MAYLIB

A symbolic number evaluator library.
Edition 0.7.2
June 2008

Patrick Pélissier
Patrick.Pelissier@removeme@gmail.com

Table of Contents

Introduction to the MAYLIB Library

Description

MAYLIB is a C library for symbolic mathematical calculations. It is not a CAS (Computer Algebra System): it doesn't have any internal programming language and it is very limited in functionalities. It doesn't support global variables, neither global functions: this is let to the user of the library.

MAYLIB is a tool for programmer, which means that he must be able to understand completely and fully what the tool can do and can not. This implies that MAYLIB specifications should be precise and exhaustive.

MAYLIB is not designed to be efficient on pure numerical computations. If you plan to do heavy computations involving integer polynomial or float matrix (for example), you should use another tool or library, and/or write the code yourself.

MAYLIB should be portable to any systems that support GCC (or a port of GCC) with the following assumptions :

- A `void (*)()` must be castable to a `void *` and vice-versa: we don't lose the value by applying the casting.
- We can't have the address of a function which is inside an array of char.
- `int *p; memset(&p, 0, sizeof p);` set p to NULL.

This is the case for the majority of the computers. It should work fine on 16, 32 or 64 bits systems (Windows, Linux, *BSD).

Even if it only supports GCC, a port to a C99 compiler (not a C++ compiler but a real C99 compiler) should be quite easy: just replace GCC's extensions in the internal header file.

Features

- Arbitrary precision integer and rational through GMP.
- Arbitrary precision float through MPFR.
- Automatic and user controlled simplification.
- Expansion and differentation of expressions.
- Substitutions and pattern matching.
- Calculations with symbolic and numeric matrices.
- Default atomic types: integer, rational, float, complex, identifier and undefined data.
- Extension through user defined types and user defined functions.

It doesn't do:

- Support of a tuning complete language.
- Limits, series and integrations.
- Factorization of polynomials.
- Solving a set of polynomial equations and even solving.
- Much more.

Computation on floating point numbers returns always the same output for a given sequence of operation, regardless of the architecture or the system (Thanks to MPFR).

How to use this Manual

Everyone should read Chapter 3 [Basics], page 6. If you need to install the library yourself, you need to read Chapter 1 [Installing], page 4, too.

The rest of the manual can be used for later reference, although it is probably a good idea to glance through it.

Copying Conditions

1 Installing the MAYLIB Library

1.1 How to install

Here are the steps needed to install the library on Unix systems:

1. To build MAYLIB, you first have to install GNU MP (version 4.1 or higher) on your computer and MPFR (version 2.1.0 or higher). You need '`GCC`' (and really GCC) and a '`make`' program. You can get GMP at http://www.gmplib.org/, MPFR at http://www.mpfr.org/ and GCC at http://gcc.gnu.org/.

   Please see their respective documentation to see how to install them.
2. In the MAYLIB source directory, type:

   '`make`'

   Or if you have not installed GMP and/or MPFR in /usr/local/, type:

   '`make GMP=$GMP_DIR MPFR=$MPFR_DIR`'

   This will compile MAYLIB, and create a library archive file '`libmay.a`'.

   By default, MAYLIB is built with the assertions turned on, which slows down a lot the library (a factor of 3), but is a lot safer. To have an optimized version, type:

   '`make CFLAGS="-O2 -fomit-frame-pointer" GMP=$GMP_DIR MPFR=$MPFR_DIR`'
3. '`make check [GMP=...]`'

   This will make sure the built is correct. If you get error messages, please report them (See Chapter 2 [Reporting Bugs], page 5.).
4. '`make install`'

   This will copy the file '`may.h`' to the directory '`/usr/local/include`', the file '`libmay.a`' to the directory '`/usr/local/lib`', and the file '`may.info`' to the directory '`/usr/local/info`'.

   Type: '`make install PREFIX=$PREFIX`'

   to install them in another directory.

1.2 Known Build Problems

MAYLIB suffers from all bugs from GMP and MPFR, plus many, many more.

Please report other problems. See Chapter 2 [Reporting Bugs], page 5.

1.3 Getting the Latest Version

The latest version of MAYLIB is available from http://www.yaronet.com/t3.

1.4 Known Problems

You may get a segmentation fault using MAYLIB. This is mainly because the stack size is too small. Uses:

- under BASH/ZSH: ulimit -s unlimited
- under TCSH: unlimited

2 Reporting Bugs

If you think you have found a bug in the MAYLIB Library, first have a look on http://www.yaronet.com/t3: perhaps this bug is already known, in which case you may find there a workaround for it. Otherwise, please investigate and report it. We have made this library available to you, and it is not to ask too much from you, to ask you to report the bugs that you find.

There are a few things you should think about when you put your bug report together.

You have to send us a test case that makes it possible for us to reproduce the bug. Include instructions on how to run the test case.

You also have to explain what is wrong; if you get a crash, or if the results printed are incorrect and in that case, in what way.

Please include compiler version information in your bug report.

If your bug report is good, we will do our best to help you to get a corrected version of the library; if the bug report is poor, we won't do anything about it (aside of chiding you to send better bug reports).

Send your bug report to: '`Patrick.Pelissier@removeme@gmail.com`'.

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please send a note to the same address.

3 Basics

All declarations needed to use MAYLIB are collected in the include file '`may.h`'. It is designed to work with both C and C++ compilers. You should include that file in any program using the library:

```
#include <may.h>
```

It is recommended that you read the basic section of GMP and MPFR manuals before continuing reading this one.

3.1 Nomenclature and Types

3.1.1 Symbolic number: may_t

A *symbolic number* is a symbolic expression (an integer, a rationnal, a float, an identifier, a sum of such objets, a product, a power, ...). The C data type for such objects is `may_t`. It is a pointer to an hidden and undocumented C struct. It can be seen as a tree with nodes (sum, product, functions) and leafs (Integer, Rationnal, float, identifier, complex).

Don't mix the C variable names, and the identifiers: they are completly independent.

3.1.2 Pair of symbolic number: may_pair_t

`may_pair_t` is a pair of two *symbolic numbers*.

The first element is the field `first`. The second element is the field `second`.

3.1.3 Domain of symbolic number: may_domain_e

A domain is a part of the complex plane. The C data type for such objects is `may_domain_e`. The available parts are:

- MAY_COMPLEX_D: The complex plane.
- MAY_NONZERO_D: The complex plane without zero.
- MAY_NONREPOS_D: All complex such that `Re(x)<=0`
- MAY_NONRENEG_D: All complex such that `Re(x)>=0`
- MAY_NONIMPOS_D: All complex such that `Im(x)<=0`
- MAY_NONIMNEG_D: All complex such that `Im(x)>=0`
- MAY_CINTEGER_D: All complex integers
- MAY_CRATIONAL_D: All complex rationals.
- MAY_EVEN_D: All even complex integers.
- MAY_ODD_D: All odd complex integers.
- MAY_OUT_UNIT_CIRCLE_D: All complex such that `abs(x) >= 1`
- MAY_IN_UNIT_CIRCLE_D: All complex such that `abs(x) <= 1`

You can create new parts by intersecting two parts (or more) of this list: for example, the reals are the part defined by `MAY_NONIMPOS_D|MAY_NONIMNEG_D` - you have to use the symbol `|` for the intersection.

Other parts of the complex plane are already defined:

- MAY_REAL_D: The reals
- MAY_REAL_NONPOS_D: The reals <= 0
- MAY_REAL_NONNEG_D: The reals >= 0

- MAY_REAL_POS_D: The reals > 0
- MAY_REAL_NEG_D: The reals < 0
- MAY_INTEGER_D: The integers
- MAY_INT_NONPOS_D: The integers <= 0
- MAY_INT_NONNEG_D: The integers >= 0
- MAY_INT_POS_D: The integers > 0
- MAY_INT_NEG_D: The integers < 0
- MAY_INT_EVEN_D: The even integers.
- MAY_INT_ODD_D: The odd integers.
- MAY_RATIONAL_D: The rationals.

3.2 The memory model

Contrary to the classical memory model (the `malloc` and `free` functions and friends) where you explicitely say to the memory handler which areas you don't want to use anymore, you have to say to MAYLIB which *symbolic number* you want to keep after a calculus.

It is harder than using a true garbage collector, but a lot simpler than using the classical memory model. It seems to be also very efficient.

When you start a new calculus, you put a mark (using `may_mark`). Then, you create a new symbolic expression. You evaluate it, and keep the evaluated result (using `may_keep`): all the intermediary calculus created after putting the mark are deleted except the kept expression.

The main philosophy of MAYLIB is copy-on-write: a *symbolic number* may be referenced by many other *symbolic numbers*. There is no way to known how many references there are (there is no reference counter or other tricks like this). So when you want to modify it, you have to create a new expression. This philosophy is usually quite good. But in some cases, it may be a nightmare: for example, to create accumulators or arithmetics on polynomial. So inside MAYLIB itself, there may be some exceptions but they are locals, and not visible from the outside.

3.3 The Stack

Like many other programs which work with mathematical expressions, MAYLIB also works internally with a stack. But this stack is completly hidden. Putting a mark means saving the stack pointer value. Allocating memory is only updating the stack pointer. Keeping a *symbolic number* means garbaging the stack and moving it at the saved position of the stack.

The GMP Custom Allocation functions are set to use the stack instead of the standard `malloc` functions. This may cause problems with other libraries which also redirect the GMP Custom Allocation for their own usage. MPFR caches are freed when you garbesh the stack.

3.4 Evaluated form

Usually, you construct an expression by using *constructor functions*: `may_add_c`, `may_sub_c`, etc. They return the expression in an *unevaluated form*: they are not simplified, neither in their internal canonic form. The function which transforms them in *evaluated form* is `may_eval`. During the conversion, various simplifications are performed as well. This function, contrary to all other, might not copy the expression while changing its form (for efficiency reasons).

All functions which endend with `_c` are *constructor functions*: they accept and return *unevaluated form*. All other functions, and except if documented otherwise, accept only and return only *evaluated form*.

Constructor functions may or may not corrupt the input *symbolic number* to construct the output : this liberty is let to the function for efficiency reasons since copying an image of the input may be much more expensive than performing the construction itself. Nevertheless, the default behavior is not to corrupt the input. If the function constructs the output by updating the input, it must explicitely say it. An evaluated symbolic number is never corrupted, except on garbaging.

3.5 Boolean

MAYLIB doesn't support boolean arithmetic natively (ie. without an extension). In this documentation, TRUE is assumed to be a non zero integer (of type int), and FALSE the integer zero.

4 An example of use of the MAYLIB Library

Before using MAYLIB, we need to initialize the library by calling `may_kernel_start`. Before quitting the program, it may be a good idea to call `may_kernel_end` to free the used memory.

This example initializes the library, parses an expression, expands it, and finally displays the results.

```
{
  may_t x, y;
  may_kernel_start (3000000, 0);
  x = may_parse_str ("(x+y)*(x-y)");
  y = may_expand (x);
  printf ("x=%s y=%s\n", may_get_string (NULL, 0, x),
                         may_get_string (NULL, 0, y));
  may_kernel_end ();
}
```

The first line creates two uninitialized C variables `x` and `y`.

The second line initializes the MAYLIB library.

The third line parses the string `"(x+y)*(x-y)"`, returns the evaluated *symbolic number* and saves it in `x`.

Then it expands `x` and stores the expanded result in `y` (in its evaluated form).

Then it displays the value of `x` and `y` by converting them to strings before printing them.

Finally, it quits the library.

5 Interface

5.1 Kernel functions

`void may_kernel_start` (*size_t* ***stackSize***, *int allow*) [Function]
Initialize the MAYLIB library by allocating a stack of initial size at least *stackSize* bytes. The stack is expanded automatically if needed (On supported systems) if *allow* is not zero. It sets the GMP memory functions to use this stack after saving them.

You must call this function before any-other call to MAYLIB functions, otherwise you WILL crash your program.

You can only start the MAYLIB library once, which means that you can't call `may_kernel_start` more than once: if you call this function twice without calling `may_kernel_end` between, the behavior is undefined.

`void may_kernel_end` (*void*) [Function]
End the MAYLIB library by freeing the allocated stuff. Calling any other MAYLIB functions except `may_kernel_start` after this function is undefined behaviour.

`void may_kernel_stop` (*void*) [Function]
Stop the MAYLIB Kernel. Calling any other MAYLIB functions except `may_kernel_start` after this function is undefined behaviour. It restores the original GMP memory functions. It is designed so that MAYLIB can be used with other libraries which also change the GMP memory functions: it stops MAYLIB so that you can use them.

It doesn't destroy any MAYLIB globals or state but simply pause MAYLIB.

`void may_kernel_restart` (*void*) [Function]
Restart the MAYLIB library by saving the current GMP memory functions and restoring the MAYLIB ones. You can use once again all the MAYLIB functions.

`mp_rnd_t may_kernel_rnd` (*mp_rnd_t* ***rnd***) [Function]
Set the current rounding mode used by the MPFR functions. Return the previous used rounding mode. The default is `GMP_RNDN`. It doesn't interfere with `mpfr_set_default_rnd_mode`.

`mp_prec_t may_kernel_prec` (*mp_prec_t* ***prec***) [Function]
Set the current precision used for new MPFR floats. Return the previous used precision. The default is 113 (It may changed). Previous computed floats aren't expanded to this new precision.

`may_domain_e may_kernel_domain` (*may_domain_e* ***domain***) [Function]
Set the current domain used for new anonymous identifier. Return the previous used domain. The default is `MAYLIB_COMPLEX_D` . Previous anonymous identifiers aren't set to this new domain.

`may_t may_kernel_intmod` (*may_t* ***n***) [Function]
Set the used integer for all modulo operations, ie it defines the integer arithmetics on Z (if n=NULL or n=0) or Z/nZ (otherwise). If n disapears after a heap compact, this integer is reseted to NULL. However, you should not depend on this behavior. It assumes n is an integer or NULL. It returns the previous used integer.

`unsigned long may_kernel_intmaxsize` (*unsigned long* `n`) [Function]
Set the maximum size in bit for the size of the results of operation involving integers (typically power and factorial). If the result is estimated to be bigger than the given *n* number of bits, it won't compute it. It returns the previous used number of bits.

`int may_kernel_num_presimplify` (*int* `flag`) [Function]
When you parse strings, the float numbers may be evaluated to the current precision if *flag* is set, or they may be stored as float-string, without evaluating them with the current precision: we delay the evaluation to a later step where we know the needed precision. Set the current value of this flag. Return the previous used flag. The default is `1`, e.g. we evaluate the float at parsing time.

`void may_kernel_info` (*FILE* *`stream`, *const char* *`str`) [Function]
Display various kernel information inside the stream *stream* using the string *str*.

Any change in the global settings by using the may_kernel_XXXX functions has as consequence that you have to reevaluate all the previously computed variables (using `may_reeval`) so that theses variables are computed using this new set of global settings.

5.2 Memory functions

`void * may_alloc` (*size_t* `size`) [Function]
Allocate *size* bytes and return a pointer to the allocated memory. The memory is not cleared. The value returned is a pointer to the allocated memory, which is suitably aligned for a `long`, or `NULL` if the request fails.

`void * may_realloc` (*void* *`ptr`, *size_t* `oldsize`, *size_t* `newsize`) [Function]
Change the size of the memory block pointed to by *ptr* whose size is *oldsize* bytes to *newsize* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. *ptr* can not be `NULL` and *newsize* can not be `0`, otherwise the behavior is undefined. *ptr* must have been returned by an earlier call to `may_alloc` or `may_realloc`. The value returned is a pointer to the allocated memory, which is suitably aligned for a `long`, or `NULL` if the request fails.

`void may_free` (*void* *`ptr`, *size_t* `oldsize`) [Function]
Tries to free the memory space pointed to by *ptr*, which must have been returned by a previous call to `may_alloc()` or `may_realloc()`. *oldsize* must be the size in bytes of the allocated memory. It is needed since it is not stored internally. Otherwise, or if `may_free(ptr)` has already been called before, or if *oldsize* is not equal to the size passed to `may_alloc` or `may_realloc`, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

Note 1: This function tries to free the memory, but it may fail. The only way to be sure the memory is cleaned is to call `may_compact` or friends.

Note 2: To help this function to succeed, you have to call `may_free` in the reverse order of the call to `may_alloc`, just like some push/pop functions.

`void may_mark` ([*may_mark_t* `mark`]) [Function]
Push a mark and save it in *mark*: all the current state of the memory is saved. On C99 or GNU C compilers, the parameter *mark* is optionnal and may be ommited: a dummy mark will be created at the level of the function call.

`may_t may_compact` ([*may_mark_t* **`mark`**,]*may_t* **`x`**) [Function]
Compact the *symbolic number* *x* from the mark: all the created variables between the referenced mark *mark* and the call to this function are lost, except *x* which is garbaged. *x* may be NULL, in which case it clears all the *symbolic numbers* created during the referenced mark and this function call.

Return the new valid pointer to the compacted *symbolic number*. *x* is not a valid reference to a *symbolic number* anymore. You usually use it like this:

```
may_mark_t mark;
may_mark(mark);
// Perform some computation
// Keep only x
 x = may_compact (mark, x);
```

On C99 or GNU C compilers, the parameter *mark* is optionnal and may be ommited: the dummy mark created using omitting the parameter in `may_mark` is used:

```
may_mark();
// Perform some computation
// Keep only x
 x = may_compact (x);
```

`may_t * may_compact_v` ([*may_mark_t* **`mark`**,] *size_t* **`size`**, *may_t* ***`tab`**) [Function]
Compact all the *symbolic number* stored in the array with *size* elements pointed by *tab*. If *tab* was allocated on the heap, it returns the updated *tab* if *tab* has to be compacted too. It returns *tab* otherwise. On C99 or GNU C compilers, the parameter *mark* is optionnal and may be ommited: the dummy mark created using omitting the parameter in `may_mark` is used:

`void may_compact_va` (*may_mark_t* **`mark`**, *may_t* ***`x`**, ...) [Function]
Compact all the *symbolic numbers* passed as arguments to the function (as a variable argument list). The variable argument list is assumed to be composed of the address to the symbolic numbers to compact (`may_t*`) It begins from *x* and ends when it encounters a null pointer. A maximum of 20 parameters can be compacted using this function. The parameter *mark* is not optionnal and can not be ommited.

```
may_t x, y, z;
may_mark_t mark;
may_mark(mark);
// Perform some computation
// Keep only x, y and z
may_compact_va (mark, &x, &y, &z, NULL);
```

`may_t may_keep` ([*may_mark_t* **`mark`**,]*may_t* **`x`**) [Function]
Perform a similar operation than `may_compact` except that it assumes that the parameter *mark* won't be used anymore. On C99 or GNU C compilers, the parameter *mark* is optionnal and may be ommited: the dummy mark created using omitting the parameter in `may_mark` is used.

```
may_mark();
// Perform some computation
// Keep and return the expanded result
return may_keep (may_expand (x));
```

5.3 Setting functions

`may_t may_set_si` (*long* `x`) [Function]
`may_t may_set_ui` (*unsigned long* `x`) [Function]
`may_t may_set_z` (*mpz_t* `x`) [Function]
`may_t may_set_q` (*mpq_t* `x`) [Function]
`may_t may_set_d` (*double* `x`) [Function]
`may_t may_set_ld` (*long double* `x`) [Function]
`may_t may_set_fr` (*mpfr_t* `x`) [Function]
Return a newly created *symbolic number*, setting it to *x*. The internal type is set to INTEGER (`may_set_si`, `may_set_ui` and `may_set_z`), RATIONAL (`may_set_q`) and FLOAT (`may_set_d`, `may_set_ld` and `may_set_fr`).

`may_t may_set_zz` (*mpz_t* `x`) [Function]
Return a newly created *symbolic number*, setting it to *x* regardless of the current INTEGER modulo (See `may_kernel_intmod`). The internal type is set to INTEGER.

`may_t may_set_str` (*const char* *`x`) [Function]
`may_t may_set_str_domain` (*const char* *`x`, *may_domain_e* `domain`) [Function]
Return a new *symbolic number* of type IDENTIFIER (or STRING or SYMBOL, it is the same thing), setting it to *x* inside domain *domain*. `may_set_str` uses the global domain as defined by `may_kernel_domain`, whereas `may_set_str_domain` uses the given parameter *domain*. If *x* is `NAN`, `INFINITY`, `PI` or `I`, it doesn't return an IDENTIFIER, but the *symbolic number* `NAN`, `INFINITY`, `PI` or `I` without taking care about *domain*. The domain *domain* of *x* must be consistent for all the creations of symbols involving the string *x*, until a call to `may_keep`, otherwise the behaviour is undefined.

`may_t may_set_si_ui` (*long* `num`, *unsigned long* `denom`) [Function]
Return a newly created *symbolic number*, setting it to the rational *num*/*denom*.

`may_t may_set_cx` (*may_t* `re`, *may_t* `im`) [Function]
Return a newly created *symbolic number*, setting it to the complex number $re + im \times I$.

`may_t may_parse_str` (*const char* *`str`) [Function]
Return a newly created *symbolic number*, setting it to the value of the parsed string *str*. It is different from `may_set_str` since this function parsed the string, whereas `may_set_str` just set an identifier. TODO: Explain valid input, grammar, etc.

```
x = may_parse_str ("x+y+z(x*y)");
x = may_parse_str ("x^2+y/z(x^y)");
x = may_parse_str ("(x+cos(PI))/f(x*y+3!)");
```

5.4 Getting functions

`int may_get_ui` (*unsigned long* *`val`, *may_t* `x`) [Function]
`int may_get_si` (*long* *`val`, *may_t* `x`) [Function]
`int may_get_str` (*const char* **`val`, *may_t* `x`) [Function]
`int may_get_d` (*double* *`val`, *may_t* `x`) [Function]
`int may_get_ld` (*long double* *`val`, *may_t* `x`) [Function]
`int may_get_z` (*mpz_t* `val`, *may_t* `x`) [Function]
`int may_get_q` (*mpq_t* `val`, *may_t* `x`) [Function]
`int may_get_fr` (*mpfr_t* `val`, *may_t* `x`) [Function]
Set `*val` to the *symbolic number* *x*. It may perform some convertions to fit the type needs. Return 0 if the setting was ok, and there was no convertion. Return 1 if the setting was

ok, but a convertion was done. Return -1 if the setting was impossible: the internal types mismatch. Return -2 if the setting was impossible because the C type is too small to fit the requested value.

`int may_get_cx (`*may_t* `*re`, *may_t* `*im`, *may_t* `x`*);* [Function]
Set `*re` to the real part of the *symbolic number* *x*. Set `*im` to the imaginary part of the *symbolic number* *x*. *x* must be a pure numerical type (INTEGER, RATIONAL, FLOAT or COMPLEX). Return 0 if the setting was ok, and there was no convertion. Return 1 if the setting was ok, but a convertion was done. Return -1 if the setting was impossible: the internal types mismatch.

`char * may_get_string (`*char* `*out`, *size_t* `length`, *may_t* `x`) [Function]
Transform a *symbolic number* to a string, parsable by `may_parse_str`. See `may_parse_str` for the format of the string. The string is:

- either stored at the buffer of size *length* bytes, pointed by *out*. If the string needs more space than what is available, the string is simply cut to fit inside the buffer.
- or allocated on the internal stack is if *out* is `NULL`. In which case, the string is valid until a later call to `may_keep` or friends.

In both cases, it returns a pointer to the created string, or `NULL` in case of an error.

5.5 Constructors functions

`may_t may_add_c (`*may_t* `x`, *may_t* `y`) [Function]
`may_t may_sub_c (`*may_t* `x`, *may_t* `y`) [Function]
`may_t may_mul_c (`*may_t* `x`, *may_t* `y`) [Function]
`may_t may_div_c (`*may_t* `x`, *may_t* `y`) [Function]
`may_t may_mod_c (`*may_t* `x`, *may_t* `y`) [Function]
`may_t may_pow_c (`*may_t* `x`, *may_t* `y`) [Function]
`may_t may_pow_si_c (`*may_t* `x`, *long* `y`) [Function]
`may_t may_gcd_c (`*may_t* `x`, *may_t* `y`) [Function]
Construct a new *symbolic number* defined as the sum, difference, product, quotient, remainder, power or gcd of *x* and *y*.

`may_t may_addinc_c (`*may_t* `x`, *may_t* `y`) [Function]
`may_t may_mulinc_c (`*may_t* `x`, *may_t* `y`) [Function]
Construct a new *symbolic number* defined as the sum or product of *x* and *y*. The memory used by *x* may be reused to construct the result if *x* is not evaluated, making *x* an invalid variable which can't be used anymore. They are MUCH more efficient than `may_add_c` or `may_mul_c` if you use *x* as an accumulator like in this example:

```
for (i = 0; i < 100; i++) {
  may_t tmpresult = my_function (i);
  acc = may_addinc_c (acc, tmpresult);
}
acc = may_eval (acc);
```

`may_t may_neg_c (`*may_t* `x`) [Function]
`may_t may_sqr_c (`*may_t* `x`) [Function]
`may_t may_sqrt_c (`*may_t* `x`) [Function]
`may_t may_exp_c (`*may_t* `x`) [Function]
`may_t may_log_c (`*may_t* `x`) [Function]

`may_t may_abs_c` (*may_t* `x`) [Function]
`may_t may_sin_c` (*may_t* `x`) [Function]
`may_t may_cos_c` (*may_t* `x`) [Function]
`may_t may_tan_c` (*may_t* `x`) [Function]
`may_t may_asin_c` (*may_t* `x`) [Function]
`may_t may_acos_c` (*may_t* `x`) [Function]
`may_t may_atan_c` (*may_t* `x`) [Function]
`may_t may_sinh_c` (*may_t* `x`) [Function]
`may_t may_cosh_c` (*may_t* `x`) [Function]
`may_t may_tanh_c` (*may_t* `x`) [Function]
`may_t may_asinh_c` (*may_t* `x`) [Function]
`may_t may_acosh_c` (*may_t* `x`) [Function]
`may_t may_atanh_c` (*may_t* `x`) [Function]
`may_t may_conj_c` (*may_t* `x`) [Function]
`may_t may_real_c` (*may_t* `x`) [Function]
`may_t may_imag_c` (*may_t* `x`) [Function]
`may_t may_argument_c` (*may_t* `x`) [Function]
`may_t may_sign_c` (*may_t* `x`) [Function]
`may_t may_floor_c` (*may_t* `x`) [Function]
`may_t may_ceil_c` (*may_t* `x`) [Function]
`may_t may_fact_c` (*may_t* `x`) [Function]
`may_t may_gamma_c` (*may_t* `x`) [Function]

Construct a new *symbolic number* defined as the image of the function neg, sqr, sqrt, exp, log, abs, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, conj, real, imag, argument, sign, floor, ceil, factorial or gamma at *x*. Theses functions are handled internally by MAYLIB, except ceil and factorial which are just wrappers.

`may_t may_func_c` (*const char* *`name`, *may_t* `x`) [Function]
`may_t may_func_domain_c` (*const char* *`name`, *may_t* `x`, *may_domain_e* `domain`) [Function]

Construct a new *symbolic number* defined as the image of the function whose name is *name* at *x*. The function is assumed to have its image inside either the global domain as defined by `may_kernel_domain` for `may_func_c` or *domain* for `may_func_domain_c`. *name* may be one of the internal functions of MAYLIB, in which case the appropriate constructor is called, and *domain* is ignored.

`may_t may_range_c` (*may_t* `l`, *may_t* `r`) [Function]

Construct a new *symbolic number* defined as the Real Interval from *l* to *r*. *l* and *r* must be numerical reals, otherwise the behavior is undefined.

`may_t may_list_vc` (*size_t* `size`, *const may_t* *`tab`) [Function]
`may_t may_add_vc` (*size_t* `size`, *const may_t* *`tab`) [Function]
`may_t may_mul_vc` (*size_t* `size`, *const may_t* *`tab`) [Function]

Construct a new *symbolic number* defined as the list, sum or product of the elements of the array of size *size* elements, and pointed by *tab*.

`may_t may_list_vac` (*may_t* `x`, ...) [Function]
`may_t may_add_vac` (*may_t* `x`, ...) [Function]
`may_t may_mul_vac` (*may_t* `x`, ...) [Function]

Construct a new *symbolic number* defined as the list, sum or product of the elements of the given `va_list`. The `va_list` is assumed to be composed only of type `may_t`. It begins from *x*. It ends when it encounters a null pointer (`(may_t) 0`).

`may_t may_diff_c` (*may_t* `f`, *may_t* `vx`, *may_t* `order`, *may_t* `at`) [Function]
Construct a new *symbolic number* defined as the differentiate of order *order* of the expression *f* by the variable *vx*, evaluated at point *at*. *vx* must be an identifier, otherwise the behavior is undefined. *order* may not be an integer, but the meaning of the constructed expression is undefined.

5.6 Mathematical functions

`may_t may_add` (*may_t* `x`, *may_t* `y`) [Function]
`may_t may_sub` (*may_t* `x`, *may_t* `y`) [Function]
`may_t may_mul` (*may_t* `x`, *may_t* `y`) [Function]
`may_t may_div` (*may_t* `x`, *may_t* `y`) [Function]
`may_t may_mod` (*may_t* `x`, *may_t* `y`) [Function]
`may_t may_pow` (*may_t* `x`, *may_t* `y`) [Function]
Return a new *symbolic number* defined as the evaluated sum, difference, product, quotient, remainder or power of *x* and *y*.

`may_t may_neg` (*may_t* `x`) [Function]
`may_t may_sqr` (*may_t* `x`) [Function]
`may_t may_sqrt` (*may_t* `x`) [Function]
`may_t may_exp` (*may_t* `x`) [Function]
`may_t may_log` (*may_t* `x`) [Function]
`may_t may_abs` (*may_t* `x`) [Function]
`may_t may_sin` (*may_t* `x`) [Function]
`may_t may_cos` (*may_t* `x`) [Function]
`may_t may_tan` (*may_t* `x`) [Function]
`may_t may_asin` (*may_t* `x`) [Function]
`may_t may_acos` (*may_t* `x`) [Function]
`may_t may_atan` (*may_t* `x`) [Function]
`may_t may_sinh` (*may_t* `x`) [Function]
`may_t may_cosh` (*may_t* `x`) [Function]
`may_t may_tanh` (*may_t* `x`) [Function]
`may_t may_asinh` (*may_t* `x`) [Function]
`may_t may_acosh` (*may_t* `x`) [Function]
`may_t may_atanh` (*may_t* `x`) [Function]
`may_t may_conj` (*may_t* `x`) [Function]
`may_t may_real` (*may_t* `x`) [Function]
`may_t may_imag` (*may_t* `x`) [Function]
`may_t may_argument` (*may_t* `x`) [Function]
`may_t may_sign` (*may_t* `x`) [Function]
`may_t may_floor` (*may_t* `x`) [Function]
`may_t may_ceil` (*may_t* `x`) [Function]
`may_t may_fact` (*may_t* `x`) [Function]
`may_t may_gamma` (*may_t* `x`) [Function]
Return a new *symbolic number* defined as the evaluated image of the function neg, sqr, sqrt, exp, log, abs, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, conj, real, imag, argument, sign, floor, ceil, factorial or gamma at *x*.

5.7 Replace functions

`may_t may_replace` (*may_t* `x`, *may_t* `o`, *may_t* `n`) [Function]
Traverse the *symbolic number x* and replace each references to *o* by *n*. *o* may be a symbolic number of any type. It returns the newly created *symbolic number*.

For nodes with variable number of arguments (sum, product), it won't look if a sub-expression matches o, e.g. $2 + y$ won't be replaced by z in $2 + x + y$. Use `may_rewrite` instead.

`may_t may_subs_c` (*may_t* `x`, *unsigned long* `level`, *size_t* `sv`, *const char* **const**`v`, *const void* **const**`value`) [Function]

Traverse the *symbolic number* x and substitute the matchning references to an identifier or a function to the corresponding value. v is an array of strings of size sv, and *value* is an array of *symbolic numbers* or function callbacks (MAYLIB identifies itself what it is) of size sv, otherwise the behavior is undefined.

The function traverses the *symbolic number* x and each time it reachs:

- an identifier: it checks if the identifier is inside the array v and if the corresponding value is a *symbolic number*. In which cases, it replaces the identifier by the corresponding *symbolic number*.
- a function call: it checks if the function name is inside the array v and if the corresponding value is a function callback. In which cases, it replaces the identifier by the returned value of the callback. The callback as prototype: `may_t callback_c (may_t x)`. The callback doesn't have to return an evaluated argument. If the function has multiple arguments, x is the list of the arguments (a *symbolic number* of type LIST).

level defines the number of times the function must call itself after substituing a value: it is to avoid the circular definition error. But if substituing the value doesn't change the *symbolic number* (defined as pointer equality), it won't call again `may_subs_c` even if *level* is not zero, since it assumes it has reached a fixed point.

It returns the newly created *symbolic number*.

`int may_match_p` (*may_t* ***value*, *may_t* `x`, *may_t* `pattern`, *int* `size`, *int* (****`funcp`)(*may_t*)) [Function]

Return TRUE if x matches the pattern *pattern*. A *pattern* is an algebraic expression that optionally contains wild-cards. A wild-card is an identifier of the form `'$NUMBER'` , where NUMBER is a decimal number from 0 to `size-1`, which represents an arbitrary expression. Every wild-card has an index, or a label, which is an unsigned integer number. Wild-cards behave like symbols: they are subject to the same transformations. For example, `$0+$0` is transformed to `2*$0`. If one wild-card appears multiple times in a pattern, it must match the same expression everywhere.

value is an array which must be initially filled with *size* NULL pointers (otherwise the behavior of this function is undefined). It will be filled with the values of the corresponding wild-cards if the match succeed. Otherwise, its contain is undefined.

If *funcp* is not NULL, each wild-card is first checked with its corresponding predicate function in this array. If the function returns FALSE when it is called with a *symbolic number*, `may_match_p` assumes that this *symbolic number* can not match the wild-card. If the function returns TRUE, it assumes that the wild-card matches the *symbolic number*.

Sums and products are treated in a special way. Only a maximum of one wild-card is allowed as a child of a sum or a product, otherwise it leads to an unpredictable result.

`may_t may_rewrite` (*may_t* `x`, *may_t* `pattern`, *may_t* `newpattern`) [Function]

Rewrite the *symbolic number* x by applying the rule: if the pattern *pattern* matches any sub expression of x, replace them by *new*. *pattern* and *new* are patterns as defined in the function `may_match_p`. The corresponding values of the wild-cards, found by trying to match *pattern*, are replaced when substituting *new* in the place of the matching subexpression of x.

new may not contain all the used wild-cards of the pattern *pattern*, but the inverse must be true (otherwise the behavior is undefined). For example, to apply the mathematical property $sin(x)^2 + cos(x)^2 = 1$, you may use the following code:

```
y = may_rewrite (y, may_parse_str ("$0+sin($1)^2+cos($1)^2"),
                    may_parse_str ("$0+1"));
```

Note that `$1` is not used in the *new* pattern. This function may be very slow for some *symbolic numbers*, and is currently implemented in a naive way.

If `may_rewrite` does nothing (ie. doesn't apply any transformation), it returns *x* (and not a copy of *x*). This feature can be used to detect how many times you may have to apply `may_rewrite` to rewrite the *symbolic number* (If the transformation may introduce other transformations).

5.8 Traversal functions

`const char * may_get_name` (*may_t* `x`) [Function]

Return the type of the *symbolic number* x. See next section for details on type names. This function always return a pointer to thoses corresponding names, except for user functions. For efficient check of the type of *x*, it is faster to check if the pointers are equals than checking if the strings are equals. Instead of:

```
const char *type = may_get_name (x);
if (strcmp (type, may_sum_name) == 0)
```

Do this:

```
const char *type = may_get_name (x);
if (type == may_sum_name)
```

If the node of *x* is not an internal function, it returns the name of the user function. The only way to identify it is to use `strcmp`. If the node of *x* is an user defined type, it returns the registered pointer to the type name. The fact of avoiding any name collision (between function and type) is let to the user.

`size_t may_nops` (*may_t* `x`) [Function]

If the *symbolic number* *x* is a node (a sum, a product, ...), it returns the number of elements of the node. Otherwise, it returns 0.

`may_t may_op` (*may_t* `x`, *size_t* `index`) [Function]

If the *symbolic number* *x* is a node (a sum, a product, ...) and if *index* is inside the valid range of elements of the node, it returns the sub element *index* of the node. Otherwise, it returns NULL. The range starts from `0` and ends at `may_nops(x)-1`.

`may_t may_map_c` (*may_t* `x`, *may_t* (*`func`)(*may_t*)) [Function]

If the *symbolic number* *x* is a node (a sum, a product, a list, ...) it returns a new node of the same type whose children have been composed by the call of *func* with the corresponding child in *x*. Otherwise it returns `(*func) (x)`. *func* doesn't have to return an evaluated form, but it shouldn't expect an evaluated form if *x* is not an evaluated form.

Example: `map ({x,y,z}, f)` gives `{f(x),f(y),f(z)}`.

`may_t may_map2_c` (*may_t* `x`, *may_t* (*`func`)(*may_t,void**), *void**`data`) [Function]

Same function as `may_map_c` except that the called function as an extra argument for which *data* is given. It may be usefull to pass extra arguments to a function throught a wrapper.

`may_t may_map` (*may_t* **x**, *may_t* (***func**)(*may_t*)) [Function]
Same function as `may_map_c` except that it returns an evaluated form.

`may_t may_map2` (*may_t* **x**, *may_t* (***func**)(*may_t*,*void**), *void**__data__) [Function]
Same function as `may_map2_c` except that it returns an evaluated form.

5.9 Polynomial functions

All theses functions expect their input to be a polynomial.

`int may_degree` (*may_t *__coeff__*, *mpz_srcptr* __degree__[], *may_t *__leader__*, *may_t* __expr__, *size_t* __numvar__, *const may_t* __var__[]) [Function]
Extract the degree, its corresponding coefficient and the leader term of *expr*, seen as a polynomial of the array of variable(s) *var* (of size *numvar*); store them in, respectively, the variables pointed by *degree* (an array of size *numvar*), *coeff* and *leader* (if theses pointers are not NULL). The returned degree of `0` is -1. This function returns TRUE if it has succeeded, FALSE otherwise.

`long may_degree_si` (*may_t* __expr__, *may_t* __var__) [Function]
Return the degree of *expr* view as an univariate polynomial of the variable *var*. Returns LONG_MAX if *expr* is not an univariate polynomial of the variable *var* or if the degree doesn't fit in a long. Return LONG_MIN if *expr* was 0.

`int may_ldegree` (*may_t *__coeff__*, *mpz_srcptr* __degree__[], *may_t *__leader__*, *may_t* __expr__, *size_t* __numvar__, *const may_t* __var__[]) [Function]
Extract the lowest degree, its corresponding coefficient and the leader term of *expr*, seen as a polynomial of the array of variable(s) *var* (of size *numvar*); store them in, respectively, the variables pointed by *degree* (an array of size *numvar*), *coeff* and *leader* (if theses pointers are not NULL). The returned degree of `0` is -1. This function returns TRUE if it has succeeded, FALSE otherwise.

`long may_ldegree_si` (*may_t* __expr__, *may_t* __var__) [Function]
Return the lowest degree of *expr* view as an univariate polynomial of the variable *var*. Returns LONG_MAX if *expr* is not an univariate polynomial of the variable *var* or if the degree doesn't fit in a long. Return LONG_MIN if *expr* was 0.

`void may_content` (*may_t *__content__*, *may_t *__primpart__*, *may_t* __b__, *may_t* __x__) [Function]
Extract the content and the primpart of *b* view as an univariate polynomial of the variable *x* and store them in *content* and *primpart* if there are not null. If *x* is NULL, it returns the content and the primpart of the implicit multivariate polynomial (ie. the integer content).

`may_t may_sqrfree` (*may_t* **a**, *may_t* **x**) [Function]
Return the square free factorisation of *a* view as an univariate polynomial of the variable *x*.

`may_t may_ratfactor` (*may_t* **a**, *may_t* **x**) [Function]
Return the factorisation of *a* view as an integer multivariate polynomial in the implicit variables (as return by *may_indets*) if *x* is NULL, or by the variable *x* otherwise, by extracting the terms of degree 1 over the integer from *a* after performing a square free factorisation.

`int may_div_qr` (*may_t *__q__*, *may_t *__r__*, *may_t* **a**, *may_t* **b**, *may_t* **var**) [Function]
Compute the quotient and the remainder of the univariate/multivariate polynomial of *a* by the univariate/multivariate polynomial *b* in the variable *var* (for univariate case) or the

variable list *var* (for multivariate case) so that the result satisfy `a = b*q + r` with `degree(r) < degree(b)` . If *var* is not a variable or a list of variables, the behaviour is undefined. If *a* or *b* can not be seen as polynomial of the variable *a*, it returns FALSE. It returns TRUE otherwise and stores in *q* the quotient if *q* is not NULL, and in *r* the reminder if *r* is not NULL.

`void may_div_qr_xexp` (*may_t* `*q`, *may_t* `*r`, *may_t* `a`, *may_t* `x`, *may_t* `n`) [Function]
Compute the quotient and the remainder of the univariate polynomial *a* by `x^n`. *x* must be a variable and *n* must be a strictly positive integer, otherwise the behaviour is undefined. This function doesn't check if *a* can be seen as an univariate polynomial of the variable *x*, and just return a result which satisfy `a = q*x^n + r`. It stores in *q* the quotient if *q* is not NULL, and in *r* the reminder if *r* is not NULL.

`may_t may_indets` (*may_t* `a`, *may_indets_e* `flags`); [Function]
Return the list of all the indeterminates of *a* view as a rational function. *flags* may be MAY_INDETS_NONE (No special treatment) MAY_INDETS_NUM (Return the numerical constants like PI or sqrt(2)) as indeterminates), MAY_INDETS_RECUR (Perform the search recursively over the found indeterminates) or a combinaison of theses three flags.

5.10 Type names.

`const char may_exp_name[]` [Variable]
`const char may_log_name[]` [Variable]
`const char may_sin_name[]` [Variable]
`const char may_cos_name[]` [Variable]
`const char may_tan_name[]` [Variable]
`const char may_asin_name[]` [Variable]
`const char may_acos_name[]` [Variable]
`const char may_atan_name[]` [Variable]
`const char may_sinh_name[]` [Variable]
`const char may_cosh_name[]` [Variable]
`const char may_tanh_name[]` [Variable]
`const char may_asinh_name[]` [Variable]
`const char may_acosh_name[]` [Variable]
`const char may_atanh_name[]` [Variable]
`const char may_abs_name[]` [Variable]
`const char may_sign_name[]` [Variable]
`const char may_floor_name[]` [Variable]
`const char may_mod_name[]` [Variable]
`const char may_gcd_name[]` [Variable]
`const char may_real_name[]` [Variable]
`const char may_imag_name[]` [Variable]
`const char may_conj_name[]` [Variable]
`const char may_argument_name[]` [Variable]
`const char may_gamma_name[]` [Variable]
The name of the internal `exp`, *log*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *sinh*, *cosh*, *tanh*, *asinh*, *acosh*, *atanh*, *abs*, *sign*, *floor*, *mod*, *gcd*, *real*, *imag*, *conj*, *argument* and *gamma* functions. They have only one child.

`const char may_integer_name[]` [Variable]
`const char may_rational_name[]` [Variable]
`const char may_float_name[]` [Variable]
`const char may_complex_name[]` [Variable]

`const char may_string_name[]` [Variable]
`const char may_data_name[]` [Variable]
The name of the internal leafs: INTEGER, RATIONAL, FLOAT, COMPLEX and STRING. They have no child. See Extension function section for details about DATA.

`const char may_sum_name[]` [Variable]
`const char may_product_name[]` [Variable]
`const char may_pow_name[]` [Variable]
The name of the internal nodes SUM, PRODUCT and POWER. POWER has exactly 2 children. SUM and PRODUCT have a variable number of children, but is assumed to be more than one.

`const char may_mat_name[]` [Variable]
`const char may_list_name[]` [Variable]
`const char may_range_name[]` [Variable]
`const char may_diff_name[]` [Variable]
Other node names.

5.11 Evaluating functions

`may_t may_eval` (*may_t* `x`) [Function]
Evaluate x, simplify it and return it in its evaluated form. The used transformations are:

- at most of complexity $O(n \times log(n))$, where n is related to the size of the *symbolic number*.
- algebraically correct, possibly except for a set of measure zero: y/y is simplified into 1.
- deterministic (The terms of sums and products are re-ordered into a deterministic way).

As soon as you evaluate a variable, all unevaluated variables which were used to build the evaluated variable become invalid and should not be used anymore, except as an argument to `may_eval` as long as you don't compact the stack. (They can't be used as inputs to other constructors too).

`may_t may_hold` (*may_t* `x`) [Function]
Hold x: mark it as evaluated without performing any simplification, for example to stop an infinite recursive evaluation. This function may be dangerous (from wrong results to a crash of the application), since the other functions expect an *evaluated form*.

`may_t may_reeval` (*may_t* `x`) [Function]
Reevaluate x without taking care of the internal flag saying that x was already evaluated, taking into account the new set of global rules (Integer modulo, precision of floats, ...). It simplifies it and returns it in its evaluated form.

`may_t may_evalf` (*may_t* `x`) [Function]
Evaluate x in the FLOAT domain, transforming all integers and rationals to floats with the current precision (See `may_kernel_prec`).

`may_t may_evalr` (*may_t* `x`) [Function]
Evaluate x in the REAL RANGE domain, returning a REAL RANGE. The exact value of x is inside this RANGE.

`may_t may_approx` (*may_t* `x`, *unsigned int* `base`, *unsigned long* `n`, *mp_rnd_t* `rnd`) [Function]

Assuming x is a numerical value (otherwise the behavior of this function is undefined), returns a FLOAT STRING which is an approximation of x in base *base* with n digits, rounded in the direction *rnd*. See `may_num_presimplify` for a proper usage of this function.

5.12 Predicate functions

All the predicate functions return true if the property is really verified. If the property is false or if it can't be decided, it returns false.

`int may_num_p` (*may_t* `x`) [Function]

Return true if x is numerical.

`int may_zero_p` (*may_t* `x`) [Function]

Return true if x is zero.

`int may_nonzero_p` (*may_t* `x`) [Function]

Return true if x is not zero.

`int may_zero_fastp` (*may_t* `x`) [Function]

Return true if x is zero without trying to normalize the expression.

`int may_one_p` (*may_t* `x`) [Function]

Return true if x is the integer one.

`int may_real_p` (*may_t* `x`) [Function]

Return true if x is real (i.e. not a complex).

`int may_pos_p` (*may_t* `x`) [Function]

Return true if x is greater than 0.

`int may_nonneg_p` (*may_t* `x`) [Function]

Return true if x is greater than or equal to 0.

`int may_neg_p` (*may_t* `x`) [Function]

Return true if x is less than 0..

`int may_nonpos_p` (*may_t* `x`) [Function]

Return true if x is less than or equal to 0.

`int may_integer_p` (*may_t* `x`) [Function]

Return true if x is integer.

`int may_cinteger_p` (*may_t* `x`) [Function]

Return true if x is a complex integer (such as 1+3*I or 42).

`int may_rational_p` (*may_t* `x`) [Function]

Return true if x is rational (integers are rational too).

`int may_crational_p` (*may_t* `x`) [Function]

Return true if x is a complex rational.

`int may_even_p (`*may_t* `x)` [Function]
Return true if *x* is an even integer.

`int may_odd_p (`*may_t* `x)` [Function]
Return true if *x* is an odd integer.

`int may_posint_p (`*may_t* `x)` [Function]
Return true if *x* is an integer greater than 0.

`int may_nonnegint_p (`*may_t* `x)` [Function]
Return true if *x* is an integer greater than or equal to 0.

`int may_negint_p (`*may_t* `x)` [Function]
Return true if *x* is an integer less than 0.

`int may_nonposint_p (`*may_t* `x)` [Function]
Return true if *x* is an integer less than or equal to 0.

`int may_prime_p (`*may_t* `x)` [Function]
Return true if *x* is a prime integer (See mpz_probab_prime_p for details).

`int may_nan_p (`*may_t* `x)` [Function]
Return true if *x* is NAN (Not A Number) (See MPFR manual for details).

`int may_inf_p (`*may_t* `x)` [Function]
Return true if *x* is INFINITY.

`int may_undef_p (`*may_t* `x)` [Function]
Return true if *x* is UNDEF, e.g. a *symbolic number* which contains NAN.

`int may_purenum_p (`*may_t* `x)` [Function]
Return true if *x* is an INTEGER, a RATIONAL, a FLOAT or a COMPLEX.

`int may_purereal_p (`*may_t* `x)` [Function]
Return true if *x* is an INTEGER, a RATIONNAL or a FLOAT.

`int may_independent_p (`*may_t* `x`, *may_t* `var)` [Function]
Return true if *x* is independent of *var*. *var* may be an indentifier (for example `x`) or a more complex expression (for example `exp(z)`). For non identifier searching, it does pattern matching: it doesn't try to find any algebraic dependencies between *var* and *x*. For example, if `var = exp(z/2) and x = exp(z)`, it returns true.

`int may_independent_vp (`*may_t* `x`, *may_t* `list)` [Function]
Assuming *list* is a list of identifiers otherwise the behavior is undefined, return true if *x* is independent of all those identifiers.

`int may_func_p (`*may_t* `x`, *const char* `*funcname)` [Function]
Return true if *x* contains a call to function *funcname*.

`int may_exp_p (`*may_t* `x`, *may_exp_p_flags_e* `flags)` [Function]
Return true if *x* contains a call to a exp-log functions. `may_exp_p_flags_e` is an enumeration of which functions you are searching for:

- MAY_EXP_EXP_P: The exp function.
- MAY_LOG_EXP_P: The log function.
- MAY_SIN_EXP_P: The sin function.
- MAY_COS_EXP_P: The cos function.
- MAY_TAN_EXP_P: The tan function.
- MAY_ASIN_EXP_P: The arcsin function.
- MAY_ACOS_EXP_P: The arccos function.
- MAY_ATAN_EXP_P: The arctan function.
- MAY_SINH_EXP_P: The sinh function.
- MAY_COSH_EXP_P: The cosh function.
- MAY_TANH_EXP_P: The tanh function.
- MAY_ASINH_EXP_P: The arcsinh function.
- MAY_ACOSH_EXP_P: The arccosh function.
- MAY_ATANH_EXP_P: The arctanh function.

You may use a combinaison of any theses values to specify which you are looking for: for example, `MAY_EXP_EXP_P|MAY_COSH_EXP_P|MAY_ATANH_EXP_P` searches for the exp, cosh and atanh functions.

`int may_compute_sign` (*may_t* `x`) [Function]
Try to compute the sign of x. Return 0 if the sign is unkwown, 1 if $x = 0$, 2 if $x > 0$, 3 if $x >= 0$, 4 if $x < 0$, 5 if $x <= 0$.

5.13 Comparaison functions

`int may_identical` (*may_t* `x`, *may_t* `y`) [Function]
Return 0 if x and y are structurally equals. Otherwise it returns a non-nul value. The returned value is suitable for a total ordering, which means that if $identical(a, b) < 0$ and $identical(b, c) < 0$ then $identical(a, c) < 0$. There is no garranty that the current way of sorting will be the same in the future. This function only compares the structure, and not the mathematical equality, of its arguments (`x*(y+z)` and `x*y+x*z` are different).

`int may_cmp` (*may_t* `x`, *may_t* `y`) [Function]
Return 0 if x and y are structurally equals. Otherwise it returns a non-nul value. The returned value is suitable for a total ordering. The order is lexicographic.

5.14 Operator functions

`may_t may_expand` (*may_t* `x`) [Function]
Return the expanded value of the *symbolic number* x, using the transformation $A * (B + C)$ to $A * B + A * C$.

`may_t may_collect` (*may_t* `a`, *may_t* `x`) [Function]
Return the expanded value of the *symbolic number* a, view as a polynomial of the variable x. If x is a list, it is view as a multinomial over the variables listed in x. x is assumed to be either a variable or a list of variable, otherwise the behaviour is undefined.

`may_t may_texpand` (*may_t* `x`) [Function]
Return the expanded value of the *symbolic number* x, expanding the trigonometric functions.

`may_t may_diff` (*may_t* `x`, *may_t* `vx`) [Function]
Return the differentiate of *x* by the variable *vx*. All identifiers and user functions are assumed to be independent of *vx*, except if there is an explicit use of *vx*.

`may_t may_antidiff` (*may_t* `x`, *may_t* `vx`) [Function]
Return the antidifferentiate of *x* by the variable *vx* if it exists, or NULL if it fails to compute it. All identifiers and user functions are assumed to be independent of *vx*, except if there is an explicit use of *vx*.

`may_t may_trig2exp` (*may_t* `x`) [Function]
Transform all trigonometrical functions to their exp/log equivalents.

`may_t may_trig2tan2` (*may_t* `x`) [Function]
Transform *sin*, *cos* and *tan* in terms of $tan(z/2)$.

`may_t may_tan2sincos` (*may_t* `x`) [Function]
Transform *tan* to sin/cos.

`may_t may_sin2tancos` (*may_t* `x`) [Function]
Transform *sin* to $tan * cos$.

`may_t may_sqrtsimp` (*may_t* `x`) [Function]
Simplify the square roots.

`void may_rectform` (*may_t* *`re`, *may_t* *`im`, *may_t* `x`) [Function]
Extract the real part and the imaginary part of *x*. Store at `*re` the real part and at `*im` the imaginary part.

`void may_comdenom` (*may_t* *`num`, *may_t* *`denom`, *may_t* `x`) [Function]
Extract the numerator and the denominator of *x*, regarding *x* as a rational function over the integers. Store at `*num` the evaluated numerator and at `*denom` the evaluated denominator. This function doesn't try to return a canonical form (See `may_rationalize`).

`may_t may_rationalize` (*may_t* `x`) [Function]
Convert *x* to canonical rational expression (canceling both numerator and denominator by their Greatest Common Divisor).

`may_t may_taylor` (*may_t* `f`, *may_t* `x`, *may_t* `a`, *unsigned long* `m`) [Function]
Return sum(diff(*f*,*x*,n)(*x*=>a)/n!*(*x*-a)^n,n=0,*m*). It may introduce NAN in its result since it won't compute the limit.

`may_t may_series` (*may_t* `f`, *may_t* `x`, *unsigned long* `m`) [Function]
Return the TAYLOR series of *f* view as a function of *x* up to the order *m-1* over 0. It may throw MAY_VALUATION_NOT_POS_ERR.

`may_t may_gcd` (*unsigned long* `size`, *may_t* *const `tab`) [Function]
Compute the Greatest Common Divisor (GCD for short) of all the elements of the array *tab* of size *size*, view as polynomomial over their implicit variables.

`may_t may_smod` (*may_t* `a`, *may_t* `b`) [Function]
Return the Modulus (in symetric representation): return *a* mod *b* in the range [-iquo(abs(b)-1,2), iquo(abs(b),2)]. *b* must be a positive integer, otherwise the bahaviour is undefined.

5.15 Iterator functions

`int may_sum_p (`*may_t* `x)` [Function]
Return TRUE if x is a sum.

`int may_product_p (`*may_t* `x)` [Function]
Return TRUE if x is a product.

`may_t may_sum_iterator_init (`*may_iterator_t* `it`, *may_t* `x)` [Function]
Initialize the iterator *it* so that it will traverse x view as a sum

$$x = \sum_{i=1}^{N} a_i * b_i$$

with `a[i]` a pure numerical number and `b[i]` the base.

It returns the pure-numerical part of the sum (ie. of base `0`) or 0 if there isn't one.

`void may_sum_iterator_next (`*may_iterator_t* `it)` [Function]
Increase the iterator *it* to point to the next term.

`int may_sum_iterator_end (`*may_t* `*factor`, *may_t* `*base`, *may_iterator_t* `it)` [Function]
Return TRUE if the iteration over all the terms of the sum is not finished, and in such a case, return the current pointed term by updating *factor* to the pure numerical part and *base* to the base so that `factor*base = the current term`

Return FALSE if the iteraton has finished.

`may_t may_sum_iterator_ref (`*may_iterator_t* `it)` [Function]
Return the current term pointed by *it*

`may_t may_product_iterator_init (`*may_iterator_t* `it`, *may_t* `x)` [Function]
Initialize the iterator *it* so that it will traverse x view as a product

$$x = \Pi_{i=1}^{N} b_i^{a_i}$$

with `a[i]` a pure integer number and `b[i]` the base.

It returns the pure-numerical part of the product (ie. of base `1`) or 1 if there isn't one.

`void may_product_iterator_next (`*may_iterator_t* `it)` [Function]
Increase the iterator *it* to point to the next term.

`int may_product_iterator_end (`*may_t* `*power`, *may_t* `*base`, *may_iterator_t* `it)` [Function]
Return TRUE if the iteration over all the terms of the product is not finished, and in such a case, return the current pointed term by updating *power* to the pure integer part and *base* to the base so that `base^power = the current term`

Return FALSE if the iteraton has finished.

`int may_product_iterator_end2 (`*may_t* `*power`, *may_t* `*base`, *may_iterator_t* `it)` [Function]
Return TRUE if the iteration over all the terms of the product is not finished, and in such a case, return the current pointed term by updating *power* to the power (may be not an integer) and *base* to the base so that `base^power = the current term`

Return FALSE if the iteraton has finished.

`may_t may_product_iterator_ref` (*may_iterator_t* `it`) [Function]
Return the current term pointed by *it*

5.16 IO functions

`void may_dump` (*may_t* `x`) [Function]
Dump the *symbolic number* to stdout.

`size_t may_in_string` (*may_t* `*x`, *FILE* `*in`) [Function]
Input a string from stream *in*, and put the read *symbolic number* to `*x`. Return the number of bytes read, or if an error occurred, return 0.

`size_t may_out_string` (*FILE* `*out`, *may_t* `x`) [Function]
Output *x* on stream *out*. Return the number of bytes written, or if an error occurred, return 0.

5.17 Error handling and exception

MAYLIB has three ways for handling errors:

- The functions return an error code.
- The functions return NAN for inputs with syntax errors.
- The functions throw exception.

This section describres the exception mechanisms of MAYLIB. All functions may throw the MEMORY ERROR exception. All other throwed exceptions should be documented.

`may_error_e` is an enumeration of the internal failure that MAYLIB may throws:

- MAY_MEMORY_ERR: No more memory free and the heap can't be extended.
- MAY_CANT_BE_CONVERTED_ERR: Can't convert an expression to another form.
- MAY_DIMENSION_ERR: Can't sums different expressions of different size (like list or matrix)
- MAY_SINGULAR_MATRIX_ERR: Can't inverse a matrix.
- MAY_VALUATION_NOT_POS_ERR: The valuation of the series is not strictly positive.

`void may_error_catch` (*void* (`*handler`)(*may_error_e* `error`, *const char* `*error_desc`, *const void* `*data`), *const void* `*data`) [Function]
Register the value of some global variables (including the previous error handler), the function *handler* and its associated *data* as the new error handler. The handler is a C function which doesn't return in normal usage but may throw exceptions using `longjmp` (C code) or `throw` (C++ code). It is called by `may_error_throw` after it restores some global variables.

The first parameter *error* of the error handler is the error number (See `may_error_e`) which is an enumeratation. The second parameter *error_desc* is a string containing additional details about the error, or NULL. The last parameter *data* is the parameter *data* given to the function `may_error_catch`: it may be used to give to the error handler additional information about how to do its job (like a `jmpbuf` buffer).

`void may_error_uncatch` (*void*) [Function]
Unregister the last saved error handler (See `may_error_catch`) and restore the previous error handler. The other global variables are not restored to their original values.

`void may_error_get (`*may_error_e *`**error_ptr**`, const char **error_desc_ptr*`)` [Function]
Save in **error_ptr* the value of the last reported error if *error_ptr* is not NULL.

Save in **error_desc_ptr* the value of the last reported error description if *error_ptr_desc* is not NULL.

`void may_error_throw (`*may_error_e* **`error`**`, const char *`**`error_desc`**`)` [Function]
Restore the global variables then throw the error *error* and the error string *error_desc* (which may be NULL) using the last registered error handler (See `may_error_catch`), just before unregistering it. *error_desc* must be a pointer to a global area (not a pointer to a buffer allocated in the stack since it is destroyed during the throw).

`const char * may_error_what (`*may_error_e* **`error`**`)` [Function]
Return a string explaining in human language what is the error associated to *error*

Example in C:

```
{
  jmp_buf buffer;
  int error_code = setjmp(_buffer);
  if (error_code == 0) {
     may_error_catch (error_setjmp_handler, &buffer);
     /* Do some computing */
     may_error_uncatch ();
  } else {
     fprintf (stderr, "ERROR reported: %s\", may_error_what (error_code));
  }
}

void
error_setjmp_handler (may_error_e e, const char *desc, const void *data)
{
  longjmp ( *(jmp_buf*)data, e);
}
```

Example in C++:

```
{
  try {
    may_error_catch (error_throw_handler, NULL);
     /* Do some computing */
    may_error_uncatch ();
  } catch (may_error_e error_code) {
    std::cerr << "ERROR reported: " << may_error_what (error_code) << std::endl;
  }
}

extern "C" void
error_throw_handler (may_error_e e, const char *desc, const void *data)
{
   throw e;
}
```

5.18 Extensions

5.18.1 Introduction

You may extend MAYLIB with your own types but they have to meet the following requirements:

- The operators `+` and * are associative: `(A1+A2)+A3=A1+A2+A3`
- The operator `+` is commutative `A1+A2=A2+A1`
- The operator * is commutative outside the extension class: `A1+B1+A2=B1+A1+A2`
- If the operator * is not commutative, the extension must absorb the product by a numerical into its own extension class at evaluating time: `Num*A1-->A2`

An *extension* is a `may_t` variable which is an instance of an *extension class*.

You may overload the behaviour of the standard operators (`+`, *, `^`, eval) for the extension class. Note that `A-B` is always parsed as `A+(-1)*B` and `A/B` as `A*B^(-1)`.

An extension is a node whose type is the extention class and whose children are `may_t` variables. You can't create an extension with no child. If you want to add some data which are not `may_t` variables (counter, values, ...), you have to use the *user data* functions to create a `may_t` variable which encapsulates all theses variables: when you design the format of your type, you have to separate between the `may_t` variables and the other variables. It is needed so that the MAYLIB garbage collector knows which pointers it has to update.

5.18.2 Interface

Once you have decided of the extension format, you have to register its class so that MAYLIB knows what to do with it:

`may_ext_t may_ext_register (`*const may_extdef_t* `*class`, `*may_extreg_t* `way)` [Function]

Register if (*way* = MAY_EXT_INSTALL) or update (if *way* = MAY_EXT_UPDATE) an extension class *class* which defines what to do with the extensions. If it fails (no more room to register an extension, or the extension to update is not found), it returns 0. Otherwise it returns the *Extension Identifying Number* of type `may_ext_t`: this number identifies the registered extension class inside MAYLIB in a unique way. The class is a structure containing all the needed information to handle the extensions. The structure *class* of type `may_extdef_t` must remains permanent since MAYLIB only keeps a pointer on it. The following fields of this structure are mandatory and must be defined:

- `name` : a pointer to the permanent string which defines the extension name. It is used as the name of the constructor method. It must be unique accross all extension class.
- `priority` : an integer between 1 and 1000. It defines its priority accross all extension class. The priority defines the order of the calls to the overloaded operators if there is more than one extension class as an argument to the operator: it calls the overloaded function which corresponds to the extension class of upper priority if doing inter-class computing is not needed (for example in `A^B`) or call the overloaded functions in the same order than their priority otherwise. For example, in the expression `2+A1+B1+A2+B2` if the priority of A is lower than B, the overloaded function of A is called before with `2+A1+A2` as an argument. Its returned value is then given as input to the overload function of B. The case whose two or more extension class have the same priority is undefined. 1 is the lowest priority, and 1000 the highest.

All other fields are optional and are callback functions. But you should set at least one, otherwise the extension does nothing. The following callbacks may be defined:

- int zero_p (may_t *extension*);

This function is called to determine if *extension* is the absorbing element of the field of the extension (ie. for all element of the fields of the extension class, `extension*element = element*extension = extension`). It returns TRUE in this case, or FALSE otherwise.

- int nonzero_p (may_t *extension*);

  This function is called to determine if *extension* is not the absorbing element of the field of the extension. It returns TRUE in this case, or FALSE otherwise.

- int one_p (may_t *extension*);

  This function is called to determine if *extension* is the unitary element of the field of the extension (ie. for all element of the fields of the extension class, `extension*element = element*extension = element`). It returns TRUE in this case, or FALSE otherwise.

- may_t eval (may_t *extension*);

  This function has to handle the pre-evaluation step: transformation of the expression into its canonical form and evaluationg of the children. *extension* is an element of the extension class.

- unsigned long add (unsigned long *start*, unsigned long *end*, may_pair_t **tab*);

  This function has to add the extensions by updating the given array: *tab* is an array of *end* elements of type `may_pair_t` which are: from 0 to *start-1*, internal types (non-extensions) or extensions of lower priority, and from *start* to *end-1*, extensions of the registered class. The function should update the array by doing as much as it can by summing the terms and returns the new number of terms in the array: this number must be lower or equal to *end* otherwise the behaviour is undefined. The array as input contains only evaluated elements. It must contain only evaluated elements as output too.

  Each `may_pair_t` element is a pair of `may_t` element: `.first` and `.second`. The extension is `.second` whereas `.first` may be an optionnal multiplier (which must be a numerical term). It is the product of both which is added to the sum.

  If the function does nothing, it returns *end*. If the function works only within the extension class, it sums all the extensions terms from *start* to *end-1*, store the new term in `tab[start]` and returns *start+1* as the new number of elements of the array.

  It is assumed that *start* < *end* otherwise the behaviour is undefined. If the function does not know what to do whith an extension of lower priority, it should do nothing and leave it in its place (maybe garbaging the array).

- unsigned long mul (unsigned long *start*, unsigned long *end*, may_t **tab*);

  Same function than the addition one but for multiplications.

  Each `may_pair_t` element is a pair of `may_t` element: `.first` and `.second`. The extension is `.second` whereas `.first` may be an optionnal exponent (which must be a numerical term). It is the power of both which is multiplied to the product.

- may_t pow (may_t *base*, may_t *power*);

  Compute `base^power`. One of *base* or *power* is of the extension class. The other is either of the extension class, of a lower class or an internal expression.

- void stringify (may_t extension, int level, void (*put_may)(may_t ext,int level), void (*put_string)(const char *string));

  Convert the extension `extension` into a string using the given callbacks: `put_may` is the callback to use to convert recursively the other child of the node. `put_string` is the callback to use to put a translated string into the internal buffer. `level` is the priority level of the current operator (0 for `+`, 1 for *, 2 for pow, 3 for func) if the priority of the extension is greater, some brackets might be needed to have a proper convertion (default: 3).

- may_t constructor (may_t *list*);

  If this callback is defined, `may_parse_str` translates the expressions which look like `NAME(a,b,c,d...)` with NAME is the name of the extension, into extensions: it calls this constructor with the list `{a,b,c,d...}` as argument. *list* is a list containing the elements given by the constructor. It shall return an extension of the same class. It shouldn't evaluate its argument. The `eval` callback will be used to do this job.
- unsigned int flags;

  A bit-field integer defining some properties of the extension class. The bit 0 defines if the if the extension class is commutative for the product (if set to '0') or not (if set to '0').
- may_t exp (may_t *extension*)
- may_t log (may_t *extension*)
- may_t cos (may_t *extension*)
- may_t sin (may_t *extension*)
- may_t tan (may_t *extension*)
- may_t cosh (may_t *extension*)
- may_t sinh (may_t *extension*)
- may_t tanh (may_t *extension*)
- may_t acos (may_t *extension*)
- may_t asin (may_t *extension*)
- may_t atan (may_t *extension*)
- may_t acosh (may_t *extension*)
- may_t asinh (may_t *extension*)
- may_t atanh (may_t *extension*)
- may_t floor (may_t *extension*)
- may_t sign (may_t *extension*)
- may_t gamma (may_t *extension*)
- may_t conj (may_t *extension*)
- may_t real (may_t *extension*)
- may_t imag (may_t *extension*)
- may_t argument (may_t *extension*)
- may_t abs (may_t *extension*)

  Assuming *extension* is a member of the extension class, these functions has to perform the evaluation of the mathematical functions of the given argument.

Here is an example of how to use this function (my_class is similar to the internal type list). We have only overloaded the function eval and add.

```
static may_ext_t my_class_ext;

static unsigned long
my_class_add (unsigned long start, unsigned long end, may_pair_t *tab)
{
  unsigned long i, j, size;
  may_t temp[end];
  may_t my;
```

```
  /* We assumed that all extensions have the same size */
  size = may_nops (tab[start].second);
  my = may_ext_c (my_class_ext, size);
  for (j = 0; j < start; j++)
    temp[j] = may_mul_c (tab[j].first, tab[j].second);
  for (i = 0; i < size ; i++) {
    for (j = start ; j < end;j++)
      temp[j] = may_mul_c (tab[j].first, may_op (tab[j].second, i));
    may_ext_set_c (my, i, may_eval (may_add_vc (end, temp)));
  }
  tab[0].first  = may_set_ui (1);
  tab[0].second = may_eval (my);
  return 1;
}

static const may_extdef_t my_class = {
  .name = "MY_CLASS",
  .priority = 66,
  .eval = my_class_eval,
  .add = my_class_add
};

{
  my_class_ext = may_ext_register (&my_class, MAY_EXT_INSTALL);
}
```

Note that in this example we don't define all the operators, but only the overloaded ones.

`int may_ext_unregister` (*const char* ***name***) [Function]
Unregister the extension of name is *name*. Return true if success or false if the extension is not found or cannot be unregistered.

`may_ext_t may_ext_find` (*const char* ***name***) [Function]
Search for the extension whose name is *name* and returns its Extension Identifying Number or 0 if not found.

`may_ext_t may_ext_p` (*may_t* **x**) [Function]
Return the Extension Identifying Number linked to *x* or 0 if *x* is not an extension.

`const may_extdef_t * may_ext_get` (*may_ext_t* **class**) [Function]
Return the extension class corresponding to the given Extension Identifying Number or NULL if *class* is not a registered Extension Identifying Number.

`may_t may_ext_c` (*may_ext_t* **class**, *size_t* **size**) [Function]
Return an unevaluated extension of class *class* with *size* children which have to be set using *may_ext_set_c*.

`void may_ext_set_c` (*may_t* **x**, *size_t* **i**, *may_t* **value**) [Function]
Set the child *i* of *x* which is an extension to *value*. *i* must be between 0 and the size of *x* minus one.

5.18.3 User Data

A *user data* type is an unknown type from MAYLIB point of view. The only thing MAYLIB knowns about it is its size, and the fact that it doesn't contain any pointer to any *symbolic number* or more generally any pointer computed by `may_alloc`, so that it can garbage it by moving its memory location, without updating the pointers and it can compute a hash by reading the memory. By itself, it has no meaning since it doesn't contain any type information. It is mainly used as a child node to store information about the node since the node can't store any information.

Note that `mpz_t`, `mpq_t` and `mpfr_t` variables use the GMP memory allocator, which is by default `may_alloc`. It means you can't use such variables too.

`may_t may_data_c` (*size_t* ***size***) [Function]
Create a new unevaluated user data of size *size* bytes.

`size_t may_data_size` (*may_t* ***x***) [Function]
Return the size of the user data *x*. If *x* is not a user data, the behavior of this function is undefined.

`void * may_data_ptr` (*may_t* ***x***) [Function]
Return the pointer to the user data area. The returned pointer is suitable to store a `void*`. The user data must not be in the evaluated form, otherwise the behavior of this function is undefined. You may fill this area until it is evaluated. If *x* is not a user data, the behavior of this function is undefined.

`const void * may_data_srcptr` (*may_t* ***x***) [Function]
Return the pointer to the user data area. The returned pointer is suitable to store a `void*`. If *x* is not a user data, the behavior of this function is undefined. It may be or not in an evaluated form.

6 Contributors

MAYLIB has been developped by Patrick Pélissier.

Concept Index

Function and Type Index