code is used in chunk 39.}
\]
43. The Sqrt-Function.

We reduce the square root operation for the data type bigfloat to the one of integers. The function is split into three main parts:

1. treatment of special cases
2. calculation of the square root
3. rounding

\textbf{arithmetical functions} \texttt{30} \equiv \\
\textbf{bigfloat} \texttt{sqrt(const bigfloat} \&a, \texttt{long} \texttt{prec, rounding\_modes} \texttt{mode, bool} \&\texttt{is\_exact}) \\
\{ \\
\texttt{bigfloat result;} \\
\texttt{integer s;} \\
\{\text{special cases of sqrt 46}\} \\
\{\text{calculate sqrt 44}\} \\
\{\text{rounding of sqrt 45}\} \\
\}\n
44. We rely on the fact that the LEDA data type integer offers a function \texttt{n.sqrt()} = \lfloor \sqrt{n} \rfloor. Furthermore, we know that the delivered bigfloat is positive and non special since we successfully pass the special case section. We now have to ensure that resulting bigfloat has precision of \texttt{prec} digits.

Let now \( l \) be the \textit{significant}'s length, \( sig = (1 + \delta) \cdot 2^{l-1} \) be the \textit{significant} of \( a \) with \( 0 \leq \delta < 1 \), \texttt{exp} its exponent and let \( k \) be the smallest value with such that

1. \( \texttt{exp} - k \) is even,
2. \( 2^{\texttt{prec}-1} \leq \sqrt{\texttt{sig} \cdot 2^k} < 2^{\texttt{prec}} \)

Let \( s = \lfloor \sqrt{\texttt{sig} \cdot 2^k} \rfloor \). Then \( \sqrt{a} = \sqrt{\texttt{sig} \cdot 2^k \cdot 2^{\texttt{exp}-k}} = (s + \epsilon) \cdot 2^{(\texttt{exp}-k)/2} \) for some \( \epsilon \) with \( 0 \leq \epsilon < 1 \). With inequality (2) it follows that \( 2 \cdot (\texttt{prec} - 1) \leq \log_2 \texttt{sig} \cdot 2^k < 2 \cdot \texttt{prec} \). Since \( \texttt{sig} = (1 + \delta) \cdot 2^{(l-1)} \) it follows that \( \log_2 \texttt{sig} = (l-1) + \log_2(1 + \delta) \). Thus the following applies:

\[ 2 \cdot \texttt{prec} - 2 \leq (k + l - 1) + \log_2(1 + \delta) \leq 2 \cdot \texttt{prec} \]

Notice that \( \log_2(1 + \delta) < 1 \). Hence we can simplify the expression:

\[ 2 \cdot \texttt{prec} - 2 \leq k + l - \epsilon \leq 2 \cdot \texttt{prec} \]

where \( 0 \leq \epsilon < 1 \). Since \( k, l \) and \( \texttt{prec} \) are integer values we conclude that

\[ 2 \cdot \texttt{prec} - 2 \leq k + l \leq 2 \cdot \texttt{prec} + 1 \]
\[ 2 \cdot \texttt{prec} - l - 2 \leq k \leq 2 \cdot \texttt{prec} - l + 1 \]
\[ 2 \cdot \texttt{prec} - l - 1 \leq k \leq 2 \cdot \texttt{prec} - l \]

Hence we choose \( k = 2 \cdot \texttt{prec} - l \). If \( \texttt{exp} - k \) is odd we decrement \( k \) by one.
\[ \text{long } k = \text{max}(0, 2 * \text{prec} - \text{a.significant}.\text{length}()); \quad /* \text{check if exp - k is odd */} \\
\text{if } (((\text{a.exponent} - k) \% 2) \neq 0) \text{ } k--; \\
\text{integer } r = \text{a.significant} \ll k; \\
\text{s = } \text{sqrt}(r); \]

This code is used in chunk 43.

45. We come to the rounding of the result. If \( s^2 = \text{sig} \cdot 2^k \) the result is \( s \cdot 2^{(\text{exp} - k)/2} \). Hence we assume that \( s < \sqrt{\text{sig} \cdot 2^k} \). We differ between 3 rounding cases:

1. **TO_ZERO, TO_N_INF**: the result is \( s \cdot 2^{(\text{exp} - k)/2} \)
2. **TO_INF, TO_P_INF**: the result is \((s + 1) \cdot 2^{(\text{exp} - k)/2}\)
3. **TO_NEAREST, EXACT**: We have to decide whether \( s \) or \( s + 1 \) is the nearest value to \( \sqrt{\text{sig} \cdot 2^k} \). If \( s \) is the best approximation we have \( \sqrt{\text{sig} \cdot 2^k} - s < (s+1) - \sqrt{\text{sig} \cdot 2^k} \) which is equivalent to \( 4 \cdot \text{sig} \cdot 2^k - s < 4 \cdot s^2 + 4 \cdot s + 1 \).

Remember that \( r = \text{sig} \cdot 2^k \).

\[ \text{integer } s2 = s * s; \]

\[ \text{if } ((s2 \equiv r) \lor (\text{mode } \equiv \text{TO_ZERO}) \lor (\text{mode } \equiv \text{TO_N_INF})) \]

\[ \text{return } \text{bigfloat}(r, (\text{a.exponent} - k)/2); \]

\[ \text{if } ((\text{mode } \equiv \text{TO_INF}) \lor (\text{mode } \equiv \text{TO_P_INF})) \]

\[ \text{return } \text{bigfloat}(s + 1, (\text{a.exponent} - k)/2); \]

\[ /* \text{mode is either TO_ZERO or EXACT */} \]

\[ \text{if } (4 * r < 4 * s2 + 4 * s + 1) \text{ return } \text{bigfloat}(r, (\text{a.exponent} - k)/2); \]

\[ \text{else return } \text{bigfloat}(s + 1, (\text{a.exponent} - k)/2); \]

This code is used in chunk 43.

46. The rules for the special case treatment are simple. If it is strictly negative we return \( NaN \). Otherwise, if it is any special value beside \(-\infty\), we return the same value.

\[ ((\text{bigfloat } \& a).\text{normalize}()); \]

\[ \text{if } (\text{sign}(a) < 0) \{ \text{is_exact } = \text{false}; \text{return } \text{bigfloat}(\text{NAN}); \} \]

\[ \text{if } (\text{isSpecial}(a)) \{ \]

\[ \text{if } (\text{isZero}(a)) \text{is_exact } = \text{true}; \text{else is_exact } = \text{false}; \]

\[ \text{return } a; \]

\[ \}

This code is used in chunk 43.

47. We now come to the implementation of the unary minus operator.
arithmetical functions

bigfloat operator—(const bigfloat &a)
{
    if (isSpecial(a)) {
        if (isZero(a)) return bigfloat(PZERO);
        if (isInf(a)) return bigfloat(PINF);
        return bigfloat(NAN);
    }
    return bigfloat(-a.significant, a.exponent);
}
48. Comparison Operators

We still have to implement the operators $\equiv$ and $>$. Remember that the other comparison operators have been reduced to these two cases. Let us begin with operator $\equiv$. We first normalize the operands to get a unique representation and afterwards check for special cases. Then we only have to compare significant and exponent.

\[
\text{bool operator}\equiv(const \text{ bigfloat} \; \& a, \text{ const bigfloat} \; \& b) \\
\quad \{ \\
\quad \quad ((\text{bigfloat} \; \& a).\text{normalize}()) ; \\
\quad \quad ((\text{bigfloat} \; \& b).\text{normalize}()) ; \\
\quad \quad \{\text{special case checking for operator } \equiv \; 49\} \\
\quad \quad \text{return } ((a.\text{significant} \equiv b.\text{significant}) \land (a.\text{exponent} \equiv b.\text{exponent})) ; \\
\quad \} \\
\]

See also chunk 50.

This code is used in chunk 10.

49. The rules of special-case checking can be summarized as follows. Comparisons with a $NaN$ not allowed. If both operands are zero, $true$ is returned. Otherwise, $true$ is returned if and only if the operands represent the same special value.

\[
\{\text{special case checking for operator } \equiv \; 49\} \\
\quad \text{if } (\text{isSpecial}(a) \lor \text{isSpecial}(b)) \{ \\
\quad \quad \text{if } (\text{isZero}(a) \land \text{isZero}(b)) \text{ return } true ; \\
\quad \quad \text{if } (\text{isNaN}(a) \lor \text{isNaN}(b)) \\
\quad \quad \text{error handler}(1, "\text{bigfloat}::\text{operator} \equiv \equiv: \text{NaN case} \text{ occurred}"); \\
\quad \quad \text{return } (a.\text{special} \equiv b.\text{special}) ; \\
\quad \} \\
\]

This code is used in chunk 48.

50. We come to operator $>$. For non-special bigfloats we first check the signs of the operands. If the operand’s signs are not equal the result may be easily computed. In case of equal signs we compare the binary lengths of the arguments. Let now $a = s_a \cdot 2^{e_a}$ and $b = s_b \cdot 2^{e_b}$. Furthermore, let $l_a$ and $l_b$ be the binary length of $s_a$ and $s_b$ respectively. We know that $2^{l_a-1} \leq s_a < 2^{l_a}$ and $2^{l_b-1} \leq s_b < 2^{l_b}$. Let $bl_a = l_a + e_a$ and $bl_b = l_b + e_b$. It follows that $a > b$ if $2^{l_a-1} \cdot 2^{e_a} > 2^{l_b-1} \cdot 2^{e_b}$. That is $a > b$ if $bl_a - bl_b > 1$. Analogously, one can see that $a \leq b$ if $bl_a - bl_b \leq -1$. So we have to watch $bl_a - bl_b$. If none of the latter cases apply we find out the operand with the bigger exponent and shift its significant by the exponential difference to the left. Afterwards we return the integer comparison of the significants.

\[
\{\text{comparison operators } \equiv \} + \equiv \\
\quad \text{bool operator } > (const \text{ bigfloat} \; \& a, \text{ const bigfloat} \; \& b) \\
\quad \{ \\
\quad \}
\]

\(^2\)Since \text{normalize} is a not a constant member function we first cast the operands to type \text{bigfloat} &. This is allowed since \text{normalize} does not change the value of a bigfloat but only its representation. Here we use the idea of logical constance.

27
\[(\text{bigfloat} \& a).\text{normalize}();
\text{bigfloat} \& b).\text{normalize}();
\}
\{\text{special case checking of operator} > 51\}
\int \text{sign}_a = ::\text{sign}(a.\text{significant});
\int \text{sign}_b = ::\text{sign}(b.\text{significant});
\text{if} (\text{sign}_a \neq \text{sign}_b) \text{ return } (\text{sign}_a > \text{sign}_b);
\text{integer} \ bl_{\text{diff}} = (a.\text{exponent} + a.\text{precision}) - (b.\text{exponent} + b.\text{precision});
\text{if} (bl_{\text{diff}} > 1) \text{ return } 1;
\text{else if} (bl_{\text{diff}} \leq -1) \text{ return } 0;
\text{integer} \ \text{sig}, \ \text{diff} = a.\text{exponent} - b.\text{exponent};
\text{if} (::\text{sign}(\text{diff}) \geq 0) \{\n\quad \text{sig} = a.\text{significant} \leq (\text{diff}.\text{tolong}());
\quad \text{return} (\text{sig} > b.\text{significant});
\}
\text{else} \{\n\quad \text{sig} = b.\text{significant} \leq ((-\text{diff}).\text{tolong}());
\quad \text{return} (a.\text{significant} > \text{sig});
\}\}

51. Finally we come to the special case checking for operator \(\text{>}\). As before, \(\text{NaN}\) comparisons are not allowed. If the \(\equiv\) operator returns true, operator \(\text{>}\) returns false. The remaining cases are straightforward.
\{\text{special case checking of operator} > 51\} \equiv
\text{if} (\text{isSpecial}(a) \lor \text{isSpecial}(b)) \{\n\text{if} (\text{isNaN}(a) \lor \text{isNaN}(b)) \text{ error} \_\text{handler}(1,"\text{bigfloat}::\text{operator}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\text{NaN}\text{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_case\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_occurred!"});
\text{if} (\text{isZero}(a) \land \text{isZero}(b)) \text{ return } \text{false};
\text{if} (\text{ispInf}(a)) \text{ return } \neg\text{ispInf}(b);
\text{if} (\text{isnInf}(b)) \text{ return } \neg\text{isnInf}(a);
\text{return} (\text{sign}(a) > \text{sign}(b));
\}\}

This code is used in chunk 50.
52.  Conversion between the data types bigfloat and double

It is possible to convert a double into a bigfloat and vice versa. If the significant of the bigfloat has more than 53 bits or if its exponent is too large there will be a loss of information. For a double \( d \) we will always guarantee the consistency property \( \text{bigfloat}(d).\text{toDouble}( ) \equiv d. \)

53.  Some facts about doubles.

A double is specified by a sign \( s \in \{0,1\} \), an exponent \( e \in [0,2047] \) and a binary fraction \( F = f_1 f_2 \ldots f_{52} \) with \( f_i \in 0,1 \) for all \( 1 \leq i \leq 52 \). For \( 0 < e < 2047 \) the triple \((s,f,e)\) represents the number

\[
(-1)^s \cdot 1.f \cdot 2^{e-1023}.
\]

Such a number is called normalized. If \( e \) is zero the number is called denormalized. and the value represented by \((s,f,e)\) is

\[
(-1)^s \cdot 0.f \cdot 2^{-1023}.
\]

Denormalized numbers are smaller than the smallest normalized number \( \text{double.min} = 2^{-1022} \). Due to the limited exponent range doubles can over- and underflow. To get more security in arithmetical operations there are the values \( \pm \infty \) signaling overflow and the value \( \text{NaN} \) signaling invalid operations like \( 0 \cdot \infty \). \( \pm \infty \) are represented by \( s \in \{0,1\} \), \( e = 2047 \) and \( f = 0 \), and \( \text{NaN} \) is represented by any triple \((s,e,f)\) with \( e = 2047 \) and \( f \neq 0 \).

54.  The function \( \text{compose.parts} \) composes the parameters \((s,e,f)\) to a double value. The sign \( s \) is delivered in \( \text{sign.1} \), and \( \text{exp.11} \) is the exponent \( e \). Since we use 32 bit wide longs, the 52 bit of \( f \) are split into the lower part \( \text{least.sig.32} \) and the higher part \( \text{most.sig.20} \). The resulting double is made of two longs, the higher 32 bit containing \( \text{sign.1}, \text{exp.11} \) and \( \text{most.sig.20} \), and the lower 32 bit of \( \text{least.sig.32} \).

\[
\text{double compose.parts( long sign.1, long exp.11, long most.sig.20, long least.sig.32 )}
\]

\[
\{ 
\text{double a;}
\text{long high32 = 0;}
\text{calculate high32 55}
\text{put it all together 56}
\text{return a;}
\}
\]

55.  First we compute the higher 32 bit of the double.

\[
\text{calculate high32 55}
\]

\[
\text{if ( sign.1 ) high32 = high32 \* 80000000;}
\text{exp.11 = exp.11 \&lt; 20;}
\text{high32 = high32 \& exp.11;}
\text{high32 = high32 \& most.sig.20;}
\]

This code is used in chunk 54.
To compose the two long parts to one double value we use pointer arithmetic. For this we need a pointer to a long variable that initially points to the beginning of the resulting double a. Since we would like to achieve machine independent code we have to care for different byte ordering mechanisms. Sun Sparc stations use BIG ENDIAN byte ordering which means that always the higher part is saved before the lower part in memory. Intel PCs use LITTLE ENDIAN byte ordering in which the lower part is saved first.

\[
\begin{align*}
\text{put it all together} & \equiv \\
\text{long } *p; & \\
\quad p = (\text{long } *) \& a; & \\
\text{#ifndef } \text{LITTLE\_ENDIAN} & \\
\quad (*p) = \text{high}_32; \ p++; \ (*p) = \text{least\_sig}_32; & \\
\text{#else} & \\
\quad (*p) = \text{least\_sig}_32; \ p++; \ (*p) = \text{high}_32; & \\
\text{#endif} & \\
\end{align*}
\]

This code is used in chunk 54.

The mathematical function \textit{pow2} for doubles that computes \(2^{\text{exp}}\) for a given exponent \textit{exp} can be realized efficiently using the new function \textit{compose\_parts}.

\[
\text{functions for internal use } 4 \begin{array}{l} \equiv \end{array} \\
\text{double } \text{pow2}(\text{long } \text{exp}) \begin{array}{l} \{ \text{ return } \text{ compose\_parts}(0, \text{exp} + 1023, 0, 0); \} \end{array}
\]

Now we are able to define the special double values that we need in the following sections.

\[
\begin{align*}
\text{global identifiers} & \equiv \begin{array}{l} /* \text{ since we need the } \text{pow2} \text{ and } \text{compose\_parts} \text{ functions we have to use prototyping } */ \end{array} \\
\text{double } \text{compose\_parts}(\text{long, long, long, long}); & \\
\text{double } \text{pow2}(\text{long}); & \\
\text{const double } \text{double\_min} = \text{pow2}(-1022); & \\
\text{const double } \text{NaN\_double} = \text{compose\_parts}(0, 2047, 0, 1); & \\
\text{const double } \text{pInf\_double} = \text{compose\_parts}(0, 2047, 0, 0); & \\
\text{const double } \text{nInf\_double} = -\text{pInf\_double}; & \\
\text{const double } \text{pZero\_double} = \text{compose\_parts}(0, 0, 0, 0); & \\
\text{const double } \text{nZero\_double} = \text{compose\_parts}(1, 0, 0, 0); & \\
\end{align*}
\]

See also chunks 66 and 74.

This code is used in chunk 10.

The next constructor transforms a double \(d\) into a bigfloat. First we check whether \(d\) is denormalized. In the case that \(d\) is denormalized, we correct the exponent of \(d\) to make it normalized (if it is nonzero) but set a flag that allows us to take back our changes later. After that we split \(d\) into a high and a low part. From these two parts we compute the sign, significant and exponent of \(d\).
Constructors 12

\textbf{bigfloat :: bigfloat(double } d) \{
  \textbf{int } sign;
  \textbf{long } flag = 0;
  \textbf{unsigned long } mh = 0, ml = 0;
  \textbf{long } *p;
  \langle \text{check for denormalized number } 60 \rangle
  \langle \text{determine high and low part } 61 \rangle
  \langle \text{get the double’s sign } 62 \rangle
  \langle \text{get the significant's value } 63 \rangle
  \langle \text{get the exponent’s value } 64 \rangle
  \langle \text{check for special values } 65 \rangle \text{normalize( )};
\}

60. If \( d \) is smaller than \( \text{double_min} \) then it is denormalized and we multiply \( d \) with \( 2^{52} \) to get a normalized number.

\langle \text{check for denormalized number } 60 \rangle \equiv
  \begin{align*}
    \text{if } (\text{fabs}(d) < \text{double_min}) & \{ \\
    d = d \times \text{pow2}(52); \\
    \text{flag} = 1;
  \}
  \end{align*}

This code is used in chunk 59.

61. We split the 64 bit wide double representation in two 32 bit wide longs. To do this we use a pointer to a long value and assign the casted address of the double value to it. The long pointer works like an \texttt{array}. If the \texttt{BIG_ENDIAN} byte ordering is active, the first component holds the higher 32 bit part \( mh \) and if the \texttt{LITTLE_ENDIAN} byte ordering is active, the first component holds the lower 32 bit part \( ml \).

\langle \text{determine high and low part } 61 \rangle \equiv
  \begin{align*}
    p = (\textbf{long } *) \& d; \\
    \#ifndef \texttt{LITTLE_ENDIAN} \\
    mh = *p; p++; ml = *p; \\
    \#else \\
    ml = *p; p++; mh = *p; \\
    \#endif
  \end{align*}

This code is used in chunk 59.

62. The \texttt{sign} of a double is denoted by its highest bit.

\langle \text{get the double’s sign } 62 \rangle \equiv
  \begin{align*}
    \text{if } (mh \& *80000000) & \text{sign} = -1; \\
    \text{else } \text{sign} = 1;
  \end{align*}

This code is used in chunk 59.
63. The significant of the double is composed out of \( ml \) and the least 20 bits of \( mh \). As we have ensured normalized representation in the beginning of this function we have to keep in mind that the significant represents the binary fraction and that there is an implicit one bit that leads this fraction.

\[
\text{get the significant's value 63} \equiv \\
\textbf{long} \text{ sig} = 0; \quad /* \text{get the significant bits out of the highword */} \\
\text{sig} = \text{mh} \& \#000fffff; \quad /* \text{we have ensured that } d \text{ is normalized } \Rightarrow \text{ add the leading one bit */} \\
\text{sig} = \text{sig} | \#00100000; \quad /* \text{compose significant */} \\
\text{significant} = \textbf{integer}(\text{sig}); \\
\text{significant} = \text{significant} \ll 32; \\
\text{significant} = \text{significant} + \textbf{integer}(\text{ml}); \\
\text{if} \ (\text{sign} \equiv -1) \text{ significant} = -\text{significant}; \\
\text{This code is used in chunk 59.}
\]

64. The double exponent takes the bits 2 to 12 of \( mh \). We use bit manipulation to extract this value out of \( mh \). To transform the double exponent into the exponent of the resulting bigfloat, we have to correct its value by

1. the bias \(-1023\),
2. \(-\text{(\text{significant}.\text{length()} - 1)} \equiv -52\), due to the bigfloat's representation
3. \(-52\), if \( d \) was denormalized

\[
\text{get the exponent's value 64} \equiv \\
\textbf{long} \text{ e} = 0; \quad /* \text{get the 11 bits of the exponent */} \\
e = \text{mh} \& \#7ff00000; \quad /* \text{shift the result to get the right value */} \\
e = e \gg 20; \quad /* \text{shift subtract bias and 52 */} \\
e -= 1075; \quad /* \text{subtract 52 if } d \text{ was denormalized */} \\
\text{if} \ (\text{flag}) \ e -= 52; \\
\text{exponent} = e; \\
\text{This code is used in chunk 59.}
\]

65. Finally, we have to check whether \( d \) was in a special state.

\[
\text{check for special values 65} \equiv \\
\text{special} = \text{NOT}; \\
\text{if} \ (e \equiv 972) \quad /* \text{Inf or NaN-Case */} \\
\{ \\
\text{if} \ (\text{significant} \equiv 0) \text{ special} = (\text{sign} > 0) \ ? \text{PINF : NINF}; \\
\text{else special} = \text{NAN}; \\
\} \\
\text{if} \ (d \equiv 0) \text{ special} = (\text{sign} > 0) \ ? \text{PZERO : NZERO}; \\
\text{This code is used in chunk 59.}
\]
In several functions we use integer values which have - in their binary representation - one single bit set.

\[
\text{const integer } \text{integer}_1 = \text{integer}(1); \\
\text{const integer } \text{integer}_52 = (\text{integer}_1 \ll 52) - \text{integer}_1; \\
\text{const integer } \text{integer}_32 = (\text{integer}_1 \ll 32) - \text{integer}_1; \\
\text{const integer } \text{integer}_20 = (\text{integer}_1 \ll 20) - \text{integer}_1;
\]

The \textit{todouble} function.

This function takes the given bigfloat and converts it into double format. If the bigfloat is in a special state (\textit{NAN}, \textit{PINF}, \ldots), we return the corresponding special double value. A bigfloat \( x \) is called \textit{approximable} if

\[2^{-1074} \leq |x| < 2^{1024}.
\]

For approximable values of \( x \), we return the double nearest to \( x \), otherwise bigfloats in special states \textit{PINF}, \textit{NINF}, \textit{PZERO} or \textit{NZERO} are returned.

In the main part of the function we distinguish the cases that the returned double is normalized or denormalized. At the end the significant \( s \), the exponent \( t_{\exp} \) and the significant \( t_{\sig} \) of the double are put together.

\[
\text{double } \text{todouble(const bigfloat } &x) \\
\{
\text{long } s = \text{::sign}(x.\text{significant}); \\
\text{long } t_{\exp} = 0; \\
\text{integer } t_{\sig} = 0; \\
\text{bigfloat } \text{rounded_value} = \text{abs}(x); \\
\text{if (normal)} \\
\text{else (denormal case 71)} \\
\text{double } a; \\
\text{return } a;
\}
\]

The special cases are first.

\[
\text{(special case checking of todouble 68) } \equiv \\
\text{((bigfloat } &x).\text{normalize()); \\
\text{if (isSpecial(x)) } \{ \\
\text{if (ispZero(x)) return pZero_double; \\
\text{if (isnZero(x)) return nZero_double; \\
\text{if (isNaN(x)) return NaN_double; 
}\}
\]
\[
\text{if } (ispInf(x)) \text{ return } pInf\text{.double};; \\
\text{if } (isnInf(x)) \text{ return } nInf\text{.double};; 
\]

This code is used in chunk 67.

69. We round the bigfloat such that all bits of the significant just fit into double format. We distinguish cases according to the quantity \(\log_2 = \text{exponent} + \text{precision}\).

- If \(\log_2\) is between \(-1021\) and \(1024\), we have to round to 53 places.
- If \(\log_2\) is between \(-1073\) and \(-1022\), we have to round to \(1074 + \log_2\) places.

If \(\log_2\) is outside of \([-1073, 1023]\) the conversion overflows or underflows. Note that \(\log_2\) might change through the rounding.

\(\text{rounding step of todouble 69} \equiv \)
\[\text{integer } \log_2 = x.\text{exponent} + x.\text{precision};\]
\[\text{if } (\log_2 > 1024) \text{ return } \text{sign}(x) \times pInf\text{.double};\]
\[\text{if } (\log_2 < -1073) \text{ return } \text{sign}(x) \times pZero\text{.double};\]
\[\text{if } (\log_2 > -1021) \text{ rounded.value} = \text{round}(\text{rounded.value}, 53, \text{TO\_NEAREST});\]
\[\text{else } \text{rounded.value} = \text{round}(\text{rounded.value}, 1074 + \log_2\text{.tolong}( ), \text{TO\_NEAREST});\]

This code is used in chunk 67.

70. Now we decide whether the bigfloat is approximable by a normalized double or not. This is the case if and only if
\[2^{-1022} \leq | \text{rounded.value} | < 2^{1024}.\]

Note that \(\log_2\) can be 1025 because of rounding, in which case we return infinity.

\(\text{check for normal or denormal return value 70} \equiv \)
\[\text{long } normal = 1;\]
\[\log_2 = \text{rounded.value}.\text{exponent} + \text{rounded.value}.\text{precision};\]
\[\text{if } (\log_2 \equiv 1025) \text{ return } \text{sign}(x) \times pInf\text{.double};\]
\[\text{if } (\log_2 < -1021) \text{ normal} = 0;\]

This code is used in chunk 67.

71. First we assume that we have a denormalized value to store. Remember that a denormalized double has an implicitly leading 0 bit and that its unbiased exponent is \(-1022\). Let us consider the number \(k\) of zeros such that \(\text{rounded.value} = \text{sig} \times 2^c\) equals \(0.00\ldots0\text{ sig} \times 2^{-1022}\). This is equivalent to \(k = -1022 - e - l\) where \(l\) is the length of the significant \(\text{sig}\) of \(\text{rounded.value}\). We have to ensure that the bigfloat significant has length \(52 - k\), that is, we shift the significant by \(L_{\text{shift}} = 52 - k - l\) places to the left. (If \(L_{\text{shift}}\) is negative rightshifts have to be performed). Simplifying we see that \(L_{\text{shift}} = e + 1074\).

\(\text{denormal case 71} \equiv \)
\[\{\text{long } L_{\text{shift}} = \text{rounded.value}.\text{exponent}.\text{tolong}( ) + 1074;\]
if (l_shift > 0) t_sig = rounded_value.significant \ll l_shift;
else t_sig = rounded_value.significant \gg (-l_shift);
}

This code is used in chunk 67.

72. Now we assume that our bigfloat rounded_value = sig \cdot 2^e is in normalized double range. As normalized doubles have an implicit leading one, the biased exponent of the returned double is

\[ t.exp = e + (l - 1) + 1023 \]

where again \( l \) is the length of sig. Finally we cut off the first bit of sig (which is necessarily one) and obtain the double significant \( t.sig \).

\langle normal case 72 \rangle \equiv

\{
  t.exp = rounded_value.exponent.tolong() + (rounded_value.precision - 1) + 1023;
  /\* if the length of the rounded significant is lower than 53 we have to shift it to the left *\/
  t_sig = rounded_value.significant \ll (53 - rounded_value.significant.length());
  t_sig = t_sig & integer_52;
\}

This code is used in chunk 67.

73. Finally we set the bits of the returned double. That means that we have to lay down our specified values for sign, significant. For this we use the function compose_parts and so we just need to calculate the arguments for this function.

\langle set the bits of a 73 \rangle \equiv

\begin{verbatim}
  unsigned long sign, h_sig, l_sig;
  sign = (s \equiv (-1));
  h_sig = ((t_sig \gg 32) \& integer_20).tolong();
  l_sig = (t_sig \& integer_32).tolong();
  a = compose_parts(sign, t_exp, h_sig, l_sig);
\end{verbatim}

This code is used in chunk 67.
There are two operators which perform input/output.

- The first one is the **operator `<`**. It outputs a bigfloat in either binary or decimal representation. The global variable `output_mode` determines the output form. The two different modes are
  - binary (`BIN_OUT`)
  - decimal (`DEC_OUT`)

  This operator uses the `binout` function for binary output and function `decimal_output` for decimal output.

- The second one is the **operator `>`**. It performs decimal input of a bigfloat using function `outofchar`. Binary input is not yet implemented.

We define the maximal size of a string field that is allocated during input and output.

```c
const long bin_maxlen = 10000;
```

### 75. Procedure `binout` takes an output stream and an integer `b` as input. It produces an unsigned, binary output of `b`.

```c
void binout (ostream &os, integer b)
{
  char temp[bin_maxlen];
  long flag = 0, count = 0;
  if (b < 0) b *= -1;
  do {
    temp[count++] = (char) (b % 2).tolong() + '0';
    if (b <= 1) flag = 1;
    else b /= 2;
  } while (~flag);
  for (long i = count - 1; i >= 0; i--) os << temp[i];
}
```

### 76. The following function computes the `n`th power of a bigfloat in precision `prec` and rounding mode `mode`.  

```c
bigfloat powl(const bigfloat &x, long n, long prec = 1, rounding_modes mode = EXACT)
{
  bigfloat z = 1, y = x;
  long n_prefix = n;
  ...
```

---

3 This does not mean that the result is correct up to `prec` digits. Only every operation within the procedure is carried out in that precision.
while (n_prefix > 0) {
    if (n_prefix % 2) z = mul(z, y, prec, mode);
    n_prefix = n_prefix / 2;
    y = mul(y, y, prec, mode);
}
return z;

77. Procedure `decimal_output` displays a bigfloat `b` in decimal floating point notation, with `prec` decimal places in the significant. It first treats the sign of the given `bigfloat` `b`. The further procedure works with `|b|` instead of `b`. Then we compute the decimal exponent of `b` which allows us to get the wanted `prec` decimal places of the output. Finally we write the output to `ostream os`.

```cpp
(void for internal use 4) +≡

void decimal_output(ostream &os, bigfloat b, long prec) {
    if (!b.get_exponent().islong())
        error_handler(1, "decimal_output: not implemented for large exponents");
    if (prec == 0)
        error_handler(1, "decimal_output: prec has to be bigger than 0!");
    ostrstream oss; /* string stream needed for output */
    long dd = 10;
    /* calculate the sign 78 */
    /* compute decimal logarithm of b 79 */
    /* compute decimal significant of b 80 */
    /* output of the result 81 */
}
```

78. First we calculate the sign of `b` and take the modulus of `b`.

```cpp
(void calculate the sign 78) ≡
    if (sign(b) < 0) { b = -b; os << "-"; }
This code is used in chunk 77.
```

79. We need to know the decimal logarithm of `b`, rounded up to the next integer. For this we use the function `log10` from the standard math library that takes a double input. To avoid the problem of overflow in the double calculation we first determine `k = [log2(b)]`, write `b = 2^k b_{rem}` and finally compute `\log_{10}(b) = \log_{10}(2^k b_{rem}) = k \cdot \log_{10}(2) + \log_{10}(b_{rem})`.

```cpp
(void compute decimal logarithm of b 79) ≡
    long log2_b = b.get_precision() + b.get_exponent().tolog();
    bigfloat b_rem(b.get_significant(), -b.get_precision());
    long log10_b = (long) ceil(log10(2) * log2_b + log10(todouble(b_rem)));
This code is used in chunk 77.
```
80. Now that we know that $b$ has decimal length $\log_{10} b$ we proceed with the further preparation of the output. We have to ensure that the significant of the exponential output has exactly $\text{prec}$ decimal digits. Therefor let $\text{diff} = \text{prec} - \log_{10} b$ and if $\text{diff}$ is positive we multiply $b$ with $10^{\text{diff}}$ and if it is negative we divide it by $10^{-\text{diff}}$. We use $\log_2 b + \log_2 10^{\text{diff}}$ as precision for these operations. Afterwards the function $\text{tolong}$ is used to get a rounding to the next integer value.

\[
\begin{align*}
\text{compute decimal significant of } b &\equiv \\
\text{long } \text{diff} &\equiv \text{prec} - \log_{10} b; \\
\text{long } \text{digits} &\equiv (\text{long}) \ \text{ceil}(\log_2((\text{double}) \ \text{dd}) \ast \text{diff}) + \log_2 b; \\
\text{bigfloat } &b_\text{shift}; \\
\text{integer } &\text{significant}; \\
\text{if } (\text{diff} \geq 0) &\text{ } b_\text{shift} = \text{mul}(b, \text{powl}(\text{dd}, \text{diff}, \text{digits, TO_NEAREST}), \text{digits}, \text{TO_NEAREST}); \\
\text{else } &b_\text{shift} = \text{div}(b, \text{powl}(\text{dd}, -\text{diff}, \text{digits, TO_NEAREST}), \text{digits, TO_NEAREST}); \\
&\text{significant } = \text{tolong}(b_\text{shift, TO_NEAREST}); \\
&\text{oss } \ll \text{significant};
\end{align*}
\]
This code is used in chunk 77.

81. At last, we have to output the computed value. We take a floating point format with one digit in front of the decimal point. We cut off final zeros to get a clear output.

\[
\begin{align*}
\text{output of the result } &\equiv \\
\text{char } &\ast \text{str }\equiv \text{oss.str( )}; \\
\text{char } &\ast \text{help }\equiv \text{str }+ \text{strlen}(\text{str}) - 1; \\
\text{while } &(*\text{help }\equiv \text{'0'}) \ \text{help }--; \\
&(*\text{help }+1) = 0; \\
\text{if } & (\text{prec }> 1) \ \text{oss } \ll (\ast(\text{str }++)) \ll \text{"." }\ll \text{str}; \\
\text{else } &\text{oss } \ll (\ast\text{str}); \\
\text{if } & (\log_{10} b \neq 1) \ \text{oss } \ll \text{"E" }\ll \log_{10} b - 1;
\end{align*}
\]
This code is used in chunk 77.

82. Now we are ready to implement the stream operators. We start with the output. In the case of decimal output we have to determine the decimal precision for the significant of the exponential output. Therefor we pass the decimal logarithm of the significant of $b$ as $\text{prec}$ to the function $\text{decimal.output}$.

\[
\begin{align*}
\text{input/output operators } &\equiv \\
\text{ostream } &\& \text{operator }\ll(\text{ostream } \& \text{os}, \text{const} \text{ bigfloat } \& b) \\
\{ \\
&\text{if } (\text{isSpecial}(b)) \{ \\
&\text{if } (\text{isNaN}(b)) \ \text{return } \text{os } \ll \text{"NaN"}; \\
&\text{if } (\text{ispInf}(b)) \ \text{return } \text{os } \ll \text{"+Inf"}; \\
&\text{if } (\text{isnInf}(b)) \ \text{return } \text{os } \ll \text{"-Inf"}; \\
&\text{if } (\text{ispZero}(b)) \ \text{return } \text{os } \ll \text{"+0"}; \\
&\text{if } (\text{isnZero}(b)) \ \text{return } \text{os } \ll \text{"-0"}; \\
&\}
\}
\text{int } &\text{sign.}b = \text{sign}(b, \text{significant});
\]

38
if (bigfloat::output_mode == BIN_OUT) {
    if (sign_b < 0) os <<= "-";
    os <<= "0.";
    if (sign_b >= 0) binout(os, b.significant);
    else binout(os, -b.significant);
    os <<= "E";
    if (b.exponent < 0) os <<= "-";
    else os <<= "+";
    binout(os, b.exponent);
}
if (bigfloat::output_mode == DEC_OUT)
    decimal_output(os, b, max(1, (long) floor(log10(2) * b.significant.length())));
    return os;
}
See also chunk 83.
This code is used in chunk 10.

83. We turn to the input operator. The hard work is done by function outofchar.

\langle\text{input/output operators 82}\rangle \equiv \langle\text{istream} &\text{operator}\rangle(\text{istream} &is, \text{bigfloat} &b)
{
    \text{char} \ temp\ [\text{bin-maxlen}] ;
    is := temp ;
    b = \text{outofchar}(\text{temp}) ;
    return \ is ;
}

84. Function outofchar provides the conversion of \text{string} to bigfloat. Currently, only decimal input is possible. The expected format is:

\[ \pm dd \cdots d[.dd \cdots d] E \pm dd\cdots d] \]

where \( d \) is out of \([0,9]\) and \( \cdots \) stands for arbitrarily many \( d \)'s. The procedure works as follows. We read the sign, the integer part, the decimal fraction and the exponent of the decimal floating point representation in turn. Then we concatenate the integer part and the decimal fraction into one decimal significant \( \text{sig} \) such that the input bigfloat is \( \text{sig} \cdot 10^{\text{exp-fl}} \) where \( \text{fl} \) is the length of the decimal fraction and \( \text{exp} \) is the scanned exponent. In order to get a binary representation we either multiply or divide \( \text{sig} \) by \( 10^{\text{exp-fl}} \) according to the sign of \( \text{exp-diff} = \text{exp} - \text{fl} \). This operations are performed by the bigfloat operations \text{mul} and \text{div} respectively. It remains to specify an appropriate value for parameter \text{prec}. We know that the binary length of \( \text{sig} \) is \( l \). In both cases the integer part of the result should be not larger than \( 10^{l+exp-diff+fl} \). The fraction should be smaller than \( 10^{fl} \). Thus a precision of \( \left\lfloor \log_2 10^{l+exp-diff+fl} \right\rfloor \) should be sufficient for both computations.
Functions for internal use

```c
bigfloat outofchar(char *rep, long prec = 0)
{
    integer sig = 0, exp = 0;
    bigfloat result, pow;
    long dd = 10;
    double log2dd = log2(dd);
    long int_length = 0, frac_length = 0;
    int s;
    (scan signs 85)
    int sign = s;
    (scan integer part 87)
    (scan fraction 88)
    (scan exponent 89)
    long l = int_length + frac_length;
    long exp_diff = exp.tolong() - frac_length;
    if (prec <= 0) prec = (long) ceil(log2dd * (l + exp_diff + frac_length));
    pow = powl(dd, abs(exp_diff), 1, EXACT);
    if (exp_diff > 0) result = mul(sig, pow, prec, TO_NEAREST);
    else result = div(sig, pow, prec, TO_NEAREST);
    if (sign == 1) return result; else return -result;
}```

85. First, we scan the optional sign at the beginning of the input.

```c
(scan sign s 85) ==
    s = 1;
    if (rep[0] == ' -') { s = -1; rep++; }
    else if (rep[0] == '+') rep++;
```

This code is used in chunks 84 and 89.

86. We need a function isnum to test if a scanned character is a digit.

```c
(auxiliary functions 16) ==
    bool isnum(char ch)
    {
        if ((ch >= '0') && (ch <= '9')) return true;
        else return false;
    }
```

87. Now we scan the input up to the decimal point if there is one. We pass every character and convert it to int. Each number is added to the temporary variable sig that is decimally shifted by one to the left.
\{scan integer part 87\} 

while (isnum(*rep)) {
    int_length++;
    sig = sig * dd + (*(rep++) - '0');
}

This code is cited in chunk 88.
This code is used in chunk 84.

88. The fraction scan works quite similar as \{scan integer part 87\}. We step over the input up to the character 'E' or until the input's end is reached.

\{scan fraction 88\} 

if (*rep == '. ') {
    rep++;
    \{scan fraction 88\} 
        if (*rep == 'E') {
            rep++;
            \{scan fraction 88\} 
        }
}

This code is used in chunk 84.

89. To scan the exponent we first read the optional sign and otherwise proceed as before.

\{scan exponent 89\} 

if (*rep == 'E') {
    rep++;
    \{scan sign s 85\} 
        while (isnum(*rep)) exp = exp * dd + (*(rep++) - '0');
    if (s == -1) exp = -exp;
}

This code is used in chunk 84.
90. References

References


[IE_Go] David Goldberg. What every computer scientist should know about floating-point arithmetic


Index

a: 5, 12, 30, 36, 37, 39, 43, 47, 48, 50, 54, 67.

a_ptr: 31.
aa: 40, 41.
abs: 5, 67, 84.
absolute: 31.
add: 5, 30, 36.
b: 5, 19, 30, 36, 37, 39, 48, 50, 75.

b_ptr: 31.
b_rem: 79.
b_shift: 80.
bias: 5, 7, 17, 18, 20, 26, 39, 41.
bigfloat: 5, 12, 59.
BIGFLOAT.H: 5.
bin_maxlen: 74, 75, 83.
BIN_OUT: 5, 74, 82.
binout: 74, 75, 82.
bl_diff: 50.
ceil: 5, 79, 80, 84.
ch: 86.
compare: 6.
compose_parts: 54, 57, 58, 73.
conversion: 10.
count: 75.
cut: 19, 20, 22, 23, 24, 25.
d: 16, 40, 59.
dbool: 5, 7, 8.

dd: 77, 80, 84, 87, 88, 89.
DEC_OUT: 5, 8, 74, 82.
decimal_output: 74, 77, 82.
diff: 31, 32, 50, 80.
digits: 5, 7, 17, 18, 20, 22, 23, 24, 25, 26, 80.
div: 5, 30, 39, 80, 84.
double_min: 53, 58, 60.
e: 5, 12, 64.
error: 30, 32.
error_handler: 14, 15, 28, 34, 49, 51, 77.
error_in_rounding: 7.
EXACT: 1, 5, 18, 26, 30, 32, 41, 45, 76, 84.
exp: 44, 57, 84, 89.
exp_diff: 31, 34, 84.
exp_11: 54, 55.
exponent: 3, 4, 5, 9, 12, 13, 18, 27, 28, 31, 33, 34, 37, 40, 41, 44, 45, 47, 48, 50, 64, 69, 70, 71, 72, 82.
fabs: 60.
false: 41, 46, 51, 86.
fl: 84.
flag: 59, 60, 64, 75.
floor: 5, 82.
frac_length: 84, 88.
gep_exponent: 5, 77, 79.
gep_precision: 5, 79.
gep_significant: 5, 79.
global_prec: 5, 8.
h_sign: 73.
help: 81.
high32: 54, 55, 56.
i: 16, 75.
int_length: 84, 87.
integer_1: 66.
integer_20: 66, 73.
integer_32: 66, 73.
integer_52: 66, 72.
is: 5, 83.
is_exact: 5, 7, 17, 18, 30, 36, 37, 39, 41, 43, 46.
isInf: 5, 28, 35, 38, 42, 47.
islong: 28, 34, 77.
isNaN: 5, 28, 35, 38, 42, 49, 51, 68, 82.
isInf: 5, 51, 68, 82.
isnum: 86, 87, 88, 89.
isZero: 5, 68, 82.
ispInf: 5, 51, 68, 82.
ispZero: 5, 68, 82.
isSpecial: 5, 18, 35, 38, 42, 46, 47, 49, 51, 68, 82.
isZero: 5, 28, 35, 38, 42, 46, 47, 49, 51.
k: 44.
l: 84.
L_shift: 71.
L_sig: 73.
least_sig_32: 54, 56.
length: 4, 12, 13, 19, 28, 40, 41, 44, 64, 72, 82.
LITTLE_ENDIAN: 56, 61.
log: 16.
\( \log_x : 31. \)
\( \log_y : 31. \)
\( \log_2 : 69, 70. \)
\( \log_{10} : 79, 82. \)
\( \log_{10b} : 79, 80, 81. \)
\( \log_{2} : 5, 16, 80, 84. \)
\( \log_{2b} : 79, 80. \)
\( \log_{2dd} : 84. \)
\( l_1 : 16. \)
\( l_2 : 16. \)
\( m : 5. \)
\( \max : 16, 28, 32, 44, 82. \)
\( \maxh : 59, 61, 62, 63, 64. \)
\( \mll : 59, 61, 63. \)
\( \mode : 5, 7, 17, 18, 26, 30, 32, 36, 37, 39, 41, 43, 45, 76. \)
\( \most_{\text{sig}}_{20} : 54, 55. \)
\( \mul : 5, 30, 37, 38, 76, 80, 84. \)
\( n : 76. \)
\( n_{\text{prefix}} : 76. \)
\( \text{NAN} : 3, 5, 14, 15, 35, 38, 42, 46, 47, 65, 67. \)
\( \text{NaN}_{\text{double}} : 58, 68. \)
\( \text{NINF} : 3, 5, 14, 38, 42, 65, 67. \)
\( \text{nInf}_{\text{double}} : 58, 68. \)
\( \text{normal} : 67, 70. \)
\( \text{normalize} : 4, 7, 10, 15, 18, 28, 35, 38, 42, 46, 48, 50, 59, 68. \)
\( \text{NOT} : 3, 4, 5, 12, 13, 14, 15, 41, 65. \)
\( \text{NZERO} : 3, 4, 5, 14, 38, 42, 65, 67. \)
\( \text{nZero}_{\text{double}} : 58, 68. \)
\( o_{\text{mode}} : 5. \)
\( \op : 16. \)
\( \text{operator} : 5, 47, 48, 50, 82, 83. \)
\( \os : 5, 75, 77, 78, 81, 82. \)
\( \os : 77, 80, 81. \)
\( \out : 6. \)
\( \text{outofchar} : 74, 83, 84. \)
\( \text{output}_{\text{mode}} : 5, 8, 74, 82. \)
\( \text{output}_{\text{modes}} : 5. \)
\( p : 5, 56, 59. \)
\( \text{PINF} : 3, 5, 14, 15, 38, 42, 47, 65, 67. \)
\( \text{pInf}_{\text{double}} : 58, 68, 69, 70. \)
\( \text{pow} : 84. \)
\( \text{powl} : 76, 80, 84. \)
\( \text{pow2} : 5, 57, 58, 60. \)
\( \text{prec} : 5, 19, 30, 32, 36, 37, 39, 40, 43, 44, 76, 77, 80, 81, 82, 84. \)
\( \text{precision} : 1, 3, 4, 5, 9, 12, 13, 18, 20, 26, 28, 31, 32, 41, 50, 69, 70, 72. \)
\( \text{Print} : 6. \)
\( \text{PZERO} : 3, 4, 5, 12, 13, 14, 15, 38, 42, 47, 65, 67. \)
\( \text{pZero}_{\text{double}} : 58, 68, 69. \)
\( R : 41. \)
\( r : 44. \)
\( \text{Read} : 6. \)
\( \text{rep} : 84, 85, 87, 88, 89. \)
\( \text{result} : 37, 39, 41, 43, 84. \)
\( \text{rmode} : 5, 28. \)
\( \text{round} : 5, 10, 17, 18, 28, 30, 37, 39, 69. \)
\( \text{round}_{\text{mode}} : 5, 7, 8. \)
\( \text{rounded}_{\text{value}} : 67, 69, 70, 71, 72. \)
\( \text{rounding}_{\text{modes}} : 5. \)
\( \text{s} : 5, 12, 43, 67, 84. \)
\( \text{set}_{\text{glob}_{\text{prec}}} : 5. \)
\( \text{set}_{\text{output}_{\text{mode}}} : 5. \)
\( \text{set}_{\text{round}_{\text{mode}}} : 5. \)
\( \text{shift} : 18, 26, 27. \)
\( \text{sig} : 50, 63, 84, 87, 88. \)
\( \text{sign} : 4, 5, 6, 10, 14, 15, 16, 20, 23, 24, 25, 26, 30, 41, 46, 50, 51, 59, 62, 63, 65, 67, 69, 70, 73, 78, 82, 84. \)
\( \text{sign}_{a} : 50. \)
\( \text{sign}_{b} : 50, 82. \)
\( \text{sign}_{\text{of}_{\text{special}_{\text{value}}} : 5, 15, 35, 38, 42. \)
\( \text{sign}_{\text{result}} : 38, 42. \)
\( \text{sign}_{1} : 54, 55. \)
\( \text{significant} : 3, 4, 5, 9, 12, 13, 14, 18, 20, 21, 22, 23, 24, 25, 26, 27, 28, 33, 34, 37, 39, 40, 41, 44, 47, 48, 50, 63, 64, 65, 67, 71, 72, 80, 82. \)
\( \text{signum} : 4. \)
\( \text{sp} : 5, 12. \)
\( \text{special} : 3, 4, 5, 9, 12, 13, 14, 15, 41, 49, 65. \)
\( \text{special}_{\text{values}} : 5. \)
\( \text{sq} : 16. \)
\( \text{sqrt} : 5, 30, 43, 44. \)
\( \text{str} : 81. \)
\( \text{strlen} : 81. \)
\( \text{sub} : 5, 30, 36. \)
\( \text{sum} : 30, 32, 33. \)
\begin{itemize}
  \item \texttt{s2}: 45.
  \item \texttt{t\_exp}: 67, 72, 73.
  \item \texttt{t\_sig}: 67, 71, 72, 73.
  \item \texttt{temp}: 75, 83.
  \item \texttt{test}: 18, 20.
  \item \texttt{TO\_INF}: 1, 5, 18, 23, 26, 45.
  \item \texttt{to\_integer}: 28.
  \item \texttt{TO\_N\_INF}: 1, 5, 18, 25, 26, 45.
  \item \texttt{TO\_NEAREST}: 1, 5, 8, 18, 20, 26, 41, 45, 69, 80, 84.
  \item \texttt{TO\_P\_INF}: 1, 5, 18, 24, 25, 26, 45.
  \item \texttt{TO\_ZERO}: 1, 5, 18, 22, 26, 45.
  \item \texttt{todouble}: 5, 52, 67, 79.
  \item \texttt{to\_integer}: 5, 28, 80.
  \item \texttt{tolong}: 28, 34, 50, 69, 71, 72, 73, 75, 79, 84.
  \item \texttt{true}: 8, 17, 18, 46, 49, 86.
  \item \texttt{Type\_Name}: 6.
  \item \texttt{values}: 3.
  \item \texttt{void}: 4.
  \item \texttt{x}: 5, 6, 14, 15, 18, 28, 30, 67, 76.
  \item \texttt{y}: 6, 30, 76.
  \item \texttt{z}: 4, 76.
  \item \texttt{zeros}: 4.
\end{itemize}
List of Refinements

- Initialization of static members 8
- LED A functions 6
- arithmetical functions 30, 36, 37, 39, 43, 47
- auxiliary functions 16, 86
- bias rounding 26
- bigfloat.c 10
- bigfloat.h 5
- calculate high 32
- calculate sqrt 44
- calculate sum 33
- calculate the sign 78
- check for denormalized number 60
- check for normal or denormal return value 70
- check for special values 65
- comparison operators 48, 50
- compute approximative result 41
- compute decimal logarithm of b 79
- compute decimal significant of b 80
- compute sum and error 32
- constructor body for integer data type 13
- constructors 12, 59
- data members of class bigfloat 9
- denormal case 71
- determine high and low part 61
- exact addition 34
- find bigger operand 31
- functions for internal use 4, 19, 54, 57, 75, 76, 77, 84
- general functions 14, 15, 18, 28, 67
- get the double’s sign 62
- get the exponent’s value 64
- get the significant’s value 63
- global identifiers 58, 66, 74
- handle special cases 35
- input/output operators 82, 83
- normal case 72
- output of the result 81
- private functions 7
- put it all together 56
- round to infinity 23
- round to minus infinity 25
- round to nearest 20
- round to plus infinity 24
- round to zero 22
- rounding of sqrt 45
- rounding step of todouble 69

Used in chunk 10
Used in chunk 5
Used in chunk 10
Used in chunk 5
Used in chunk 18
Used in chunk 5
Used in chunk 5
Used in chunk 10
Used in chunk 5
Used in chunk 10
Used in chunk 10
Used in chunk 10
Used in chunk 5
Used in chunk 10
Used in chunk 10
Used in chunk 10
Used in chunk 10
Used in chunk 10
Used in chunk 10
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 18
Used in chunk 43
Used in chunk 67
\{ scan exponent 89 \} Used in chunk 84.
\{ scan fraction 88 \} Used in chunk 84.
\{ scan integer part 87 \} Cited in chunk 88. Used in chunk 84.
\{ scan signs 85 \} Used in chunks 84 and 89.
\{ set the bits of a 73 \} Used in chunk 67.
\{ shift dividend's significant 40 \} Used in chunk 39.
\{ shift significant 27 \} Used in chunk 26.
\{ significant is even 21 \} Used in chunk 20.
\{ special case checking for operator $\equiv$ 49 \} Used in chunk 48.
\{ special case checking of operator $>$ 51 \} Used in chunk 50.
\{ special case checking of todouble 68 \} Used in chunk 67.
\{ special cases for div 42 \} Used in chunk 39.
\{ special cases for mul 38 \} Used in chunk 37.
\{ special cases of sqrt 46 \} Used in chunk 43.