Users' manual for the `Sollya` tool

Release 3.0

Sylvain Chevillard
sylvain.chevillard@ens-lyon.org

Christoph Lauter
christoph.lauter@ens-lyon.org

Mioara Joldeş
mioara.joldes@ens-lyon.fr

April 11, 2012

License

The `Sollya` tool is Copyright © 2006-2011 by

Laboratoire de l'Informatique du Parallélisme - UMR CNRS - ENS Lyon - UCB Lyon 1 - INRIA 5668, Lyon, France,

LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), Nancy, France,

Laboratoire d'Informatique de Paris 6, Équipe PEQUAN, UPMC Université Paris 06 - CNRS - UMR 7606 - LIP6, Paris, France,

and by

INRIA Sophia-Antipolis Méditerranée, APICS Team, Sophia-Antipolis, France.

All rights reserved.

The `Sollya` tool is open software. It is distributed and can be used, modified and redistributed under the terms of the CeCILL-C licence available at `http://www.cecill.info/` and reproduced in the `COPYING` file of the distribution. The distribution contains parts of other libraries as a support for but not integral part of `Sollya`. These libraries are reigned by the GNU Lesser General Public License that is available at `http://www.gnu.org/licenses/` and reproduced in the `COPYING` file of the distribution.

This software (`Sollya`) is distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Contents

1 Compilation and installation of Sollya

Sollya comes in two flavors:

- Either as an interactive tool. This is achieved by running the Sollya executable file.
- Or as a C library that provides all the features of the tool within the C programming language.

The installation of the tool and the library follow the same steps, desribed below. The present documentation focuses more on the interactive tool. As a matter of fact, the library works exactly the same way as the tool, so it is necessary to know a little about the tool in order to correctly use the library. The reader who is only interested in the library should at least read the following Sections 2, 3 and 5. A documentation specifically describing the library usage is available in Appendix 10 at the end of the present documentation.

1.1 Compilation dependencies

The Sollya distribution can be compiled and installed using the usual ./configure, make, make install procedure. Besides a C and a C++ compiler, Sollya needs the following software libraries and tools to be installed. The ./configure script checks for the installation of the libraries. However Sollya will build without error if some of its external tools are not installed. In this case an error will be displayed at runtime.

- GMP
- MPFR
- MPFI
- fplll
- libxml2
- gnuplot (external tool)

The use of the external tool rlwrap is highly recommended but not required to use the Sollya interactive tool. Use the -A option of rlwrap for correctly displayed ANSI X3.64/ ISO/IEC 6429 colored prompts (see below).

1.2 Sollya command line options

Sollya can read input on standard input or in a file whose name is given as an argument when Sollya is invoked. The tool will always produce its output on standard output, unless specificly instructed by a particular Sollya command that writes to a file. The following lines are valid invocations of Sollya, assuming that bash is used as a shell:

```
~/% sollya
...
~/% sollya myfile.sollya
...
~/% sollya < myfile.sollya
```

If a file given as an input does not exist, an error message is displayed.

All configurations of the internal state of the tool are done by commands given on the `Sollya` prompt or in `Sollya` scripts. Nevertheless, some command line options are supported; they work at a very basic I/O-level and can therefore not be implemented as commands.

The following options are supported when calling `Sollya`:

- `--donotmodifystacksize`: When invoked, `Sollya` trys to increase the stack size that is available to a user process to the maximum size supported by the kernel. On some systems, the correspondent `ioctl` does not work properly. Use the option to prevent `Sollya` from changing the stack size.
- `--flush`: When this option is given, `Sollya` will flush all its input and output buffers after parsing and executing each command resp. sequence of commands. This option is needed when pipes are used to communicate with `Sollya` from another program.
- `--help`: Prints help on the usage of the tool and quits.
- `--nocolor`: `Sollya` supports coloring of the output using ANSI X3.64/ ISO/IEC 6429 escape sequences. Coloring is deactivated when `Sollya` is connected on standard input to a file that is not a terminal. This option forces the deactivation of ANSI coloring. This might be necessary on very old grey-scale terminals or when encountering problems with old versions of `rlwrap`.
- `--noprompt`: `Sollya` prints a prompt symbol when connected on standard input to a pseudo-file that is a terminal. The option deactivates the prompt.
- `--oldautoprint`: The behaviour of an undocumented feature for displaying values has changed in `Sollya` from version 1.1 to version 2.0. The old feature is deprecated. If you wish to use it nevertheless, use this deprecated option.
- `--oldrlwrapcompatible`: This option is deprecated. It makes `Sollya` emit a non ANSI X3.64 compliant coloring escape sequence for making it compatible with versions of `rlwrap` that do not support the `-A` option. The option is considered a hack since it is known to garble the output of the tool under some particular circumstances.
- `--warninfile[append] <file>`: Normally, `Sollya` emits warning and information messages together with all other displayed information on either standard output or standard error. This option allows all warning and information messages to get redirected to a file. The filename to be used must be given after the option. When `--warninfile` is used, the existing content (if any) of the file is first removed before writing to the file. With `--warninfileappend`, the messages are appended to an existing file. Even if coloring is used for the displaying all other `Sollya` output, no coloring sequences are ever written to the file. Let us emphasize on the fact that any file of a unixoid system can be used for output, for instance also a named pipe. This allows for error messaging to be performed on a separate terminal. The use of this option is mutually exclusive with the `--warnonstderr` option.
- `--warnonstderr`: Normally, `Sollya` prints warning and information messages on standard output, using a warning color when coloring is activated. When this option is given, `Sollya` will output all warning and information messages on standard error. Coloring will be used even on standard error, when activated. The use of this option is mutually exclusive with the `--warninfile[append]` option.

2 Introduction

`Sollya` is an interactive tool for handling numerical functions and working with arbitrary precision. It can evaluate functions accurately, compute polynomial approximations of functions, automatically implement polynomials for use in math libraries, plot functions, compute infinity norms, etc. `Sollya` is also a full-featured script programming language with support for procedures etc.

Let us begin this manual with an example. `Sollya` does not allow command line edition; since this may quickly become uncomfortable, we highly suggest to use the `rlwrap` tool with `Sollya`:

```
~/% rlwrap -A sollya
>
```

`Sollya` manipulates only functions in one variable. The first time that an unbound variable is used, this name is fixed. It will be used to refer to the free variable. For instance, try

```
> f = sin(x)/x;
> g = cos(y)-1;
Warning: the identifier "y" is neither assigned to, nor bound to a library funct
ion nor external procedure, nor equal to the current free variable.
Will interpret "y" as "x".
> g;
cos(x) - 1
```

Now, the name x can only be used to refer to the free variable:

```
> x = 3;
Warning: the identifier "x" is already bound to the free variable, to a library
function, library constant or to an external procedure.
The command will have no effect.
Warning: the last assignment will have no effect.
```

If you really want to unbind x, you can use the `rename` command and change the name of the free variable:

```
> rename(x,y);
Information: the free variable has been renamed from "x" to "y".
> g;
cos(y) - 1
> x=3;
> x;
3
```

`Sollya` has a reserved keyword that can always be used to refer to the free variable. This keyword is `_x_`. This is particularly useful in contexts when the name of the variable is not known: typically when refering to the free variable in a pattern matching or inside a procedure.

```
> f == sin(_x_)/_x_;
true
```

As you have seen, you can name functions and easily work with them. The basic thing to do with a function is to evaluate it at some point:

```
> f(-2);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
0.45464871341284084769800993295587242135112748572394
> evaluate(f,-2);
0.45464871341284084769800993295587242135112748572394
```

The printed value is generally a faithful rounding of the exact value at the working precision (i.e. one of the two floating-point numbers enclosing the exact value). Internally `Sollya` represents numbers as floating-point numbers in arbitrary precision with radix 2: the fact that a faithful rounding is performed in binary does not imply much on the exactness of the digits displayed in decimal. The working precision is controlled by the global variable `prec`:

```
> prec;
165
> prec=200;
The precision has been set to 200 bits.
> prec;
200
> f(-2);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
0.45464871341284084769800993295587242135112748572394513418948 65
```

Sometimes a faithful rounding cannot easily be computed. In such a case, a value is printed that was obtained using floating-point approximations without control on the final accuracy:

```
> log2(5)/log2(17) - log(5)/log(17);
Warning: rounding may have happened.
If there is rounding, the displayed value is *NOT* guaranteed to be a faithful r
ounding of the true result.
0
```

The philosophy of `Sollya` is: *Whenever something is not exact, print a warning.* This explains the warnings in the previous examples. If the result can be shown to be exact, there is no warning:

```
> sin(0);
0
```

Let us finish this Section with a small complete example that shows a bit of what can be done with `Sollya`:

```
> restart;
The tool has been restarted.
> prec=50;
The precision has been set to 50 bits.
> f=cos(2*exp(x));
> d=[-1/8;1/8];
> p=remez(f,2,d);
> derivativeZeros = dirtyfindzeros(diff(p-f),d);
> derivativeZeros = inf(d).:derivativeZeros:.sup(d);
> maximum=0;
> for t in derivativeZeros do {
    r = evaluate(abs(p-f), t);
    if r > maximum then { maximum=r; argmaximum=t; };
  };
> print("The infinity norm of", p-f, "is", maximum, "and is reached at", argmaxi
mum);
The infinity norm of -0.416265572875373 + x * (-1.798067209218835 + x * (-3.8971
0727747639e-2)) - cos(2 * exp(x)) is 8.630659443624325e-4 and is reached at -5.8
01672331417684e-2
```

In this example, we define a function f, an interval d and we compute the best degree-2 polynomial approximation of f on d with respect to the infinity norm. In other words, $\max_{x \in d}\{|p(x) - f(x)|\}$ is minimal amongst polynomials with degree not greater than 2. Then, we compute the list of the zeros of the derivative of $p - f$ and add the bounds of d to this list. Finally, we evaluate $|p - f|$ for each point in the list and store the maximum and the point where it is reached. We conclude by printing the result in a formatted way.

Let us mention as a sidenote that you do not really need to use such a script for computing an infinity norm; as we will see, the command `dirtyinfnorm` does this for you.

3 General principles

The first purpose of `Sollya` is to help people using numerical functions and numerical algorithms in a safe way. It is first designed to be used interactively but it can also be used in scripts[1].

One of the particularities of `Sollya` is to work with multi-precision arithmetic (it uses the `MPFR` library). For safety purposes, `Sollya` knows how to use interval arithmetic. It uses interval arithmetic to produce tight and safe results with the precision required by the user.

The general philosophy of `Sollya` is: *When you can perform a computation exactly and sufficiently quickly, do it; when you cannot, do not, unless you have been explicitly asked for.*

The precision of the tool is set by the global variable `prec`. In general, the variable `prec` determines the precision of the outputs of commands: more precisely, the command will internally determine how much precision should be used during the computations in order to ensure that the output is a faithfully rounded result with `prec` bits.

For decidability and efficiency reasons, this general principle cannot be applied every time, so be careful. Moreover certain commands are known to be unsafe: they give in general excellent results and give almost `prec` correct bits in output for everyday examples. However they are merely based on heuristics and should not be used when the result must be safe. See the documentation of each command to know precisely how confident you can be with their result.

A second principle (that comes together with the first one) is the following one: *When a computation leads to inexact results, inform the user with a warning.* This can be quite irritating in some circumstances: in particular if you are using `Sollya` within other scripts. The global variable `verbosity` lets you change the level of verbosity of `Sollya`. When the variable is set to 0, `Sollya` becomes completely silent on standard output and prints only very important messages on standard error. Increase `verbosity` if you want more information about what `Sollya` is doing. Please keep in mind that when you affect a value to a global variable, a message is always printed even if `verbosity` is set to 0. In order to silently affect a global variable, use `!`:

```
> prec=30;
The precision has been set to 30 bits.
> prec=30!;
>
```

For conviviality reasons, values are displayed in decimal by default. This lets a normal human being understand the numbers they manipulate. But since constants are internally represented in binary, this causes permanent conversions that are sources of roundings. Thus you are loosing in accuracy and `Sollya` is always complaining about inexact results. If you just want to store or communicate your results (to another tools for instance) you can use bit-exact representations available in `Sollya`. The global variable `display` defines the way constants are displayed. Here is an example of the five available modes:

[1] Remark: some of the behaviors of `Sollya` slightly change when it is used in scripts. For example, no prompt is printed.

```
> prec=30!;
> a = 17.25;
> display=decimal;
Display mode is decimal numbers.
> a;
1.725e1
> display=binary;
Display mode is binary numbers.
> a;
1.000101_2 * 2^(4)
> display=powers;
Display mode is dyadic numbers in integer-power-of-2 notation.
> a;
69 * 2^(-2)
> display=dyadic;
Display mode is dyadic numbers.
> a;
69b-2
> display=hexadecimal;
Display mode is hexadecimal numbers.
> a;
0x1.14p4
```

Please keep in mind that it is possible to maintain the general verbosity level at some higher setting while deactivating all warnings on roundings. This feature is controlled using the `roundingwarnings` global variable. It may be set to `on` or `off`. By default, the warnings are activated (`roundingwarnings = on`) when `Sollya` is connected on standard input to a pseudo-file that represents a terminal. They are deactivated when `Sollya` is connected on standard input to a real file. See 8.154 for further details; the behavior is illustrated with examples there.

As always, the symbol `e` means $\times 10^{\square}$. The same way the symbol `b` means $\times 2^{\square}$. The symbol `p` means $\times 16^{\square}$ and is used only with the `0x` prefix. The prefix `0x` indicates that the digits of the following number until a symbol `p` or white-space are hexadecimal. The suffix `_2` indicates to `Sollya` that the previous number has been written in binary. `Sollya` can parse these notations even if you are not in the corresponding `display` mode, so you can always use them.

You can also use memory-dump hexadecimal notation frequently used to represent IEEE 754 `double` and `single` precision numbers. Since this notation does not allow for exactly representing numbers with arbitrary precision, there is no corresponding `display` mode. However, the commands `printdouble` respectively `printsingle` round the value to the nearest `double` respectively `single`. The number is then printed in hexadecimal as the integer number corresponding to the memory representation of the IEEE 754 `double` or `single` number:

```
> printdouble(a);
0x4031400000000000
> printsingle(a);
0x418a0000
```

`Sollya` can parse these memory-dump hexadecimal notation back in any `display` mode. The difference of this memory-dump notation with the hexadecimal notation (as defined above) is made by the presence or absence of a `p` indicator.

4 Variables

As already explained, `Sollya` can manipulate variate functional expressions in one variable. These expressions contain a unique free variable the name of which is fixed by its first usage in an expression

that is not a left-hand-side of an assignment. This global and unique free variable is a variable in the mathematical sense of the term.

Sollya also provides variables in the sense programming languages give to the term. These variables, which must be different in their name from the global free variable, may be global or declared and attached to a block of statements, i.e. a begin-end-block. These programming language variables may hold any object of the Sollya language, as for example functional expressions, strings, intervals, constant values, procedures, external functions and procedures, etc.

Global variables need not to be declared. They start existing, i.e. can be correctly used in expressions that are not left-hand-sides of assignments, when they are assigned a value in an assignment. Since they are global, this kind of variables is recommended only for small Sollya scripts. Larger scripts with code reuse should use declared variables in order to avoid name clashes for example in loop variables.

Declared variables are attached to a begin-end-block. The block structure builds scopes for declared variables. Declared variables in inner scopes shadow (global and declared) variables of outer scopes. The global free variable, i.e. the mathematical variable for variate functional expressions in one variable, cannot be shadowed. Variables are declared using the var keyword. See section 8.182 for details on its usage and semantic.

The following code examples illustrate the use of variables.

```
> f = exp(x);
> f;
exp(x)
> a = "Hello world";
> a;
Hello world
> b = 5;
> f(b);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
1.484131591025766034211155800405522796234876675938 78e2
> {var b; b = 4; f(b); };
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
5.459815003314423907811026120286087840279073703861 37e1
> {var x; x = 3; };
Warning: the identifier "x" is already bound to the current free variable.
It cannot be declared as a local variable. The declaration of "x" will have no e
ffect.
Warning: the identifier "x" is already bound to the free variable, to a library
function, library constant or to an external procedure.
The command will have no effect.
Warning: the last assignment will have no effect.
> {var a, b; a=5; b=3; {var a; var b; b = true; a = 1; a; b;}; a; b; };
1
true
5
3
> a;
Hello world
```

Let us state that a variable identifier, just as every identifier in Sollya, contains at least one character, starts with a ASCII letter and continues with ASCII letters or numerical digits.

5 Data types

Sollya has a (very) basic system of types. If you try to perform an illicit operation (such as adding a number and a string, for instance), you will get a typing error. Let us see the available data types.

5.1 Booleans

There are two special values `true` and `false`. Boolean expressions can be constructed using the boolean connectors `&&` (and), `||` (or), `!` (not), and comparisons.

The comparison operators `<`, `<=`, `>` and `>=` can only be used between two numbers or constant expressions.

The comparison operators `==` and `!=` are polymorphic. You can use them to compare any two objects, like two strings, two intervals, etc. As a matter of fact, polymorphism is allowed on both sides: it is possible to compare objects of different type. Such objects of different type, as they can never be syntactically equal, will always compare unequal (see exception for `error`, section 8.52) and never equal. It is important to remember that testing the equality between two functions will return `true` if and only if the expression trees representing the two functions are exactly the same. See 8.52 for an exception concerning the special object `error`. Example:

```
> 1+x==1+x;
true
```

5.2 Numbers

`Sollya` represents numbers as binary multi-precision floating-point values. For integer values and values in dyadic, binary, hexadecimal or memory dump notation, it automatically uses a precision needed for representing the value exactly (unless this behaviour is overridden using the syntax given below). Additionally, automatic precision adaption takes place for all integer values (even in decimal notation) written without the exponent sign `e` or with the exponent sign `e` and an exponent sufficiently small that they are less than 10^{999}. Otherwise the values are represented with the current precision `prec`. When a number must be rounded, it is rounded to the precision `prec` before the expression get evaluated:

```
> prec=12!;
> 4097.1;
Warning: Rounding occurred when converting the constant "4097.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
4098
> 4098.1;
Warning: Rounding occurred when converting the constant "4098.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
4098
> 4097.1+1;
Warning: Rounding occurred when converting the constant "4097.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
4099
```

As a matter of fact, each variable has its own precision that corresponds to its intrinsic precision or, if it cannot be represented, to the value of `prec` when the variable was set. Thus you can work with variables having a precision higher than the current precision.

The same way, if you define a function that refers to some constant, this constant is stored in the function with the current precision and will keep this value in the future, even if `prec` becomes smaller.

If you define a function that refers to some variable, the precision of the variable is kept, independently of the current precision:

```
> prec = 50!;
> a = 4097.1;
Warning: Rounding occurred when converting the constant "4097.1" to floating-poi
nt with 50 bits.
If safe computation is needed, try to increase the precision.
> prec = 12!;
> f = x + a;
> g = x + 4097.1;
Warning: Rounding occurred when converting the constant "4097.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
> prec = 120;
The precision has been set to 120 bits.
> f;
4.097099999999985448084771633148193 35e3 + x
> g;
4098 + x
```

In some rare cases, it is necessary to read in decimal constants with a particular precision being used in the conversion to the binary floating-point format, which `Sollya` uses. Setting `prec` to that precision may prove to be an insufficient means for doing so, for example when several different precisions have to be used in one expression. For these rare cases, `Sollya` provides the following syntax: decimal constants may be written `%`*precision*`%`*constant*, where *precision* is a constant integer, written in decimal, and *constant* is the decimal constant. `Sollya` will convert the constant *constant* with precision *precision*, regardless of the global variable `prec` and regardless if *constant* is an integer or would otherwise be representable.

```
> prec = 24;
The precision has been set to 24 bits.
> a = 0.1;
Warning: Rounding occurred when converting the constant "0.1" to floating-point
with 24 bits.
If safe computation is needed, try to increase the precision.
> b = 33554433;
> prec = 64;
The precision has been set to 64 bits.
> display = binary;
Display mode is binary numbers.
> a;
1.10011001100110011001101_2 * 2^(-4)
> 0.1;
Warning: Rounding occurred when converting the constant "0.1" to floating-point
with 64 bits.
If safe computation is needed, try to increase the precision.
1.1001100110011001100110011001100110011001100110011001100110011001101_2 * 2^(-4)
> %24%0.1;
Warning: Rounding occurred when converting the constant "0.1" to floating-point
with 24 bits.
If safe computation is needed, try to increase the precision.
1.10011001100110011001101_2 * 2^(-4)
> c = 33554433;
> b;
1.00000000000000000000000_2 * 2^(25)
> c;
1.00000000000000000000001_2 * 2^(25)
> %24%33554433;
Warning: Rounding occurred when converting the constant "33554433" to floating-p
oint with 24 bits.
If safe computation is needed, try to increase the precision.
1_2 * 2^(25)
>
>
```

`Sollya` is an environment that uses floating-point arithmetic. The IEEE 754-2008 standard on floating-point arithmetic does not only define floating-point numbers that represent real numbers but also floating-point data representing infinities and Not-a-Numbers (NaNs). `Sollya` also supports infinities and NaNs in the spirit of the IEEE 754-2008 standard without taking the standard's choices literally.

- Signed infinities are available through the `Sollya` objects `infty, -infty, @Inf@` and `-@Inf@`.
- Not-a-Numbers are supported through the `Sollya` objects `NaN` and `@NaN@`. `Sollya` does not have support for NaN payloads, signaling or quiet NaNs or signs of NaNs. Signaling NaNs are supported on input for single and double precision memory notation (see section 3). However, they immediately get converted to plain `Sollya` NaNs.

The evaluation of an expression involving a NaN or the evaluation of a function at a point being NaN always results in a NaN.

Infinities are considered to be the limits of expressions tending to infinity. They are supported as bounds of intervals in some cases. However, particular commands might prohibit their use even though there might be a mathematical meaning attached to such expressions. For example, while `Sollya` will evaluate expressions such as $\lim_{x \to -\infty} e^x$, expressed e.g. through `evaluate(exp(x),[-infty;0])`, it will

not accept to compute the (finite) value of

$$\int_{-\infty}^{0} e^x \, \mathrm{d}x.$$

The following examples give an idea of what can be done with `Sollya` infinities and NaNs. Here is what can be done with infinities:

```
> f = exp(x) + 5;
> f(-infty);
5
> evaluate(f,[-infty;infty]);
[5;@Inf@]
> f(infty);
Warning: the given expression is undefined or numerically unstable.
@NaN@
> [-infty;5] * [3;4];
[-@Inf@;20]
> -infty < 5;
true
> log(0);
Warning: the given expression is undefined or numerically unstable.
@NaN@
> [log(0);17];
Warning: the given expression is not a constant but an expression to evaluate
and a faithful evaluation is not possible.
Will use a plain floating-point evaluation, which might yield a completely wrong
 value.
Warning: inclusion property is satisfied but the diameter may be greater than th
e least possible.
[-@Inf@;17]
>
```

And the following example illustrates NaN behavior.

```
> 3/0;
Warning: the given expression is undefined or numerically unstable.
@NaN@
> (-3)/0;
Warning: the given expression is undefined or numerically unstable.
@NaN@
> infty/infty;
Warning: the given expression is undefined or numerically unstable.
@NaN@
> infty + infty;
Warning: the given expression is undefined or numerically unstable.
@Inf@
> infty - infty;
Warning: the given expression is undefined or numerically unstable.
@NaN@
> f = exp(x) + 5;
> f(NaN);
@NaN@
> NaN == 5;
false
> NaN == NaN;
false
> NaN != NaN;
false
> X = "Vive la Republique!";
> !(X == X);
false
> X = 5;
> !(X == X);
false
> X = NaN;
> !(X == X);
true
>
```

5.3 Rational numbers and rational arithmetic

The `Sollya` tool is mainly based on floating-point arithmetic: wherever possible, floating-point algorithms, including algorithms using interval arithmetic, are used to produce approximate but safe results. For some particular cases, floating-point arithmetic is not sufficient: some algorithms just require natural and rational numbers to be handled exactly. More importantly, for these applications, it is required that rational numbers be displayed as such.

`Sollya` implements a particular mode that offers a lightweight support for rational arithmetic. When needed, it can be enabled by assigning `on` to the global variable `rationalmode`. It is disabled by assigning `off`; the default is `off`.

When the mode for rational arithmetic is enabled, `Sollya`'s behavior will change as follows:

- When a constant expression is given at the `Sollya` prompt, `Sollya` will first try to simplify the expression to a rational number. If such an evaluation to a rational number is possible, `Sollya` will display that number as an integer or a fraction of two integers. Only if `Sollya` is not able to simplify the constant expression to a rational number, it will launch the default behavior of evaluating constant expressions to floating-point numbers that are generally faithful roundings of the expressions.

- When the global mode `autosimplify` is `on`, which is the default, `Sollya` will additionally use rational arithmetic while trying to simplify expressions given in argument of commands.

Even when rationalmode is on, Sollya will not be able to exhibit integer ratios between transcendental quantities. For example, Sollya will not display $\frac{1}{6}$ for $\arcsin\left(\frac{1}{2}\right)/\pi$ but $0.16666\ldots$. Sollya's evaluator for rational arithmetic is only able to simplify rational expressions based on addition, subtraction, multiplication, division, negation, perfect squares (for square root) and integer powers.

The following example illustrates what can and what cannot be done with Sollya's mode for rational arithmetic:

```
> 1/3 - 1/7;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
0.190476190476190476190476190476190476190476190476190476190476190476190476190476
> rationalmode = on;
Rational mode has been activated.
> 1/3 - 1/7;
4 / 21
> (2 + 1/7)^2 + (6/7)^2 + 2 * (2 + 1/7) * 6/7;
9
> rationalmode = off;
Rational mode has been deactivated.
> (2 + 1/7)^2 + (6/7)^2 + 2 * (2 + 1/7) * 6/7;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
9
> rationalmode = on;
Rational mode has been activated.
> asin(1)/pi;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
0.5
> sin(1/6 * pi);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
0.5
> exp(1/7 - 3/21) / 7;
1 / 7
> rationalmode = off;
Rational mode has been deactivated.
> exp(1/7 - 3/21) / 7;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
0.142857142857142857142857142857142857142857145
> print(1/7 - 3/21);
1 / 7 - 3 / 21
> rationalmode = on;
Rational mode has been activated.
> print(1/7 - 3/21);
0
```

5.4 Intervals and interval arithmetic

Sollya can manipulate intervals that are closed subsets of the real numbers. Several ways of defining intervals exist in Sollya. There is the most common way where intervals are composed of two numbers or constant expressions representing the lower and the upper bound. These values are separated either by commas or semi-colons. Interval bound evaluation is performed in a way that ensures the inclusion property: all points in the original, unevaluated interval will be contained in the interval with its bounds evaluated to floating-point numbers.

```
> d=[1;2];
> d2=[1,1+1];
> d==d2;
true
> prec=12!;
> 8095.1;
Warning: Rounding occurred when converting the constant "8095.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
8096
> [8095.1; 8096.1];
Warning: Rounding occurred when converting the constant "8095.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
Warning: Rounding occurred when converting the constant "8096.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
[8094;8098]
```

`Sollya` has a mode for printing intervals that are that thin that their bounds have a number of decimal digits in common when printed. That mode is called `midpointmode`; see below for an introduction and section 8.102 for details. As `Sollya` must be able to parse back its own output, a syntax is provided to input intervals in midpoint mode. However, please pay attention to the fact that the notation used in midpoint mode generally increases the width of intervals: hence when an interval is displayed in midpoint mode and read again, the resulting interval may be wider than the original interval.

```
> midpointmode = on!;
> [1.725e4;1.75e4];
0.17~2/5~e5
> 0.17~2/5~e5;
0.17~2/5~e5
> midpointmode = off!;
> 0.17~2/5~e5;
[17200;17500]
```

In some cases, intervals become infinitely thin in theory, in which case one tends to think of point intervals even if their floating-point representation is not infinitely thin. `Sollya` provides a very covenient way for input of such point intervals. Instead of writing `[a;a]`, it is possible to just write `[a]`. `Sollya` will expand the notation while making sure that the inclusion property is satisfied:

```
> [3];
[3;3]
> [1/7];
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
[0.14285713;0.14285716]
> [exp(8)];
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
[2.980957e3;2.9809589e3]
```

When the mode `midpointmode` is set to `on` (see 8.102), `Sollya` will display intervals that are provably reduced to one point in this extended interval syntax. It will use `midpointmode` syntax for intervals that are sufficiently thin but not reduced to one point (see section 8.102 for details):

```
> midpointmode = off;
Midpoint mode has been deactivated.
> [17;17];
[17;17]
> [exp(pi);exp(pi)];
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
[2.31406926327792690057290863679485473802661062425987e1;2.3140692632779269005729
08636794854738026610624260l5e1]
> midpointmode = on;
Midpoint mode has been activated.
> [17;17];
[17]
> [exp(pi);exp(pi)];
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
0.23140692632779269005729086367948547380266106242~5/7~e2
>
```

`Sollya` intervals are internally represented with floating-point numbers as bounds; rational numbers are not supported here. If bounds are defined by constant expressions, these are evaluated to floating-point numbers using the current precision. Numbers or variables containing numbers keep their precision for the interval bounds.

Constant expressions get evaluated to floating-point values immediately; this includes π and rational numbers, even when `rationalmode` is `on` (see section 5.3 for this mode).

```
> prec = 300!;
> a = 4097.1;
Warning: Rounding occurred when converting the constant "4097.1" to floating-poi
nt with 300 bits.
If safe computation is needed, try to increase the precision.
> prec = 12!;
> d = [4097.1; a];
Warning: Rounding occurred when converting the constant "4097.1" to floating-poi
nt with 12 bits.
If safe computation is needed, try to increase the precision.
> prec = 300!;
> d;
[4096;4.0971e3]
> prec = 30!;
> [-pi;pi];
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
[-3.141592659;3.141592659]
```

You can get the upper-bound (respectively the lower-bound) of an interval with the command `sup` (respectively `inf`). The middle of the interval can be computed with the command `mid`. Let us also mention that these commands can also be used on numbers (in that case, the number is interpreted as an interval containing only one single point. In that case the commands `inf`, `mid` and `sup` are just the identity):

```
> d=[1;3];
> inf(d);
1
> mid(d);
2
> sup(4);
4
```

Let us mention that the `mid` operator never provokes a rounding. It is rewritten as an unevaluated expression in terms of `inf` and `sup`.

`Sollya` permits intervals to also have non-real bounds, such as infinities or NaNs. When evaluating certain expressions, in particular given as interval bounds, `Sollya` will itself generate intervals containing infinities or NaNs. When evaluation yields an interval with a NaN bound, the given expression is most likely undefined or numerically unstable. Such results should not be trusted; a warning is displayed.

While computations on intervals with bounds being NaN will always fail, `Sollya` will try to interpret infinities in the common way as limits. However, this is not guaranteed to work, even if it is guaranteed that no unsafe results will be produced. See also section 5.2 for more detail on infinities in `Sollya`. The behavior of interval arithmetic on intervals containing infinities or NaNs is subject to debate; moreover, there is no complete consensus on what should be the result of the evaluation of a function f over an interval I containing points where f is not defined. `Sollya` has its own philosophy regarding these questions. This philosophy is explained in Appendix 9 at the end of this document.

```
> evaluate(exp(x),[-infty;0]);
[0;1]
> dirtyinfnorm(exp(x),[-infty;0]);
Warning: a bound of the interval is infinite or NaN.
This command cannot handle such intervals.
@NaN@
>
> f = log(x);
> [f(0); f(1)];
Warning: the given expression is not a constant but an expression to evaluate
and a faithful evaluation is not possible.
Will use a plain floating-point evaluation, which might yield a completely wrong
 value.
Warning: inclusion property is satisfied but the diameter may be greater than th
e least possible.
[-@Inf@;0]
>
```

`Sollya` internally uses interval arithmetic extensively to provide safe answers. In order to provide for algorithms written in the `Sollya` language being able to use interval arithmetic, `Sollya` offers native support of interval arithmetic. Intervals can be added, subtracted, multiplied, divided, raised to powers, for short, given in argument to any `Sollya` function. The tool will apply the rules of interval arithmetic in order to compute output intervals that safely encompass the hull of the image of the function on the given interval:

```
> [1;2] + [3;4];
[4;6]
> [1;2] * [3;4];
[3;8]
> sqrt([9;25]);
[3;5]
> exp(sin([10;100]));
[0.36787942;2.7182819]
```

When such expressions involving intervals are given, Sollya will follow the rules of interval arithmetic in precision prec for immediately evaluating them to interval enclosures. While Sollya's evaluator always guarantees the inclusion property, it also applies some optimisations in some cases in order to make the image interval as thin as possible. For example, Sollya will use a Taylor expansion based evaluation if a composed function, call it f, is applied to an interval. In other words, in this case Sollya will behave as if the evaluate command (see section 8.53) were implicitly used. In most cases, the result will be different from the one obtained by replacing all occurences of the free variable of a function by the interval the function is to be evaluated on:

```
> f = x - sin(x);
> [-1b-10;1b-10] - sin([-1b-10;1b-10]);
[-1.95312484477957829894e-3;1.95312484477957829894e-3]
> f([-1b-10;1b-10]);
[-1.55220421701117626897e-10;1.55220421701117626897e-10]
> evaluate(f,[-1b-10;1b-10]);
[-1.55220421701117626897e-10;1.55220421701117626897e-10]
```

5.5 Functions

Sollya knows only about functions with one single variable. The first time in a session that an unbound name is used (without being assigned) it determines the name used to refer to the free variable.

The basic functions available in Sollya are the following:

- +, -, *, /, ^
- sqrt
- abs
- sin, cos, tan, sinh, cosh, tanh
- asin, acos, atan, asinh, acosh, atanh
- exp, expm1 (defined as $\mathrm{expm1}(x) = \exp(x) - 1$)
- log (natural logarithm), log2 (binary logarithm), log10 (decimal logarithm), log1p (defined as $\mathrm{log1p}(x) = \log(1 + x)$)
- erf, erfc
- halfprecision, single, double, doubleextended, doubledouble, quad, tripledouble (see sections 8.73, 8.162, 8.45, 8.47, 8.46, 8.136 and 8.180)
- HP, SG, D, DE, DD, QD, TD (see sections 8.73, 8.162, 8.45, 8.47, 8.46, 8.136 and 8.180)
- floor, ceil, nearestint.

The constant π is available through the keyword pi as a 0-ary function:

```
> display=binary!;
> prec=12!;
> a=pi;
> a;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
1.10010010001_2 * 2^(1)
> prec=30!;
> a;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
1.10010010000111111011010101001_2 * 2^(1)
```

The reader may wish to see Sections 8.91 and 8.69 for ways of dynamically adding other base functions to Sollya.

5.6 Strings

Anything written between quotes is interpreted as a string. The infix operator `@` concatenates two strings. To get the length of a string, use the `length` function. You can access the i-th character of a string using brackets (see the example below). There is no character type in `Sollya`: the i-th character of a string is returned as a string itself.

```
> s1 = "Hello "; s2 = "World!";
> s = s1@s2;
> length(s);
12
> s[0];
H
> s[11];
!
```

Strings may contain the following escape sequences: \\, \", \?, \', \n, \t, \a, \b, \f, \r, \v, \x[hexadecimal number] and \[octal number]. Refer to the C99 standard for their meaning.

5.7 Particular values

`Sollya` knows about some particular values. These values do not really have a type. They can be stored in variables and in lists. A (possibly not exhaustive) list of such values is the following one:

- `on`, `off` (see sections 8.113 and 8.112)
- `dyadic`, `powers`, `binary`, `decimal`, `hexadecimal` (see sections 8.48, 8.124, 8.18, 8.33 and 8.75)
- `file`, `postscript`, `postscriptfile` (see sections 8.62, 8.121 and 8.122)
- `RU`, `RD`, `RN`, `RZ` (see sections 8.155, 8.141, 8.150 and 8.156)
- `absolute`, `relative` (see sections 8.2 and 8.144)
- `floating`, `fixed` (see sections 8.65 and 8.64)
- `halfprecision`, `single`, `double`, `doubleextended`, `doubledouble`, `quad`, `tripledouble` (see sections 8.73, 8.162, 8.45, 8.47, 8.46, 8.136 and 8.180)
- `HP`, `SG`, `D`, `DE`, `DD`, `QD`, `TD` (see sections 8.73, 8.162, 8.45, 8.47, 8.46, 8.136 and 8.180)
- `perturb` (see section 8.116)
- `honorcoeffprec` (see section 8.76)
- `default` (see section 8.34)
- `error` (see section 8.52)
- `void` (see section 8.184)

5.8 Lists

Objects can be grouped into lists. A list can contain elements with different types. As for strings, you can concatenate two lists with `@`. The function `length` also gives the length of a list.

You can prepend an element to a list using `.:` and you can append an element to a list using `:.` The following example illustrates some features:

```
> L = [| "foo" |];
> L = L:.1;
> L = "bar".:L;
> L;
[|"bar", "foo", 1|]
> L[1];
foo
> L@L;
[|"bar", "foo", 1, "bar", "foo", 1|]
```

Lists can be considered arrays and elements of lists can be referenced using brackets. Possible indices start at 0. The following example illustrates this point:

```
> L = [|1,2,3,4,5|];
> L;
[|1, 2, 3, 4, 5|]
> L[3];
4
```

Please be aware of the fact that the complexity for accessing an element of the list using indices is $\mathcal{O}(n)$, where n is the length of the whole list.

Lists may contain ellipses indicated by `,...,` between elements that are constant and evaluate to integers that are incrementally ordered. `Sollya` translates such ellipses to the full list upon evaluation. The use of ellipses between elements that are not constants is not allowed. This feature is provided for ease of programming; remark that the complexity for expanding such lists is high. For illustration, see the following example:

```
> [|1,...,5|];
[|1, 2, 3, 4, 5|]
> [|-5,...,5|];
[|-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5|]
> [|3,...,1|];
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
> [|true,...,false|];
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
```

Lists may be continued to infinity by means of the `...` indicator after the last element given. At least one element must explicitly be given. If the last element given is a constant expression that evaluates to an integer, the list is considered as continued to infinity by all integers greater than that last element. If the last element is another object, the list is considered as continued to infinity by re-duplicating this last element. Let us remark that bracket notation is supported for such end-elliptic lists even for implicitly given elements. However, evaluation complexity is high. Combinations of ellipses inside a list and in its end are possible. The usage of lists described here is best illustrated by the following examples:

```
> L = [|1,2,true,3...|];
> L;
[|1, 2, true, 3...|]
> L[2];
true
> L[3];
3
> L[4];
4
> L[1200];
1200
> L = [|1,...,5,true...|];
> L;
[|1, 2, 3, 4, 5, true...|]
> L[1200];
true
```

5.9 Structures

In a similar way as in lists, `Sollya` allows data to be grouped in – untyped – structures. A structure forms an object to which other objects can be added as elements and identified by their names. The elements of a structure can be retrieved under their name and used as usual. The following sequence shows that point:

```
> s.a = 17;
> s.b = exp(x);
> s.a;
17
> s.b;
exp(x)
> s.b(1);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
2.71828182845904523536028747135266249775724709369998
> s.d.a = [-1;1];
> s.d.b = sin(x);
> inf(s.d.a);
-1
> diff(s.d.b);
cos(x)
```

Structures can also be defined literally using the syntax illustrated in the next example. They will also be printed in that syntax.

```
> a = { .f = exp(x), .dom = [-1;1] };
> a;
{ .f = exp(x), .dom = [-1;1] }
> a.f;
exp(x)
> a.dom;
[-1;1]
> b.f = sin(x);
> b.dom = [-1b-5;1b-5];
> b;
{ .dom = [-3.125e-2;3.125e-2], .f = sin(x) }
> { .f = asin(x), .dom = [-1;1] }.f(1);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
1.57079632679489661923132169163975144209858469968754
```

If the variable `a` is bound to an existing structure, it is possible to use the "dot notation" `a.b` to assign the value of the field `b` of the structure `a`. This works even if `b` is not yet a field of `a`: in this case a new field is created inside the structure `a`.

Besides, the dot notation can be used even when `a` is unassigned. In this case a new structure is created with a field `b`, and this structure is bound to `a`. However, the dot notation cannot be used if `a` is already bound to something that is not a structure.

These principles apply recursively: for instance, if `a` is a structure that contains only one field `d`, the command `a.b.c = 3` creates a new field named `b` inside the structure `a`; this field itself is a structure containing the field `c`. The command `a.d.c = 3` is allowed if `a.d` is already a structure, but forbidden otherwise (e.g. if `a.d` was equal to `sin(x)`). This is summed up in the following example.

```
> restart;
The tool has been restarted.
> a.f = exp(x);
> a.dom = [-1;1];
> a.info.text = "My akrnoximation problem";
> a;
{ .info = { .text = "My akrnoximation problem" }, .dom = [-1;1], .f = exp(x) }
>
> a.info.text = "My approximation problem";
> a;
{ .info = { .text = "My approximation problem" }, .dom = [-1;1], .f = exp(x) }
>
> b = exp(x);
> b.a = 5;
Warning: cannot modify an element of something that is not a structure.
Warning: the last assignment will have no effect.
> b;
exp(x)
>
> a.dom.a = -1;
Warning: cannot modify an element of something that is not a structure.
Warning: the last assignment will have no effect.
> a;
{ .info = { .text = "My approximation problem" }, .dom = [-1;1], .f = exp(x) }
```

When printed, the elements of a structure are not sorted in any manner. They get printed in an arbitrary order that just maintains the order given in the definition of literate structures. That said, when compared, two structures compare equal iff they contain the same number of identifiers, with the

same names and iff the elements of corresponding names all compare equal. This means the order does not matter in comparisons and otherwise does only for printing.

The following example illustrates this matter:

```
> a = { .f = exp(x), .a = -1, .b = 1 };
> a;
{ .f = exp(x), .a = -1, .b = 1 }
> a.info = "My function";
> a;
{ .info = "My function", .f = exp(x), .a = -1, .b = 1 }
>
> b = { .a = -1, .f = exp(x), .info = "My function", .b = 1 };
> b;
{ .a = -1, .f = exp(x), .info = "My function", .b = 1 }
>
> a == b;
true
>
> b.info = "My other function";
> a == b;
false
>
> b.info = "My function";
> a == b;
true
> b.something = true;
> a == b;
false
```

6 Iterative language elements: assignments, conditional statements and loops

6.1 Blocks

Statements in `Sollya` can be grouped in blocks, so-called begin-end-blocks. This can be done using the key tokens { and }. Blocks declared this way are considered to be one single statement. As already explained in section 4, using begin-end-blocks also opens the possibility of declaring variables through the keyword `var`.

6.2 Assignments

`Sollya` has two different assignment operators, `=` and `:=`. The assignment operator `=` assigns its right-hand-object "as is", i.e. without evaluating functional expressions. For instance, `i = i + 1;` will dereferentiate the identifier `i` with some content, notate it y, build up the expression (function) $y + 1$ and assign this expression back to `i`. In the example, if `i` stood for the value 1000, the statement `i = i + 1;` would assign "$1000 + 1$" – and not "1001" – to `i`. The assignment operator `:=` evaluates constant functional expressions before assigning them. On other expressions it behaves like `=`. Still in the example, the statement `i := i + 1;` really assigns 1001 to `i`.

Both `Sollya` assignment operators support indexing of lists or strings elements using brackets on the left-hand-side of the assignment operator. The indexed element of the list or string gets replaced by the right-hand-side of the assignment operator. When indexing strings this way, that right-hand side must evaluate to a string of length 1. End-elliptic lists are supported with their usual semantic for this kind of assignment. When referencing and assigning a value in the implicit part of the end-elliptic list, the list gets expanded to the corresponding length.

The following examples well illustrate the behavior of assignment statements:

```
> autosimplify = off;
Automatic pure tree simplification has been deactivated.
> i = 1000;
> i = i + 1;
> print(i);
1000 + 1
> i := i + 1;
> print(i);
1002
> L = [|1,...,5|];
> print(L);
[|1, 2, 3, 4, 5|]
> L[3] = L[3] + 1;
> L[4] := L[4] + 1;
> print(L);
[|1, 2, 3, 4 + 1, 6|]
> L[5] = true;
> L;
[|1, 2, 3, 5, 6, true|]
> s = "Hello world";
> s;
Hello world
> s[1] = "a";
> s;
Hallo world
> s[2] = "foo";
Warning: the string to be assigned is not of length 1.
This command will have no effect.
> L = [|true,1,...,5,9...|];
> L;
[|true, 1, 2, 3, 4, 5, 9...|]
> L[13] = "Hello";
> L;
[|true, 1, 2, 3, 4, 5, 9, 10, 11, 12, 13, 14, 15, "Hello"...|]
```

The indexing of lists on left-hand sides of assignments is reduced to the first order. Multiple indexing of lists of lists on assignment is not supported for complexity reasons. Multiple indexing is possible in right-hand sides.

```
> L = [| 1, 2, [|"a", "b", [|true, false|] |] |];
> L[2][2][1];
false
> L[2][2][1] = true;
Warning: the first element of the left-hand side is not an identifier.
This command will have no effect.
> L[2][2] = "c";
Warning: the first element of the left-hand side is not an identifier.
This command will have no effect.
> L[2] = 3;
> L;
[|1, 2, 3|]
```

6.3 Conditional statements

`Sollya` supports conditional statements expressed with the keywords `if`, `then` and optionally `else`. Let us mention that only conditional statements are supported and not conditional expressions.

The following examples illustrate both syntax and semantic of conditional statements in `Sollya`. Concerning syntax, be aware that there must not be any semicolon before the `else` keyword.

```
> a = 3;
> b = 4;
> if (a == b) then print("Hello world");
> b = 3;
> if (a == b) then print("Hello world");
Hello world
> if (a == b) then print("You are telling the truth") else print("Liar!");
You are telling the truth
```

6.4 Loops

`Sollya` supports three kinds of loops. General *while-condition* loops can be expressed using the keywords `while` and `do`. One has to be aware of the fact that the condition test is executed always before the loop, there is no *do-until-condition* loop. Consider the following examples for both syntax and semantic:

```
> verbosity = 0!;
> prec = 30!;
> i = 5;
> while (expm1(i) > 0) do { expm1(i); i := i - 1; };
1.474131591e2
5.359815e1
1.908553692e1
6.3890561
1.718281827
> print(i);
0
```

The second kind of loops are loops on a variable ranging from a numerical start value and a end value. These kind of loops can be expressed using the keywords `for`, `from`, `to`, `do` and optionally `by`. The `by` statement indicates the width of the steps on the variable from the start value to the end value. Once again, syntax and semantic are best explained with an example:

```
> for i from 1 to 5 do print ("Hello world",i);
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5
> for i from 2 to 1 by -0.5 do print("Hello world",i);
Hello world 2
Hello world 1.5
Hello world 1
```

The third kind of loops are loops on a variable ranging on values contained in a list. In order to ensure the termination of the loop, that list must not be end-elliptic. The loop is expressed using the keywords `for`, `in` and `do` as in the following examples:

```
> L = [|true, false, 1,...,4, "Hello", exp(x)|];
> for i in L do i;
true
false
1
2
3
4
Hello
exp(x)
```

For both types of `for` loops, assigning the loop variable is allowed and possible. When the loop terminates, the loop variable will contain the value that made the loop condition fail. Consider the following examples:

```
> for i from 1 to 5 do { if (i == 3) then i = 4 else i; };
1
2
5
> i;
6
```

7 Functional language elements: procedures and pattern matching

7.1 Procedures

`Sollya` has some elements of functional languages. In order to avoid confusion with mathematical functions, the associated programming objects are called *procedures* in `Sollya`.

`Sollya` procedures are common objects that can be, for example, assigned to variables or stored in lists. Procedures are declared by the `proc` keyword; see section 8.133 for details. The returned procedure object must then be assigned to a variable. It can hence be applied to arguments with common application syntax. The `procedure` keyword provides an abbreviation for declaring and assigning a procedure; see section 8.134 for details.

`Sollya` procedures can return objects using the `return` keyword at the end of the begin-end-block of the procedure. Section 8.148 gives details on the usage of `return`. Procedures further can take any type of object in argument, in particular also other procedures that are then applied to arguments. Procedures can be declared inside other procedures.

Common `Sollya` procedures are declared with a certain number of formal parameters. When the procedure is applied to actual parameters, a check is performed if the right number of actual parameters is given. Then the actual parameters are applied to the formal parameters. In some cases, it is required that the number of parameters of a procedure be variable. `Sollya` provides support for the case with procedures with an arbitrary number of actual arguments. When the procedure is called, those actual arguments are gathered in a list which is applied to the only formal list parameter of a procedure with an arbitrary number of arguments. See section 8.134 for the exact syntax and details; an example is given just below.

Let us remark that declaring a procedure does not involve any evaluation or other interpretation of the procedure body. In particular, this means that constants are evaluated to floating-point values inside `Sollya` when the procedure is applied to actual parameters and the global precision valid at this moment.

`Sollya` procedures are well illustrated with the following examples:

```
> succ = proc(n) { return n + 1; };
> succ(5);
6
> 3 + succ(0);
4
> succ;
proc(n)
{
nop;
return (n) + (1);
}
```

```
> add = proc(m,n) { var res; res := m + n; return res; };
> add(5,6);
11
> hey = proc() { print("Hello world."); };
> hey();
Hello world.
> print(hey());
Hello world.
void
> hey;
proc()
{
print("Hello world.");
return void;
}
```

```
> fac = proc(n) { var res; if (n == 0) then res := 1 else res := n * fac(n - 1);
 return res; };
> fac(5);
120
> fac(11);
39916800
> fac;
proc(n)
{
var res;
if (n) == (0) then
res := 1
else
res := (n) * (fac((n) - (1)));
return res;
}
```

```
> sumall = proc(args = ...) { var i, acc; acc = 0; for i in args do acc = acc +
i; return acc; };
> sumall;
proc(args = ...)
{
var i, acc;
acc = 0;
for i in args do
acc = (acc) + (i);
return acc;
}
> sumall();
0
> sumall(1);
1
> sumall(1,5);
6
> sumall(1,5,9);
15
> sumall @ [|1,5,9,4,8|];
27
>
```

Let us note that, when writing a procedure, one does not know what will be the name of the free variable at run-time. This is typically the context when one should use the special keyword `_x_`:

```
> ChebPolynomials = proc(n) {
    var i, res;
    if (n<0) then res = [||]
    else if (n==0) then res = [|1|]
    else {
       res = [|1, _x_|];
       for i from 2 to n do res[i] = horner(2*_x_*res[i-1]-res[i-2]);
    };
    return res;
  };
>
> f = sin(x);
> T = ChebPolynomials(4);
> canonical = on!;
> for i from 0 to 4 do T[i];
1
x
-1 + 2 * x^2
-3 * x + 4 * x^3
1 + -8 * x^2 + 8 * x^4
```

`Sollya` also supports external procedures, i.e. procedures written in `C` (or some other language) and dynamically bound to `Sollya` identifiers. See 8.60 for details.

7.2 Pattern matching

Starting with version 3.0, `Sollya` supports matching expressions with expression patterns. This feature is important for an extended functional programming style. Further, and most importantly, it allows expression trees to be recursively decomposed using native constructs of the `Sollya` language. This means no help from external procedures or other compiled-language mechanisms is needed here anymore.

Basically, pattern matching supports relies on one `Sollya` construct:

```
match expr with
pattern1 :  (return-expr1)
pattern2 :  (return-expr2)
...
patternN : (return-exprN)
```

That construct has the following semantic: try to match the expression *expr* with the patterns *pattern1* through *patternN*, proceeding in natural order. If a pattern *patternI* is found that matches, evaluate the whole `match ... with` construct to the return expression *return-exprI* associated with the matching pattern *patternI*. If no matching pattern is found, display an error warning and return `error`. Note that the parentheses around the expressions *return-exprI* are mandatory.

Matching a pattern means the following:

- If a pattern does not contain any programming-language-level variables (different from the free mathematical variable), it matches expressions that are syntactically equal to itself. For instance, the pattern `exp(sin(3 * x))` will match the expression `exp(sin(3 * x))`, but it does not match `exp(sin(x * 3))` because the expressions are not syntactically equal.
- If a pattern does contain variables, it matches an expression *expr* if these variables can be bound to subexpressions of *expr* such that once the pattern is evaluated with that variable binding, it becomes syntactically equal to the expression *expr*. For instance, the pattern `exp(sin(a * x))` will match the expression `exp(sin(3 * x))` as it is possible to bind `a` to `3` such that `exp(sin(a * x))` evaluates to `exp(sin(3 * x))`.

If a pattern *patternI* with variables is matched in a `match ... with` construct, the variables in the pattern stay bound during the evaluation of the corresponding return expression *return-exprI*. This allows subexpressions to be extracted from expressions and/or recursively handled as needed.

The following examples illustrate the basic principles of pattern matching in `Sollya`. One can remark that it is useful to use the keyword `_x_` when one wants to be sure to refer to the free variable in a pattern matching:

```
> match exp(x) with
      exp(x)      : (1)
      sin(x)      : (2)
      default     : (3);
1
>
> match sin(x) with
      exp(x)      : (1)
      sin(x)      : (2)
      default     : (3);
2
>
> match exp(sin(x)) with
      exp(x)      : ("Exponential of x")
      exp(sin(x)) : ("Exponential of sine of x")
      default     : ("Something else");
Exponential of sine of x
>
> match exp(sin(x)) with
      exp(x)      : ("Exponential of x")
      exp(a)      : ("Exponential of " @ a)
      default     : ("Something else");
Exponential of sin(x)
>
>
> procedure differentiate(f) {
      return match f with
          g + h   : (differentiate(g) + differentiate(h))
          g * h   : (differentiate(g) * h + differentiate(h) * g)
          g / h   : ((differentiate(g) * h - differentiate(h) * g) / (h^2))
          exp(_x_)  : (exp(_x_))
          sin(_x_)  : (cos(_x_))
          cos(_x_)  : (-sin(_x_))
          g(h)    : ((differentiate(g))(h) * differentiate(h))
          _x_       : (1)
          h(_x_)    : (NaN)
          c       : (0);
  };
>
> rename(x,y);
Information: the free variable has been renamed from "x" to "y".
> differentiate(exp(sin(y + y)));
exp(sin(y * 2)) * cos(y * 2) * 2
> diff(exp(sin(y + y)));
exp(sin(y * 2)) * cos(y * 2) * 2
>
```

As `Sollya` is not a purely functional language, the `match ... with` construct can also be used in a more imperative style, which makes it become closer to constructs like `switch` in `C` or `Perl`. In lieu of a simple return expression, a whole block of imperative statements can be given. The expression to be returned by that block is indicated in the end of the block, using the `return` keyword. That syntax is illustrated in the next example:

```
> match exp(sin(x)) with
      exp(a)  : {
                   write("Exponential of ", a, "\n");
                   return a;
                }
      sin(x)  : {
                   var foo;
                   foo = 17;
                   write("Sine of x\n");
                   return foo;
                }
      default : {
                   write("Something else\n");
                   bashexecute("LANG=C date");
                   return true;
                };
Exponential of sin(x)
sin(x)
>
> match sin(x) with
      exp(a)  : {
                   write("Exponential of ", a, "\n");
                   return a;
                }
      sin(x)  : {
                   var foo;
                   foo = 17;
                   write("Sine of x\n");
                   return foo;
                }
      default : {
                   write("Something else\n");
                   bashexecute("LANG=C date");
                   return true;
                };
Sine of x
17
>
> match acos(17 * pi * x) with
      exp(a)  : {
                   write("Exponential of ", a, "\n");
                   return a;
                }
      sin(x)  : {
                   var foo;
                   foo = 17;
                   write("Sine of x\n");
                   return foo;
                }
      default : {
                   write("Something else\n");
                   bashexecute("LANG=C date");
                   return true;
                };
Something else
Mon May  2 10:36:35 CEST 2011
true
```

In the case when no return statement is indicated for a statement-block in a `match ... with` construct, the construct evaluates to the special value `void` if that pattern matches.

In order to well understand pattern matching in `Sollya`, it is important to realize the meaning of variables in patterns. This meaning is different from the one usually found for variables. In a pattern, variables are never evaluated to whatever they might have set before the pattern is executed. In contrast, all variables in patterns are new, free variables that will freshly be bound to subexpressions of the matching expression. If a variable of the same name already exists, it will be shadowed during the evaluation of the statement block and the return expression corresponding to the matching expression. This type of semantic implies that patterns can never be computed at run-time, they must always be hard-coded beforehand. However this is necessary to make pattern matching context-free.

As a matter of course, all variables figuring in the expression *expr* to be matched are evaluated before pattern matching is attempted. In fact, *expr* is a usual `Sollya` expression, not a pattern.

In `Sollya`, the use of variables in patterns does not need to be linear. This means the same variable might appear twice or more in a pattern. Such a pattern will only match an expression if it contains the same subexpression, associated with the variable, in all places indicated by the variable in the pattern.

The following examples illustrate the use of variables in patterns in detail:

```
> a = 5;
> b = 6;
> match exp(x + 3) with
        exp(a + b) : {
                           print("Exponential");
                           print("a = ", a);
                           print("b = ", b);
                     }
        sin(x)     : {
                           print("Sine of x");
                     };
Exponential
a =  x
b =  3
> print("a = ", a, ", b = ", b);
a =  5 , b =  6
>
> a = 5;
> b = 6;
> match exp(x + 3) with
        exp(a + b) : {
                           var a, c;
                           a = 17;
                           c = "Hallo";
                           print("Exponential");
                           print("a = ", a);
                           print("b = ", b);
                           print("c = ", c);
                     }
        sin(x)     : {
                           print("Sine of x");
                     };
Exponential
a =  17
b =  3
c =  Hallo
> print("a = ", a, ", b = ", b);
a =  5 , b =  6
```

```
> match exp(sin(x)) + sin(x) with
        exp(a) + a : {
                        print("Winner");
                        print("a = ", a);
                     }
        default    : {
                        print("Loser");
                     };
Winner
a =  sin(x)
>
> match exp(sin(x)) + sin(3 * x) with
        exp(a) + a : {
                        print("Winner");
                        print("a = ", a);
                     }
        default    : {
                        print("Loser");
                     };
Loser
>
> f = exp(x);
> match f with
        sin(x) : (1)
        cos(x) : (2)
        exp(x) : (3)
        default : (4);
3
```

Pattern matching is meant to be a means to decompose expressions structurally. For this reason and in an analogous way to variables, no evaluation is performed at all on (sub-)expressions that form constant functions. As a consequence, patterns match constant expressions only if they are structurally identical. For example $5 + 1$ only matches $5 + 1$ and not $1 + 5$, $3 + 3$ nor 6.

This general rule on constant expressions admits one exception. Intervals in `Sollya` can be defined using constant expressions as bounds. These bounds are immediately evaluated to floating-point constants, though. In order to permit pattern matching on intervals, constant expressions given as bounds of intervals that form patterns are evaluated before pattern matching. However, in order not conflict with the rules of no evaluation of variables, these constant expressions as bounds of intervals in patterns must not contain free variables.

```
> match 5 + 1 with
      1 + 5 : ("One plus five")
      6     : ("Six")
      5 + 1 : ("Five plus one");
Five plus one
>
> match 6 with
      1 + 5 : ("One plus five")
      6     : ("Six")
      5 + 1 : ("Five plus one");
Six
>
> match 1 + 5 with
      1 + 5 : ("One plus five")
      6     : ("Six")
      5 + 1 : ("Five plus one");
One plus five
>
> match [1; 5 + 1] with
      [1; 1 + 5] : ("Interval from one to one plus five")
      [1; 6]     : ("Interval from one to six")
      [1; 5 + 1] : ("Interval from one to five plus one");
Interval from one to one plus five
>
> match [1; 6] with
      [1; 1 + 5] : ("Interval from one to one plus five")
      [1; 6]     : ("Interval from one to six")
      [1; 5 + 1] : ("Interval from one to five plus one");
Interval from one to one plus five
>
```

The `Sollya` keyword `default` has a special meaning in patterns. It acts like a wild-card, matching any (sub-)expression, as long as the whole expression stays correctly typed. Upon matching with `default`, no variable gets bound. This feature is illustrated in the next example:

```
> match exp(x) with
      sin(x)    : ("Sine of x")
      atan(x^2) : ("Arctangent of square of x")
      default   : ("Something else")
      exp(x)    : ("Exponential of x");
Something else
>
> match atan(x^2) with
      sin(x)          : ("Sine of x")
      atan(default^2) : ("Arctangent of the square of something")
      default         : ("Something else");
Arctangent of the square of something
>
> match atan(exp(x)^2) with
      sin(x)          : ("Sine of x")
      atan(default^2) : ("Arctangent of the square of something")
      default         : ("Something else");
Arctangent of the square of something
>
> match exp("Hello world") with
      exp(default)    : ("A miracle has happened")
      default         : ("Something else");
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
```

In `Sollya`, pattern matching is possible on the following `Sollya` types and operations defined on them:

- Expressions that define univariate functions, as explained above,
- Intervals with one, two or no bound defined in the pattern by a variable,
- Character sequences, literate or defined using the `@` operator, possibly with a variable on one of the sides of the `@` operator,
- Lists, literate, literate with variables or defined using the `.:`, `:.` and `@` operators, possibly with a variable on one of the sides of the `@` operator or one or two variables for `.:` and `:.`,
- Structures, literate or literate with variables, and
- All other `Sollya` objects, matchable with themselves (`DE` matches `DE`, `on` matches `on`, `perturb` matches `perturb` etc.)

```
> procedure detector(obj) {
      match obj with
          exp(a * x)            : { "Exponential of ", a, " times x"; }
          [ a; 17 ]             : { "An interval from ", a, " to 17"; }
          [| |]                 : { "Empty list"; }
          [| a, b, 2, exp(c) |] : { "A list of ", a, ", ", b, ", 2 and ",
                                    "exponential of ", c; }
          a @ [| 2, 3 |]        : { "Concatenation of the list ", a, " and ",
                                    "the list of 2 and 3"; }
          a .: [| 9 ... |]      : { a, " prepended to all integers >= 9"; }
          "Hello" @ w           : { "Hello concatenated with ", w; }
          { .a = sin(b);
            .b = [c;d] }        : { "A structure containing as .a the ",
                                    "sine of ", b,
                                    " and as .b the range from ", c,
                                    " to ", d; }
          perturb               : { "The special object perturb"; }
          default               : { "Something else"; };
  };
>
> detector(exp(5 * x));
Exponential of 5 times x
> detector([3.25;17]);
An interval from 3.25 to 17
> detector([||]);
Empty list
> detector([| sin(x), nearestint(x), 2, exp(5 * atan(x)) |]);
A list of sin(x), nearestint(x), 2 and exponential of 5 * atan(x)
> detector([| sin(x), cos(5 * x), "foo", 2, 3 |]);
Concatenation of the list [|sin(x), cos(x * 5), "foo"|] and the list of 2 and 3
> detector([| DE, 9... |]);
doubleextended prepended to all integers >= 9
> detector("Hello world");
Hello concatenated with  world
> detector({ .a = sin(x); .c = "Hello"; .b = [9;10] });
A structure containing as .a the sine of x and as .b the range from 9 to 10
> detector(perturb);
The special object perturb
> detector([13;19]);
Something else
```

Concerning intervals, please pay attention to the fact that expressions involving intervals are immediately evaluated and that structural pattern matching on functions on intervals is not possible. This point is illustrated in the next example:

```
> match exp([1;2]) with
        [a;b]              : {
                                a,", ",b;
                             }
        default            : {
                                "Something else";
                             };
2.71828182845904523536028747135266249775724709369989, 7.389056098930650227230427
460575007813180315570551 8
>
> match exp([1;2]) with
        exp([a;b])         : {
                                a,", ", b;
                             }
        default            : {
                                "Something else";
                             };
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
>
> match exp([1;2]) with
    exp(a)  : {
                "Exponential of ", a;
              }
    default : {
                "Something else";
              };
Something else
```

With respect to pattern matching on lists or character sequences defined using the `@` operator, the following is to be mentionned:

- Patterns like `a @ b` are not allowed as they would need to perform an ambiguous cut of the list or character sequence to be matched. This restriction is maintained even if the variables (here `a` and `b`) are constrained by other occurrences in the pattern (for example in a list) which would make the cut unambiguous.
- Recursive use of the `@` operator (even mixed with the operators `.:` and `:.`) is possible under the condition that there must not exist any other parenthezation of the term in concatenations (`@`) such that the rule of one single variable for `@` above gets violated. For instance, `( [| 1 |] @ a) @ (b @ [| 4 |])` is not possible as it can be re-parenthesized `[| 1 |] @ (a @ b) @ [| 4 |]`, which exhibits the ambiguous case.

These points are illustrated in this example:

```
> match [| exp(sin(x)), sin(x), 4, DE(x), 9... |] with
        exp(a) .: (a .: (([||] :. 4) @ (b @ [| 13... |]))) :
                              { "a = ", a, ", b = ", b; };
a = sin(x), b = [|doubleextended(x), 9, 10, 11, 12|]
>
> match [| 1, 2, 3, 4, D... |] with
        a @ [| 4, D...|] : (a);
[|1, 2, 3|]
>
> match [| 1, 2, 3, 4, D... |] with
        a @ [| D...|] : (a);
[|1, 2, 3, 4|]
>
> match [| 1, 2, 3, 4... |] with
        a @ [| 3...|] : (a);
[|1, 2|]
>
> match [| 1, 2, 3, 4... |] with
        a @ [| 4...|] : (a);
[|1, 2, 3|]
>
> match [| 1, 2, 3, 4... |] with
        a @ [| 17...|] : (a);
[|1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16|]
>
> match [| 1, 2, 3, 4... |] with
        a @ [| 17, 18, 19 |] : (a)
        default              : ("Something else");
Something else
```

As mentionned above, pattern matching on `Sollya` structures is possible. Patterns for such a match are given in a literately, i.e. using the syntax { `.a =` *exprA*, `.b =` *exprB*, ... }. A structure pattern *sp* will be matched by a structure *s* iff that structure *s* contains at least all the elements (like `.a`, `.b` etc.) of the structure pattern *sp* and iff each of the elements of the structure *s* matches the pattern in the corresponding element of the structure pattern *sp*. The user should be aware of the fact that the structure to be matched is only supposed to have at least the elements of the pattern but that it may contain more elements is a particular `Sollya` feature. For instance with pattern matching, it is hence possible to ensure that access to particular elements will be possible in a particular code segment. The following example is meant to clarify this point:

```
> structure.f = exp(x);
> structure.dom = [1;2];
> structure.formats = [| DD, D, D, D |];
> match structure with
       { .f = sin(x);
         .dom = [a;b]
       }                    : { "Sine, ",a,", ",b; }
       { .f = exp(c);
         .dom = [a;b];
         .point = default
       }                    : { "Exponential, ",a, ", ", b, ", ", c; }
       { .f = exp(x);
         .dom = [a;b]
       }                    : { "Exponential, ",a, ", ", b; }
       default              : { "Something else"; };
Exponential, 1, 2
>
> structure.f = sin(x);
> match structure with
       { .f = sin(x);
         .dom = [a;b]
       }                    : { "Sine, ",a,", ",b; }
       { .f = exp(c);
         .dom = [a;b];
         .point = default
       }                    : { "Exponential, ",a, ", ", b, ", ", c; }
       { .f = exp(x);
         .dom = [a;b]
       }                    : { "Exponential, ",a, ", ", b; }
       default              : { "Something else"; };
Sine, 1, 2
>
> structure.f = exp(x + 2);
> structure.point = 23;
> match structure with
       { .f = sin(x);
         .dom = [a;b]
       }                    : { "Sine, ",a,", ",b; }
       { .f = exp(c);
         .dom = [a;b];
         .point = default
       }                    : { "Exponential, ",a, ", ", b, ", ", c; }
       { .f = exp(x);
         .dom = [a;b]
       }                    : { "Exponential, ",a, ", ", b; }
       default              : { "Something else"; };
Exponential, 1, 2, 2 + x
```

8 Commands and functions

8.1 abs

Name: **abs**
the absolute value.
Description:

- **abs** is the absolute value function. **abs**(x)=$\begin{cases} x & x > 0 \\ -x & x \leq 0 \end{cases}$.

8.2 absolute

Name: **absolute**
indicates an absolute error for **externalplot**, **fpminimax** or **supnorm**
Usage:

absolute : absolute|relative

Description:

- The use of **absolute** in the command **externalplot** indicates that during plotting in **externalplot** an absolute error is to be considered.

  See **externalplot** for details.

- Used with **fpminimax**, **absolute** indicates that **fpminimax** must try to minimize the absolute error.

  See **fpminimax** for details.

- When given in argument to **supnorm**, **absolute** indicates that an absolute error is to be considered for supremum norm computation.

  See **supnorm** for details.

Example 1:

```
> bashexecute("gcc -fPIC -c externalplotexample.c");
> bashexecute("gcc -shared -o externalplotexample externalplotexample.o -lgmp -l
mpfr");
> externalplot("./externalplotexample",absolute,exp(x),[-1/2;1/2],12,perturb);
```

See also: **externalplot** (8.59), **fpminimax** (8.67), **relative** (8.144), **bashexecute** (8.17), **supnorm** (8.170)

8.3 accurateinfnorm

Name: **accurateinfnorm**
computes a faithful rounding of the infinity norm of a function

Usage:

accurateinfnorm(*function*,*range*,*constant*) : (function, range, constant) → constant
accurateinfnorm(*function*,*range*,*constant*,*exclusion range 1*,...,*exclusion range n*) : (function, range, constant, range, ..., range) → constant

Parameters:

- *function* represents the function whose infinity norm is to be computed
- *range* represents the infinity norm is to be considered on
- *constant* represents the number of bits in the significant of the result
- *exclusion range 1* through *exclusion range n* represent ranges to be excluded

Description:

- The command **accurateinfnorm** computes an upper bound to the infinity norm of function *function* in *range*. This upper bound is the least floating-point number greater than the value of the infinity norm that lies in the set of dyadic floating point numbers having *constant* significant mantissa bits. This means the value **accurateinfnorm** evaluates to is at the time an upper bound and a faithful rounding to *constant* bits of the infinity norm of function *function* on range *range*.

  If given, the fourth and further arguments of the command **accurateinfnorm**, *exclusion range 1* through *exclusion range n* the infinity norm of the function *function* is not to be considered on.

- The command **accurateinfnorm** is now considered DEPRECATED in `Sollya`. Users should be aware about the fact that the algorithm behind **accurateinfnorm** is highly inefficient and that other, better suited algorithms, such as **supnorm**, are available inside `Sollya`. As a matter of fact, while **accurateinfnorm** is maintained for compatibility reasons with legacy `Sollya` codes, users are advised to avoid using **accurateinfnorm** in new `Sollya` scripts and to replace it, where possible, by the **supnorm** command.

Example 1:

```
> p = remez(exp(x), 5, [-1;1]);
> accurateinfnorm(p - exp(x), [-1;1], 20);
4.52055246569216251373291015625e-5
> accurateinfnorm(p - exp(x), [-1;1], 30);
4.52055210757862369064241647720336914062 5e-5
> accurateinfnorm(p - exp(x), [-1;1], 40);
4.52055210438673249484509142348542809486389160156 25e-5
```

Example 2:

```
> p = remez(exp(x), 5, [-1;1]);
> midpointmode = on!;
> infnorm(p - exp(x), [-1;1]);
0.45205~5/7~e-4
> accurateinfnorm(p - exp(x), [-1;1], 40);
4.52055210438673249484509142348542809486389160156 25e-5
```

See also: **infnorm** (8.84), **dirtyinfnorm** (8.41), **supnorm** (8.170), **checkinfnorm** (8.23), **remez** (8.145), **diam** (8.37)

8.4 acos

Name: **acos**
the arccosine function.
Description:

- **acos** is the inverse of the function **cos**: **acos**(y) is the unique number $x \in [0; \pi]$ such that **cos**(x)=y.
- It is defined only for $y \in [-1; 1]$.

See also: **cos** (8.28)

8.5 acosh

Name: **acosh**
the arg-hyperbolic cosine function.
Description:

- **acosh** is the inverse of the function **cosh**: **acosh**(y) is the unique number $x \in [0; +\infty]$ such that **cosh**(x)=y.
- It is defined only for $y \in [0; +\infty]$.

See also: **cosh** (8.29)

8.6 &&

Name: &&
boolean AND operator
Usage:

expr1 && *expr2* : (boolean, boolean) → boolean

Parameters:

- *expr1* and *expr2* represent boolean expressions

Description:

- && evaluates to the boolean AND of the two boolean expressions *expr1* and *expr2*. && evaluates to true iff both *expr1* and *expr2* evaluate to true.

Example 1:

```
> true && false;
false
```

Example 2:

```
> (1 == exp(0)) && (0 == log(1));
true
```

See also: || (8.114), **!** (8.109)

8.7 :.

Name: **:.**
add an element at the end of a list.
Usage:

L**:.**x : (list, any type) → list

Parameters:

- L is a list (possibly empty).
- x is an object of any type.

Description:

- **:.** adds the element x at the end of the list L.
- Note that since x may be of any type, it can in particular be a list.

Example 1:

```
> [|2,3,4|]:.5;
[|2, 3, 4, 5|]
```

Example 2:

```
> [|1,2,3|]:.[|4,5,6|];
[|1, 2, 3, [|4, 5, 6|]|]
```

Example 3:

```
> [||]:.1;
[|1|]
```

See also: **.:** (8.127), **@** (8.26)

8.8 ~

Name: ~
floating-point evaluation of a constant expression
Usage:

~ *expression* : function → constant
~ *something* : any type → any type

Parameters:

- *expression* stands for an expression that is a constant
- *something* stands for some language element that is not a constant expression

Description:

- ~ *expression* evaluates the *expression* that is a constant term to a floating-point constant. The evaluation may involve a rounding. If *expression* is not a constant, the evaluated constant is a faithful rounding of *expression* with **precision** bits, unless the *expression* is exactly 0 as a result of cancellation. In the latter case, a floating-point approximation of some (unknown) accuracy is returned.
- ~ does not do anything on all language elements that are not a constant expression. In other words, it behaves like the identity function on any type that is not a constant expression. It can hence be used in any place where one wants to be sure that expressions are simplified using floating-point computations to constants of a known precision, regardless of the type of actual language elements.
- ~ **error** evaluates to error and provokes a warning.
- ~ is a prefix operator not requiring parentheses. Its precedence is the same as for the unary + and − operators. It cannot be repeatedly used without brackets.

Example 1:

```
> print(exp(5));
exp(5)
> print(~ exp(5));
1.484131591025766034211155800405522796234876675938 78e2
```

Example 2:

```
> autosimplify = off!;
```

Example 3:

```
> print(~sin(5 * pi));
0
```

Example 4:

```
> print(~exp(x));
exp(x)
> print(~ "Hello");
Hello
```

Example 5:

```
> print(~exp(x*5*Pi));
exp((pi) * 5 * x)
> print(exp(x* ~(5*Pi)));
exp(x * 1.57079632679489661923132169163975144209858469968757e1)
```

Example 6:

```
> print(~exp(5)*x);
1.484131591025766034211155800405522796234876675938 78e2 * x
> print( (~exp(5))*x);
1.484131591025766034211155800405522796234876675938 78e2 * x
> print(~(exp(5)*x));
exp(5) * x
```

See also: **evaluate** (8.53), **prec** (8.125), **error** (8.52)

8.9 asciiplot

Name: **asciiplot**
plots a function in a range using ASCII characters
Usage:

asciiplot(*function*, *range*) : (function, range) → void

Parameters:

- *function* represents a function to be plotted
- *range* represents a range the function is to be plotted in

Description:

- **asciiplot** plots the function *function* in range *range* using ASCII characters. On systems that provide the necessary `TIOCGWINSZ ioctl`, `Sollya` determines the size of the terminal for the plot size if connected to a terminal. If it is not connected to a terminal or if the test is not possible, the plot is of fixed size 77×25 characters. The function is evaluated on a number of points equal to the number of columns available. Its value is rounded to the next integer in the range of lines available. A letter `x` is written at this place. If zero is in the hull of the image domain of the function, an x-axis is displayed. If zero is in range, a y-axis is displayed. If the function is constant or if the range is reduced to one point, the function is evaluated to a constant and the constant is displayed instead of a plot.

Example 1:

```
> asciiplot(exp(x),[1;2]);
                                                                      x
                                                                     xx
                                                                   xxx
                                                                 xx
                                                               xx
                                                             xxx
                                                           xx
                                                        xxx
                                                      xx
                                                    xxx
                                                  xxx
                                                xxx
                                              xxx
                                            xxx
                                          xxx
                                        xxxx
                                      xxxx
                                    xxx
                                  xxxx
                               xxxxx
                            xxxxx
                         xxxx
                      xxxxx
xxxx
```

Example 2:

```
> asciiplot(expm1(x),[-1;2]);
                           |                                                x
                           |                                               x
                           |                                              x
                           |                                             x
                           |                                            x
                           |                                          xx
                           |                                         x
                           |                                        x
                           |                                      xx
                           |                                    xx
                           |                                  xx
                           |                                 xx
                           |                                x
                           |                             xxx
                           |                           xx
                           |                         xxx
                           |                       xxx
                           |                   xxxx
                           |               xxxx
                           |           xxxx
                           |     xxxxxx
---------------------xxxxxxxx-------------------------------------------------
         xxxxxxxxxxxx      |
xxxxxxxxx                  |
```

Example 3:

```
> asciiplot(5,[-1;1]);
5
```

Example 4:

```
> asciiplot(exp(x),[1;1]);
2.7182818284590452353602874713526624977572470936999 8
```

See also: **plot** (8.118), **externalplot** (8.59)

8.10 asin

Name: **asin**
the arcsine function.
Description:

- **asin** is the inverse of the function **sin**: **asin**(y) is the unique number $x \in [-\pi/2; \pi/2]$ such that **sin**(x)=y.
- It is defined only for $y \in [-1; 1]$.

See also: **sin** (8.161)

8.11 asinh

Name: **asinh**
the arg-hyperbolic sine function.
Description:

- **asinh** is the inverse of the function **sinh**: **asinh**(y) is the unique number $x \in [-\infty; +\infty]$ such that **sinh**(x)=y.
- It is defined for every real number y.

See also: **sinh** (8.163)

8.12 atan

Name: **atan**
the arctangent function.
Description:

- **atan** is the inverse of the function **tan**: **atan**(y) is the unique number $x \in [-\pi/2; +\pi/2]$ such that **tan**(x)=y.
- It is defined for every real number y.

See also: **tan** (8.172)

8.13 atanh

Name: **atanh**
the hyperbolic arctangent function.
Description:

- **atanh** is the inverse of the function **tanh**: **atanh**(y) is the unique number $x \in [-\infty; +\infty]$ such that **tanh**(x)=y.
- It is defined only for $y \in [-1; 1]$.

See also: **tanh** (8.173)

8.14 autodiff

Name: **autodiff**
Computes the first n derivatives of a function at a point or over an interval.
Usage:

autodiff(f, n, x_0) : (function, integer, constant) → list
autodiff(f, n, I) : (function, integer, range) → list

Parameters:

- f is the function to be differentiated.
- n is the order of differentiation.
- x_0 is the point at which the function is differentiated.
- I is the interval over which the function is differentiated.

Description:

- **autodiff** computes the first n derivatives of f at point x_0. The computation is performed numerically, without symbolically differentiating the expression of f. Yet, the computation is safe since small interval enclosures are produced. More precisely, **autodiff** returns a list $[f_0, \ldots, f_n]$ such that, for each i, f_i is a small interval enclosing the exact value of $f^{(i)}(x_0)$.
- Since it does not perform any symbolic differentiation, **autodiff** is much more efficient than **diff** and should be prefered when only numerical values are necessary.
- If an interval I is provided instead of a point x_0, the list returned by **autodiff** satisfies: $\forall i$, $f^{(i)}(I) \subseteq f_i$. A particular use is when one wants to know the successive derivatives of a function at a non representable point such as π. In this case, it suffices to call **autodiff** with the (almost) point interval $I =$ [**pi**].
- When I is almost a point interval, the returned enclosures f_i are also almost point intervals. However, when the interval I begins to be fairly large, the enclosures can be deeply overestimated due to the dependecy phenomenon present with interval arithmetic.
- As a particular case, f_0 is an enclosure of the image of f over I. However, since the algorithm is not specially designed for this purpose it is not very efficient for this particular task. In particular, it is not able to return a finite enclosure for functions with removable singularities (e.g. $\sin(x)/x$ at 0). The command **evaluate** is much more efficient for computing an accurate enclosure of the image of a function over an interval.

Example 1:

```
> L = autodiff(exp(cos(x))+sin(exp(x)), 5, 0);
> midpointmode = on!;
> for f_i in L do f_i;
0.3559752813266941742012789792982961497379810154498~2/4~e1
0.5403023058681397174009366074429766037323104206179~0/3~
-0.3019450507398802024611853185539984893647499733880~6/2~e1
-0.2524412954423689519957506964890896998867689182397~6/4~e1
0.3122789875648103314521452918413972974632057906~1/3~e1
-0.1663430795900669603348405357933995688395595497~8/1~e2
```

Example 2:

```
> f = log(cos(x)+x);
> L = autodiff(log(cos(x)+x), 5, [2,4]);
> L[0];
[0;1.27643852425465597132446653114905059102580436018893]
> evaluate(f, [2,4]);
[0.45986058925497069206106494332976097408234056912429;1.20787210589964169595901037621103012113048821362855]
> fprime = diff(f);
> L[1];
[2.53086745013099407167484456656211083053393118778677e-2;1.756802495307928251372639094511829094135912887336 49]
> evaluate(fprime,[2,4]);
[2.71048755415961996452136364304380881763456815673085e-2;1.109195306639432908373972257886235314055584312799 49]
```

Example 3:

```
> L = autodiff(sin(x)/x, 0, [-1,1]);
> L[0];
[-@Inf@;@Inf@]
> evaluate(sin(x)/x, [-1,1]);
[0.54030230586813971740093660744297660373231042061 79;1]
```

See also: **diff** (8.39), **evaluate** (8.53)

8.15 autosimplify

Name: **autosimplify**
activates, deactivates or inspects the value of the automatic simplification state variable
Usage:

autosimplify = *activation value* : on|off → void
autosimplify = *activation value* ! : on|off → void
autosimplify : on|off

Parameters:

- *activation value* represents **on** or **off**, i.e. activation or deactivation

Description:

- An assignment **autosimplify** = *activation value*, where *activation value* is one of **on** or **off**, activates respectively deactivates the automatic safe simplification of expressions of functions generated by the evaluation of commands or in argument of other commands.

  `Sollya` commands like **remez**, **taylor** or **rationalapprox** sometimes produce expressions that can be simplified. Constant subexpressions can be evaluated to dyadic floating-point numbers, monomials with coefficients 0 can be eliminated. Further, expressions indicated by the user perform better in many commands when simplified before being passed in argument to a command. When the automatic simplification of expressions is activated, `Sollya` automatically performs a safe (not value changing) simplification process on such expressions.

  The automatic generation of subexpressions can be annoying, in particular if it takes too much time for not enough benefit. Further the user might want to inspect the structure of the expression tree returned by a command. In this case, the automatic simplification should be deactivated.

  If the assignment **autosimplify** = *activation value* is followed by an exclamation mark, no message indicating the new state is displayed. Otherwise the user is informed of the new state of the global mode by an indication.

Example 1:

```
> autosimplify = on !;
> print(x - x);
0
> autosimplify = off ;
Automatic pure tree simplification has been deactivated.
> print(x - x);
x - x
```

Example 2:

```
> autosimplify = on !;
> print(rationalapprox(sin(pi/5.9),7));
0.5
> autosimplify = off !;
> print(rationalapprox(sin(pi/5.9),7));
1 / 2
```

See also: **print** (8.128), **prec** (8.125), **points** (8.120), **diam** (8.37), **display** (8.43), **verbosity** (8.183), **canonical** (8.21), **taylorrecursions** (8.176), **timing** (8.179), **fullparentheses** (8.68), **midpointmode** (8.102), **hopitalrecursions** (8.77), **remez** (8.145), **rationalapprox** (8.139), **taylor** (8.174)

8.16 bashevaluate

Name: **bashevaluate**
executes a shell command and returns its output as a string
Usage:

bashevaluate(*command*) : string → string
bashevaluate(*command*,*input*) : (string, string) → string

Parameters:

- *command* is a command to be interpreted by the shell.
- *input* is an optional character sequence to be fed to the command.

Description:

- **bashevaluate**(*command*) will execute the shell command *command* in a shell. All output on the command's standard output is collected and returned as a character sequence.
- If an additional argument *input* is given in a call to **bashevaluate**(*command*,*input*), this character sequence is written to the standard input of the command *command* that gets executed.
- All characters output by *command* are included in the character sequence to which **bashevaluate** evaluates but two exceptions. Every NULL character ('\0') in the output is replaced with '?' as `Sollya` is unable to handle character sequences containing that character. Additionally, if the output ends in a newline character ('\n'), this character is stripped off. Other newline characters which are not at the end of the output are left as such.

Example 1:

```
> bashevaluate("LANG=C date");
Mon May 23 17:36:50 CEST 2011
```

Example 2:

```
> [| bashevaluate("echo Hello") |];
[|"Hello"|]
```

Example 3:

```
> a = bashevaluate("sed -e 's/a/e/g;'", "Hallo");
> a;
Hello
```

See also: **bashexecute** (8.17)

8.17 bashexecute

Name: **bashexecute**
executes a shell command.
Usage:

bashexecute(*command*) : string → void

Parameters:

- *command* is a command to be interpreted by the shell.

Description:

- **bashexecute**(*command*) lets the shell interpret *command*. It is useful to execute some external code within `Sollya`.
- **bashexecute** does not return anything. It just executes its argument. However, if *command* produces an output in a file, this result can be imported in `Sollya` with help of commands like **execute**, **readfile** and **parse**.

Example 1:

```
> bashexecute("LANG=C date");
Mon May 23 17:36:53 CEST 2011
```

See also: **execute** (8.54), **readfile** (8.142), **parse** (8.115), **bashevaluate** (8.16)

8.18 binary

Name: **binary**
special value for global state **display**
Description:

- **binary** is a special value used for the global state **display**. If the global state **display** is equal to **binary**, all data will be output in binary notation.

  As any value it can be affected to a variable and stored in lists.

See also: **decimal** (8.33), **dyadic** (8.48), **powers** (8.124), **hexadecimal** (8.75), **display** (8.43)

8.19 bind

Name: **bind**
partially applies a procedure to an argument, returning a procedure with one argument less
Usage:

bind(*proc*, *ident*, *obj*) : (procedure, identifier type, any type) → procedure

Parameters:

- *proc* is a procedure to be partially applied to an argument
- *ident* is one of the formal arguments of *proc*

- *obj* is any `Sollya` object *ident* is to be bound to

Description:

- **bind** allows a formal parameter *ident* of a procedure *proc* to be bound to an object *obj*, hence *proc* to be partially applied. The result of this curryfied application, returned by **bind**, is a procedure with one argument less. This way, **bind** permits specialization of a generic procedure, parameterized e.g. by a function or range.
- In the case when *proc* does not have a formal parameter named *ident*, **bind** prints a warning and returns the procedure *proc* unmodified.
- **bind** always returns a procedure, even if *proc* only has one argument, which gets bound to *ident*. In this case, **bind** returns a procedure which does not take any argument. Hence evaluation, which might provoke side effects, is only performed once the procedure gets used.
- **bind** does not work on procedures with an arbitrary number of arguments.

Example 1:

```
> procedure add(X,Y) { return X + Y; };
> succ = bind(add,X,1);
> succ(5);
6
> succ;
proc(Y)
{
nop;
return (proc(X, Y)
{
nop;
return (X) + (Y);
})(1, Y);
}
```

Example 2:

```
> procedure add(X,Y) { return X + Y; };
> succ = bind(add,X,1);
> five = bind(succ,Y,4);
> five();
5
> five;
proc()
{
nop;
return (proc(Y)
{
nop;
return (proc(X, Y)
{
nop;
return (X) + (Y);
})(1, Y);
})(4);
}
```

Example 3:

```
> verbosity = 1!;
> procedure add(X,Y) { return X + Y; };
> foo = bind(add,R,1);
Warning: the given procedure has no argument named "R". The procedure is returne
d unchanged.
> foo;
proc(X, Y)
{
nop;
return (X) + (Y);
}
```

See also: **procedure** (8.134), **proc** (8.133), **function** (8.69), **@** (8.26)

8.20 boolean

Name: **boolean**
keyword representing a boolean type
Usage:

boolean : type type

Description:

- **boolean** represents the boolean type for declarations of external procedures by means of **externalproc**.

  Remark that in contrast to other indicators, type indicators like **boolean** cannot be handled outside the **externalproc** context. In particular, they cannot be assigned to variables.

See also: **externalproc** (8.60), **constant** (8.27), **function** (8.69), **integer** (8.85), **list of** (8.93), **range** (8.138), **string** (8.166)

8.21 canonical

Name: **canonical**
brings all polynomial subexpressions of an expression to canonical form or activates, deactivates or checks canonical form printing
Usage:

canonical(*function*) : function → function
canonical = *activation value* : on|off → void
canonical = *activation value* ! : on|off → void

Parameters:

- *function* represents the expression to be rewritten in canonical form
- *activation value* represents **on** or **off**, i.e. activation or deactivation

Description:

- The command **canonical** rewrites the expression representing the function *function* in a way such that all polynomial subexpressions (or the whole expression itself, if it is a polynomial) are written in canonical form, i.e. as a sum of monomials in the canonical base. The canonical base is the base of the integer powers of the global free variable. The command **canonical** does not endanger the safety of computations even in `Sollya`'s floating-point environment: the function returned is mathematically equal to the function *function*.

- An assignment **canonical** = *activation value*, where *activation value* is one of **on** or **off**, activates respectively deactivates the automatic printing of polynomial expressions in canonical form, i.e. as a sum of monomials in the canonical base. If automatic printing in canonical form is deactivated, automatic printing yields to displaying polynomial subexpressions in Horner form.

  If the assignment **canonical** = *activation value* is followed by an exclamation mark, no message indicating the new state is displayed. Otherwise the user is informed of the new state of the global mode by an indication.

Example 1:

```
> print(canonical(1 + x * (x + 3 * x^2)));
1 + x^2 + 3 * x^3
> print(canonical((x + 1)^7));
1 + 7 * x + 21 * x^2 + 35 * x^3 + 35 * x^4 + 21 * x^5 + 7 * x^6 + x^7
```

Example 2:

```
> print(canonical(exp((x + 1)^5) - log(asin(((x + 2) + x)^4 * (x + 1)) + x)));
exp(1 + 5 * x + 10 * x^2 + 10 * x^3 + 5 * x^4 + x^5) - log(asin(16 + 80 * x + 16
0 * x^2 + 160 * x^3 + 80 * x^4 + 16 * x^5) + x)
```

Example 3:

```
> canonical;
off
> (x + 2)^9;
512 + x * (2304 + x * (4608 + x * (5376 + x * (4032 + x * (2016 + x * (672 + x *
 (144 + x * (18 + x))))))))
> canonical = on;
Canonical automatic printing output has been activated.
> (x + 2)^9;
512 + 2304 * x + 4608 * x^2 + 5376 * x^3 + 4032 * x^4 + 2016 * x^5 + 672 * x^6 +
 144 * x^7 + 18 * x^8 + x^9
> canonical;
on
> canonical = off!;
> (x + 2)^9;
512 + x * (2304 + x * (4608 + x * (5376 + x * (4032 + x * (2016 + x * (672 + x *
 (144 + x * (18 + x))))))))
```

See also: **horner** (8.78), **print** (8.128), **autosimplify** (8.15)

8.22 ceil

Name: **ceil**
the usual function ceil.
Description:

- **ceil** is defined as usual: **ceil**(x) is the smallest integer y such that $y \geq x$.
- It is defined for every real number x.

See also: **floor** (8.66), **nearestint** (8.106), **round** (8.151), **RU** (8.155)

8.23 checkinfnorm

Name: **checkinfnorm**
checks whether the infinity norm of a function is bounded by a value
Usage:

checkinfnorm(*function*,*range*,*constant*) : (function, range, constant) → boolean

Parameters:

- *function* represents the function whose infinity norm is to be checked
- *range* represents the infinity norm is to be considered on
- *constant* represents the upper bound the infinity norm is to be checked to

Description:

- The command **checkinfnorm** checks whether the infinity norm of the given function *function* in the range *range* can be proven (by `Sollya`) to be less than the given bound *bound*. This means, if **checkinfnorm** evaluates to **true**, the infinity norm has been proven (by `Sollya`'s interval arithmetic) to be less than the bound. If **checkinfnorm** evaluates to **false**, there are two possibilities: either the bound is less than or equal to the infinity norm of the function or the bound is greater than the infinity norm but `Sollya` could not conclude using its internal interval arithmetic.

  checkinfnorm is sensitive to the global variable **diam**. The smaller **diam**, the more time `Sollya` will spend on the evaluation of **checkinfnorm** in order to prove the bound before returning **false** although the infinity norm is bounded by the bound. If **diam** is equal to 0, `Sollya` will eventually spend infinite time on instances where the given bound *bound* is less or equal to the infinity norm of the function *function* in range *range*. In contrast, with **diam** being zero, **checkinfnorm** evaluates to **true** iff the infinity norm of the function in the range is bounded by the given bound.

Example 1:

```
> checkinfnorm(sin(x),[0;1.75], 1);
true
> checkinfnorm(sin(x),[0;1.75], 1/2); checkinfnorm(sin(x),[0;20/39],1/2);
false
true
```

Example 2:

```
> p = remez(exp(x), 5, [-1;1]);
> b = dirtyinfnorm(p - exp(x), [-1;1]);
> checkinfnorm(p - exp(x), [-1;1], b);
false
> b1 = round(b, 15, RU);
> checkinfnorm(p - exp(x), [-1;1], b1);
true
> b2 = round(b, 25, RU);
> checkinfnorm(p - exp(x), [-1;1], b2);
false
> diam = 1b-20!;
> checkinfnorm(p - exp(x), [-1;1], b2);
true
```

See also: **infnorm** (8.84), **dirtyinfnorm** (8.41), **supnorm** (8.170), **accurateinfnorm** (8.3), **remez** (8.145), **diam** (8.37)

8.24 coeff

Name: **coeff**
gives the coefficient of degree n of a polynomial
Usage:

coeff(f,n) : (function, integer) → constant

Parameters:

- f is a function (usually a polynomial).
- n is an integer

Description:

- If f is a polynomial, **coeff**(f, n) returns the coefficient of degree n in f.
- If f is a function that is not a polynomial, **coeff**(f, n) returns 0.

Example 1:

```
> coeff((1+x)^5,3);
10
```

Example 2:

```
> coeff(sin(x),0);
0
```

See also: **degree** (8.35), **roundcoefficients** (8.152), **subpoly** (8.167)

8.25 composepolynomials

Name: **composepolynomials**
computes an approximation to the composition of two polynomials and bounds the error
Usage:

composepolynomials(p,q) : (function, function) → structure

Parameters:

- p and q are polynomials

Description:

- Given two polynomials p and q, **composepolynomials**(p, q) computes an approximation r to the polynomial $(p \circ q)$ and bounds the error polynomial $r - (p \circ q)$ using interval arithmetic.
- **composepolynomials** always returns a structure containing two elements, `poly` and `radii`. The element `poly` is contains the approximate composed polynomial r. The element `radii` contains a list of $n + 1$ intervals a_i bounding the coefficients of the error polynomial, which is of the same degree n as is the composed polynomial $(p \circ q)$. This is, there exist $\alpha_i \in a_i$ such that

$$\sum_{i=0}^{n} \alpha_i \, x^i = r(x) - (p \circ q)(x).$$

- In the case when either of p or q is not a polynomial, **composepolynomials** behaves like **substitute** used in a literate structure. The list of intervals bounding the coefficients of the error polynomial is returned empty.

Example 1:

```
> composepolynomials(1 + 2 * x + 3 * x^2 + 4 * x^3, 5 + 6 * x + 7 * x^2);
{ .radii = [|[0;0], [0;0], [0;0], [0;0], [0;0], [0;0], [0;0]|], .poly = 586 + x
* (1992 + x * (4592 + x * (6156 + x * (6111 + x * (3528 + x * 1372))))) }
```

Example 2:

```
> print(composepolynomials(1/5 * x + exp(17) + log(2) * x^2, x^4 + 1/3 * x^2));
{ .radii = [|[-7.17464813734306340312949546644437059215494114240 77e-43;7.1746481
37343063403129495466444370592154941142407 7e-43], [0;0], [-2.67276471009219564614
05364671514818788151968 8010505e-51;2.67276471009219564614053646715148187881519688
8010505e-51], [0;0], [-1.06910588403687825845621458686059275152607875204202e-50;
1.06910588403687825845621458686059275152607875204202e-50], [0;0], [-2.1382117680
73756516912429173721185503052157504 08404e-50;2.13821176807375651691242917372118
5503052157504 08404e-50], [0;0], [-2.13821176807375651691242917372118550305215750
4084 04e-50;2.13821176807375651691242917372118550305215750408404e-50]|], .poly = 2
.41549527535752982147754351803858238798675673527228e7 + x^2 * (6.666666666666666
6666666666666666666666666666666666e-2 + x^2 * (0.277016353395549478824136902384 2
41840897277792706698 + x^2 * (0.46209812037329687294482141430545104538366675624 0
17 + x^2 * 0.693147180559945309417232121458176568075500134360259))) }
```

Example 3:

```
> composepolynomials(sin(x),x + x^2);
{ .radii = [| |], .poly = sin(x * (1 + x)) }
```

See also: **substitute** (8.168)

8.26 @

Name: **@**
concatenates two lists or strings or applies a list as arguments to a procedure
Usage:

$$L1\mathbf{@}L2 : (\mathsf{list}, \mathsf{list}) \to \mathsf{list}$$
$$string1\mathbf{@}string2 : (\mathsf{string}, \mathsf{string}) \to \mathsf{string}$$
$$proc\mathbf{@}L1 : (\mathsf{procedure}, \mathsf{list}) \to \mathsf{any\ type}$$

Parameters:

- *L1* and *L2* are two lists.
- *string1* and *string2* are two strings.
- *proc* is a procedure.

Description:

- In its first usage form, @ concatenates two lists or strings.
- In its second usage form, @ applies the elements of a list as arguments to a procedure. In the case when *proc* is a procedure with a fixed number of arguments, a check is done if the number of elements in the list corresponds to the number of formal parameters of the procedure. An empty list can therefore applied only to a procedure that does not take any argument. In the case of a procedure with an arbitrary number of arguments, no such check is performed.

Example 1:

```
> [|1,...,3|]@[|7,8,9|];
[|1, 2, 3, 7, 8, 9|]
```

Example 2:

```
> "Hello "@"World!";
Hello World!
```

Example 3:

```
> procedure cool(a,b,c) {
  write(a,", ", b," and ",c," are cool guys.\n");
  };
> cool @ [| "Christoph", "Mioara", "Sylvain" |];
Christoph, Mioara and Sylvain are cool guys.
```

Example 4:

```
> procedure sayhello() {
  "Hello! how are you?";
  };
> sayhello();
Hello! how are you?
> sayhello @ [||];
Hello! how are you?
```

Example 5:

```
> procedure add(L = ...) {
  var acc, i;
  acc = 0;
  for i in L do acc = i + acc;
  return acc;
  };
> add(1,2);
3
> add(1,2,3);
6
> add @ [|1, 2|];
3
> add @ [|1, 2, 3|];
6
> add @ [||];
0
```

See also: **.:** (8.127), **:.** (8.7), **procedure** (8.134), **proc** (8.133), **bind** (8.19)

8.27 constant

Name: **constant**
keyword representing a constant type
Usage:

constant : type type

Description:

- **constant** represents the constant type for declarations of external procedures **externalproc**.

  Remark that in contrast to other indicators, type indicators like **constant** cannot be handled outside the **externalproc** context. In particular, they cannot be assigned to variables.

See also: **externalproc** (8.60), **boolean** (8.20), **function** (8.69), **integer** (8.85), **list of** (8.93), **range** (8.138), **string** (8.166)

8.28 cos

Name: **cos**
the cosine function.
Description:

- **cos** is the usual cosine function.
- It is defined for every real number x.

See also: **acos** (8.4), **sin** (8.161), **tan** (8.172)

8.29 cosh

Name: **cosh**
the hyperbolic cosine function.
Description:

- **cosh** is the usual hyperbolic function: $\cosh(x) = \frac{e^x+e^{-x}}{2}$.
- It is defined for every real number x.

See also: **acosh** (8.5), **sinh** (8.163), **tanh** (8.173), **exp** (8.55)

8.30 D

Name: **D**
short form for **double**
See also: **double** (8.45)

8.31 DD

Name: **DD**
short form for **doubledouble**
See also: **doubledouble** (8.46)

8.32 DE

Name: **DE**
short form for **doubleextended**
See also: **doubleextended** (8.47)

8.33 decimal

Name: **decimal**
special value for global state **display**
Description:

- **decimal** is a special value used for the global state **display**. If the global state **display** is equal to **decimal**, all data will be output in decimal notation.

  As any value it can be affected to a variable and stored in lists.

See also: **dyadic** (8.48), **powers** (8.124), **hexadecimal** (8.75), **binary** (8.18), **display** (8.43)

8.34 default

Name: **default**
default value for some commands.
Description:

- **default** is a special value and is replaced by something depending on the context where it is used. It can often be used as a joker, when you want to specify one of the optional parameters of a command and not the others: set the value of uninteresting parameters to **default**.
- Global variables can be reset by affecting them the special value **default**.

Example 1:

```
> p = remez(exp(x),5,[0;1],default,1e-5);
> q = remez(exp(x),5,[0;1],1,1e-5);
> p==q;
true
```

Example 2:

```
> prec;
165
> prec=200;
The precision has been set to 200 bits.
> prec=default;
The precision has been set to 165 bits.
```

8.35 degree

Name: **degree**
gives the degree of a polynomial.
Usage:

$$\textbf{degree}(f) : \mathsf{function} \to \mathsf{integer}$$

Parameters:

- f is a function (usually a polynomial).

Description:

- If f is a polynomial, **degree**(f) returns the degree of f.
- Contrary to the usage, `Sollya` considers that the degree of the null polynomial is 0.
- If f is a function that is not a polynomial, **degree**(f) returns -1.

Example 1:

```
> degree((1+x)*(2+5*x^2));
3
> degree(0);
0
```

Example 2:

```
> degree(sin(x));
-1
```

See also: **coeff** (8.24), **subpoly** (8.167), **roundcoefficients** (8.152)

8.36 denominator

Name: **denominator**
gives the denominator of an expression
Usage:

$$\textbf{denominator}(\textit{expr}) : \mathsf{function} \to \mathsf{function}$$

Parameters:

- *expr* represents an expression

Description:

- If *expr* represents a fraction *expr1*/*expr2*, **denominator**(*expr*) returns the denominator of this fraction, i.e. *expr2*.

  If *expr* represents something else, **denominator**(*expr*) returns 1.

  Note that for all expressions *expr*, **numerator**(*expr*) **/ denominator**(*expr*) is equal to *expr*.

Example 1:

```
> denominator(5/3);
3
```

Example 2:

```
> denominator(exp(x));
1
```

Example 3:

```
> a = 5/3;
> b = numerator(a)/denominator(a);
> print(a);
5 / 3
> print(b);
5 / 3
```

Example 4:

```
> a = exp(x/3);
> b = numerator(a)/denominator(a);
> print(a);
exp(x / 3)
> print(b);
exp(x / 3)
```

See also: **numerator** (8.111), **rationalmode** (8.140)

8.37 diam

Name: **diam**
parameter used in safe algorithms of `Sollya` and controlling the maximal length of the involved intervals.
Usage:

$$\textbf{diam} = \textit{width} : \mathsf{constant} \to \mathsf{void}$$
$$\textbf{diam} = \textit{width}\ ! : \mathsf{constant} \to \mathsf{void}$$
$$\textbf{diam} : \mathsf{constant}$$

Parameters:

- *width* represents the maximal relative width of the intervals used

Description:

- **diam** is a global variable. Its value represents the maximal width allowed for intervals involved in safe algorithms of `Sollya` (namely **infnorm**, **checkinfnorm**, **accurateinfnorm**, **integral**, **findzeros**, **supnorm**).
- More precisely, **diam** is relative to the width of the input interval of the command. For instance, suppose that **diam**=1e-5: if **infnorm** is called on interval [0, 1], the maximal width of an interval will be 1e-5. But if it is called on interval [0, 1e−3], the maximal width will be 1e-8.

See also: **infnorm** (8.84), **checkinfnorm** (8.23), **accurateinfnorm** (8.3), **integral** (8.86), **findzeros** (8.63), **supnorm** (8.170)

8.38 dieonerrormode

Name: **dieonerrormode**
global variable controlling if `Sollya` is exited on an error or not.
Usage:

dieonerrormode = *activation value* : on|off → void
dieonerrormode = *activation value* ! : on|off → void
dieonerrormode : on|off

Parameters:

- *activation value* controls if `Sollya` is exited on an error or not.

Description:

- **dieonerrormode** is a global variable. When its value is **off**, which is the default, `Sollya` will not exit on any syntax, typing, side-effect errors. These errors will be caught by the tool, even if a memory might be leaked at that point. On evaluation, the **error** special value will be produced.
- When the value of the **dieonerrormode** variable is **on**, `Sollya` will exit on any syntax, typing, side-effect errors. A warning message will be printed in these cases at appropriate **verbosity** levels.

Example 1:

```
> verbosity = 1!;
> dieonerrormode = off;
Die-on-error mode has been deactivated.
> for i from true to false do i + "Salut";
Warning: one of the arguments of the for loop does not evaluate to a constant.
The for loop will not be executed.
> exp(17);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
2.41549527535752982147754351803858238798675673527224e7
```

Example 2:

```
> verbosity = 1!;
> dieonerrormode = off!;
> 5 */  4;
Warning: syntax error, unexpected DIVTOKEN.
The last symbol read has been "/".
Will skip input until next semicolon after the unexpected token. May leak memory
.
  exp(17);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
2.41549527535752982147754351803858238798675673527224e7
```

Example 3:

```
> verbosity = 1!;
> dieonerrormode;
off
> dieonerrormode = on!;
> dieonerrormode;
on
> for i from true to false do i + "Salut";
Warning: one of the arguments of the for loop does not evaluate to a constant.
The for loop will not be executed.
Warning: some syntax, typing or side-effect error has occurred.
As the die-on-error mode is activated, the tool will be exited.
```

Example 4:

```
> verbosity = 1!;
> dieonerrormode = on!;
> 5 */  4;
Warning: syntax error, unexpected DIVTOKEN.
The last symbol read has been "/".
Will skip input until next semicolon after the unexpected token. May leak memory
.
Warning: some syntax, typing or side-effect error has occurred.
As the die-on-error mode is activated, the tool will be exited.
```

Example 5:

```
> verbosity = 0!;
> dieonerrormode = on!;
> 5 */  4;
```

See also: **on** (8.113), **off** (8.112), **verbosity** (8.183), **error** (8.52)

8.39 diff

Name: **diff**
differentiation operator
Usage:

$$\textbf{diff}(\textit{function}) : \mathsf{function} \rightarrow \mathsf{function}$$

Parameters:

- *function* represents a function

Description:

- **diff**(*function*) returns the symbolic derivative of the function *function* by the global free variable.

  If *function* represents a function symbol that is externally bound to some code by **library**, the derivative is performed as a symbolic annotation to the returned expression tree.

Example 1:

```
> diff(sin(x));
cos(x)
```

Example 2:

```
> diff(x);
1
```

Example 3:

```
> diff(x^x);
x^x * (1 + log(x))
```

See also: **library** (8.91), **autodiff** (8.14), **taylor** (8.174), **taylorform** (8.175)

8.40 dirtyfindzeros

Name: **dirtyfindzeros**
gives a list of numerical values listing the zeros of a function on an interval.
Usage:

dirtyfindzeros(f,I) : (function, range) → list

Parameters:

- f is a function.
- I is an interval.

Description:

- **dirtyfindzeros**(f,I) returns a list containing some zeros of f in the interval I. The values in the list are numerical approximation of the exact zeros. The precision of these approximations is approximately the precision stored in **prec**. If f does not have two zeros very close to each other, it can be expected that all zeros are listed. However, some zeros may be forgotten. This command should be considered as a numerical algorithm and should not be used if safety is critical.
- More precisely, the algorithm relies on global variables **prec** and **points** and it performs the following steps: let n be the value of variable **points** and t be the value of variable **prec**.
  - Evaluate $|f|$ at n evenly distributed points in the interval I. The working precision to be used is automatically chosen in order to ensure that the sign is correct.
  - Whenever f changes its sign for two consecutive points, find an approximation x of its zero with precision t using Newton's algorithm. The number of steps in Newton's iteration depends on t: the precision of the approximation is supposed to be doubled at each step.
  - Add this value to the list.

Example 1:

```
> dirtyfindzeros(sin(x),[-5;5]);
[|-3.14159265358979323846264338327950288419716939937508, 0, 3.141592653589793238
462643383279502884197169399375081]
```

Example 2:

```
> L1=dirtyfindzeros(x^2*sin(1/x),[0;1]);
> points=1000!;
> L2=dirtyfindzeros(x^2*sin(1/x),[0;1]);
> length(L1); length(L2);
18
25
```

See also: **prec** (8.125), **points** (8.120), **findzeros** (8.63), **dirtyinfnorm** (8.41), **numberroots** (8.110)

8.41 dirtyinfnorm

Name: **dirtyinfnorm**
computes a numerical approximation of the infinity norm of a function on an interval.
Usage:

dirtyinfnorm(f,I) : (function, range) → constant

Parameters:

- f is a function.
- I is an interval.

Description:

- **dirtyinfnorm**(f,I) computes an approximation of the infinity norm of the given function f on the interval I, e.g. $\max_{x \in I}\{|f(x)|\}$.
- The interval must be bound. If the interval contains one of -Inf or +Inf, the result of **dirtyinfnorm** is NaN.
- The result of this command depends on the global variables **prec** and **points**. Therefore, the returned result is generally a good approximation of the exact infinity norm, with precision **prec**. However, the result is generally underestimated and should not be used when safety is critical. Use **infnorm** instead.
- The following algorithm is used: let n be the value of variable **points** and t be the value of variable **prec**.
  - Evaluate $|f|$ at n evenly distributed points in the interval I. The evaluation are faithful roundings of the exact results at precision t.
  - Whenever the derivative of f changes its sign for two consecutive points, find an approximation x of its zero with precision t. Then compute a faithful rounding of $|f(x)|$ at precision t.
  - Return the maximum of all computed values.

Example 1:

```
> dirtyinfnorm(sin(x),[-10;10]);
1
```

Example 2:

```
> prec=15!;
> dirtyinfnorm(exp(cos(x))*sin(x),[0;5]);
1.45856
> prec=40!;
> dirtyinfnorm(exp(cos(x))*sin(x),[0;5]);
1.458528537135
> prec=100!;
> dirtyinfnorm(exp(cos(x))*sin(x),[0;5]);
1.458528537136237644438147455024
> prec=200!;
> dirtyinfnorm(exp(cos(x))*sin(x),[0;5]);
1.458528537136237644438147455023841718299214087993682374094153
```

Example 3:

```
> dirtyinfnorm(x^2, [log(0);log(1)]);
@NaN@
```

See also: **prec** (8.125), **points** (8.120), **infnorm** (8.84), **checkinfnorm** (8.23), **supnorm** (8.170)

8.42 dirtyintegral

Name: **dirtyintegral**
computes a numerical approximation of the integral of a function on an interval.
Usage:

dirtyintegral(*f*,*I*) : (function, range) → constant

Parameters:

- *f* is a function.
- *I* is an interval.

Description:

- **dirtyintegral**(*f*,*I*) computes an approximation of the integral of *f* on *I*.
- The interval must be bound. If the interval contains one of -Inf or +Inf, the result of **dirtyintegral** is NaN, even if the integral has a meaning.
- The result of this command depends on the global variables **prec** and **points**. The method used is the trapezium rule applied at n evenly distributed points in the interval, where n is the value of global variable **points**.
- This command computes a numerical approximation of the exact value of the integral. It should not be used if safety is critical. In this case, use command **integral** instead.
- Warning: this command is currently known to be unsatisfactory. If you really need to compute integrals, think of using an other tool or report a feature request to sylvain.chevillard@ens-lyon.org.

Example 1:

```
> sin(10);
-0.544021110889369813404747661851377281683643012916 21
> dirtyintegral(cos(x),[0;10]);
-0.544003049051526298224480588824753820365362983562 81
> points=2000!;
> dirtyintegral(cos(x),[0;10]);
-0.544019977511583219722226973125831990359958379268 92
```

See also: **prec** (8.125), **points** (8.120), **integral** (8.86)

8.43 display

Name: **display**
sets or inspects the global variable specifying number notation
Usage:

display = *notation value* : decimal|binary|dyadic|powers|hexadecimal → void
display = *notation value* ! : decimal|binary|dyadic|powers|hexadecimal → void
display : decimal|binary|dyadic|powers|hexadecimal

Parameters:

- *notation value* represents a variable of type decimal|binary|dyadic|powers|hexadecimal

Description:

- An assignment **display** = *notation value*, where *notation value* is one of **decimal**, **dyadic**, **powers**, **binary** or **hexadecimal**, activates the corresponding notation for output of values in **print**, **write** or at the `Sollya` prompt.

  If the global notation variable **display** is **decimal**, all numbers will be output in scientific decimal notation. If the global notation variable **display** is **dyadic**, all numbers will be output as dyadic numbers with Gappa notation. If the global notation variable **display** is **powers**, all numbers will be output as dyadic numbers with a notation compatible with Maple and PARI/GP. If the global notation variable **display** is **binary**, all numbers will be output in binary notation. If the global notation variable **display** is **hexadecimal**, all numbers will be output in C99/ IEEE754-2008 notation. All output notations can be parsed back by `Sollya`, inducing no error if the input and output precisions are the same (see **prec**).

  If the assignment **display** = *notation value* is followed by an exclamation mark, no message indicating the new state is displayed. Otherwise the user is informed of the new state of the global mode by an indication.

Example 1:

```
> display = decimal;
Display mode is decimal numbers.
> a = evaluate(sin(pi * x), 0.25);
> a;
0.707106781186547524400844362104849039284835937688 47
> display = binary;
Display mode is binary numbers.
> a;
1.01101010000010011110011001100111111100111011110011001001000010001011001011111 0
11000100110110011011101010100101010111110100111110001110101101111011000001011101
010001_2 * 2^(-1)
> display = hexadecimal;
Display mode is hexadecimal numbers.
> a;
0xb.504f333f9de6484597d89b3754abe9f1d6f60ba88p-4
> display = dyadic;
Display mode is dyadic numbers.
> a;
330700069911015586133239834882209443600671071332 65b-165
> display = powers;
Display mode is dyadic numbers in integer-power-of-2 notation.
> a;
330700069911015586133239834882209443600671071332 65 * 2^(-165)
```

See also: **print** (8.128), **write** (8.186), **decimal** (8.33), **dyadic** (8.48), **powers** (8.124), **binary** (8.18), **hexadecimal** (8.75), **prec** (8.125)

8.44 /

Name: /
division function
Usage:

$$\begin{aligned} \textit{function1} \ / \ \textit{function2} &: (\mathsf{function}, \mathsf{function}) \to \mathsf{function} \\ \textit{interval1} \ / \ \textit{interval2} &: (\mathsf{range}, \mathsf{range}) \to \mathsf{range} \\ \textit{interval1} \ / \ \textit{constant} &: (\mathsf{range}, \mathsf{constant}) \to \mathsf{range} \\ \textit{interval1} \ / \ \textit{constant} &: (\mathsf{constant}, \mathsf{range}) \to \mathsf{range} \end{aligned}$$

Parameters:

- *function1* and *function2* represent functions
- *interval1* and *interval2* represent intervals (ranges)
- *constant* represents a constant or constant expression

Description:

- / represents the division (function) on reals. The expression *function1* / *function2* stands for the function composed of the division function and the two functions *function1* and *function2*, where *function1* is the numerator and *function2* the denominator.
- / can be used for interval arithmetic on intervals (ranges). / will evaluate to an interval that safely encompasses all images of the division function with arguments varying in the given intervals. If the intervals given contain points where the division function is not defined, infinities and NaNs will be produced in the output interval. Any combination of intervals with intervals or constants (resp. constant expressions) is supported. However, it is not possible to represent families of functions using an interval as one argument and a function (varying in the free variable) as the other one.

Example 1:

```
> 5 / 2;
2.5
```

Example 2:

```
> x / 2;
x * 0.5
```

Example 3:

```
> x / x;
1
```

Example 4:

```
> 3 / 0;
@NaN@
```

Example 5:

```
> diff(sin(x) / exp(x));
(exp(x) * cos(x) - sin(x) * exp(x)) / exp(x)^2
```

Example 6:

```
> [1;2] / [3;4];
[0.25;0.66666666666666666666666666666666666666666666666668]
> [1;2] / 17;
[5.88235294117647058823529411764705882352941176470588e-2;0.11764705882352941176470
588235294117647058823529412]
> -13 / [4;17];
[-3.25;-0.76470588235294117647058823529411764705882352941175]
```

See also: + (8.119), − (8.104), ∗ (8.105), ^ (8.123)

8.45 double

Names: **double**, **D**
rounding to the nearest IEEE 754 double (binary64).
Description:

- **double** is both a function and a constant.
- As a function, it rounds its argument to the nearest IEEE 754 double precision (i.e. IEEE754-2008 binary64) number. Subnormal numbers are supported as well as standard numbers: it is the real rounding described in the standard.
- As a constant, it symbolizes the double precision format. It is used in contexts when a precision format is necessary, e.g. in the commands **round**, **roundcoefficients** and **implementpoly**. See the corresponding help pages for examples.

Example 1:

```
> display=binary!;
> D(0.1);
1.10011001100110011001100110011001100110011001100110011001101_2 * 2^(-4)
> D(4.17);
1.00001010111000010100011110101110000101000111101011 1_2 * 2^(2)
> D(1.011_2 * 2^(-1073));
1.1_2 * 2^(-1073)
```

See also: **halfprecision** (8.73), **single** (8.162), **doubleextended** (8.47), **doubledouble** (8.46), **quad** (8.136), **tripledouble** (8.180), **roundcoefficients** (8.152), **implementpoly** (8.81), **round** (8.151), **printdouble** (8.129)

8.46 doubledouble

Names: **doubledouble**, **DD**
represents a number as the sum of two IEEE doubles.
Description:

- **doubledouble** is both a function and a constant.
- As a function, it rounds its argument to the nearest number that can be written as the sum of two double precision numbers.
- The algorithm used to compute **doubledouble**(x) is the following: let x_h = **double**(x) and let x_l = **double**($x - x_h$). Return the number $x_h + x_l$. Note that if the current precision is not sufficient to exactly represent $x_h + x_l$, a rounding will occur and the result of **doubledouble**(x) will be useless.
- As a constant, it symbolizes the double-double precision format. It is used in contexts when a precision format is necessary, e.g. in the commands **round**, **roundcoefficients** and **implementpoly**. See the corresponding help pages for examples.

Example 1:

```
> verbosity=1!;
> a = 1+ 2^(-100);
> DD(a);
1.00000000000000000000000000000078886090522101180 54
> prec=50!;
> DD(a);
Warning: double rounding occurred on invoking the double-double rounding operato
r.
Try to increase the working precision.
1
```

See also: **halfprecision** (8.73), **single** (8.162), **double** (8.45), **doubleextended** (8.47), **quad** (8.136), **tripledouble** (8.180), **roundcoefficients** (8.152), **implementpoly** (8.81), **round** (8.151)

8.47 doubleextended

Names: **doubleextended**, **DE**
computes the nearest number with 64 bits of mantissa.
Description:

- **doubleextended** is a function that computes the nearest floating-point number with 64 bits of mantissa to a given number. Since it is a function, it can be composed with other `Sollya` functions such as **exp**, **sin**, etc.
- **doubleextended** now does handle subnormal numbers for a presumed exponent width of the double-extended format of 15 bits. This means, with respect to rounding, **doubleextended** behaves as a IEEE 754-2008 binary79 with a 64 bit significand (with a hidden bit normal range), one sign bit and a 15 bit exponent field would behave. This behavior may be different from the one observed on Intel-based IA32/Intel64 processors (or compatible versions from other vendors). However it is the one seen on HP/Intel Itanium when the precision specifier is double-extended and pseudo-denormals are activated.
- Since it is a function and not a command, its behavior is a bit different from the behavior of **round**(x,64,RN) even if the result is exactly the same. **round**(x,64,RN) is immediately evaluated whereas **doubleextended**(x) can be composed with other functions (and thus be plotted and so on).

Example 1:

```
> display=binary!;
> DE(0.1);
1.10011001100110011001100110011001100110011001100110011001100110011001101_2 * 2^(-4)
> round(0.1,64,RN);
1.10011001100110011001100110011001100110011001100110011001100110011001101_2 * 2^(-4)
```

Example 2:

```
> D(2^(-2000));
0
> DE(2^(-20000));
0
```

Example 3:

```
> verbosity=1!;
> f = sin(DE(x));
> f(pi);
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
-5.0165576126683320235573270803307570138315616702549e-20
> g = sin(round(x,64,RN));
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
```

See also: **roundcoefficients** (8.152), **halfprecision** (8.73), **single** (8.162), **double** (8.45), **doubledouble** (8.46), **quad** (8.136), **tripledouble** (8.180), **round** (8.151)

8.48 dyadic

Name: **dyadic**
special value for global state **display**
Description:

- **dyadic** is a special value used for the global state **display**. If the global state **display** is equal to **dyadic**, all data will be output in dyadic notation with numbers displayed in Gappa format.

  As any value it can be affected to a variable and stored in lists.

See also: **decimal** (8.33), **powers** (8.124), **hexadecimal** (8.75), **binary** (8.18), **display** (8.43)

8.49 ==

Name: ==
equality test operator
Usage:

$$expr1 == expr2 : (\text{any type}, \text{any type}) \rightarrow \text{boolean}$$

Parameters:

- *expr1* and *expr2* represent expressions

Description:

- The operator == evaluates to true iff its operands *expr1* and *expr2* are syntactically equal and different from **error** or constant expressions that are not constants and that evaluate to the same floating-point number with the global precision **prec**. The user should be aware of the fact that because of floating-point evaluation, the operator == is not exactly the same as the mathematical equality. Further remark that according to IEEE 754-2008 floating-point rules, which `Sollya` emulates, floating-point data which are NaN do not compare equal to any other floating-point datum, including NaN.

Example 1:

```
> "Hello" == "Hello";
true
> "Hello" == "Salut";
false
> "Hello" == 5;
false
> 5 + x == 5 + x;
true
```

Example 2:

```
> 1 == exp(0);
true
> asin(1) * 2 == pi;
true
> exp(5) == log(4);
false
```

Example 3:

```
> sin(pi/6) == 1/2 * sqrt(3);
false
```

Example 4:

```
> prec = 12;
The precision has been set to 12 bits.
> 16384.1 == 16385.1;
true
```

Example 5:

```
> error == error;
false
```

Example 6:

```
> a = "Biba";
> b = NaN;
> a == a;
true
> b == b;
false
```

See also: **!=** (8.107), > (8.71), >= (8.70), <= (8.89), < (8.98), **in** (8.82), **!** (8.109), && (8.6), || (8.114), **error** (8.52), **prec** (8.125)

8.50 erf

Name: **erf**
the error function.
Description:

- **erf** is the error function defined by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t.$$

- It is defined for every real number x.

See also: **erfc** (8.51), **exp** (8.55)

8.51 erfc

Name: **erfc**
the complementary error function.
Description:

- **erfc** is the complementary error function defined by $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.
- It is defined for every real number x.

See also: **erf** (8.50)

8.52 error

Name: **error**
expression representing an input that is wrongly typed or that cannot be executed
Usage:

error : error

Description:

- The variable **error** represents an input during the evaluation of which a type or execution error has been detected or is to be detected. Inputs that are syntactically correct but wrongly typed evaluate to **error** at some stage. Inputs that are correctly typed but containing commands that depend on side-effects that cannot be performed or inputs that are wrongly typed at meta-level (cf. **parse**), evaluate to **error**.

  Remark that in contrast to all other elements of the `Sollya` language, **error** compares neither equal nor unequal to itself. This provides a means of detecting syntax errors inside the `Sollya` language itself without introducing issues of two different wrongly typed inputs being equal.

Example 1:

```
> print(5 + "foo");
error
```

Example 2:

```
> error;
error
```

Example 3:

```
> error == error;
false
> error != error;
false
```

Example 4:

```
> correct = 5 + 6;
> incorrect = 5 + "foo";
> correct == correct;
true
> incorrect == incorrect;
false
> errorhappened = !(incorrect == incorrect);
> errorhappened;
true
```

See also: **void** (8.184), **parse** (8.115), == (8.49), **!**= (8.107)

8.53 evaluate

Name: **evaluate**
evaluates a function at a constant point or in a range
Usage:

evaluate(*function*, *constant*) : (function, constant) → constant | range
evaluate(*function*, *range*) : (function, range) → range
evaluate(*function*, *function2*) : (function, function) → function

Parameters:

- *function* represents a function
- *constant* represents a constant point
- *range* represents a range
- *function2* represents a function that is not constant

Description:

- If its second argument is a constant *constant*, **evaluate** evaluates its first argument *function* at the point indicated by *constant*. This evaluation is performed in a way that the result is a faithful rounding of the real value of the *function* at *constant* to the current global precision. If such a faithful rounding is not possible, **evaluate** returns a range surely encompassing the real value of the function *function* at *constant*. If even interval evaluation is not possible because the expression is undefined or numerically unstable, NaN will be produced.
- If its second argument is a range *range*, **evaluate** evaluates its first argument *function* by interval evaluation on this range *range*. This ensures that the image domain of the function *function* on the preimage domain *range* is surely enclosed in the returned range.
- In the case when the second argument is a range that is reduced to a single point (such that [1; 1] for instance), the evaluation is performed in the same way as when the second argument is a constant but it produces a range as a result: **evaluate** automatically adjusts the precision of the intern computations and returns a range that contains at most three floating-point consecutive numbers in precision **prec**. This correponds to the same accuracy as a faithful rounding of the actual result. If such a faithful rounding is not possible, **evaluate** has the same behavior as in the case when the second argument is a constant.
- If its second argument is a function *function2* that is not a constant, **evaluate** replaces all occurrences of the free variable in function *function* by function *function2*.

Example 1:

```
> midpointmode=on!;
> print(evaluate(sin(pi * x), 2.25));
0.70710678118654752440084436210484903928483593768847
> print(evaluate(sin(pi * x), [2.25; 2.25]));
0.70710678118654752440084436210484903928483593768688~4/5~
```

Example 2:

```
> print(evaluate(sin(pi * x), 2));
[-1.72986452514381269516508615031098129542836767991679e-12715;7.5941198201187963
145069564314525661706039084390067e-12716]
```

Example 3:

```
> print(evaluate(sin(pi * x), [2, 2.25]));
[-5.14339027267725463004699891996191240734922416542 1e-50;0.707106781186547524400
844362104849039284835937688 66]
```

Example 4:

```
> print(evaluate(sin(pi * x), 2 + 0.25 * x));
sin((pi) * (2 + 0.25 * x))
```

Example 5:

```
> print(evaluate(sin(pi * 1/x), 0));
[@NaN@;@NaN@]
```

See also: **isevaluable** (8.88)

8.54 execute

Name: **execute**
executes the content of a file
Usage:

$$\textbf{execute}(\textit{filename}) : \mathsf{string} \rightarrow \mathsf{void}$$

Parameters:

- *filename* is a string representing a file name

Description:

- **execute** opens the file indicated by *filename*, and executes the sequence of commands it contains. This command is evaluated at execution time: this way you can modify the file *filename* (for instance using **bashexecute**) and execute it just after.
- If *filename* contains a command **execute**, it will be executed recursively.
- If *filename* contains a call to **restart**, it will be neglected.
- If *filename* contains a call to **quit**, the commands following **quit** in *filename* will be neglected.

Example 1:

```
> a=2;
> a;
2
> print("a=1;") > "example.sollya";
> execute("example.sollya");
> a;
1
```

Example 2:

```
> verbosity=1!;
> print("a=1; restart; a=2;") > "example.sollya";
> execute("example.sollya");
Warning: a restart command has been used in a file read into another.
This restart command will be neglected.
> a;
2
```

Example 3:

```
> verbosity=1!;
> print("a=1; quit; a=2;") > "example.sollya";
> execute("example.sollya");
Warning: the execution of a file read by execute demanded stopping the interpret
ation but it is not stopped.
> a;
1
```

See also: **parse** (8.115), **readfile** (8.142), **write** (8.186), **print** (8.128), **bashexecute** (8.17), **quit** (8.137), **restart** (8.147)

8.55 exp

Name: **exp**
the exponential function.
Description:

- **exp** is the usual exponential function defined as the solution of the ordinary differential equation $y' = y$ with $y(0) = 1$.
- **exp**(x) is defined for every real number x.

See also: **exp** (8.55), **log** (8.94)

8.56 expand

Name: **expand**
expands polynomial subexpressions
Usage:

expand(*function*) : function → function

Parameters:

- *function* represents a function

Description:

- **expand**(*function*) expands all polynomial subexpressions in function *function* as far as possible. Factors of sums are multiplied out, power operators with constant positive integer exponents are replaced by multiplications and divisions are multiplied out, i.e. denomiators are brought at the most interior point of expressions.

Example 1:

```
> print(expand(x^3));
x * x * x
```

Example 2:

```
> print(expand((x + 2)^3 + 2 * x));
8 + 12 * x + 6 * x * x + x * x * x + 2 * x
```

Example 3:

```
> print(expand(exp((x + (x + 3))^5)));
exp(243 + 405 * x + 270 * x * x + 90 * x * x * x + 15 * x * x * x * x + x * x *
x * x * x + x * 405 + 108 * x * 5 * x + 54 * x * x * 5 * x + 12 * x * x * x * 5
* x + x * x * x * x * 5 * x + x * x * 270 + 27 * x * x * x * 10 + 9 * x * x * x
* x * 10 + x * x * x * x * x * 10 + x * x * x * 90 + 6 * x * x * x * x * 10 + x
* x * x * x * x * 10 + x * x * x * x * 5 * x + 15 * x * x * x * x * x + x * x *
x * x)
```

See also: **simplify** (8.159), **simplifysafe** (8.160), **horner** (8.78), **coeff** (8.24), **degree** (8.35)

8.57 expm1

Name: **expm1**
shifted exponential function.
Description:

- **expm1** is defined by $\text{expm1}(x) = \exp(x) - 1$.
- It is defined for every real number x.

See also: **exp** (8.55)

8.58 exponent

Name: **exponent**
returns the scaled binary exponent of a number.
Usage:

exponent(x) : constant → integer

Parameters:

- x is a dyadic number.

Description:

- **exponent**(x) is by definition 0 if x is one of 0, NaN, or Inf.
- If x is not zero, it can be uniquely written as $x = m \cdot 2^e$ where m is an odd integer and e is an integer. **exponent**(x) returns e.

Example 1:

```
> a=round(Pi,20,RN);
> e=exponent(a);
> e;
-17
> m=mantissa(a);
> a-m*2^e;
0
```

See also: **mantissa** (8.99), **precision** (8.126)

8.59 externalplot

Name: **externalplot**
plots the error of an external code with regard to a function
Usage:

externalplot(*filename*, *mode*, *function*, *range*, *precision*) : (string, absolute|relative, function, range, integer) → void
externalplot(*filename*, *mode*, *function*, *range*, *precision*, *perturb*) : (string, absolute|relative, function, range, integer, perturb) → void
externalplot(*filename*, *mode*, *function*, *range*, *precision*, *plot mode*, *result filename*) : (string, absolute|relative, function, range, integer, file|postscript|postscriptfile, string) → void
externalplot(*filename*, *mode*, *function*, *range*, *precision*, *perturb*, *plot mode*, *result filename*) : (string, absolute|relative, function, range, integer, perturb, file|postscript|postscriptfile, string) → void

Description:

- The command **externalplot** plots the error of an external function evaluation code sequence implemented in the object file named *filename* with regard to the function *function*. If *mode* evaluates to *absolute*, the difference of both functions is considered as an error function; if *mode* evaluates to *relative*, the difference is divided by the function *function*. The resulting error function is plotted on all floating-point numbers with *precision* significant mantissa bits in the range *range*.

  If the sixth argument of the command **externalplot** is given and evaluates to **perturb**, each of the floating-point numbers the function is evaluated at gets perturbed by a random value that is uniformly distributed in ±1 ulp around the original *precision* bit floating-point variable.

  If a sixth and seventh argument, respectively a seventh and eighth argument in the presence of **perturb** as a sixth argument, are given that evaluate to a variable of type file|postscript|postscriptfile respectively to a character sequence of type string, **externalplot** will plot (additionally) to a file in the same way as the command **plot** does. See **plot** for details.

The external function evaluation code given in the object file name *filename* is supposed to define a function name `f` as follows (here in C syntax): `void f(mpfr_t rop, mpfr_ op)`. This function is supposed to evaluate `op` with an accuracy corresponding to the precision of `rop` and assign this value to `rop`.

Example 1:

```
> bashexecute("gcc -fPIC -c externalplotexample.c");
> bashexecute("gcc -shared -o externalplotexample externalplotexample.o -lgmp -l
mpfr");
> externalplot("./externalplotexample",relative,exp(x),[-1/2;1/2],12,perturb);
```

See also: **plot** (8.118), **asciiplot** (8.9), **perturb** (8.116), **absolute** (8.2), **relative** (8.144), **file** (8.62), **postscript** (8.121), **postscriptfile** (8.122), **bashexecute** (8.17), **externalproc** (8.60), **library** (8.91)

8.60 externalproc

Name: **externalproc**
binds an external code to a `Sollya` procedure
Usage:

externalproc(*identifier*, *filename*, *argumenttype* $->$ *resulttype*) : (identifier type, string, type type, type type) $\rightarrow$ void

Parameters:

- *identifier* represents the identifier the code is to be bound to
- *filename* of type string represents the name of the object file where the code of procedure can be found
- *argumenttype* represents a definition of the types of the arguments of the `Sollya` procedure and the external code
- *resulttype* represents a definition of the result type of the external code

Description:

- **externalproc** allows for binding the `Sollya` identifier *identifier* to an external code. After this binding, when `Sollya` encounters *identifier* applied to a list of actual parameters, it will evaluate these parameters and call the external code with these parameters. If the external code indicated success, it will receive the result produced by the external code, transform it to `Sollya`'s internal representation and return it.

  In order to allow correct evaluation and typing of the data in parameter and in result to be passed to and received from the external code, **externalproc** has a third parameter *argumenttype* $->$ *resulttype*. Both *argumenttype* and *resulttype* are one of **void**, **constant**, **function**, **range**, **integer**, **string**, **boolean**, **list of constant**, **list of function**, **list of range**, **list of integer**, **list of string**, **list of boolean**.

  If upon a usage of a procedure bound to an external procedure the type of the actual parameters given or its number is not correct, `Sollya` produces a type error. An external function not applied to arguments represents itself and prints out with its argument and result types.

  The external function is supposed to return an integer indicating success. It returns its result depending on its `Sollya` result type as follows. Here, the external procedure is assumed to be implemented as a C function.

  If the `Sollya` result type is void, the C function has no pointer argument for the result. If the `Sollya` result type is **constant**, the first argument of the C function is of C type `mpfr_t *`, the result is returned by affecting the MPFR variable. If the `Sollya` result type is **function**, the first argument of the C function is of C type `node **`, the result is returned by the `node *` pointed with

a new `node *`. If the `Sollya` result type is **range**, the first argument of the C function is of C type `mpfi_t *`, the result is returned by affecting the interval-arithmetic variable. If the `Sollya` result type is **integer**, the first argument of the C function is of C type `int *`, the result is returned by affecting the int variable. If the `Sollya` result type is **string**, the first argument of the C function is of C type `char **`, the result is returned by the `char *` pointed with a new `char *`. If the `Sollya` result type is **boolean**, the first argument of the C function is of C type `int *`, the result is returned by affecting the int variable with a boolean value. If the `Sollya` result type is **list of** type, the first argument of the C function is of C type `chain **`, the result is returned by the `chain *` pointed with a new `chain *`. This chain contains for `Sollya` type **constant** pointers `mpfr_t *` to new MPFR variables, for `Sollya` type **function** pointers `node *` to new nodes, for `Sollya` type **range** pointers `mpfi_t *` to new interval-arithmetic variables, for `Sollya` type **integer** pointers `int *` to new int variables for `Sollya` type **string** pointers `char *` to new `char *` variables and for `Sollya` type **boolean** pointers `int *` to new int variables representing boolean values.

The external procedure affects its possible pointer argument if and only if it succeeds. This means, if the function returns an integer indicating failure, it does not leak any memory to the encompassing environment.

The external procedure receives its arguments as follows: If the `Sollya` argument type is **void**, no argument array is given. Otherwise the C function receives a C `void **` argument representing an array of size equal to the arity of the function where each entry (of C type `void *`) represents a value with a C type depending on the corresponding `Sollya` type. If the `Sollya` type is **constant**, the C type the `void *` is to be casted to is `mpfr_t *`. If the `Sollya` type is **function**, the C type the `void *` is to be casted to is `node *`. If the `Sollya` type is **range**, the C type the `void *` is to be casted to is `mpfi_t *`. If the `Sollya` type is **integer**, the C type the `void *` is to be casted to is `int *`. If the `Sollya` type is **string**, the C type the `void *` is to be casted to is `char *`. If the `Sollya` type is **boolean**, the C type the `void *` is to be casted to is `int *`. If the `Sollya` type is **list of** type, the C type the `void *` is to be casted to is `chain *`. Here depending on type, the values in the chain are to be casted to `mpfr_t *` for `Sollya` type **constant**, `node *` for `Sollya` type **function**, `mpfi_t *` for `Sollya` type **range**, `int *` for `Sollya` type **integer**, `char *` for `Sollya` type **string** and `int *` for `Sollya` type **boolean**.

The external procedure is not supposed to alter the memory pointed by its array argument `void **`.

In both directions (argument and result values), empty lists are represented by `chain * NULL` pointers.

In contrast to internal procedures, externally bounded procedures can be considered to be objects inside `Sollya` that can be assigned to other variables, stored in list etc.

Example 1:

```
> bashexecute("gcc -fPIC -Wall -c externalprocexample.c");
> bashexecute("gcc -fPIC -shared -o externalprocexample externalprocexample.o");

> externalproc(foo, "./externalprocexample", (integer, integer) -> integer);
> foo;
foo(integer, integer) -> integer
> foo(5, 6);
11
> verbosity = 1!;
> foo();
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
> a = foo;
> a(5,6);
11
```

See also: **library** (8.91), **libraryconstant** (8.92), **externalplot** (8.59), **bashexecute** (8.17), **void** (8.184), **constant** (8.27), **function** (8.69), **range** (8.138), **integer** (8.85), **string** (8.166), **boolean** (8.20), **list of** (8.93)

8.61 false

Name: **false**
the boolean value representing the false.
Description:

- **false** is the usual boolean value.

Example 1:

```
> true && false;
false
> 2<1;
false
```

See also: **true** (8.181), && (8.6), || (8.114)

8.62 file

Name: **file**
special value for commands **plot** and **externalplot**
Description:

- **file** is a special value used in commands **plot** and **externalplot** to save the result of the command in a data file.
- As any value it can be affected to a variable and stored in lists.

Example 1:

```
> savemode=file;
> name="plotSinCos";
> plot(sin(x),0,cos(x),[-Pi,Pi],savemode, name);
```

See also: **externalplot** (8.59), **plot** (8.118), **postscript** (8.121), **postscriptfile** (8.122)

8.63 findzeros

Name: **findzeros**
gives a list of intervals containing all zeros of a function on an interval.
Usage:

$$\textbf{findzeros}(f,I) : (\mathsf{function}, \mathsf{range}) \to \mathsf{list}$$

Parameters:

- f is a function.
- I is an interval.

Description:

- **findzeros**(f,I) returns a list of intervals $I_1, \ldots, I_n$ such that, for every zero z of f, there exists some k such that $z \in I_k$.
- The list may contain intervals I_k that do not contain any zero of f. An interval Ik may contain many zeros of f.

- This command is meant for cases when safety is critical. If you want to be sure not to forget any zero, use **findzeros**. However, if you just want to know numerical values for the zeros of f, **dirtyfindzeros** should be quite satisfactory and a lot faster.
- If δ denotes the value of global variable **diam**, the algorithm ensures that for each k, $|I_k| \leq \delta \cdot |I|$.
- The algorithm used is basically a bisection algorithm. It is the same algorithm that the one used for **infnorm**. See the help page of this command for more details. In short, the behavior of the algorithm depends on global variables **prec**, **diam**, **taylorrecursions** and **hopitalrecursions**.

Example 1:

```
> findzeros(sin(x),[-5;5]);
[|[-3.14208984375;-3.140869140625], [-1.220703125e-3;1.220703125e-3], [3.1408691
40625;3.14208984375]|]
> diam=1e-10!;
> findzeros(sin(x),[-5;5]);
[|[-3.14159265370108187198638916015625;-3.14159265253692865371704101562 5], [-1.1
64153218269348144531 25e-9;1.164153218269348144531 25e-9], [3.14159265253692865371
7041015625;3.14159265370108187198638916015625]|]
```

See also: **dirtyfindzeros** (8.40), **infnorm** (8.84), **prec** (8.125), **diam** (8.37), **taylorrecursions** (8.176), **hopitalrecursions** (8.77), **numberroots** (8.110)

8.64 fixed

Name: **fixed**
indicates that fixed-point formats should be used for **fpminimax**
Usage:

fixed : fixed|floating

Description:

- The use of **fixed** in the command **fpminimax** indicates that the list of formats given as argument is to be considered to be a list of fixed-point formats. See **fpminimax** for details.

Example 1:

```
> fpminimax(cos(x),6,[|32,32,32,32,32,32,32|],[-1;1],fixed);
0.99999974807724356651306152343 75 + x^2 * (-0.49999286932870745658874511718 75 +
x^2 * (4.16335149202495813369750976562 5e-2 + x^2 * (-1.33822392672300338745117187 5e-3)))
```

See also: **fpminimax** (8.67), **floating** (8.65)

8.65 floating

Name: **floating**
indicates that floating-point formats should be used for **fpminimax**
Usage:

floating : fixed|floating

Description:

- The use of **floating** in the command **fpminimax** indicates that the list of formats given as argument is to be considered to be a list of floating-point formats. See **fpminimax** for details.

Example 1:

```
> fpminimax(cos(x),6,[|D...|],[-1;1],floating);
0.99999974816012948686250183527590706944465637207031 + x * (5.521004406122249513
178203580244316832191390012 6185e-14 + x * (-0.499992869801976880239635647740215 0
630950927734375 + x * (-3.95371609372064761555136192612768146546591008227978e-13
 + x * (4.16335155285858099505347240665287245064973831176758e-2 + x * (5.2492670
39583512274801498093883432767038643707024 9e-13 + x * (-1.33822408807599468535953
76836665309383533895015 7166e-3))))))
```

See also: **fpminimax** (8.67), **fixed** (8.64)

8.66 floor

Name: **floor**
the usual function floor.
Description:

- **floor** is defined as usual: **floor**(x) is the greatest integer y such that $y \leq x$.
- It is defined for every real number x.

See also: **ceil** (8.22), **nearestint** (8.106), **round** (8.151), **RD** (8.141)

8.67 fpminimax

Name: **fpminimax**
computes a good polynomial approximation with fixed-point or floating-point coefficients
Usage:

fpminimax(*f*, *n*, *formats*, *range*, *indic1*, *indic2*, *indic3*, *P*) : (function, integer, list, range, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, function) → function
fpminimax(*f*, *monomials*, *formats*, *range*, *indic1*, *indic2*, *indic3*, *P*) : (function, list, list, range, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, function) → function
fpminimax(*f*, *n*, *formats*, *L*, *indic1*, *indic2*, *indic3*, *P*) : (function, integer, list, list, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, function) → function
fpminimax(*f*, *monomials*, *formats*, *L*, *indic1*, *indic2*, *indic3*, *P*) : (function, list, list, list, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, absolute|relative | fixed|floating | function, function) → function

Parameters:

- f is the function to be approximated
- n is the degree of the polynomial that must approximate f
- *monomials* is a list of integers or a list of function. It indicates the basis for the approximation of f
- *formats* is a list indicating the formats that the coefficients of the polynomial must have
- *range* is the interval where the function must be approximated
- L is a list of interpolation points used by the method
- *indic1* (optional) is one of the optional indication parameters. See the detailed description below.
- *indic2* (optional) is one of the optional indication parameters. See the detailed description below.
- *indic3* (optional) is one of the optional indication parameters. See the detailed description below.

- P (optional) is the minimax polynomial to be considered for solving the problem.

Description:

- **fpminimax** uses a heuristic (but practically efficient) method to find a good polynomial approximation of a function f on an interval *range*. It implements the method published in the article:
  Efficient polynomial L^∞-approximations
  Nicolas Brisebarre and Sylvain Chevillard
  Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH 18)
  pp. 169-176

- The basic usage of this command is **fpminimax**(*f*, *n*, *formats*, *range*). It computes a polynomial approximation of f with degree at most n on the interval *range*. *formats* is a list of integers or format types (such as **double**, **doubledouble**, etc.). The polynomial returned by the command has its coefficients that fit the formats indications. For instance, if formats[0] is 35, the coefficient of degree 0 of the polynomial will fit a floating-point format of 35 bits. If formats[1] is D, the coefficient of degree 1 will be representable by a floating-point number with a precision of 53 bits (which is not necessarily an IEEE 754 double precision number. See the remark below), etc.

- The second argument may be either an integer, a list of integers or a list of functions. An integer indicates the degree of the desired polynomial approximation. A list of integers indicates the list of desired monomials. For instance, the list [|0, 2, 4, 6|] indicates that the polynomial must be even and of degree at most 6. Giving an integer n as second argument is equivalent as giving [|0, ..., n|]. Finally, a list of function g_k indicates that the desired approximation must be a linear combination of the g_k.
  The list of formats is interpreted with respect to the list of monomials. For instance, if the list of monomials is [|0, 2, 4, 6|] and the list of formats is [|161, 107, 53, 24|], the coefficients of degree 0 is searched as a floating-point number with precision 161, the coefficient of degree 2 is searched as a number of precision 107, and so on.

- The list of formats may contain either integers or format types (**halfprecision**, **single**, **double**, **doubledouble**, **tripledouble**, **doubleextended** and **quad**). The list may be too large or even infinite. Only the first indications will be considered. For instance, for a degree n polynomial, formats[$n+1$] and above will be discarded. This lets one use elliptical indications for the last coefficients.

- The floating-point coefficients considered by **fpminimax** do not have an exponent range. In particular, in the format list, **double** is an exact synonym for 53. Currently, **fpminimax** only ensures that the corresponding coefficient has at most 53 bits of mantissa. It does not imply that it is an IEEE-754 double.

- By default, the list of formats is interpreted as a list of floating-point formats. This may be changed by passing **fixed** as an optional argument (see below). Let us take an example: **fpminimax**(f, 2, [107, DD, 53], [0;1]). Here the optional argument is missing (we could have set it to **floating**). Thus, **fpminimax** will search for a polynomial of degree 2 with a constant coefficient that is a 107 bits floating-point number, etc.
  Currently, **doubledouble** is just a synonym for 107 and **tripledouble** a synonym for 161. This behavior may change in the future (taking into account the fact that some double-doubles are not representable with 107 bits).
  Second example: **fpminimax**(f, 2, [25, 18, 30], [0;1], **fixed**). In this case, **fpminimax** will search for a polynomial of degree 2 with a constant coefficient of the form $m/2^{25}$ where m is an integer. In other words, it is a fixed-point number with 25 bits after the point. Note that even with argument **fixed**, the formats list is allowed to contain **halfprecision**, **single**, **double**, **doubleextended**, **doubledouble**, **quad** or **tripledouble**. In this this case, it is just a synonym for 11, 24, 53, 64, 107, 113 or 161. This is deprecated and may change in the future.

- The fourth argument may be a range or a list. Lists are for advanced users that know what they are doing. The core of the method is a kind of approximated interpolation. The list given here is a list of points that must be considered for the interpolation. It must contain at least as many points

as unknown coefficients. If you give a list, it is also recommended that you provide the minimax polynomial as last argument. If you give a range, the list of points will be automatically computed.

- The fifth, sixth and seventh arguments are optional. By default, **fpminimax** will approximate f while optimizing the relative error, and interpreting the list of formats as a list of floating-point formats.
  This default behavior may be changed with these optional arguments. You may provide zero, one, two or three of the arguments in any order. This lets the user indicate only the non-default arguments.
  The three possible arguments are:
  - **relative** or **absolute**: the error to be optimized;
  - **floating** or **fixed**: formats of the coefficients;
  - a constrained part q.

  The constrained part lets the user assign in advance some of the coefficients. For instance, for approximating $\exp(x)$, it may be interesting to search for a polynomial p of the form

  $$p = 1 + x + \frac{x^2}{2} + a_3x^3 + a_4x^4.$$

  Thus, there is a constrained part $q = 1+x+x^2/2$ and the unknown polynomial should be considered in the monomial basis $[|3, 4|]$. Calling **fpminimax** with monomial basis [|3, 4|] and constrained part q, will return a polynomial with the right form.

- The last argument is for advanced users. It is the minimax polynomial that approximates the function f in the given basis. If it is not given this polynomial will be automatically computed by **fpminimax**.
  This minimax polynomial is used to compute the list of interpolation points required by the method. It is also used, when floating-point coefficients are desired, to give an initial assumption for the exponents of the coeffcients. In general, you do not have to provide this argument. But if you want to obtain several polynomials of the same degree that approximate the same function on the same range, just changing the formats, you should probably consider computing only once the minimax polynomial and the list of points instead of letting **fpminimax** recompute them each time.
  Note that in the case when a constrained part is given, the minimax polynomial must take that into account. For instance, in the previous example, the minimax would be obtained by the following command:

  ```
  P = remez(1-(1+x+x^2/2)/exp(x), [|3,4|], range, 1/exp(x));
  ```

  Note that the constrained part is not to be added to P.
  In the case when the second argument is an integer or a list of integers, there is no restriction for P, as long as it is a polynomial. However, when the second argument is a list of functions, and even if these functions are all polynomials, P must be expanded in the given basis. For instance, if the second argument is 2 or [|0, 1, 2|], P can be given in Horner form. However, if the second argument is [|1, x, x^2|], P must be written as a linear combination of 1, x and x^2, otherwise, the algorithm will fail to recover the coefficients of P and will fail with an error message.
  Please also note that recovering the coefficients of P in an arbitrary basis is performed heuristically and no verification is performed to check that P does not contain other functions than the functions of the basis.

- Note that **fpminimax** internally computes a minimax polynomial (using the same algorithm as **remez** command). Thus **fpminimax** may encounter the same problems as **remez**. In particular, it may be very slow when Haar condition is not fulfilled. Another consequence is that currently **fpminimax** has to be run with a sufficiently high working precision.

Example 1:

```
> P = fpminimax(cos(x),6,[|DD, DD, D...|],[-1b-5;1b-5]);
> printexpansion(P);
(0x3ff0000000000000 + 0xbc09fda20235c100) + x * ((0x3b29ecd485d34781 + 0xb7c1cbc
97120359a) + x * (0xbfdfffffffffff98 + x * (0xbbfa6e0b3183cb0d + x * (0x3fa55555
55145337 + x * (0x3ca3540480618939 + x * 0xbf56c138142d8c3b)))))
```

Example 2:

```
> P = fpminimax(sin(x),6,[|32...|],[-1b-5;1b-5], fixed, absolute);
> display = powers!;
> P;
x * (1 + x^2 * (-357913941 * 2^(-31) + x^2 * (35789873 * 2^(-32))))
```

Example 3:

```
> P = fpminimax(exp(x), [|3,4|], [|D,24|], [-1/256; 1/246], 1+x+x^2/2);
> display = powers!;
> P;
1 + x * (1 + x * (1 * 2^(-1) + x * (375300225001191 * 2^(-51) + x * (5592621 * 2
^(-27)))))
```

Example 4:

```
> f = cos(exp(x));
> pstar = remez(f, 5, [-1b-7;1b-7]);
> listpoints = dirtyfindzeros(f-pstar, [-1b-7; 1b-7]);
> P1 = fpminimax(f, 5, [|DD...|], listpoints, absolute, default, default, pstar)
;
> P2 = fpminimax(f, 5, [|D...|], listpoints, absolute, default, default, pstar);

> P3 = fpminimax(f, 5, [|D, D, D, 24...|], listpoints, absolute, default, defaul
t, pstar);
> print("Error of pstar: ", dirtyinfnorm(f-pstar, [-1b-7; 1b-7]));
Error of pstar:  7.90484413054597351028798313257187453993793294531 02e-16
> print("Error of P1:    ", dirtyinfnorm(f-P1, [-1b-7; 1b-7]));
Error of P1:     7.90484413054597351598486470891926674420474694048 83e-16
> print("Error of P2:    ", dirtyinfnorm(f-P2, [-1b-7; 1b-7]));
Error of P2:     8.24771445799508710611470215974060779936577145752 38e-16
> print("Error of P3:    ", dirtyinfnorm(f-P3, [-1b-7; 1b-7]));
Error of P3:     1.08454277156993282593701156841863009789063333951055e-15
```

Example 5:

```
> L = [|exp(x), sin(x), cos(x)-1, sin(x^3)|];
> g = (2^x-1)/x;
> p = fpminimax(g, L, [|D...|], [-1/16;1/16],absolute);
> display = powers!;
> p;
-6535769594871261 * 2^(-55) * sin(x^3) + 5247089102535871 * 2^(-53) * (cos(x) -
1) + -8159095033730773 * 2^(-54) * sin(x) + 6243315658446641 * 2^(-53) * exp(x)
```

Example 6:

```
> n = 9;
> T = [|1, x|];
> for i from 2 to n do T[i] = canonical(2*x*T[i-1]-T[i-2]);
> g = (2^x-1)/x;
> PCheb = fpminimax(g, T, [|DD,DE...|], [-1/16;1/16],absolute);
> display = dyadic!;
> print(PCheb);
1746786017920488573 5b-99 * (9 * x + -120 * x^3 + 432 * x^5 + -576 * x^7 + 256 *
x^9) + 7875248523371081439b-93 * (1 + -32 * x^2 + 160 * x^4 + -256 * x^6 + 128 *
 x^8) + 1293476066180903623 1b-89 * (-7 * x + 56 * x^3 + -112 * x^5 + 64 * x^7) +
 9342762606926463323b-84 * (-1 + 18 * x^2 + -48 * x^4 + 32 * x^6) + 590726068372
7596799b-79 * (5 * x + -20 * x^3 + 16 * x^5) + 1281095894865714451 9b-76 * (1 + -
8 * x^2 + 8 * x^4) + 5792228662390969179b-71 * (-3 * x + 4 * x^3) + 167797053124
47201213b-69 * (-1 + 2 * x^2) + 1826501428099735904 9b-66 * x + 11705449744817514
3910939038333811b-107
```

See also: **remez** (8.145), **dirtyfindzeros** (8.40), **absolute** (8.2), **relative** (8.144), **fixed** (8.64), **floating** (8.65), **default** (8.34), **halfprecision** (8.73), **single** (8.162), **double** (8.45), **doubleextended** (8.47), **doubledouble** (8.46), **quad** (8.136), **tripledouble** (8.180), **implementpoly** (8.81), **coeff** (8.24), **degree** (8.35), **roundcoefficients** (8.152), **guessdegree** (8.72)

8.68 fullparentheses

Name: **fullparentheses**
activates, deactivates or inspects the state variable controlling output with full parenthesising
Usage:

fullparentheses = *activation value* : on|off → void
fullparentheses = *activation value* ! : on|off → void

Parameters:

- *activation value* represents **on** or **off**, i.e. activation or deactivation

Description:

- An assignment **fullparentheses** = *activation value*, where *activation value* is one of **on** or **off**, activates respectively deactivates the output of expressions with full parenthesising. In full parenthesising mode, `Sollya` commands like **print**, **write** and the implicit command when an expression is given at the prompt will output expressions with parenthesising at all places where it is necessary for expressions containing infix operators to be parsed back with the same result. Otherwise parentheses around associative operators are omitted.

  If the assignment **fullparentheses** = *activation value* is followed by an exclamation mark, no message indicating the new state is displayed. Otherwise the user is informed of the new state of the global mode by an indication.

Example 1:

```
> autosimplify = off!;
> fullparentheses = off;
Full parentheses mode has been deactivated.
> print(1 + 2 + 3);
1 + 2 + 3
> fullparentheses = on;
Full parentheses mode has been activated.
> print(1 + 2 + 3);
(1 + 2) + 3
```

See also: **print** (8.128), **write** (8.186), **autosimplify** (8.15)

8.69 function

Name: **function**
keyword for declaring a procedure-based function or a keyword representing a function type
Usage:

function(*procedure*) : procedure → function
function : type type

Parameters:

- *procedure* is a procedure of type (range, integer, integer) → range

Description:

- For the sake of safety and mathematical consistency, `Sollya` distinguishes clearly between functions, seen in the mathematical sense of the term, i.e. mappings, and procedures, seen in the sense Computer Science gives to functions, i.e. pieces of code that compute results for arguments following an algorithm. In some cases however, it is interesting to use such Computer Science procedures as realisations of mathematical functions, e.g. in order to plot them or even to perform polynomial approximation on them. The **function** keyword allows for such a transformation of a `Sollya` procedure into a `Sollya` function.
- The procedure to be used as a function through **function**(*procedure*) must be of type (range, integer, integer) → range. This means it must take in argument an interval X, a degree of differentiation n and a working precision p. It must return in result an interval Y encompassing the image $f^{(n)}(X)$ of the n-th derivative of the implemented function f, i.e. $f^{(n)}(X) \subseteq Y$. In order to allow `Sollya`'s algorithms to work properly, the procedure must ensure that, whenever $(p, \mathrm{diam}(X))$ tends to $(+\infty, 0)$, the computed over-estimated bounding Y tends to the actual image $f^{(n)}(X)$.
- The user must be aware that they are responsible of the correctness of the procedure. If, for some n and X, *procedure* returns an interval Y such that $f^{(n)}(X) \not\subseteq Y$, **function** will successfully return a function without any complain, but this function might behave inconsistently in further computations.
- For cases when the procedure does not have the correct signature or does not return a finite interval as a result **function**(*procedure*) evaluates to Not-A-Number (resp. to an interval of Not-A-Numbers for interval evaluation).
- **function** also represents the function type for declarations of external procedures by means of **externalproc**.

  Remark that in contrast to other indicators, type indicators like **function** cannot be handled outside the **externalproc** context. In particular, they cannot be assigned to variables.

Example 1:

```
> procedure EXP(X,n,p) {
        var res, oldPrec;
        oldPrec = prec;
        prec = p!;

        res = exp(X);

        prec = oldPrec!;
        return res;
  };
> f = function(EXP);
> f(1);
2.71828182845904523536028747135266249775724709369998
> exp(1);
2.71828182845904523536028747135266249775724709369998
> f(x + 3);
(function(proc(X, n, p)
{
var res, oldPrec;
oldPrec = prec;
prec = p!;
res = exp(X);
prec = oldPrec!;
return res;
}))(3 + x)
> diff(f);
diff(function(proc(X, n, p)
{
var res, oldPrec;
oldPrec = prec;
prec = p!;
res = exp(X);
prec = oldPrec!;
return res;
}))
> (diff(f))(0);
1
> g = f(sin(x));
> g(17);
0.38235816999386683402690554641655641359573458342087 6
> diff(g);
(diff(function(proc(X, n, p)
{
var res, oldPrec;
oldPrec = prec;
prec = p!;
res = exp(X);
prec = oldPrec!;
return res;
}))(sin(x)) * cos(x)
> (diff(g))(1);
1.25338076749344683697237458088447611474812675164344
> p = remez(f,3,[-1/2;1/2]);
> p;
0.99967120901425193658110435888409366679868809033 78 + x * (0.9997370298357005328
02338697856944389400672232655 05 + x * (0.5104972936025655553580020205228144445 13
04355667385 + x * 0.16981432460717676177005021986415491524474293027 16))
```

See also: **proc** (8.133), **library** (8.91), **procedure** (8.134), **externalproc** (8.60), **boolean** (8.20), **constant** (8.27), **integer** (8.85), **list of** (8.93), **range** (8.138), **string** (8.166)

8.70 >=

Name: >=
greater-than-or-equal-to operator
Usage:

expr1 >= *expr2* : (constant, constant) → boolean

Parameters:

- *expr1* and *expr2* represent constant expressions

Description:

- The operator >= evaluates to true iff its operands *expr1* and *expr2* evaluate to two floating-point numbers a_1 respectively a_2 with the global precision **prec** and a_1 is greater than or equal to a_2. The user should be aware of the fact that because of floating-point evaluation, the operator >= is not exactly the same as the mathematical operation *greater-than-or-equal-to*.

Example 1:

```
> 5 >= 4;
true
> 5 >= 5;
true
> 5 >= 6;
false
> exp(2) >= exp(1);
true
> log(1) >= exp(2);
false
```

Example 2:

```
> prec = 12;
The precision has been set to 12 bits.
> 16384.1 >= 16385.1;
true
```

See also: == (8.49), **!**= (8.107), > (8.71), <= (8.89), < (8.98), **in** (8.82), **!** (8.109), && (8.6), || (8.114), **prec** (8.125), **max** (8.100), **min** (8.103)

8.71 >

Name: >
greater-than operator
Usage:

expr1 > *expr2* : (constant, constant) → boolean

Parameters:

- *expr1* and *expr2* represent constant expressions

Description:

- The operator > evaluates to true iff its operands *expr1* and *expr2* evaluate to two floating-point numbers a_1 respectively a_2 with the global precision **prec** and a_1 is greater than a_2. The user should be aware of the fact that because of floating-point evaluation, the operator > is not exactly the same as the mathematical operation *greater-than*.

Example 1:

```
> 5 > 4;
true
> 5 > 5;
false
> 5 > 6;
false
> exp(2) > exp(1);
true
> log(1) > exp(2);
false
```

Example 2:

```
> prec = 12;
The precision has been set to 12 bits.
> 16385.1 > 16384.1;
false
```

See also: == (8.49), **!**= (8.107), >= (8.70), <= (8.89), < (8.98), **in** (8.82), **!** (8.109), && (8.6), || (8.114), **prec** (8.125), **max** (8.100), **min** (8.103)

8.72 guessdegree

Name: **guessdegree**
returns the minimal degree needed for a polynomial to approximate a function with a certain error on an interval.
Usage:

guessdegree(*f*,*I*,*eps*,*w*,*bound*) : (function, range, constant, function, constant) → range

Parameters:

- f is the function to be approximated.
- I is the interval where the function must be approximated.
- *eps* is the maximal acceptable error.
- w (optional) is a weight function. Default is 1.
- *bound* (optional) is a bound on the degree. Default is currently 128.

Description:

- **guessdegree** tries to find the minimal degree needed to approximate f on I by a polynomial with an error $\epsilon = pw - f$ whose infinity norm not greater than *eps*. More precisely, it finds n minimal such that there exists a polynomial p of degree n such that $\|pw - f\|_\infty <$ eps.
- **guessdegree** returns an interval: for common cases, this interval is reduced to a single number (i.e. the minimal degree). But in certain cases, **guessdegree** does not succeed in finding the minimal degree. In such cases the returned interval is of the form $[n, p]$ such that:
  - no polynomial of degree $n - 1$ gives an error less than *eps*.

– there exists a polynomial of degree p giving an error less than *eps*.

- The fifth optional argument *bound* is used to prevent **guessdegree** from trying to find too large degrees. If **guessdegree** does not manage to find a degree n satisfying the error and such that $n \leq$ bound, an interval of the form $[\cdot, +\infty]$ is returned. Note that *bound* must be a positive integer.

Example 1:

```
> guessdegree(exp(x),[-1;1],1e-10);
[10;10]
```

Example 2:

```
> guessdegree(exp(x),[-1;1],1e-10,default, 9);
[10;@Inf@]
```

Example 3:

```
> guessdegree(1, [-1;1], 1e-8, 1/exp(x));
[8;9]
```

See also: **dirtyinfnorm** (8.41), **remez** (8.145), **fpminimax** (8.67), **degree** (8.35)

8.73 halfprecision

Names: **halfprecision**, **HP**
rounding to the nearest IEEE 754 half-precision number (binary16).
Description:

- **halfprecision** is both a function and a constant.
- As a function, it rounds its argument to the nearest IEEE 754 half-precision (i.e. IEEE754-2008 binary16) number. Subnormal numbers are supported as well as standard numbers: it is the real rounding described in the standard.
- As a constant, it symbolizes the half-precision format. It is used in contexts when a precision format is necessary, e.g. in the commands **round** and **roundcoefficients**. It is not supported for **implementpoly**. See the corresponding help pages for examples.

Example 1:

```
> display=binary!;
> HP(0.1);
1.100110011_2 * 2^(-4)
> HP(4.17);
1.00001011_2 * 2^(2)
> HP(1.011_2 * 2^(-23));
1.1_2 * 2^(-23)
```

See also: **single** (8.162), **double** (8.45), **doubleextended** (8.47), **doubledouble** (8.46), **quad** (8.136), **tripledouble** (8.180), **roundcoefficients** (8.152), **fpminimax** (8.67), **implementpoly** (8.81), **round** (8.151), **printsingle** (8.131)

8.74 head

Name: **head**
gives the first element of a list.
Usage:

$$\textbf{head}(L) : \mathsf{list} \rightarrow \mathsf{any\ type}$$

Parameters:

- L is a list.

Description:

- **head**(L) returns the first element of the list L. It is equivalent to L[0].
- If L is empty, the command will fail with an error.

Example 1:

```
> head([|1,2,3|]);
1
> head([|1,2...|]);
1
```

See also: **tail** (8.171)

8.75 hexadecimal

Name: **hexadecimal**
special value for global state **display**
Description:

- **hexadecimal** is a special value used for the global state **display**. If the global state **display** is equal to **hexadecimal**, all data will be output in hexadecimal C99/ IEEE 754-2008 notation.

  As any value it can be affected to a variable and stored in lists.

See also: **decimal** (8.33), **dyadic** (8.48), **powers** (8.124), **binary** (8.18), **display** (8.43)

8.76 honorcoeffprec

Name: **honorcoeffprec**
indicates the (forced) honoring the precision of the coefficients in **implementpoly**
Usage:

honorcoeffprec : honorcoeffprec

Description:

- Used with command **implementpoly**, **honorcoeffprec** makes **implementpoly** honor the precision of the given polynomial. This means if a coefficient needs a double-double or a triple-double to be exactly stored, **implementpoly** will allocate appropriate space and use a double-double or triple-double operation even if the automatic (heuristic) determination implemented in command **implementpoly** indicates that the coefficient could be stored on less precision or, respectively, the operation could be performed with less precision. See **implementpoly** for details.

Example 1:

```
> verbosity = 1!;
> q = implementpoly(1 - simplify(TD(1/6)) * x^2,[-1b-10;1b-10],1b-60,DD,"p","imp
lementation.c");
Warning: at least one of the coefficients of the given polynomial has been round
ed in a way
that the target precision can be achieved at lower cost. Nevertheless, the imple
mented polynomial
is different from the given one.
> printexpansion(q);
0x3ff0000000000000 + x^2 * 0xbfc5555555555555
> r = implementpoly(1 - simplify(TD(1/6)) * x^2,[-1b-10;1b-10],1b-60,DD,"p","imp
lementation.c",honorcoeffprec);
Warning: the infered precision of the 2th coefficient of the polynomial is great
er than
the necessary precision computed for this step. This may make the automatic dete
rmination
of precisions useless.
> printexpansion(r);
0x3ff0000000000000 + x^2 * (0xbfc5555555555555 + 0xbc65555555555555 + 0xb9055555
55555555)
```

See also: **implementpoly** (8.81), **printexpansion** (8.130), **fpminimax** (8.67)

8.77 hopitalrecursions

Name: **hopitalrecursions**
controls the number of recursion steps when applying L'Hopital's rule.
Usage:

hopitalrecursions = n : integer → void
hopitalrecursions = n ! : integer → void
hopitalrecursions : integer

Parameters:

- n represents the number of recursions

Description:

- **hopitalrecursions** is a global variable. Its value represents the number of steps of recursion that are tried when applying L'Hopital's rule. This rule is applied by the interval evaluator present in the core of `Sollya` (and particularly visible in commands like **infnorm**).
- If an expression of the form f/g has to be evaluated by interval arithmetic on an interval I and if f and g have a common zero in I, a direct evaluation leads to NaN. `Sollya` implements a safe heuristic to avoid this, based on L'Hopital's rule: in such a case, it can be shown that $(f/g)(I) \subseteq (f'/g')(I)$. Since the same problem may exist for f'/g', the rule is applied recursively. The number of step in this recursion process is controlled by **hopitalrecursions**.
- Setting **hopitalrecursions** to 0 makes `Sollya` use this rule only once; setting it to 1 makes `Sollya` use the rule twice, and so on. In particular: the rule is always applied at least once, if necessary.

Example 1:

```
> hopitalrecursions=0;
The number of recursions for Hopital's rule has been set to 0.
> evaluate(log(1+x)^2/x^2,[-1/2; 1]);
[-@Inf@;@Inf@]
> hopitalrecursions=1;
The number of recursions for Hopital's rule has been set to 1.
> evaluate(log(1+x)^2/x^2,[-1/2; 1]);
[-2.52258872223978123766892848583270627230200053744108;6.77258872223978123766892
84858327062723020005374411]
```

See also: **taylorrecursions** (8.176), **infnorm** (8.84), **findzeros** (8.63), **evaluate** (8.53)

8.78 horner

Name: **horner**
brings all polynomial subexpressions of an expression to Horner form
Usage:

horner(*function*) : function → function

Parameters:

- *function* represents the expression to be rewritten in Horner form

Description:

- The command **horner** rewrites the expression representing the function *function* in a way such that all polynomial subexpressions (or the whole expression itself, if it is a polynomial) are written in Horner form. The command **horner** does not endanger the safety of computations even in `Sollya`'s floating-point environment: the function returned is mathematically equal to the function *function*.

Example 1:

```
> print(horner(1 + 2 * x + 3 * x^2));
1 + x * (2 + x * 3)
> print(horner((x + 1)^7));
1 + x * (7 + x * (21 + x * (35 + x * (35 + x * (21 + x * (7 + x))))))
```

Example 2:

```
> print(horner(exp((x + 1)^5) - log(asin(x + x^3) + x)));
exp(1 + x * (5 + x * (10 + x * (10 + x * (5 + x))))) - log(asin(x * (1 + x^2)) +
 x)
```

See also: **canonical** (8.21), **print** (8.128), **coeff** (8.24), **degree** (8.35), **autosimplify** (8.15), **simplifysafe** (8.160)

8.79 HP

Name: **HP**
short form for **halfprecision**
See also: **halfprecision** (8.73)

8.80 implementconstant

Name: **implementconstant**
implements a constant in arbitrary precision
Usage:

implementconstant(*expr*) : constant → void
implementconstant(*expr*,*filename*) : (constant, string) → void
implementconstant(*expr*,*filename*,*functionname*) : (constant, string, string) → void

Description:

- The command **implementconstant** implements the constant expression *expr* in arbitrary precision. More precisely, it generates the source code (written in C, and using MPFR) of a C function `const_something` with the following signature:

  ```
  void const_something (mpfr_ptr y, mp_prec_t prec)
  ```

  Let us denote by c the exact mathematical value of the constant defined by the expression *expr*. When called with arguments y and prec (where the variable y is supposed to be already initialized), the function `mpfr_const_something` sets the precision of y to a suitable precision and stores in it an approximate value of c such that

  $$|y - c| \leq |c|\, 2^{1-\mathrm{prec}}.$$

- When no filename *filename* is given or if **default** is given as *filename*, the source code produced by **implementconstant** is printed on standard output. Otherwise, when *filename* is given as a string of characters, the source code is output to a file named *filename*. If that file cannot be opened and/or written to, **implementconstant** fails and has no other effect.

- When *functionname* is given as an argument to **implementconstant** and *functionname* evaluates to a string of characters, the default name for the C function `const_something` is replaced by *functionname*. When **default** is given as *functionname*, the default name is used nevertheless, as if no *functionname* argument were given. When choosing a character sequence for *functionname*, the user should keep attention to the fact that *functionname* must be a valid C identifier in order to enable error-free compilation of the produced code.

- If *expr* refers to a constant defined with **libraryconstant**, the produced code uses the external code implementing this constant. The user should keep in mind that it is up to them to make sure the symbol for that external code can get resolved when the newly generated code is to be loaded.

- If a subexpression of *expr* evaluates to 0, **implementconstant** will most likely fail with an error message.

- **implementconstant** is unable to implement constant expressions *expr* that contain procedure-based functions, i.e. functions created from `Sollya` procedures using the **function** construct. If *expr* contains such a procedure-based function, **implementconstant** prints a warning and fails silently. The reason for this lack of functionality is that the produced C source code, which is supposed to be compiled, would have to call back to the `Sollya` interpreter in order to evaluate the procedure-based function.

- Similarily, **implementconstant** is currently unable to implement constant expressions *expr* that contain library-based functions, i.e. functions dynamically bound to `Sollya` using the **library** construct. If *expr* contains such a library-based function, **implementconstant** prints a warning and fails silently. Support for this feature is in principle feasible from a technical standpoint and might be added in a future release of `Sollya`.

- Currently, non-differentiable functions such as **double**, **doubledouble**, **tripledouble**, **single**, **halfprecision**, **quad**, **doubleextended**, **floor**, **ceil**, **nearestint** are not supported by **implementconstant**. If **implementconstant** encounters one of them, a warning message is displayed and no code is produced. However, if **autosimplify** equals on, it is possible that `Sollya` silently simplifies subexpressions of *expr* containing such functions and that **implementconstant** successfully produces code for evaluating *expr*.

- While it produces an MPFR-based C source code for *expr*, **implementconstant** takes architectural and system-dependent parameters into account. For example, it checks whether literal constants figuring in *expr* can be represented on a C `long int` type or if they must be stored in a different manner not to affect their accuracy. These tests, performed by `Sollya` during execution of **implementconstant**, depend themselves on the architecture `Sollya` is running on. Users should keep this matter in mind, especially when trying to compile source code on one machine whilst it has been produced on another.

Example 1:

```
> implementconstant(exp(1)+log(2)/sqrt(1/10));
#include <mpfr.h>

void
const_something (mpfr_ptr y, mp_prec_t prec)
{
  /* Declarations */
  mpfr_t tmp1;
  mpfr_t tmp2;
  mpfr_t tmp3;
  mpfr_t tmp4;
  mpfr_t tmp5;
  mpfr_t tmp6;
  mpfr_t tmp7;

  /* Initializations */
  mpfr_init2 (tmp2, prec+5);
  mpfr_init2 (tmp1, prec+3);
  mpfr_init2 (tmp4, prec+8);
  mpfr_init2 (tmp3, prec+7);
  mpfr_init2 (tmp6, prec+11);
  mpfr_init2 (tmp7, prec+11);
  mpfr_init2 (tmp5, prec+11);

  /* Core */
  mpfr_set_prec (tmp2, prec+4);
  mpfr_set_ui (tmp2, 1, MPFR_RNDN);
  mpfr_set_prec (tmp1, prec+3);
  mpfr_exp (tmp1, tmp2, MPFR_RNDN);
  mpfr_set_prec (tmp4, prec+8);
  mpfr_set_ui (tmp4, 2, MPFR_RNDN);
  mpfr_set_prec (tmp3, prec+7);
  mpfr_log (tmp3, tmp4, MPFR_RNDN);
  mpfr_set_prec (tmp6, prec+11);
  mpfr_set_ui (tmp6, 1, MPFR_RNDN);
  mpfr_set_prec (tmp7, prec+11);
  mpfr_set_ui (tmp7, 10, MPFR_RNDN);
  mpfr_set_prec (tmp5, prec+11);
  mpfr_div (tmp5, tmp6, tmp7, MPFR_RNDN);
  mpfr_set_prec (tmp4, prec+7);
  mpfr_sqrt (tmp4, tmp5, MPFR_RNDN);
  mpfr_set_prec (tmp2, prec+5);
  mpfr_div (tmp2, tmp3, tmp4, MPFR_RNDN);
  mpfr_set_prec (y, prec+3);
  mpfr_add (y, tmp1, tmp2, MPFR_RNDN);

  /* Cleaning stuff */
  mpfr_clear(tmp1);
  mpfr_clear(tmp2);
  mpfr_clear(tmp3);
  mpfr_clear(tmp4);
  mpfr_clear(tmp5);
  mpfr_clear(tmp6);
  mpfr_clear(tmp7);
}
```

Example 2:

```
> implementconstant(sin(13/17),"sine_of_thirteen_seventeenth.c");
> readfile("sine_of_thirteen_seventeenth.c");
#include <mpfr.h>

void
const_something (mpfr_ptr y, mp_prec_t prec)
{
  /* Declarations */
  mpfr_t tmp1;
  mpfr_t tmp2;
  mpfr_t tmp3;

  /* Initializations */
  mpfr_init2 (tmp2, prec+6);
  mpfr_init2 (tmp3, prec+6);
  mpfr_init2 (tmp1, prec+6);

  /* Core */
  mpfr_set_prec (tmp2, prec+6);
  mpfr_set_ui (tmp2, 13, MPFR_RNDN);
  mpfr_set_prec (tmp3, prec+6);
  mpfr_set_ui (tmp3, 17, MPFR_RNDN);
  mpfr_set_prec (tmp1, prec+6);
  mpfr_div (tmp1, tmp2, tmp3, MPFR_RNDN);
  mpfr_set_prec (y, prec+2);
  mpfr_sin (y, tmp1, MPFR_RNDN);

  /* Cleaning stuff */
  mpfr_clear(tmp1);
  mpfr_clear(tmp2);
  mpfr_clear(tmp3);
}
```

Example 3:

```
> implementconstant(asin(1/3 * pi),default,"arcsin_of_one_third_pi");
#include <mpfr.h>

void
arcsin_of_one_third_pi (mpfr_ptr y, mp_prec_t prec)
{
  /* Declarations */
  mpfr_t tmp1;
  mpfr_t tmp2;
  mpfr_t tmp3;

  /* Initializations */
  mpfr_init2 (tmp2, prec+8);
  mpfr_init2 (tmp3, prec+8);
  mpfr_init2 (tmp1, prec+8);

  /* Core */
  mpfr_set_prec (tmp2, prec+8);
  mpfr_const_pi (tmp2, MPFR_RNDN);
  mpfr_set_prec (tmp3, prec+8);
  mpfr_set_ui (tmp3, 3, MPFR_RNDN);
  mpfr_set_prec (tmp1, prec+8);
  mpfr_div (tmp1, tmp2, tmp3, MPFR_RNDN);
  mpfr_set_prec (y, prec+2);
  mpfr_asin (y, tmp1, MPFR_RNDN);

  /* Cleaning stuff */
  mpfr_clear(tmp1);
  mpfr_clear(tmp2);
  mpfr_clear(tmp3);
}
```

Example 4:

```
> implementconstant(ceil(log(19 + 1/3)),"constant_code.c","magic_constant");
> readfile("constant_code.c");
#include <mpfr.h>

void
magic_constant (mpfr_ptr y, mp_prec_t prec)
{
  /* Initializations */

  /* Core */
  mpfr_set_prec (y, prec);
  mpfr_set_ui (y, 3, MPFR_RNDN);
}
```

Example 5:

```
> bashexecute("gcc -fPIC -Wall -c libraryconstantexample.c -I$HOME/.local/includ
e");
> bashexecute("gcc -shared -o libraryconstantexample libraryconstantexample.o -l
gmp -lmpfr");
> euler_gamma = libraryconstant("./libraryconstantexample");
> implementconstant(euler_gamma^(1/3));
#include <mpfr.h>

void
const_something (mpfr_ptr y, mp_prec_t prec)
{
  /* Declarations */
  mpfr_t tmp1;

  /* Initializations */
  mpfr_init2 (tmp1, prec+1);

  /* Core */
  euler_gamma (tmp1, prec+1);
  mpfr_set_prec (y, prec+2);
  mpfr_root (y, tmp1, 3, MPFR_RNDN);

  /* Cleaning stuff */
  mpfr_clear(tmp1);
}
```

See also: **implementpoly** (8.81), **libraryconstant** (8.92), **library** (8.91), **function** (8.69)

8.81 implementpoly

Name: **implementpoly**
implements a polynomial using double, double-double and triple-double arithmetic and generates a Gappa proof
Usage:

implementpoly(*polynomial*, *range*, *error bound*, *format*, *functionname*, *filename*) : (function, range, constant, D|double|DD|doubledouble|TD|tripledouble, string, string) → function
implementpoly(*polynomial*, *range*, *error bound*, *format*, *functionname*, *filename*, *honor coefficient precisions*) : (function, range, constant, D|double|DD|doubledouble|TD|tripledouble, string, string, honorcoeffprec) → function
implementpoly(*polynomial*, *range*, *error bound*, *format*, *functionname*, *filename*, *proof filename*) : (function, range, constant, D|double|DD|doubledouble|TD|tripledouble, string, string, string) → function
implementpoly(*polynomial*, *range*, *error bound*, *format*, *functionname*, *filename*, *honor coefficient precisions*, *proof filename*) : (function, range, constant, D|double|DD|doubledouble|TD|tripledouble, string, string, honorcoeffprec, string) → function

Description:

- The command **implementpoly** implements the polynomial *polynomial* in range *range* as a function called *functionname* in `C` code using double, double-double and triple-double arithmetic in a way that the rounding error (estimated at its first order) is bounded by *error bound*. The produced code is output in a file named *filename*. The argument *format* indicates the double, double-double or triple-double format of the variable in which the polynomial varies, influencing also in the signature of the `C` function.

  If a seventh or eighth argument *proof filename* is given and if this argument evaluates to a variable of type string, the command **implementpoly** will produce a `Gappa` proof that the rounding error is less than the given bound. This proof will be output in `Gappa` syntax in a file name *proof filename*.

The command **implementpoly** returns the polynomial that has been implemented. As the command **implementpoly** tries to adapt the precision needed in each evaluation step to its strict minimum and as it applies renormalization to double-double and triple-double precision coefficients to bring them to a round-to-nearest expansion form, the returned polynomial may differ from the polynomial *polynomial*. Nevertheless the difference will be small enough that the rounding error bound with regard to the polynomial *polynomial* (estimated at its first order) will be less than the given error bound.

If a seventh argument *honor coefficient precisions* is given and evaluates to a variable **honorcoeffprec** of type honorcoeffprec, **implementpoly** will honor the precision of the given polynomial *polynomials*. This means if a coefficient needs a double-double or a triple-double to be exactly stored, **implementpoly** will allocate appropriate space and use a double-double or triple-double operation even if the automatic (heuristic) determination implemented in command **implementpoly** indicates that the coefficient could be stored on less precision or, respectively, the operation could be performed with less precision. The use of **honorcoeffprec** has advantages and disadvantages. If the polynomial *polynomial* given has not been determined by a process considering directly polynomials with floating-point coefficients, **honorcoeffprec** should not be indicated. The **implementpoly** command can then determine the needed precision using the same error estimation as used for the determination of the precisions of the operations. Generally, the coefficients will get rounded to double, double-double and triple-double precision in a way that minimizes their number and respects the rounding error bound *error bound*. Indicating **honorcoeffprec** may in this case short-circuit most precision estimations leading to sub-optimal code. On the other hand, if the polynomial *polynomial* has been determined with floating-point precisions in mind, **honorcoeffprec** should be indicated because such polynomials often are very sensitive in terms of error propagation with regard to their coefficients' values. Indicating **honorcoeffprec** prevents the **implementpoly** command from rounding the coefficients and altering by many orders of magnitude the approximation error of the polynomial with regard to the function it approximates.

The implementer behind the **implementpoly** command makes some assumptions on its input and verifies them. If some assumption cannot be verified, the implementation will not succeed and **implementpoly** will evaluate to a variable **error** of type error. The same behaviour is observed if some file is not writable or some other side-effect fails, e.g. if the implementer runs out of memory.

As error estimation is performed only on the first order, the code produced by the **implementpoly** command should be considered valid iff a `Gappa` proof has been produced and successfully run in `Gappa`.

Example 1:

```
> implementpoly(1 - 1/6 * x^2 + 1/120 * x^4, [-1b-10;1b-10], 1b-30, D, "p","impl
ementation.c");
1 + x^2 * (-0.16666666666666665741480812812369549646973609924316406250000
3333333333217685101601546193705871701240539550781250000
> readfile("implementation.c");
#define p_coeff_0h 1.00000000000000000000000000000000000000000000000000000000000
0000000000000000000e+00
#define p_coeff_2h -1.66666666666666657414808128123695496469736099243164062500000
0000000000000000000e-01
#define p_coeff_4h 8.33333333333333217685101601546193705871701240539550781250000
0000000000000000000e-03


void p(double *p_resh, double x) {
double p_x_0_pow2h;


p_x_0_pow2h = x * x;


double p_t_1_0h;
double p_t_2_0h;
double p_t_3_0h;
double p_t_4_0h;
double p_t_5_0h;



p_t_1_0h = p_coeff_4h;
p_t_2_0h = p_t_1_0h * p_x_0_pow2h;
p_t_3_0h = p_coeff_2h + p_t_2_0h;
p_t_4_0h = p_t_3_0h * p_x_0_pow2h;
p_t_5_0h = p_coeff_0h + p_t_4_0h;
*p_resh = p_t_5_0h;


}
```

Example 2:

```
> implementpoly(1 - 1/6 * x^2 + 1/120 * x^4, [-1b-10;1b-10], 1b-30, D, "p","impl
ementation.c","implementation.gappa");
1 + x^2 * (-0.16666666666666665741480812812369549646973609924316406250000
3333333333217685101601546193705871701240539550781250000)
```

Example 3:

```
> verbosity = 1!;
> q = implementpoly(1 - simplify(TD(1/6)) * x^2,[-1b-10;1b-10],1b-60,DD,"p","imp
lementation.c");
Warning: at least one of the coefficients of the given polynomial has been round
ed in a way
that the target precision can be achieved at lower cost. Nevertheless, the imple
mented polynomial
is different from the given one.
> printexpansion(q);
0x3ff0000000000000 + x^2 * 0xbfc5555555555555
> r = implementpoly(1 - simplify(TD(1/6)) * x^2,[-1b-10;1b-10],1b-60,DD,"p","imp
lementation.c",honorcoeffprec);
Warning: the infered precision of the 2th coefficient of the polynomial is great
er than
the necessary precision computed for this step. This may make the automatic dete
rmination
of precisions useless.
> printexpansion(r);
0x3ff0000000000000 + x^2 * (0xbfc5555555555555 + 0xbc65555555555555 + 0xb9055555
55555555)
```

Example 4:

```
> p = 0x3ff0000000000000 + x * (0x3ff0000000000000 + x * (0x3fe0000000000000 + x
 * (0x3fc5555555555559 + x * (0x3fa55555555555bd + x * (0x3f811111111106e2 + x
 * (0x3f56c16c16bf5eb7 + x * (0x3f2a01a01a292dcd + x * (0x3efa01a0218a016a + x
 * (0x3ec71de360331aad + x * (0x3e927e42e3823bf3 + x * (0x3e5ae6b2710c2c9a + x
 * (0x3e2203730c0a7c1d + x * 0x3de5da557e0781df))))))))));
> q = implementpoly(p,[-1/2;1/2],1b-60,D,"p","implementation.c",honorcoeffprec,"
implementation.gappa");
> if (q != p) then print("During implementation, rounding has happened.") else p
rint("Polynomial implemented as given.");
Polynomial implemented as given.
```

See also: **honorcoeffprec** (8.76), **roundcoefficients** (8.152), **double** (8.45), **doubledouble** (8.46), **tripledouble** (8.180), **readfile** (8.142), **printexpansion** (8.130), **error** (8.52), **remez** (8.145), **fpminimax** (8.67), **taylor** (8.174), **implementconstant** (8.80)

8.82 in

Name: **in**
containment test operator
Usage:

expr **in** *range1* : (constant, range) → boolean
range1 **in** *range2* : (range, range) → boolean

Parameters:

- *expr* represents a constant expression
- *range1* and *range2* represent ranges (intervals)

Description:

- When its first operand is a constant expression *expr*, the operator **in** evaluates to true iff the constant value of the expression *expr* is contained in the interval *range1*.

- When both its operands are ranges (intervals), the operator **in** evaluates to true iff all values in *range1* are contained in the interval *range2*.
- **in** is also used as a keyword for loops over the different elements of a list.

Example 1:

```
> 5 in [-4;7];
true
> 4 in [-1;1];
false
> 0 in sin([-17;17]);
true
```

Example 2:

```
> [5;7] in [2;8];
true
> [2;3] in [4;5];
false
> [2;3] in [2.5;5];
false
```

Example 3:

```
> for i in [|1,...,5|] do print(i);
1
2
3
4
5
```

See also: == (8.49), **!**= (8.107), >= (8.70), > (8.71), <= (8.89), < (8.98), **!** (8.109), && (8.6), || (8.114), **prec** (8.125), **print** (8.128)

8.83 inf

Name: **inf**
gives the lower bound of an interval.
Usage:

$$\mathbf{inf}(I) : \mathsf{range} \to \mathsf{constant}$$
$$\mathbf{inf}(x) : \mathsf{constant} \to \mathsf{constant}$$

Parameters:

- I is an interval.
- x is a real number.

Description:

- Returns the lower bound of the interval I. Each bound of an interval has its own precision, so this command is exact, even if the current precision is too small to represent the bound.
- When called on a real number x, **inf** considers it as an interval formed of a single point: $[x, x]$. In other words, **inf** behaves like the identity.

Example 1:

```
> inf([1;3]);
1
> inf(0);
0
```

Example 2:

```
> display=binary!;
> I=[0.111110000011111_2; 1];
> inf(I);
1.11110000011111_2 * 2^(-1)
> prec=12!;
> inf(I);
1.11110000011111_2 * 2^(-1)
```

See also: **mid** (8.101), **sup** (8.169), **max** (8.100), **min** (8.103)

8.84 infnorm

Name: **infnorm**
computes an interval bounding the infinity norm of a function on an interval.
Usage:

infnorm(*f*,*I*,*filename*,*Ilist*) : (function, range, string, list) → range

Parameters:

- f is a function.
- I is an interval.
- *filename* (optional) is the name of the file into a proof will be saved.
- *IList* (optional) is a list of intervals to be excluded.

Description:

- **infnorm**(*f*,*range*) computes an interval bounding the infinity norm of the given function f on the interval I, e.g. computes an interval J such that $\max_{x\in I}\{|f(x)|\} \subseteq J$.
- If *filename* is given, a proof in English will be produced (and stored in file called *filename*) proving that $\max_{x\in I}\{|f(x)|\} \subseteq J$.
- If a list *IList* of intervals $I_1, \ldots, I_n$ is given, the infinity norm will be computed on $I\backslash(I_1 \cup \ldots \cup I_n)$.
- The function f is assumed to be at least twice continuous on I. More generally, if f is $\mathcal{C}^k$, global variables **hopitalrecursions** and **taylorrecursions** must have values not greater than k.
- If the interval is reduced to a single point, the result of **infnorm** is an interval containing the exact absolute value of f at this point.
- If the interval is not bound, the result will be $[0, +\infty]$ which is correct but perfectly useless. **infnorm** is not meant to be used with infinite intervals.
- The result of this command depends on the global variables **prec**, **diam**, **taylorrecursions** and **hopitalrecursions**. The contribution of each variable is not easy even to analyse.
  - The algorithm uses interval arithmetic with precision **prec**. The precision should thus be set high enough to ensure that no critical cancellation will occur.

– When an evaluation is performed on an interval $[a, b]$, if the result is considered being too large, the interval is split into $[a, \frac{a+b}{2}]$ and $[\frac{a+b}{2}, b]$ and so on recursively. This recursion step is not performed if the $(b-a) < \delta \cdot |I|$ where δ is the value of variable **diam**. In other words, **diam** controls the minimum length of an interval during the algorithm.

– To perform the evaluation of a function on an interval, Taylor's rule is applied, e.g. $f([a, b]) \subseteq f(m) + [a - m, b - m] \cdot f'([a, b])$ where $m = \frac{a+b}{2}$. This rule is recursively applied n times where n is the value of variable **taylorrecursions**. Roughly speaking, the evaluations will avoid decorrelation up to order n.

– When a function of the form $\frac{g}{h}$ has to be evaluated on an interval $[a, b]$ and when g and h vanish at a same point z of the interval, the ratio may be defined even if the expression $\frac{g(z)}{h(z)} = \frac{0}{0}$ does not make any sense. In this case, L'Hopital's rule may be used and $\left(\frac{g}{h}\right)([a, b]) \subseteq \left(\frac{g'}{h'}\right)([a, b])$. Since the same can occur with the ratio $\frac{g'}{h'}$, the rule is applied recursively. The variable **hopitalrecursions** controls the number of recursion steps.

- The algorithm used for this command is quite complex to be explained here. Please find a complete description in the following article:
  S. Chevillard and C. Lauter
  A certified infinity norm for the implementation of elementary functions
  LIP Research Report number RR2007-26
  http://prunel.ccsd.cnrs.fr/ensl-00119810

- Users should be aware about the fact that the algorithm behind **infnorm** is inefficient in most cases and that other, better suited algorithms, such as **supnorm**, are available inside `Sollya`. As a matter of fact, while **infnorm** is maintained for compatibility reasons with legacy `Sollya` codes, users are advised to avoid using **infnorm** in new `Sollya` scripts and to replace it, where possible, by the **supnorm** command.

Example 1:

```
> infnorm(exp(x),[-2;3]);
[2.008553692318766774092852965458171789698790783855 37e1;2.008553692318766774092852965458171789698790783855 44e1]
```

Example 2:

```
> infnorm(exp(x),[-2;3],"proof.txt");
[2.008553692318766774092852965458171789698790783855 37e1;2.008553692318766774092852965458171789698790783855 44e1]
```

Example 3:

```
> infnorm(exp(x),[-2;3],[| [0;1], [2;2.5] |]);
[2.008553692318766774092852965458171789698790783855 37e1;2.008553692318766774092852965458171789698790783855 44e1]
```

Example 4:

```
> infnorm(exp(x),[-2;3],"proof.txt", [| [0;1], [2;2.5] |]);
[2.008553692318766774092852965458171789698790783855 37e1;2.008553692318766774092852965458171789698790783855 44e1]
```

Example 5:

```
> infnorm(exp(x),[1;1]);
[2.718281828459045235360287471352662497757247093699 89;2.718281828459045235360287471352662497757247093699 98]
```

Example 6:

```
> infnorm(exp(x), [log(0);log(1)]);
[0;@Inf@]
```

See also: **prec** (8.125), **diam** (8.37), **hopitalrecursions** (8.77), **dirtyinfnorm** (8.41), **checkinfnorm** (8.23), **supnorm** (8.170), **findzeros** (8.63), **diff** (8.39), **taylorrecursions** (8.176), **autodiff** (8.14), **numberroots** (8.110), **taylorform** (8.175)

8.85 integer

Name: **integer**
keyword representing a machine integer type
Usage:

integer : type type

Description:

- **integer** represents the machine integer type for declarations of external procedures **externalproc**.

  Remark that in contrast to other indicators, type indicators like **integer** cannot be handled outside the **externalproc** context. In particular, they cannot be assigned to variables.

See also: **externalproc** (8.60), **boolean** (8.20), **constant** (8.27), **function** (8.69), **list of** (8.93), **range** (8.138), **string** (8.166)

8.86 integral

Name: **integral**
computes an interval bounding the integral of a function on an interval.
Usage:

integral(f,I) : (function, range) $\rightarrow$ range

Parameters:

- f is a function.
- I is an interval.

Description:

- **integral**(f,I) returns an interval J such that the exact value of the integral of f on I lies in J.
- This command is safe but very inefficient. Use **dirtyintegral** if you just want an approximate value.
- The result of this command depends on the global variable **diam**. The method used is the following: I is cut into intervals of length not greater then $\delta \cdot |I|$ where δ is the value of global variable **diam**. On each small interval J, an evaluation of f by interval is performed. The result is multiplied by the length of J. Finally all values are summed.

Example 1:

```
> sin(10);
-0.54402111088936981340474766185137728168364301291621
> integral(cos(x),[0;10]);
[-0.54710197983579690224097637163525943075698599257332;-0.540940151300131838481505408813733707440537411917 28]
> diam=1e-5!;
> integral(cos(x),[0;10]);
[-0.54432915685955427101857780295936956775293876382777;-0.543713064012499695080396442219274890104258031735 55]
```

See also: **diam** (8.37), **dirtyintegral** (8.42), **prec** (8.125)

8.87 isbound

Name: **isbound**
indicates whether a variable is bound or not.
Usage:

isbound(*ident*) : boolean

Parameters:

- *ident* is a name.

Description:

- **isbound**(*ident*) returns a boolean value indicating whether the name *ident* is used or not to represent a variable. It returns true when *ident* is the name used to represent the global variable or if the name is currently used to refer to a (possibly local) variable.
- When a variable is defined in a block and has not been defined outside, **isbound** returns true when called inside the block, and false outside. Note that **isbound** returns true as soon as a variable has been declared with **var**, even if no value is actually stored in it.
- If *ident1* is bound to a variable and if *ident2* refers to the global variable, the command **rename**(*ident2*, *ident1*) hides the value of *ident1* which becomes the global variable. However, if the global variable is again renamed, *ident1* gets its value back. In this case, **isbound**(*ident1*) returns true. If *ident1* was not bound before, **isbound**(*ident1*) returns false after that *ident1* has been renamed.

Example 1:

```
> isbound(x);
false
> isbound(f);
false
> isbound(g);
false
> f=sin(x);
> isbound(x);
true
> isbound(f);
true
> isbound(g);
false
```

Example 2:

```
> isbound(a);
false
> { var a; isbound(a); };
true
> isbound(a);
false
```

Example 3:

```
> f=sin(x);
> isbound(x);
true
> rename(x,y);
> isbound(x);
false
```

Example 4:

```
> x=1;
> f=sin(y);
> rename(y,x);
> f;
sin(x)
> x;
x
> isbound(x);
true
> rename(x,y);
> isbound(x);
true
> x;
1
```

See also: **rename** (8.146)

8.88 isevaluable

Name: **isevaluable**
tests whether a function can be evaluated at a point

Usage:

isevaluable(*function*, *constant*) : (function, constant) → boolean

Parameters:

- *function* represents a function
- *constant* represents a constant point

Description:

- **isevaluable** applied to function *function* and a constant *constant* returns a boolean indicating whether or not a subsequent call to **evaluate** on the same function *function* and constant *constant* will produce a numerical result or NaN. This means **isevaluable** returns false iff **evaluate** will return NaN.
- The command **isevaluable** is now considered DEPRECATED in `Sollya`. As checks for NaNs are now possible in Sollya, the command **isevaluable** can be fully emulated with a call to evaluate and a couple of tests, as shown below in the last example.

Example 1:

```
> isevaluable(sin(pi * 1/x), 0.75);
true
> print(evaluate(sin(pi * 1/x), 0.75));
-0.866025403784438646763723170752936183471402626905 18
```

Example 2:

```
> isevaluable(sin(pi * 1/x), 0.5);
true
> print(evaluate(sin(pi * 1/x), 0.5));
[-1.729864525143812695165086150310981295428367679916 79e-12715;7.5941198201187963
145069564314525661706039084390067e-12716]
```

Example 3:

```
> isevaluable(sin(pi * 1/x), 0);
false
> print(evaluate(sin(pi * 1/x), 0));
[@NaN@;@NaN@]
```

Example 4:

```
> procedure isEvaluableEmulation(f, c) {
       return match evaluate(f, c) with
                   NaN : (false)
                 [NaN;NaN] : (false)
                 default : (true);
  }
  isEvaluableEmulation(sin(pi * 1/x), 0.75);
  isEvaluableEmulation(sin(pi * 1/x), 0.5);
error
> isEvaluableEmulation(sin(pi * 1/x), 0);
error
```

See also: **evaluate** (8.53)

8.89 <=

Name: <=
less-than-or-equal-to operator
Usage:

$$expr1 <= expr2 : (\mathsf{constant}, \mathsf{constant}) \to \mathsf{boolean}$$

Parameters:

- *expr1* and *expr2* represent constant expressions

Description:

- The operator <= evaluates to true iff its operands *expr1* and *expr2* evaluate to two floating-point numbers a_1 respectively a_2 with the global precision **prec** and a_1 is less than or equal to a_2. The user should be aware of the fact that because of floating-point evaluation, the operator <= is not exactly the same as the mathematical operation *less-than-or-equal-to*.

Example 1:

```
> 5 <= 4;
false
> 5 <= 5;
true
> 5 <= 6;
true
> exp(2) <= exp(1);
false
> log(1) <= exp(2);
true
```

Example 2:

```
> prec = 12;
The precision has been set to 12 bits.
> 16385.1 <= 16384.1;
true
```

See also: == (8.49), **!=** (8.107), >= (8.70), > (8.71), < (8.98), **in** (8.82), **!** (8.109), && (8.6), || (8.114), **prec** (8.125), **max** (8.100), **min** (8.103)

8.90 length

Name: **length**
computes the length of a list or string.
Usage:

$$\textbf{length}(L) : \mathsf{list} \to \mathsf{integer}$$
$$\textbf{length}(s) : \mathsf{string} \to \mathsf{integer}$$

Parameters:

- L is a list.
- s is a string.

Description:

- **length** returns the length of a list or a string, e.g. the number of elements or letters.
- The empty list or string have length 0. If L is an end-elliptic list, **length** returns +Inf.

Example 1:

```
> length("Hello World!");
12
```

Example 2:

```
> length([|1,...,5|]);
5
```

Example 3:

```
> length([| |]);
0
```

Example 4:

```
> length([|1,2...|]);
@Inf@
```

8.91 library

Name: **library**
binds an external mathematical function to a variable in `Sollya`
Usage:

$$\textbf{library}(path) : \mathsf{string} \to \mathsf{function}$$

Description:

- The command **library** lets you extend the set of mathematical functions known to `Sollya`. By default, `Sollya` knows the most common mathematical functions such as **exp**, **sin**, **erf**, etc. Within `Sollya`, these functions may be composed. This way, `Sollya` should satisfy the needs of a lot of users. However, for particular applications, one may want to manipulate other functions such as Bessel functions, or functions defined by an integral or even a particular solution of an ODE.

- **library** makes it possible to let `Sollya` know about new functions. In order to let it know, you have to provide an implementation of the function you are interested in. This implementation is a C file containing a function of the form:

  ```
  int my_ident(sollya_mpfi_t result, sollya_mpfi_t op, int n)
  ```

  The semantic of this function is the following: it is an implementation of the function and its derivatives in interval arithmetic. `my_ident(result, I, n)` shall store in `result` an enclosure of the image set of the n-th derivative of the function f over `I`: $f^{(n)}(I) \subseteq$ result.

- The integer value returned by the function implementation currently has no meaning.

- You do not need to provide a working implementation for any `n`. Most functions of `Sollya` requires a relevant implementation only for f, f' and f''. For higher derivatives, its is not so critical and the implementation may just store $[-\infty, +\infty]$ in result whenever $n > 2$.

- Note that you should respect somehow interval-arithmetic standards in your implementation: `result` has its own precision and you should perform the intermediate computations so that `result` is as tight as possible.

- You can include sollya.h in your implementation and use library functionnalities of `Sollya` for your implementation. However, this requires to have compiled `Sollya` with `-fPIC` in order to make the `Sollya` executable code position independent and to use a system on with programs, using `dlopen` to open dynamic routines can dynamically open themselves.

- To bind your function into `Sollya`, you must use the same identifier as the function name used in your implementation file (`my_ident` in the previous example). Once the function code has been bound to an identifier, you can use a simple assignment to assign the bound identifier to yet another identifier. This way, you may use convenient names inside `Sollya` even if your implementation environment requires you to use a less convenient name.

Example 1:

```
> bashexecute("gcc -fPIC -Wall -c libraryexample.c -I$HOME/.local/include");
> bashexecute("gcc -shared -o libraryexample libraryexample.o -lgmp -lmpfr");
> myownlog = library("./libraryexample");
> evaluate(log(x), 2);
0.693147180559945309417232121458176568075500134360 24
> evaluate(myownlog(x), 2);
0.693147180559945309417232121458176568075500134360 24
```

See also: **function** (8.69), **bashexecute** (8.17), **externalproc** (8.60), **externalplot** (8.59), **diff** (8.39), **evaluate** (8.53), **libraryconstant** (8.92)

8.92 libraryconstant

Name: **libraryconstant**
binds an external mathematical constant to a variable in `Sollya`
Usage:

libraryconstant(*path*) : string → function

Description:

- The command **libraryconstant** lets you extend the set of mathematical constants known to `Sollya`. By default, the only mathematical constant known by `Sollya` is **pi**. For particular applications, one may want to manipulate other constants, such as Euler's gamma constant, for instance.

- **libraryconstant** makes it possible to let `Sollya` know about new constants. In order to let it know, you have to provide an implementation of the constant you are interested in. This implementation is a C file containing a function of the form:

  ```
  void my_ident(mpfr_t result, mp_prec_t prec)
  ```

  The semantic of this function is the following: it is an implementation of the constant in arbitrary precision. `my_ident(result, prec)` shall set the precision of the variable result to a suitable precision (the variable is assumed to be already initialized) and store in result an approximate value of the constant with a relative error not greater than $2^{1-\mathrm{prec}}$. More precisely, if c is the exact value of the constant, the value stored in result should satisfy

  $$|\mathrm{result} - c| \leq |c|\, 2^{1-\mathrm{prec}}.$$

- You can include sollya.h in your implementation and use library functionnalities of `Sollya` for your implementation. However, this requires to have compiled `Sollya` with `-fPIC` in order to make the `Sollya` executable code position independent and to use a system on with programs, using `dlopen` to open dynamic routines can dynamically open themselves.
- To bind your constant into `Sollya`, you must use the same identifier as the function name used in your implementation file (`my_ident` in the previous example). Once the function code has been bound to an identifier, you can use a simple assignment to assign the bound identifier to yet another identifier. This way, you may use convenient names inside `Sollya` even if your implementation environment requires you to use a less convenient name.
- Once your constant is bound, it is considered by `Sollya` as an infinitely accurate constant (i.e. a 0-ary function, exactly like **pi**).

Example 1:

```
> bashexecute("gcc -fPIC -Wall -c libraryconstantexample.c -I$HOME/.local/includ
e");
> bashexecute("gcc -shared -o libraryconstantexample libraryconstantexample.o -l
gmp -lmpfr");
> euler_gamma = libraryconstant("./libraryconstantexample");
> prec = 20!;
> euler_gamma;
0.577215
> prec = 100!;
> euler_gamma;
0.57721566490153286060651209008~
> midpointmode = on;
Midpoint mode has been activated.
> [euler_gamma];
0.57721566490153286060651209008~0/1~
```

See also: **bashexecute** (8.17), **externalproc** (8.60), **externalplot** (8.59), **pi** (8.117), **library** (8.91), **evaluate** (8.53), **implementconstant** (8.80)

8.93 list of

Name: **list of**
keyword used in combination with a type keyword
Description:

- **list of** is used in combination with one of the following keywords for indicating lists of the respective type in declarations of external procedures using **externalproc**: **boolean**, **constant**, **function**, **integer**, **range** and **string**.

See also: **externalproc** (8.60), **boolean** (8.20), **constant** (8.27), **function** (8.69), **integer** (8.85), **range** (8.138), **string** (8.166)

8.94 log

Name: **log**
natural logarithm.
Description:

- **log** is the natural logarithm defined as the inverse of the exponential function: $\log(y)$ is the unique real number x such that $\exp(x) = y$.
- It is defined only for $y \in [0; +\infty]$.

See also: **exp** (8.55), **log2** (8.97), **log10** (8.95)

8.95 log10

Name: **log10**
decimal logarithm.
Description:

- **log10** is the decimal logarithm defined by: $\log10(x) = \log(x)/\log(10)$.
- It is defined only for $x \in [0; +\infty]$.

See also: **log** (8.94), **log2** (8.97)

8.96 log1p

Name: **log1p**
translated logarithm.
Description:

- **log1p** is the function defined by $\log1p(x) = \log(1 + x)$.
- It is defined only for $x \in [-1; +\infty]$.

See also: **log** (8.94)

8.97 log2

Name: **log2**
binary logarithm.
Description:

- **log2** is the binary logarithm defined by: $\log2(x) = \log(x)/\log(2)$.
- It is defined only for $x \in [0; +\infty]$.

See also: **log** (8.94), **log10** (8.95)

8.98 <

Name: <
less-than operator
Usage:

$$expr1 < expr2 : (\mathsf{constant}, \mathsf{constant}) \rightarrow \mathsf{boolean}$$

Parameters:

- *expr1* and *expr2* represent constant expressions

Description:

- The operator < evaluates to true iff its operands *expr1* and *expr2* evaluate to two floating-point numbers a_1 respectively a_2 with the global precision **prec** and a_1 is less than a_2. The user should be aware of the fact that because of floating-point evaluation, the operator < is not exactly the same as the mathematical operation *less-than*.

Example 1:

```
> 5 < 4;
false
> 5 < 5;
false
> 5 < 6;
true
> exp(2) < exp(1);
false
> log(1) < exp(2);
true
```

Example 2:

```
> prec = 12;
The precision has been set to 12 bits.
> 16384.1 < 16385.1;
false
```

See also: == (8.49), **!**= (8.107), >= (8.70), > (8.71), <= (8.89), **in** (8.82), **!** (8.109), && (8.6), || (8.114), **prec** (8.125), **max** (8.100), **min** (8.103)

8.99 mantissa

Name: **mantissa**
returns the integer mantissa of a number.
Usage:

$$\textbf{mantissa}(x) : \mathsf{constant} \to \mathsf{integer}$$

Parameters:

- x is a dyadic number.

Description:

- **mantissa**(x) is by definition x if x equals 0, NaN, or Inf.
- If x is not zero, it can be uniquely written as $x = m \cdot 2^e$ where m is an odd integer and e is an integer. **mantissa**(x) returns m.

Example 1:

```
> a=round(Pi,20,RN);
> e=exponent(a);
> m=mantissa(a);
> m;
411775
> a-m*2^e;
0
```

See also: **exponent** (8.58), **precision** (8.126)

8.100 max

Name: **max**
determines which of given constant expressions has maximum value
Usage:

$$\textbf{max}(expr1,expr2,...,exprn) : (\text{constant}, \text{constant}, ..., \text{constant}) \rightarrow \text{constant}$$
$$\textbf{max}(l) : \text{list} \rightarrow \text{constant}$$

Parameters:

- *expr* are constant expressions.
- *l* is a list of constant expressions.

Description:

- **max** determines which of a given set of constant expressions *expr* has maximum value. To do so, **max** tries to increase the precision used for evaluation until it can decide the ordering or some maximum precision is reached. In the latter case, a warning is printed indicating that there might actually be another expression that has a greater value.
- Even though **max** determines the maximum expression by evaluation, it returns the expression that is maximum as is, i.e. as an expression tree that might be evaluated to any accuracy afterwards.
- **max** can be given either an arbitrary number of constant expressions in argument or a list of constant expressions. The list however must not be end-elliptic.
- Users should be aware that the behavior of **max** follows the IEEE 754-2008 standard with respect to NaNs. In particular, **max** evaluates to NaN if and only if all arguments of **max** are NaNs. This means that NaNs may disappear during computations.

Example 1:

```
> max(1,2,3,exp(5),log(0.25));
1.484131591025766034211155800405522796234876675938 78e2
> max(17);
17
```

Example 2:

```
> l = [|1,2,3,exp(5),log(0.25)|];
> max(l);
1.484131591025766034211155800405522796234876675938 78e2
```

Example 3:

```
> print(max(exp(17),sin(62)));
exp(17)
```

Example 4:

```
> verbosity = 1!;
> print(max(17 + log2(13)/log2(9),17 + log(13)/log(9)));
Warning: maximum computation relies on floating-point result that is faithfully
evaluated and different faithful roundings toggle the result.
17 + log2(13) / log2(9)
```

See also: **min** (8.103), == (8.49), **!**= (8.107), >= (8.70), > (8.71), < (8.98), <= (8.89), **in** (8.82), **inf** (8.83), **sup** (8.169)

8.101 mid

Name: **mid**
gives the middle of an interval.
Usage:

$$\textbf{mid}(I) : \mathsf{range} \to \mathsf{constant}$$
$$\textbf{mid}(x) : \mathsf{constant} \to \mathsf{constant}$$

Parameters:

- I is an interval.
- x is a real number.

Description:

- Returns the middle of the interval I. If the middle is not exactly representable at the current precision, the value is returned as an unevaluated expression.
- When called on a real number x, **mid** considers it as an interval formed of a single point: $[x, x]$. In other words, **mid** behaves like the identity.

Example 1:

```
> mid([1;3]);
2
> mid(17);
17
```

See also: **inf** (8.83), **sup** (8.169)

8.102 midpointmode

Name: **midpointmode**
global variable controlling the way intervals are displayed.
Usage:

$$\textbf{midpointmode} = \textit{activation value} : \mathsf{on}|\mathsf{off} \to \mathsf{void}$$
$$\textbf{midpointmode} = \textit{activation value}\ ! : \mathsf{on}|\mathsf{off} \to \mathsf{void}$$
$$\textbf{midpointmode} : \mathsf{on}|\mathsf{off}$$

Parameters:

- *activation value* enables or disables the mode.

Description:

- **midpointmode** is a global variable. When its value is **off**, intervals are displayed as usual (in the form $[a; b]$). When its value is **on**, and if a and b have the same first significant digits, the interval in displayed in a way that lets one immediately see the common digits of the two bounds.
- This mode is supported only with **display** set to **decimal**. In other modes of display, **midpointmode** value is simply ignored.

Example 1:

```
> a = round(Pi,30,RD);
> b = round(Pi,30,RU);
> d = [a,b];
> d;
[3.14159265160560607910156250;3.14159265533089637756347656250]
> midpointmode=on!;
> d;
0.314159265~1/6~e1
```

See also: **on** (8.113), **off** (8.112), **roundingwarnings** (8.154), **display** (8.43), **decimal** (8.33)

8.103 min

Name: **min**
determines which of given constant expressions has minimum value
Usage:

$$\textbf{min}(expr1,expr2,...,exprn) : (\mathsf{constant}, \mathsf{constant}, ..., \mathsf{constant}) \to \mathsf{constant}$$
$$\textbf{min}(l) : \mathsf{list} \to \mathsf{constant}$$

Parameters:

- *expr* are constant expressions.
- l is a list of constant expressions.

Description:

- **min** determines which of a given set of constant expressions *expr* has minimum value. To do so, **min** tries to increase the precision used for evaluation until it can decide the ordering or some maximum precision is reached. In the latter case, a warning is printed indicating that there might actually be another expression that has a lesser value.
- Even though **min** determines the minimum expression by evaluation, it returns the expression that is minimum as is, i.e. as an expression tree that might be evaluated to any accuracy afterwards.
- **min** can be given either an arbitrary number of constant expressions in argument or a list of constant expressions. The list however must not be end-elliptic.
- Users should be aware that the behavior of **min** follows the IEEE 754-2008 standard with respect to NaNs. In particular, **min** evaluates to NaN if and only if all arguments of **min** are NaNs. This means that NaNs may disappear during computations.

Example 1:

```
> min(1,2,3,exp(5),log(0.25));
-1.3862943611198906188344642429163531361510002687205
> min(17);
17
```

Example 2:

```
> l = [|1,2,3,exp(5),log(0.25)|];
> min(l);
-1.3862943611198906188344642429163531361510002687205
```

Example 3:

```
> print(min(exp(17),sin(62)));
sin(62)
```

Example 4:

```
> verbosity = 1!;
> print(min(17 + log2(13)/log2(9),17 + log(13)/log(9)));
Warning: minimum computation relies on floating-point result that is faithfully
evaluated and different faithful roundings toggle the result.
17 + log(13) / log(9)
```

See also: **max** (8.100), == (8.49), **!**= (8.107), >= (8.70), > (8.71), < (8.98), <= (8.89), **in** (8.82), **inf** (8.83), **sup** (8.169)

8.104 −

Name: −
subtraction function
Usage:

$$\begin{aligned} \mathit{function1} - \mathit{function2} &: (\mathsf{function}, \mathsf{function}) \to \mathsf{function} \\ \mathit{interval1} - \mathit{interval2} &: (\mathsf{range}, \mathsf{range}) \to \mathsf{range} \\ \mathit{interval1} - \mathit{constant} &: (\mathsf{range}, \mathsf{constant}) \to \mathsf{range} \\ \mathit{interval1} - \mathit{constant} &: (\mathsf{constant}, \mathsf{range}) \to \mathsf{range} \\ - \mathit{function1} &: \mathsf{function} \to \mathsf{function} \\ - \mathit{interval1} &: \mathsf{range} \to \mathsf{range} \end{aligned}$$

Parameters:

- *function1* and *function2* represent functions
- *interval1* and *interval2* represent intervals (ranges)
- *constant* represents a constant or constant expression

Description:

- − represents the subtraction (function) on reals. The expression *function1* − *function2* stands for the function composed of the subtraction function and the two functions *function1* and *function2*, where *function1* is the subtrahend and *function2* the subtractor.
- − can be used for interval arithmetic on intervals (ranges). − will evaluate to an interval that safely encompasses all images of the subtraction function with arguments varying in the given intervals. Any combination of intervals with intervals or constants (resp. constant expressions) is supported. However, it is not possible to represent families of functions using an interval as one argument and a function (varying in the free variable) as the other one.
- − stands also for the negation function.

Example 1:

```
> 5 - 2;
3
```

Example 2:

```
> x - 2;
-2 + x
```

Example 3:

```
> x - x;
0
```

Example 4:

```
> diff(sin(x) - exp(x));
cos(x) - exp(x)
```

Example 5:

```
> [1;2] - [3;4];
[-3;-1]
> [1;2] - 17;
[-16;-15]
> 13 - [-4;17];
[-4;17]
```

Example 6:

```
> -exp(x);
-exp(x)
> -13;
-13
> -[13;17];
[-17;-13]
```

See also: + (8.119), ∗ (8.105), / (8.44), ^ (8.123)

8.105 ∗

Name: ∗
multiplication function
Usage:

$$\begin{aligned} \textit{function1} * \textit{function2} &: (\mathsf{function}, \mathsf{function}) \to \mathsf{function} \\ \textit{interval1} * \textit{interval2} &: (\mathsf{range}, \mathsf{range}) \to \mathsf{range} \\ \textit{interval1} * \textit{constant} &: (\mathsf{range}, \mathsf{constant}) \to \mathsf{range} \\ \textit{interval1} * \textit{constant} &: (\mathsf{constant}, \mathsf{range}) \to \mathsf{range} \end{aligned}$$

Parameters:

- *function1* and *function2* represent functions
- *interval1* and *interval2* represent intervals (ranges)
- *constant* represents a constant or constant expression

Description:

- ∗ represents the multiplication (function) on reals. The expression *function1* ∗ *function2* stands for the function composed of the multiplication function and the two functions *function1* and *function2*.
- ∗ can be used for interval arithmetic on intervals (ranges). ∗ will evaluate to an interval that safely encompasses all images of the multiplication function with arguments varying in the given intervals. Any combination of intervals with intervals or constants (resp. constant expressions) is supported. However, it is not possible to represent families of functions using an interval as one argument and a function (varying in the free variable) as the other one.

Example 1:

```
> 5 * 2;
10
```

Example 2:

```
> x * 2;
x * 2
```

Example 3:

```
> x * x;
x^2
```

Example 4:

```
> diff(sin(x) * exp(x));
sin(x) * exp(x) + exp(x) * cos(x)
```

Example 5:

```
> [1;2] * [3;4];
[3;8]
> [1;2] * 17;
[17;34]
> 13 * [-4;17];
[-52;221]
```

See also: + (8.119), − (8.104), / (8.44), ^ (8.123)

8.106 nearestint

Name: **nearestint**
the function mapping the reals to the integers nearest to them.
Description:

- **nearestint** is defined as usual: **nearestint**(x) is the integer nearest to x, with the special rule that the even integer is chosen if there exist two integers equally near to x.
- It is defined for every real number x.

See also: **ceil** (8.22), **floor** (8.66), **round** (8.151), **RN** (8.150)

8.107 !=

Name: **!=**
negated equality test operator
Usage:

expr1 **!=** *expr2* : (any type, any type) → boolean

Parameters:

- *expr1* and *expr2* represent expressions

Description:

- The operator **!=** evaluates to true iff its operands *expr1* and *expr2* are syntactically unequal and both different from **error** or constant expressions that are not constants and that evaluate to two different floating-point number with the global precision **prec**. The user should be aware of the fact that because of floating-point evaluation, the operator **!=** is not exactly the same as the negation of the mathematical equality.

  Note that the expressions **!**(*expr1* **!=** *expr2*) and *expr1* == *expr2* do not evaluate to the same boolean value. See **error** for details.

Example 1:

```
> "Hello" != "Hello";
false
> "Hello" != "Salut";
true
> "Hello" != 5;
true
> 5 + x != 5 + x;
false
```

Example 2:

```
> 1 != exp(0);
false
> asin(1) * 2 != pi;
false
> exp(5) != log(4);
true
```

Example 3:

```
> sin(pi/6) != 1/2 * sqrt(3);
true
```

Example 4:

```
> prec = 12;
The precision has been set to 12 bits.
> 16384.1 != 16385.1;
false
```

Example 5:

```
> error != error;
false
```

See also: == (8.49), > (8.71), >= (8.70), <= (8.89), < (8.98), **in** (8.82), **!** (8.109), && (8.6), || (8.114), **error** (8.52), **prec** (8.125)

8.108 nop

Name: **nop**
no operation
Usage:

$$\textbf{nop} : \mathsf{void} \to \mathsf{void}$$
$$\textbf{nop}() : \mathsf{void} \to \mathsf{void}$$
$$\textbf{nop}(n) : \mathsf{integer} \to \mathsf{void}$$

Description:

- The command **nop** does nothing. This means it is an explicit parse element in the `Sollya` language that finally does not produce any result or side-effect.
- The command **nop** may take an optional positive integer argument n. The argument controls how much (useless) integer additions `Sollya` performs while doing nothing. With this behaviour, **nop** can be used for calibration of timing tests.
- The keyword **nop** is implicit in some procedure definitions. Procedures without imperative body get parsed as if they had an imperative body containing one **nop** statement.

Example 1:

```
> nop;
```

Example 2:

```
> nop(100);
```

Example 3:

```
> succ = proc(n) { return n + 1; };
> succ;
proc(n)
{
nop;
return (n) + (1);
}
> succ(5);
6
```

See also: **proc** (8.133), **time** (8.178)

8.109 !

Name: **!**
boolean NOT operator
Usage:

! *expr* : boolean → boolean

Parameters:

- *expr* represents a boolean expression

Description:

- **!** evaluates to the boolean NOT of the boolean expression *expr*. **!** *expr* evaluates to true iff *expr* does not evaluate to true.

Example 1:

```
> ! false;
true
```

Example 2:

```
> ! (1 == exp(0));
false
```

See also: && (8.6), || (8.114)

8.110 numberroots

Name: **numberroots**
Computes the number of roots of a polynomial in a given range.
Usage:

numberroots(p, I) : (function, range) → integer

Parameters:

- p is a polynomial.
- I is an interval.

Description:

- **numberroots** rigorously computes the number of roots of polynomial the p in the interval I. The technique used is Sturm's algorithm. The value returned is not just a numerical estimation of the number of roots of p in I: it is the exact number of roots.

- The command **findzeros** computes safe enclosures of all the zeros of a function, without forgetting any, but it is not guaranteed to separate them all in distinct intervals. **numberroots** is more accurate since it guarantees the exact number of roots. However, it does not compute them. It may be used, for instance, to certify that **findzeros** did not put two distinct roots in the same interval.
- Multiple roots are counted only once.
- The interval I must be bounded. The algorithm cannot handle unbounded intervals. Moreover, the interval is considered as a closed interval: if one (or both) of the endpoints of I are roots of p, they are counted.
- The argument p can be any expression, but if `Sollya` fails to prove that it is a polynomial an error is produced. Also, please note that if the coefficients of p or the endpoints of I are not exactly representable, they are first numerically evaluated, before the algorithm is used. In that case, the counted number of roots corresponds to the rounded polynomial on the rounded interval **and not** to the exact parameters given by the user. A warning is displayed to inform the user.

Example 1:

```
> numberroots(1+x-x^2, [1,2]);
1
> findzeros(1+x-x^2, [1,2]);
[|[1.617919921875;1.6180419921875]|]
```

Example 2:

```
> numberroots((1+x)*(1-x), [-1,1]);
2
> numberroots(x^2, [-1,1]);
1
```

Example 3:

```
> verbosity = 1!;
> numberroots(x-pi, [0,4]);
Warning: the 0th coefficient of the polynomial is neither a floating point
constant nor can be evaluated without rounding to a floating point constant.
Will faithfully evaluate it with the current precision (165 bits)
1
```

Example 4:

```
> verbosity = 1!;
> numberroots(1+x-x^2, [0, @Inf@]);
Warning: the given interval must have finite bounds.
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
> numberroots(exp(x), [0, 1]);
Warning: the given function must be a polynomial in this context.
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
```

See also: **dirtyfindzeros** (8.40), **findzeros** (8.63)

8.111 numerator

Name: **numerator**
gives the numerator of an expression
Usage:

numerator(*expr*) : function → function

Parameters:

- *expr* represents an expression

Description:

- If *expr* represents a fraction *expr1*/*expr2*, **numerator**(*expr*) returns the numerator of this fraction, i.e. *expr1*.

  If *expr* represents something else, **numerator**(*expr*) returns the expression itself, i.e. *expr*.

  Note that for all expressions *expr*, **numerator**(*expr*) **/ denominator**(*expr*) is equal to *expr*.

Example 1:

```
> numerator(5/3);
5
```

Example 2:

```
> numerator(exp(x));
exp(x)
```

Example 3:

```
> a = 5/3;
> b = numerator(a)/denominator(a);
> print(a);
5 / 3
> print(b);
5 / 3
```

Example 4:

```
> a = exp(x/3);
> b = numerator(a)/denominator(a);
> print(a);
exp(x / 3)
> print(b);
exp(x / 3)
```

See also: **denominator** (8.36), **rationalmode** (8.140)

8.112 off

Name: **off**
special value for certain global variables.
Description:

- **off** is a special value used to deactivate certain functionnalities of `Sollya`.
- As any value it can be affected to a variable and stored in lists.

Example 1:

```
> canonical=on;
Canonical automatic printing output has been activated.
> p=1+x+x^2;
> mode=off;
> p;
1 + x + x^2
> canonical=mode;
Canonical automatic printing output has been deactivated.
> p;
1 + x * (1 + x)
```

See also: **on** (8.113), **autosimplify** (8.15), **canonical** (8.21), **timing** (8.179), **fullparentheses** (8.68), **midpointmode** (8.102), **rationalmode** (8.140), **roundingwarnings** (8.154), **timing** (8.179), **dieonerrormode** (8.38)

8.113 on

Name: **on**
special value for certain global variables.
Description:

- **on** is a special value used to activate certain functionnalities of `Sollya`.
- As any value it can be affected to a variable and stored in lists.

Example 1:

```
> p=1+x+x^2;
> mode=on;
> p;
1 + x * (1 + x)
> canonical=mode;
Canonical automatic printing output has been activated.
> p;
1 + x + x^2
```

See also: **off** (8.112), **autosimplify** (8.15), **canonical** (8.21), **timing** (8.179), **fullparentheses** (8.68), **midpointmode** (8.102), **rationalmode** (8.140), **roundingwarnings** (8.154), **timing** (8.179), **dieonerrormode** (8.38)

8.114 ||

Name: ||
boolean OR operator
Usage:

$$\textit{expr1} \;||\; \textit{expr2} : (\mathsf{boolean}, \mathsf{boolean}) \to \mathsf{boolean}$$

Parameters:

- *expr1* and *expr2* represent boolean expressions

Description:

- || evaluates to the boolean OR of the two boolean expressions *expr1* and *expr2*. || evaluates to true iff at least one of *expr1* or *expr2* evaluates to true.

Example 1:

```
> false || false;
false
```

Example 2:

```
> (1 == exp(0)) || (0 == log(1));
true
```

See also: && (8.6), **!** (8.109)

8.115 parse

Name: **parse**
parses an expression contained in a string
Usage:

parse(*string*) : string → function | error

Parameters:

- *string* represents a character sequence

Description:

- **parse**(*string*) parses the character sequence *string* containing an expression built on constants and base functions.

  If the character sequence does not contain a well-defined expression, a warning is displayed indicating a syntax error and **parse** returns a **error** of type error.

- The character sequence to be parsed by **parse** may contain commands that return expressions, including **parse** itself. Those commands get executed after the string has been parsed. **parse**(*string*) will return the expression computed by the commands contained in the character sequence *string*.

Example 1:

```
> parse("exp(x)");
exp(x)
```

Example 2:

```
> text = "remez(exp(x),5,[-1;1])";
> print("The string", text, "gives", parse(text));
The string remez(exp(x),5,[-1;1]) gives 1.000044750290550706430770524820533987654
426158966754 + x * (1.000038346529839707352445411245040338175442330753 56 + x * (
0.499196982628829864921688244942403747719690128612 97 + x * (0.166424656075155194
415920597322727380932279602909199 + x * (4.379369638732804702712575662071834966 5
9575464236489e-2 + x * 8.738191038806555114015842027833096047996047671337 6e-3)))
)
```

Example 3:

```
> verbosity = 1!;
> parse("5 + * 3");
Warning: syntax error, unexpected MULTOKEN. Will try to continue parsing (expect
ing ";"). May leak memory.
Warning: the string "5 + * 3" could not be parsed by the miniparser.
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
```

See also: **execute** (8.54), **readfile** (8.142), **print** (8.128), **error** (8.52), **dieonerrormode** (8.38)

8.116 perturb

Name: **perturb**
indicates random perturbation of sampling points for **externalplot**
Usage:

perturb : perturb

Description:

- The use of **perturb** in the command **externalplot** enables the addition of some random noise around each sampling point in **externalplot**.

  See **externalplot** for details.

Example 1:

```
> bashexecute("gcc -fPIC -c externalplotexample.c");
> bashexecute("gcc -shared -o externalplotexample externalplotexample.o -lgmp -l
mpfr");
> externalplot("./externalplotexample",relative,exp(x),[-1/2;1/2],12,perturb);
```

See also: **externalplot** (8.59), **absolute** (8.2), **relative** (8.144), **bashexecute** (8.17)

8.117 pi

Name: **pi**
the constant π.
Description:

- **pi** is the constant π, defined as half the period of sine and cosine.
- In `Sollya`, **pi** is considered a 0-ary function. This way, the constant is not evaluated at the time of its definition but at the time of its use. For instance, when you define a constant or a function relating to π, the current precision at the time of the definition does not matter. What is important is the current precision when you evaluate the function or the constant value.
- Remark that when you define an interval, the bounds are first evaluated and then the interval is defined. In this case, **pi** will be evaluated as any other constant value at the definition time of the interval, thus using the current precision at this time.

Example 1:

```
> verbosity=1!; prec=12!;
> a = 2*pi;
> a;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
6.283
> prec=20!;
> a;
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
6.283187
```

Example 2:

```
> display=binary;
Display mode is binary numbers.
> prec=12!;
> d = [pi; 5];
> d;
[1.1001001_2 * 2^(1);1.01_2 * 2^(2)]
> prec=20!;
> d;
[1.1001001_2 * 2^(1);1.01_2 * 2^(2)]
```

See also: **cos** (8.28), **sin** (8.161), **tan** (8.172), **asin** (8.10), **acos** (8.4), **atan** (8.12), **evaluate** (8.53), **prec** (8.125), **libraryconstant** (8.92)

8.118 plot

Name: **plot**
plots one or several functions
Usage:

plot(*f1*, ... ,*fn*, *I*) : (function, ... ,function, range) → void
plot(*f1*, ... ,*fn*, *I*, **file**, *name*) : (function, ... ,function, range, **file**, string) → void
plot(*f1*, ... ,*fn*, *I*, **postscript**, *name*) : (function, ... ,function, range, **postscript**, string) → void
plot(*f1*, ... ,*fn*, *I*, **postscriptfile**, *name*) : (function, ... ,function, range, **postscriptfile**, string) → void
plot(*L*, *I*) : (list, range) → void
plot(*L*, *I*, **file**, *name*) : (list, range, **file**, string) → void
plot(*L*, *I*, **postscript**, *name*) : (list, range, **postscript**, string) → void
plot(*L*, *I*, **postscriptfile**, *name*) : (list, range, **postscriptfile**, string) → void

Parameters:

- *f1*, ..., *fn* are functions to be plotted.
- *L* is a list of functions to be plotted.
- *I* is the interval where the functions have to be plotted.
- *name* is a string representing the name of a file.

Description:

- This command plots one or several functions *f1*, ... ,*fn* on an interval *I*. Functions can be either given as parameters of **plot** or as a list *L* which elements are functions. The functions are drawn on the same plot with different colors.
- If *L* contains an element that is not a function (or a constant), an error occurs.
- **plot** relies on the value of global variable **points**. Let *n* be the value of this variable. The algorithm is the following: each function is evaluated at *n* evenly distributed points in *I*. At each point, the computed value is a faithful rounding of the exact value with a sufficiently high precision. Each point is finally plotted. This should avoid numerical artefacts such as critical cancellations.
- You can save the function plot either as a data file or as a postscript file.
- If you use argument **file** with a string *name*, `Sollya` will save a data file called name.dat and a gnuplot directives file called name.p. Invoking gnuplot on name.p will plot the data stored in name.dat.
- If you use argument **postscript** with a string *name*, `Sollya` will save a postscript file called name.eps representing your plot.

- If you use argument **postscriptfile** with a string *name*, `Sollya` will produce the corresponding name.dat, name.p and name.eps.
- This command uses gnuplot to produce the final plot. If your terminal is not graphic (typically if you use `Sollya` through ssh without -X) gnuplot should be able to detect that and produce an ASCII-art version on the standard output. If it is not the case, you can either store the plot in a postscript file to view it locally, or use **asciiplot** command.
- If every function is constant, **plot** will not plot them but just display their value.
- If the interval is reduced to a single point, **plot** will just display the value of the functions at this point.

Example 1:

```
> plot(sin(x),0,cos(x),[-Pi,Pi]);
```

Example 2:

```
> plot(sin(x),0,cos(x),[-Pi,Pi],postscriptfile,"plotSinCos");
```

Example 3:

```
> plot(exp(0), sin(1), [0;1]);
1
0.84147098480789650665250232163029899962256306079837
```

Example 4:

```
> plot(sin(x), cos(x), [1;1]);
0.84147098480789650665250232163029899962256306079837
0.54030230586813971740093660744297660373231042061792
```

See also: **externalplot** (8.59), **asciiplot** (8.9), **file** (8.62), **postscript** (8.121), **postscriptfile** (8.122), **points** (8.120)

8.119 +

Name: +
addition function
Usage:

$$\begin{aligned} \textit{function1} + \textit{function2} &: (\mathsf{function}, \mathsf{function}) \to \mathsf{function} \\ \textit{interval1} + \textit{interval2} &: (\mathsf{range}, \mathsf{range}) \to \mathsf{range} \\ \textit{interval1} + \textit{constant} &: (\mathsf{range}, \mathsf{constant}) \to \mathsf{range} \\ \textit{interval1} + \textit{constant} &: (\mathsf{constant}, \mathsf{range}) \to \mathsf{range} \end{aligned}$$

Parameters:

- *function1* and *function2* represent functions
- *interval1* and *interval2* represent intervals (ranges)
- *constant* represents a constant or constant expression

Description:

- + represents the addition (function) on reals. The expression *function1* + *function2* stands for the function composed of the addition function and the two functions *function1* and *function2*.

- + can be used for interval arithmetic on intervals (ranges). + will evaluate to an interval that safely encompasses all images of the addition function with arguments varying in the given intervals. Any combination of intervals with intervals or constants (resp. constant expressions) is supported. However, it is not possible to represent families of functions using an interval as one argument and a function (varying in the free variable) as the other one.

Example 1:

```
> 1 + 2;
3
```

Example 2:

```
> x + 2;
2 + x
```

Example 3:

```
> x + x;
x * 2
```

Example 4:

```
> diff(sin(x) + exp(x));
cos(x) + exp(x)
```

Example 5:

```
> [1;2] + [3;4];
[4;6]
> [1;2] + 17;
[18;19]
> 13 + [-4;17];
[9;30]
```

See also: − (8.104), ∗ (8.105), / (8.44), ^ (8.123)

8.120 points

Name: **points**
controls the number of points chosen by `Sollya` in certain commands.
Usage:

$$\textbf{points} = n : \mathsf{integer} \to \mathsf{void}$$
$$\textbf{points} = n\ ! : \mathsf{integer} \to \mathsf{void}$$
$$\textbf{points} : \mathsf{constant}$$

Parameters:

- n represents the number of points

Description:

- **points** is a global variable. Its value represents the number of points used in numerical algorithms of `Sollya` (namely **dirtyinfnorm**, **dirtyintegral**, **dirtyfindzeros**, **plot**).

Example 1:

```
> f=x^2*sin(1/x);
> points=10;
The number of points has been set to 10.
> dirtyfindzeros(f, [0;1]);
[|0, 0.31830988618379067153776752674502872406891929148091 8|]
> points=100;
The number of points has been set to 100.
> dirtyfindzeros(f, [0;1]);
[|0, 2.448537586029159011828980974961759415914763780622 4e-2, 3.97887357729738339
422209408431285905086149114351147e-2, 4.54728408833986673625382181064326748669884
702115589e-2, 5.30516476972984452562945877908381206781532152468 19e-2, 6.3661977
236758134307553505349005744813783858296183e-2, 7.95774715459476678844418816862 57
181017229822870229e-2, 0.10610329539459689051258917558167624135630643049363 8, 0.
159154943091895335768883763372514362034459645740459, 0.31830988618379067153776752
674502872406891929148091 8|]
```

See also: **dirtyinfnorm** (8.41), **dirtyintegral** (8.42), **dirtyfindzeros** (8.40), **plot** (8.118), **diam** (8.37), **prec** (8.125)

8.121 postscript

Name: **postscript**
special value for commands **plot** and **externalplot**
Description:

- **postscript** is a special value used in commands **plot** and **externalplot** to save the result of the command in a postscript file.
- As any value it can be affected to a variable and stored in lists.

Example 1:

```
> savemode=postscript;
> name="plotSinCos";
> plot(sin(x),0,cos(x),[-Pi,Pi],savemode, name);
```

See also: **externalplot** (8.59), **plot** (8.118), **file** (8.62), **postscriptfile** (8.122)

8.122 postscriptfile

Name: **postscriptfile**
special value for commands **plot** and **externalplot**
Description:

- **postscriptfile** is a special value used in commands **plot** and **externalplot** to save the result of the command in a data file and a postscript file.
- As any value it can be affected to a variable and stored in lists.

Example 1:

```
> savemode=postscriptfile;
> name="plotSinCos";
> plot(sin(x),0,cos(x),[-Pi,Pi],savemode, name);
```

See also: **externalplot** (8.59), **plot** (8.118), **file** (8.62), **postscript** (8.121)

8.123 ^

Name: ^
power function
Usage:

function1 ^ *function2* : (function, function) → function
interval1 ^ *interval2* : (range, range) → range
interval1 ^ *constant* : (range, constant) → range
interval1 ^ *constant* : (constant, range) → range

Parameters:

- *function1* and *function2* represent functions
- *interval1* and *interval2* represent intervals (ranges)
- *constant* represents a constant or constant expression

Description:

- ^ represents the power (function) on reals. The expression *function1* ^ *function2* stands for the function composed of the power function and the two functions *function1* and *function2*, where *function1* is the base and *function2* the exponent. If *function2* is a constant integer, ^ is defined on negative values of *function1*. Otherwise ^ is defined as $e^{y \cdot \ln x}$.
- Note that whenever several ^ are composed, the priority goes to the last ^. This corresponds to the natural way of thinking when a tower of powers is written on a paper. Thus, `2^3^5` is read as 2^{3^5} and is interpreted as $2^{(3^5)}$.
- ^ can be used for interval arithmetic on intervals (ranges). ^ will evaluate to an interval that safely encompasses all images of the power function with arguments varying in the given intervals. If the intervals given contain points where the power function is not defined, infinities and NaNs will be produced in the output interval. Any combination of intervals with intervals or constants (resp. constant expressions) is supported. However, it is not possible to represent families of functions using an interval as one argument and a function (varying in the free variable) as the other one.

Example 1:

```
> 5 ^ 2;
25
```

Example 2:

```
> x ^ 2;
x^2
```

Example 3:

```
> 3 ^ (-5);
4.11522633744855967078189300411522633744855967078 18e-3
```

Example 4:

```
> (-3) ^ (-2.5);
@NaN@
```

Example 5:

```
> diff(sin(x) ^ exp(x));
sin(x)^exp(x) * ((cos(x) * exp(x)) / sin(x) + exp(x) * log(sin(x)))
```

Example 6:

```
> 2^3^5;
1.4134776518227074636666380005943348126619871175005e73
> (2^3)^5;
32768
> 2^(3^5);
1.4134776518227074636666380005943348126619871175005e73
```

Example 7:

```
> [1;2] ^ [3;4];
[1;1.60000000000000000000000000000000000000000000000007e1]
> [1;2] ^ 17;
[1;131072]
> 13 ^ [-4;17];
[3.501277966457757081334687160813696999404782745702e-5;86504159193813379 33]
```

See also: + (8.119), − (8.104), ∗ (8.105), / (8.44)

8.124 powers

Name: **powers**
special value for global state **display**
Description:

- **powers** is a special value used for the global state **display**. If the global state **display** is equal to **powers**, all data will be output in dyadic notation with numbers displayed in a Maple and PARI/GP compatible format.

  As any value it can be affected to a variable and stored in lists.

See also: **decimal** (8.33), **dyadic** (8.48), **hexadecimal** (8.75), **binary** (8.18), **display** (8.43)

8.125 prec

Name: **prec**
controls the precision used in numerical computations.
Description:

- **prec** is a global variable. Its value represents the precision of the floating-point format used in numerical computations.
- Many commands try to adapt their working precision in order to have approximately n correct bits in output, where n is the value of **prec**.

Example 1:

```
> display=binary!;
> prec=50;
The precision has been set to 50 bits.
> dirtyinfnorm(exp(x),[1;2]);
1.11011000111001100100101110001101010011011101 1011_2 * 2^(2)
> prec=100;
The precision has been set to 100 bits.
> dirtyinfnorm(exp(x),[1;2]);
1.11011000111001100100101110001101010011011101 1010110011100001100111010001110
11101000100000011011_2 * 2^(2)
```

See also: **evaluate** (8.53), **diam** (8.37)

8.126 precision

Name: **precision**
returns the precision necessary to represent a number.
Usage:

$$\textbf{precision}(x) : \mathsf{constant} \to \mathsf{integer}$$

Parameters:

- x is a dyadic number.

Description:

- **precision**(x) is by definition $|x|$ if x equals 0, NaN, or Inf.
- If x is not zero, it can be uniquely written as $x = m \cdot 2^e$ where m is an odd integer and e is an integer. **precision**(x) returns the number of bits necessary to write m in binary (i.e. $\lceil \log_2(m) \rceil$).

Example 1:

```
> a=round(Pi,20,RN);
> precision(a);
19
> m=mantissa(a);
> ceil(log2(m));
19
```

See also: **mantissa** (8.99), **exponent** (8.58), **round** (8.151)

8.127 .:

Name: **.:**
add an element at the beginning of a list.
Usage:

$$x\textbf{.:}L : (\mathsf{any\ type}, \mathsf{list}) \to \mathsf{list}$$

Parameters:

- x is an object of any type.
- L is a list (possibly empty).

Description:

- **.:** adds the element x at the beginning of the list L.
- Note that since x may be of any type, it can be in particular a list.

Example 1:

```
> 1.:[|2,3,4|];
[|1, 2, 3, 4|]
```

Example 2:

```
> [|1,2,3|].:[|4,5,6|];
[|[|1, 2, 3|], 4, 5, 6|]
```

Example 3:

```
> 1.:[||];
[|1|]
```

See also: **:.** (8.7), **@** (8.26)

8.128 print

Name: **print**
prints an expression
Usage:

print(*expr1*,...,*exprn*) : (any type,..., any type) → void
print(*expr1*,...,*exprn*) > *filename* : (any type,..., any type, string) → void
print(*expr1*,...,*exprn*) >> *filename* : (any type,...,any type, string) → void

Parameters:

- *expr* represents an expression
- *filename* represents a character sequence indicating a file name

Description:

- **print**(*expr1*,...,*exprn*) prints the expressions *expr1* through *exprn* separated by spaces and followed by a newline.

  If a second argument *filename* is given after a single ">", the displaying is not output on the standard output of `Sollya` but if in the file *filename* that get newly created or overwritten. If a double ">>" is given, the output will be appended to the file *filename*.

  The global variables **display**, **midpointmode** and **fullparentheses** have some influence on the formatting of the output (see **display**, **midpointmode** and **fullparentheses**).

  Remark that if one of the expressions *expri* given in argument is of type string, the character sequence *expri* evaluates to is displayed. However, if *expri* is of type list and this list contains a variable of type string, the expression for the list is displayed, i.e. all character sequences get displayed surrounded by double quotes ("). Nevertheless, escape sequences used upon defining character sequences are interpreted immediately.

Example 1:

```
> print(x + 2 + exp(sin(x)));
x + 2 + exp(sin(x))
> print("Hello","world");
Hello world
> print("Hello","you", 4 + 3, "other persons.");
Hello you 7 other persons.
```

Example 2:

```
> print("Hello");
Hello
> print([|"Hello"|]);
[|"Hello"|]
> s = "Hello";
> print(s,[|s|]);
Hello [|"Hello"|]
> t = "Hello\tyou";
> print(t,[|t|]);
Hello    you [|"Hello\tyou"|]
```

Example 3:

```
> print(x + 2 + exp(sin(x))) > "foo.sol";
> readfile("foo.sol");
x + 2 + exp(sin(x))
```

Example 4:

```
> print(x + 2 + exp(sin(x))) >> "foo.sol";
```

Example 5:

```
> display = decimal;
Display mode is decimal numbers.
> a = evaluate(sin(pi * x), 0.25);
> b = evaluate(sin(pi * x), [0.25; 0.25 + 1b-50]);
> print(a);
0.707106781186547524400844362104849039284835937688 47
> display = binary;
Display mode is binary numbers.
> print(a);
1.011010100000100111100110011001111111001110111100110010010000100010110010111110
110001001101100110111010101001010101111101001111100011101011011110110000010111101
010001_2 * 2^(-1)
> display = hexadecimal;
Display mode is hexadecimal numbers.
> print(a);
0xb.504f333f9de6484597d89b3754abe9f1d6f60ba88p-4
> display = dyadic;
Display mode is dyadic numbers.
> print(a);
330700069911015586133239834882209443600671071332 65b-165
> display = powers;
Display mode is dyadic numbers in integer-power-of-2 notation.
> print(a);
330700069911015586133239834882209443600671071332 65 * 2^(-165)
> display = decimal;
Display mode is decimal numbers.
> midpointmode = off;
Midpoint mode has been deactivated.
> print(b);
[0.707106781186547524400844362104849039284835937688 44;0.707106781186549497437217
825175573477826462744170 48]
> midpointmode = on;
Midpoint mode has been activated.
> print(b);
0.7071067811865~4/5~
> display = dyadic;
Display mode is dyadic numbers.
> print(b);
[206687543694384741333274896801380902250419419582 9b-161;165350034955508254441962
370193859364144326751565 71b-164]
> display = decimal;
Display mode is decimal numbers.
> autosimplify = off;
Automatic pure tree simplification has been deactivated.
> fullparentheses = off;
Full parentheses mode has been deactivated.
> print(x + x * ((x + 1) + 1));
x + x * (x + 1 + 1)
> fullparentheses = on;
Full parentheses mode has been activated.
> print(x + x * ((x + 1) + 1));
x + (x * ((x + 1) + 1))
```

See also: **write** (8.186), **printexpansion** (8.130), **printdouble** (8.129), **printsingle** (8.131), **printxml** (8.132), **readfile** (8.142), **autosimplify** (8.15), **display** (8.43), **midpointmode** (8.102), **fullparentheses** (8.68), **evaluate** (8.53), **rationalmode** (8.140)

8.129 printdouble

Name: **printdouble**
prints a constant value as a hexadecimal double precision number
Usage:

printdouble(*constant*) : constant → void

Parameters:

- *constant* represents a constant

Description:

- Prints a constant value as a hexadecimal number on 16 hexadecimal digits. The hexadecimal number represents the integer equivalent to the 64 bit memory representation of the constant considered as a double precision number.

  If the constant value does not hold on a double precision number, it is first rounded to the nearest double precision number before displayed. A warning is displayed in this case.

Example 1:

```
> printdouble(3);
0x4008000000000000
```

Example 2:

```
> prec=100!;
> verbosity = 1!;
> printdouble(exp(5));
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
Warning: rounding down occurred before printing a value as a double.
0x40628d389970338f
```

See also: **printsingle** (8.131), **printexpansion** (8.130), **double** (8.45)

8.130 printexpansion

Name: **printexpansion**
prints a polynomial in Horner form with its coefficients written as a expansions of double precision numbers
Usage:

printexpansion(*polynomial*) : function → void

Parameters:

- *polynomial* represents the polynomial to be printed

Description:

- The command **printexpansion** prints the polynomial *polynomial* in Horner form writing its coefficients as expansions of double precision numbers. The double precision numbers themselves are displayed in hexadecimal memory notation (see **printdouble**).

  If some of the coefficients of the polynomial *polynomial* are not floating-point constants but constant expressions, they are evaluated to floating-point constants using the global precision **prec**. If a rounding occurs in this evaluation, a warning is displayed.

  If the exponent range of double precision is not sufficient to display all the mantissa bits of a coefficient, the coefficient is displayed rounded and a warning is displayed.

  If the argument *polynomial* does not a polynomial, nothing but a warning or a newline is displayed. Constants can be displayed using **printexpansion** since they are polynomials of degree 0.

Example 1:

```
> printexpansion(roundcoefficients(taylor(exp(x),5,0),[|DD...|]));
0x3ff0000000000000 + x * (0x3ff0000000000000 + x * (0x3fe0000000000000 + x * ((0
x3fc5555555555555 + 0x3c65555555555555) + x * ((0x3fa5555555555555 + 0x3c4555555
5555555) + x * (0x3f81111111111111 + 0x3c01111111111111)))))
```

Example 2:

```
> printexpansion(remez(exp(x),5,[-1;1]));
(0x3ff0002eec908ce9 + 0xbc7df99eb225af5b + 0xb8d55834b08b1f18) + x * ((0x3ff0002
835917719 + 0x3c6d82c073b25ebf + 0xb902cf062b54b7b6 + 0x35b0000000000000) + x *
((0x3fdff2d7e6a9c5e9 + 0xbc7b09a95b0d520f + 0xb915b639add55731 + 0x35a8000000000
000) + x * ((0x3fc54d67338ba09f + 0x3c4867596d0631cf + 0xb8ef0756bdb4af6e) + x *
 ((0x3fa66c209b825167 + 0x3c45ec5b6655b076 + 0xb8d8c125286400bc) + x * (0x3f81e5
5425e72ab4 + 0x3c263b25a1bf597b + 0xb8c843e0401dadd0 + 0x3540000000000000)))))
```

Example 3:

```
> verbosity = 1!;
> prec = 3500!;
> printexpansion(pi);
(0x400921fb54442d18 + 0x3ca1a62633145c07 + 0xb92f1976b7ed8fbc + 0x35c4cf98e80417
7d + 0x32631d89cd9128a5 + 0x2ec0f31c6809bbdf + 0x2b5519b3cd3a431b + 0x27e8158536
f92f8a + 0x246ba7f09ab6b6a9 + 0xa0eedd0dbd2544cf + 0x1d779fb1bd1310ba + 0x1a1a63
7ed6b0bff6 + 0x96aa485fca40908e + 0x933e501295d98169 + 0x8fd160dbee83b4e0 + 0x8c
59b6d799ae131c + 0x08f6cf70801f2e28 + 0x05963bf0598da483 + 0x023871574e69a459 +
0x8000000005702db3 + 0x8000000000000000)
Warning: the expansion is not complete because of the limited exponent range of
double precision.
Warning: rounding occurred while printing.
```

See also: **printdouble** (8.129), **horner** (8.78), **print** (8.128), **prec** (8.125), **remez** (8.145), **taylor** (8.174), **roundcoefficients** (8.152), **fpminimax** (8.67), **implementpoly** (8.81)

8.131 printsingle

Name: **printsingle**
prints a constant value as a hexadecimal single precision number
Usage:

printsingle(*constant*) : constant → void

Parameters:

- *constant* represents a constant

Description:

- Prints a constant value as a hexadecimal number on 8 hexadecimal digits. The hexadecimal number represents the integer equivalent to the 32 bit memory representation of the constant considered as a single precision number.

  If the constant value does not hold on a single precision number, it is first rounded to the nearest single precision number before it is displayed. A warning is displayed in this case.

Example 1:

```
> printsingle(3);
0x40400000
```

Example 2:

```
> prec=100!;
> verbosity = 1!;
> printsingle(exp(5));
Warning: the given expression is not a constant but an expression to evaluate. A
 faithful evaluation will be used.
Warning: rounding down occurred before printing a value as a simple.
0x431469c5
```

See also: **printdouble** (8.129), **single** (8.162)

8.132 printxml

Name: **printxml**
prints an expression as an MathML-Content-Tree
Usage:

$$\textbf{printxml}(expr) : \mathsf{function} \rightarrow \mathsf{void}$$
$$\textbf{printxml}(expr) > filename : (\mathsf{function}, \mathsf{string}) \rightarrow \mathsf{void}$$
$$\textbf{printxml}(expr) >> filename : (\mathsf{function}, \mathsf{string}) \rightarrow \mathsf{void}$$

Parameters:

- *expr* represents a functional expression
- *filename* represents a character sequence indicating a file name

Description:

- **printxml**(*expr*) prints the functional expression *expr* as a tree of MathML Content Definition Markups. This XML tree can be re-read in external tools or by usage of the **readxml** command.

  If a second argument *filename* is given after a single >, the MathML tree is not output on the standard output of `Sollya` but if in the file *filename* that get newly created or overwritten. If a double >> is given, the output will be appended to the file *filename*.

Example 1:

```
> printxml(x + 2 + exp(sin(x)));

<?xml version="1.0" encoding="UTF-8"?>
<!-- generated by sollya: http://sollya.gforge.inria.fr/ -->
<!-- syntax: printxml(...);   example: printxml(x^2-2*x+5); -->
<?xml-stylesheet type="text/xsl" href="http://sollya.gforge.inria.fr/mathmlc2p-w
eb.xsl"?>
<?xml-stylesheet type="text/xsl" href="mathmlc2p-web.xsl"?>
<!-- This stylesheet allows direct web browsing of MathML-c XML files (http:// o
r file://) -->

<math xmlns="http://www.w3.org/1998/Math/MathML">
<semantics>
<annotation-xml encoding="MathML-Content">
<lambda>
<bvar><ci> x </ci></bvar>
<apply>
<apply>
<plus/>
<apply>
<plus/>
<ci> x </ci>
<cn type="integer" base="10"> 2 </cn>
</apply>
<apply>
<exp/>
<apply>
<sin/>
<ci> x </ci>
</apply>
</apply>
</apply>
</apply>
</lambda>
</annotation-xml>
<annotation encoding="sollya/text">(x + 1b1) + exp(sin(x))</annotation>
</semantics>
</math>
```

Example 2:

```
> printxml(x + 2 + exp(sin(x))) > "foo.xml";
```

Example 3:

```
> printxml(x + 2 + exp(sin(x))) >> "foo.xml";
```

See also: **readxml** (8.143), **print** (8.128), **write** (8.186)

8.133 proc

Name: **proc**
defines a `Sollya` procedure
Usage:

proc(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) { *procedure body* } : void → procedure

proc(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) { *procedure body* **return** *expression*; } : void → procedure

proc(*formal list parameter* = ...) { *procedure body* } : void → procedure

proc(*formal list parameter* = ...) { *procedure body* **return** *expression*; } : void → procedure

Parameters:

- *formal parameter1*, *formal parameter2* through *formal parameter n* represent identifiers used as formal parameters
- *formal list parameter* represents an identifier used as a formal parameter for the list of an arbitrary number of parameters
- *procedure body* represents the imperative statements in the body of the procedure
- *expression* represents the expression **proc** shall evaluate to

Description:

- The **proc** keyword allows for defining procedures in the `Sollya` language. These procedures are common `Sollya` objects that can be applied to actual parameters after definition. Upon such an application, the `Sollya` interpreter applies the actual parameters to the formal parameters *formal parameter1* through *formal parameter n* (resp. builds up the list of arguments and applies it to the list *formal list parameter*) and executes the *procedure body*. The procedure applied to actual parameters evaluates then to the expression *expression* in the **return** statement after the *procedure body* or to **void**, if no return statement is given (i.e. a **return void** statement is implicitly given).
- `Sollya` procedures defined by **proc** have no name. They can be bound to an identifier by assigning the procedure object a **proc** expression produces to an identifier. However, it is possible to use procedures without giving them any name. For instance, `Sollya` procedures, i.e. procedure objects, can be elements of lists. They can even be given as an argument to other internal `Sollya` procedures. See also **procedure** on this subject.
- Upon definition of a `Sollya` procedure using **proc**, no type check is performed. More precisely, the statements in *procedure body* are merely parsed but not interpreted upon procedure definition with **proc**. Type checks are performed once the procedure is applied to actual parameters or to **void**. At this time, if the procedure was defined using several different formal parameters *formal parameter 1* through *formal parameter n*, it is checked whether the number of actual parameters corresponds to the number of formal parameters. If the procedure was defined using the syntax for a procedure with an arbitrary number of parameters by giving a *formal list parameter*, the number of actual arguments is not checked but only a list ¡formal list parameter¿ of appropriate length is built up. Type checks are further performed upon execution of each statement in *procedure body* and upon evaluation of the expression *expression* to be returned.

  Procedures defined by **proc** containing a **quit** or **restart** command cannot be executed (i.e. applied). Upon application of a procedure, the `Sollya` interpreter checks beforehand for such a statement. If one is found, the application of the procedure to its arguments evaluates to **error**. A warning is displayed. Remark that in contrast to other type or semantic correctness checks, this check is really performed before interpreting any other statement in the body of the procedure.
- Through the **var** keyword it is possible to declare local variables and thus to have full support of recursive procedures. This means a procedure defined using **proc** may contain in its *procedure body* an application of itself to some actual parameters: it suffices to assign the procedure (object) to an identifier with an appropriate name.
- `Sollya` procedures defined using **proc** may return other procedures. Further *procedure body* may contain assignments of locally defined procedure objects to identifiers. See **var** for the particular behaviour of local and global variables.

- The expression *expression* returned by a procedure is evaluated with regard to `Sollya` commands, procedures and external procedures. Simplification may be performed. However, an application of a procedure defined by **proc** to actual parameters evaluates to the expression *expression* that may contain the free global variable or that may be composed.

Example 1:

```
> succ = proc(n) { return n + 1; };
> succ(5);
6
> 3 + succ(0);
4
> succ;
proc(n)
{
nop;
return (n) + (1);
}
```

Example 2:

```
> add = proc(m,n) { var res; res := m + n; return res; };
> add(5,6);
11
> add;
proc(m, n)
{
var res;
res := (m) + (n);
return res;
}
> verbosity = 1!;
> add(3);
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
> add(true,false);
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
Warning: the given expression or command could not be handled.
error
```

Example 3:

```
> succ = proc(n) { return n + 1; };
> succ(5);
6
> succ(x);
1 + x
```

Example 4:

```
> hey = proc() { print("Hello world."); };
> hey();
Hello world.
> print(hey());
Hello world.
void
> hey;
proc()
{
print("Hello world.");
return void;
}
```

Example 5:

```
> fac = proc(n) { var res; if (n == 0) then res := 1 else res := n * fac(n - 1);
 return res; };
> fac(5);
120
> fac(11);
39916800
> fac;
proc(n)
{
var res;
if (n) == (0) then
res := 1
else
res := (n) * (fac((n) - (1)));
return res;
}
```

Example 6:

```
> myprocs = [| proc(m,n) { return m + n; }, proc(m,n) { return m - n; } |];
> (myprocs[0])(5,6);
11
> (myprocs[1])(5,6);
-1
> succ = proc(n) { return n + 1; };
> pred = proc(n) { return n - 1; };
> applier = proc(p,n) { return p(n); };
> applier(succ,5);
6
> applier(pred,5);
4
```

Example 7:

```
> verbosity = 1!;
> myquit = proc(n) { print(n); quit; };
> myquit;
proc(n)
{
print(n);
quit;
return void;
}
> myquit(5);
Warning: a quit or restart command may not be part of a procedure body.
The procedure will not be executed.
Warning: an error occurred while executing a procedure.
Warning: the given expression or command could not be handled.
error
```

Example 8:

```
> printsucc = proc(n) { var succ; succ = proc(n) { return n + 1; }; print("Succe
ssor of",n,"is",succ(n)); };
> printsucc(5);
Successor of 5 is 6
```

Example 9:

```
> makeadd = proc(n) { var add; print("n =",n); add = proc(m,n) { return n + m; }
; return add; };
> makeadd(4);
n = 4
proc(m, n)
{
nop;
return (n) + (m);
}
> (makeadd(4))(5,6);
n = 4
11
```

Example 10:

```
> sumall = proc(L = ...) { var acc, i; acc = 0; for i in L do acc = acc + i; ret
urn acc; };
> sumall;
proc(L = ...)
{
var acc, i;
acc = 0;
for i in L do
acc = (acc) + (i);
return acc;
}
> sumall();
0
> sumall(2);
2
> sumall(2,5);
7
> sumall(2,5,7,9,16);
39
> sumall @ [|1,...,10|];
55
```

See also: **return** (8.148), **externalproc** (8.60), **void** (8.184), **quit** (8.137), **restart** (8.147), **var** (8.182), **@** (8.26), **bind** (8.19), **error** (8.52)

8.134 procedure

Name: **procedure**
defines and assigns a `Sollya` procedure
Usage:

procedure *identifier*(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) { *procedure body* } : void → void
procedure *identifier*(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) { *procedure body* **return** *expression*; } : void → void
procedure *identifier*(*formal list parameter* = ...) { *procedure body* } : void → void
procedure *identifier*(*formal list parameter* = ...) { *procedure body* **return** *expression*; } : void → void

Parameters:

- *identifier* represents the name of the procedure to be defined and assigned
- *formal parameter1*, *formal parameter2* through *formal parameter n* represent identifiers used as formal parameters
- *formal list parameter* represents an identifier used as a formal parameter for the list of an arbitrary number of parameters
- *procedure body* represents the imperative statements in the body of the procedure
- *expression* represents the expression **procedure** shall evaluate to

Description:

- The **procedure** keyword allows for defining and assigning procedures in the `Sollya` language. It is an abbreviation to a procedure definition using **proc** with the same formal parameters, procedure body and return-expression followed by an assignment of the procedure (object) to the identifier *identifier*. In particular, all rules concerning local variables declared using the **var** keyword apply for **procedure**.

Example 1:

```
> procedure succ(n) { return n + 1; };
> succ(5);
6
> 3 + succ(0);
4
> succ;
proc(n)
{
nop;
return (n) + (1);
}
```

Example 2:

```
> procedure myprint(L = ...) { var i; for i in L do i; };
> myprint("Lyon","Nancy","Beaverton","Coye-la-Foret","Amberg","Nizhny Novgorod",
"Cluj-Napoca");
Lyon
Nancy
Beaverton
Coye-la-Foret
Amberg
Nizhny Novgorod
Cluj-Napoca
```

See also: **proc** (8.133), **var** (8.182), **bind** (8.19)

8.135 QD

Name: **QD**
short form for **quad**
See also: **quad** (8.136)

8.136 quad

Names: **quad**, **QD**
rounding to the nearest IEEE 754 quad (binary128).
Description:

- **quad** is both a function and a constant.
- As a function, it rounds its argument to the nearest IEEE 754 quad precision (i.e. IEEE754-2008 binary128) number. Subnormal numbers are supported as well as standard numbers: it is the real rounding described in the standard.
- As a constant, it symbolizes the quad precision format. It is used in contexts when a precision format is necessary, e.g. in the commands **round** and **roundcoefficients**. It is not supported for **implementpoly**. See the corresponding help pages for examples.

Example 1:

```
> display=binary!;
> QD(0.1);
1.10011001100110011001100110011001100110011001100110011001100110011001100110011001100110
01100110011001100110011001100110011001101_2 * 2^(-4)
> QD(4.17);
1.00001010111000010100011110101110000101000111101011100001010001111010111000010100011110
00011110101110000101000111101011 1_2 * 2^(2)
> QD(1.011_2 * 2^(-16493));
1.1_2 * 2^(-16493)
```

See also: **halfprecision** (8.73), **single** (8.162), **double** (8.45), **doubleextended** (8.47), **doubledouble** (8.46), **tripledouble** (8.180), **roundcoefficients** (8.152), **implementpoly** (8.81), **fpminimax** (8.67), **round** (8.151), **printsingle** (8.131)

8.137 quit

Name: **quit**
quits `Sollya`
Usage:

quit : void → void

Description:

- The command **quit**, when executed, stops the execution of a `Sollya` script and leaves the `Sollya` interpreter unless the **quit** command is executed in a `Sollya` script read into a main `Sollya` script by **execute** or #include.

  Upon exiting the `Sollya` interpreter, all state is thrown away, all memory is deallocated, all bound libraries are unbound and the temporary files produced by **plot** and **externalplot** are deleted.

  If the **quit** command does not lead to a halt of the `Sollya` interpreter, a warning is displayed.

Example 1:

```
> quit;
```

See also: **restart** (8.147), **execute** (8.54), **plot** (8.118), **externalplot** (8.59), **return** (8.148)

8.138 range

Name: **range**
keyword representing a range type
Usage:

range : type type

Description:

- **range** represents the range type for declarations of external procedures by means of **externalproc**.

  Remark that in contrast to other indicators, type indicators like **range** cannot be handled outside the **externalproc** context. In particular, they cannot be assigned to variables.

See also: **externalproc** (8.60), **boolean** (8.20), **constant** (8.27), **function** (8.69), **integer** (8.85), **list of** (8.93), **string** (8.166)

8.139 rationalapprox

Name: **rationalapprox**
returns a fraction close to a given number.
Usage:

rationalapprox(x,n) : (constant, integer) → function

Parameters:

- x is a number to approximate.
- n is a integer (representing a format).

Description:

- **rationalapprox**(x,n) returns a constant function of the form a/b where a and b are integers. The value a/b is an approximation of x. The quality of this approximation is determined by the parameter n that indicates the number of correct bits that a/b should have.
- The command is not safe in the sense that it is not ensured that the error between a/b and x is less than 2^{-n}.
- The following algorithm is used: x is first rounded downwards and upwards to a format of n bits, thus obtaining an interval $[x_l, x_u]$. This interval is then developped into a continued fraction as far as the representation is the same for every elements of $[x_l, x_u]$. The corresponding fraction is returned.
- Since rational numbers are not a primitive object of `Sollya`, the fraction is returned as a constant function. This can be quite amazing, because `Sollya` immediately simplifies a constant function by evaluating it when the constant has to be displayed. To avoid this, you can use **print** (that displays the expression representing the constant and not the constant itself) or the commands **numerator** and **denominator**.

Example 1:

```
> pi10 = rationalapprox(Pi,10);
> pi50 = rationalapprox(Pi,50);
> pi100 = rationalapprox(Pi,100);
> print( pi10, ": ", simplify(floor(-log2(abs(pi10-Pi)/Pi))), "bits." );
22 / 7 :  11 bits.
> print( pi50, ": ", simplify(floor(-log2(abs(pi50-Pi)/Pi))), "bits." );
90982559 / 28960648 :  50 bits.
> print( pi100, ": ", simplify(floor(-log2(abs(pi100-Pi)/Pi))), "bits." );
4850225745369133 / 1543874804974140 :  101 bits.
```

Example 2:

```
> a=0.1;
> b=rationalapprox(a,4);
> numerator(b); denominator(b);
1
10
> print(simplify(floor(-log2(abs((b-a)/a)))), "bits.");
166 bits.
```

See also: **print** (8.128), **numerator** (8.111), **denominator** (8.36), **rationalmode** (8.140)

8.140 rationalmode

Name: **rationalmode**
global variable controlling if rational arithmetic is used or not.
Usage:

rationalmode = *activation value* : on|off → void
rationalmode = *activation value* ! : on|off → void
rationalmode : on|off

Parameters:

- *activation value* controls if rational arithmetic should be used or not

Description:

- **rationalmode** is a global variable. When its value is **off**, which is the default, `Sollya` will not use rational arithmetic to simplify expressions. All computations, including the evaluation of constant expressions given on the `Sollya` prompt, will be performed using floating-point and interval arithmetic. Constant expressions will be approximated by floating-point numbers, which are in most cases faithful roundings of the expressions, when shown at the prompt.
- When the value of the global variable **rationalmode** is **on**, `Sollya` will use rational arithmetic when simplifying expressions. Constant expressions, given at the `Sollya` prompt, will be simplified to rational numbers and displayed as such when they are in the set of the rational numbers. Otherwise, flaoting-point and interval arithmetic will be used to compute a floating-point approximation, which is in most cases a faithful rounding of the constant expression.

Example 1:

```
> rationalmode=off!;
> 19/17 + 3/94;
1.14956195244055068836045056320400500625782222778473
> rationalmode=on!;
> 19/17 + 3/94;
1837 / 1598
```

Example 2:

```
> rationalmode=off!;
> exp(19/17 + 3/94);
3.15680977395514136754709208944824276340328162814418
> rationalmode=on!;
> exp(19/17 + 3/94);
3.15680977395514136754709208944824276340328162814418
```

See also: **on** (8.113), **off** (8.112), **numerator** (8.111), **denominator** (8.36), **simplifysafe** (8.160), **rationalapprox** (8.139), **autosimplify** (8.15)

8.141 RD

Name: **RD**
constant representing rounding-downwards mode.
Description:

- **RD** is used in command **round** to specify that the value x must be rounded to the greatest floating-point number y such that $y \leq x$.

Example 1:

```
> display=binary!;
> round(Pi,20,RD);
1.1001001000011111101_2 * 2^(1)
```

See also: **RZ** (8.156), **RU** (8.155), **RN** (8.150), **round** (8.151), **floor** (8.66)

8.142 readfile

Name: **readfile**
reads the content of a file into a string variable
Usage:

readfile(*filename*) : string → string

Parameters:

- *filename* represents a character sequence indicating a file name

Description:

- **readfile** opens the file indicated by *filename*, reads it and puts its contents in a character sequence of type string that is returned.

  If the file indicated by *filename* cannot be opened for reading, a warning is displayed and **readfile** evaluates to an **error** variable of type error.

Example 1:

```
> print("Hello world") > "myfile.txt";
> t = readfile("myfile.txt");
> t;
Hello world
```

Example 2:

```
> verbosity=1!;
> readfile("afile.txt");
Warning: the file "afile.txt" could not be opened for reading.
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
```

See also: **parse** (8.115), **execute** (8.54), **write** (8.186), **print** (8.128), **bashexecute** (8.17), **error** (8.52)

8.143 readxml

Name: **readxml**
reads an expression written as a MathML-Content-Tree in a file
Usage:

readxml(*filename*) : string → function | error

Parameters:

- *filename* represents a character sequence indicating a file name

Description:

- **readxml**(*filename*) reads the first occurrence of a lambda application with one bounded variable on applications of the supported basic functions in file *filename* and returns it as a `Sollya` functional expression.

  If the file *filename* does not contain a valid MathML-Content tree, **readxml** tries to find an "annotation encoding" markup of type "sollya/text". If this annotation contains a character sequence that can be parsed by **parse**, **readxml** returns that expression. Otherwise **readxml** displays a warning and returns an **error** variable of type error.

Example 1:

```
> readxml("readxmlexample.xml");
2 + x + exp(sin(x))
```

See also: **printxml** (8.132), **readfile** (8.142), **parse** (8.115), **error** (8.52)

8.144 relative

Name: **relative**
indicates a relative error for **externalplot**, **fpminimax** or **supnorm**
Usage:

relative : absolute|relative

Description:

- The use of **relative** in the command **externalplot** indicates that during plotting in **externalplot** a relative error is to be considered.

  See **externalplot** for details.

- Used with **fpminimax**, **relative** indicates that **fpminimax** must try to minimize the relative error.

  See **fpminimax** for details.

- When given in argument to **supnorm**, **relative** indicates that a relative error is to be considered for supremum norm computation.

  See **supnorm** for details.

Example 1:

```
> bashexecute("gcc -fPIC -c externalplotexample.c");
> bashexecute("gcc -shared -o externalplotexample externalplotexample.o -lgmp -l
mpfr");
> externalplot("./externalplotexample",absolute,exp(x),[-1/2;1/2],12,perturb);
```

See also: **externalplot** (8.59), **fpminimax** (8.67), **absolute** (8.2), **bashexecute** (8.17), **supnorm** (8.170)

8.145 remez

Name: **remez**
computes the minimax of a function on an interval.

Library names:
`sollya_obj_t sollya_lib_remez(sollya_obj_t, sollya_obj_t, sollya_obj_t, ...)`
`sollya_obj_t sollya_lib_v_remez(sollya_obj_t, sollya_obj_t, sollya_obj_t, va_list)`

Usage:

remez(*f*, *n*, *range*, *w*, *quality*) : (function, integer, range, function, constant) → function
remez(*f*, *L*, *range*, *w*, *quality*) : (function, list, range, function, constant) → function

Parameters:

- *f* is the function to be approximated
- *n* is the degree of the polynomial that must approximate *f*
- *L* is a list of integers or a list of functions and indicates the basis for the approximation of *f*

- *range* is the interval where the function must be approximated
- w (optional) is a weight function. Default is 1.
- *quality* (optional) is a parameter that controls the quality of the returned polynomial p, with respect to the exact minimax $p^\star$. Default is 1e-5.

Description:

- **remez** computes an approximation of the function f with respect to the weight function w on the interval *range*. More precisely, it searches p such that $\|pw - f\|_\infty$ is (almost) minimal among all p of a certain form. The norm is the infinity norm, e.g. $\|g\|_\infty = \max\{|g(x)|, x \in \text{range}\}$.
- If $w = 1$ (the default case), it consists in searching the best polynomial approximation of f with respect to the absolute error. If $f = 1$ and w is of the form $1/g$, it consists in searching the best polynomial approximation of g with respect to the relative error.
- If n is given, p is searched among the polynomials with degree not greater than n. If L is given and is a list of integers, p is searched as a linear combination of monomials X^k where k belongs to L. In the case when L is a list of integers, it may contain ellipses but cannot be end-elliptic. If L is given and is a list of functions g_k, p is searched as a linear combination of the g_k. In that case L cannot contain ellipses. It is the user responsability to check that the g_k are linearly independent over the interval *range*. Moreover, the functions $w \cdot g_k$ must be at least twice differentiable over *range*. If these conditions are not fulfilled, the algorithm might fail or even silently return a result as if it successfully found the minimax, though the returned p is not optimal.
- The polynomial is obtained by a convergent iteration called Remez' algorithm (and an extension of this algorithm, due to Stiefel). The algorithm computes a sequence $p_1, \ldots, p_k, \ldots$ such that $e_k = \|p_k w - f\|_\infty$ converges towards the optimal value e. The algorithm is stopped when the relative error between e_k and e is less than *quality*.

Example 1:

```
> p = remez(exp(x),5,[0;1]);
> degree(p);
5
> dirtyinfnorm(p-exp(x),[0;1]);
1.12956984638214536849843017679626063762687503980789e-6
```

Example 2:

```
> p = remez(1,[|0,2,4,6,8|],[0,Pi/4],1/cos(x));
> canonical=on!;
> p;
0.99999999994393749280444571988532724907643631727379 + -0.49999999571557467737204931630836834563663039748162
8 * x^2 + 4.16666132335010905188253972212748718651775241856
1e-2 * x^4 + -1.38865291475286141707180658383176799662601690152622e-3 * x^6 + 2.43726791911116269422173866792791676168996590663655e-5 * x^8
```

Example 3:

```
> p1 = remez(exp(x),5,[0;1],default,1e-5);
> p2 = remez(exp(x),5,[0;1],default,1e-10);
> p3 = remez(exp(x),5,[0;1],default,1e-15);
> dirtyinfnorm(p1-exp(x),[0;1]);
1.12956984638214536849843017679626063762687503980789e-6
> dirtyinfnorm(p2-exp(x),[0;1]);
1.12956980227478687332174207517728389861926662553 95e-6
> dirtyinfnorm(p3-exp(x),[0;1]);
1.12956980227478687332174207517728389861926662553 95e-6
```

Example 4:

```
> L = [|exp(x), sin(x), cos(x)-1, sin(x^3)|];
> g = (2^x-1)/x;
> p1 = remez(g, L, [-1/16;1/16]);
> p2 = remez(g, 3, [-1/16;1/16]);
> dirtyinfnorm(p1 - g, [-1/16;1/16]);
9.884132380555484596830895969139556435537529967061 4e-8
> dirtyinfnorm(p2 - g, [-1/16;1/16]);
2.543378010597542970388883892867108943131865099183 6e-9
```

See also: **dirtyinfnorm** (8.41), **infnorm** (8.84), **fpminimax** (8.67), **guessdegree** (8.72), **taylorform** (8.175), **taylor** (8.174)

8.146 rename

Name: **rename**
rename the free variable.
Usage:

rename(*ident1*,*ident2*) : void

Parameters:

- *ident1* is the current name of the free variable.
- *ident2* is a fresh name.

Description:

- **rename** permits a change of the name of the free variable. `Sollya` can handle only one free variable at a time. The first time in a session that an unbound name is used in a context where it can be interpreted as a free variable, the name is used to represent the free variable of `Sollya`. In the following, this name can be changed using **rename**.
- Be careful: if *ident2* has been set before, its value will be lost. Use the command **isbound** to know if *ident2* is already used or not.
- If *ident1* is not the current name of the free variable, an error occurs.
- If **rename** is used at a time when the name of the free variable has not been defined, *ident1* is just ignored and the name of the free variable is set to *ident2*.
- It is always possible to use the special keyword `_x_` to denote the free variable. Hence *ident1* can be `_x_`.

Example 1:

```
> f=sin(x);
> f;
sin(x)
> rename(x,y);
> f;
sin(y)
```

Example 2:

```
> a=1;
> f=sin(x);
> rename(x,a);
> a;
a
> f;
sin(a)
```

Example 3:

```
> verbosity=1!;
> f=sin(x);
> rename(y, z);
Warning: the current free variable is named "x" and not "y". Can only rename the
 free variable.
The last command will have no effect.
> rename(_x_, z);
Information: the free variable has been renamed from "x" to "z".
```

Example 4:

```
> verbosity=1!;
> rename(x,y);
Information: the free variable has been named "y".
> isbound(x);
false
> isbound(y);
true
```

See also: **isbound** (8.87)

8.147 restart

Name: **restart**
brings `Sollya` back to its initial state
Usage:

restart : void → void

Description:

- The command **restart** brings `Sollya` back to its initial state. All current state is abandoned, all libraries unbound and all memory freed.

  The **restart** command has no effect when executed inside a `Sollya` script read into a main `Sollya` script using **execute**. It is executed in a `Sollya` script included by a #include macro.

  Using the **restart** command in nested elements of imperative programming like for or while loops is possible. Since in most cases abandoning the current state of `Sollya` means altering a loop invariant, warnings for the impossibility of continuing a loop may follow unless the state is rebuilt.

Example 1:

```
> print(exp(x));
exp(x)
> a = 3;
> restart;
The tool has been restarted.
> print(x);
x
> a;
Warning: the identifier "a" is neither assigned to, nor bound to a library funct
ion nor external procedure, nor equal to the current free variable.
Will interpret "a" as "x".
x
```

Example 2:

```
> print(exp(x));
exp(x)
> for i from 1 to 10 do {
      print(i);
      if (i == 5) then restart;
  };
1
2
3
4
5
The tool has been restarted.
Warning: the tool has been restarted inside a for loop.
The for loop will no longer be executed.
```

Example 3:

```
> print(exp(x));
exp(x)
> a = 3;
> for i from 1 to 10 do {
      print(i);
      if (i == 5) then {
          restart;
          i = 7;
      };
  };
1
2
3
4
5
The tool has been restarted.
8
9
10
> print(x);
x
> a;
Warning: the identifier "a" is neither assigned to, nor bound to a library funct
ion nor external procedure, nor equal to the current free variable.
Will interpret "a" as "x".
x
```

See also: **quit** (8.137), **execute** (8.54)

8.148 return

Name: **return**
indicates an expression to be returned in a procedure
Usage:

return *expression* : void

Parameters:

- *expression* represents the expression to be returned

Description:

- The keyword **return** allows for returning the (evaluated) expression *expression* at the end of a begin-end-block (-block) used as a `Sollya` procedure body. See **proc** for further details concerning `Sollya` procedure definitions.

  Statements for returning expressions using **return** are only possible at the end of a begin-end-block used as a `Sollya` procedure body. Only one **return** statement can be given per begin-end-block.

- If at the end of a procedure definition using **proc** no **return** statement is given, a **return void** statement is implicitely added. Procedures, i.e. procedure objects, when printed out in `Sollya` defined with an implicit **return void** statement are displayed with this statement explicitly given.

Example 1:

```
> succ = proc(n) { var res; res := n + 1; return res; };
> succ(5);
6
> succ;
proc(n)
{
var res;
res := (n) + (1);
return res;
}
```

Example 2:

```
> hey = proc(s) { print("Hello",s); };
> hey("world");
Hello world
> hey;
proc(s)
{
print("Hello", s);
return void;
}
```

See also: **proc** (8.133), **void** (8.184)

8.149 revert

Name: **revert**
reverts a list.
Usage:

$$\textbf{revert}(L) : \mathsf{list} \to \mathsf{list}$$

Parameters:

- L is a list.

Description:

- **revert**(L) returns the same list, but with its elements in reverse order.
- If L is an end-elliptic list, **revert** will fail with an error.

Example 1:

```
> revert([| |]);
[| |]
```

Example 2:

```
> revert([|2,3,5,2,1,4|]);
[|4, 1, 2, 5, 3, 2|]
```

8.150 RN

Name: **RN**
constant representing rounding-to-nearest mode.
Description:

- **RN** is used in command **round** to specify that the value must be rounded to the nearest representable floating-point number.

Example 1:

```
> display=binary!;
> round(Pi,20,RN);
1.10010010000111111_2 * 2^(1)
```

See also: **RD** (8.141), **RU** (8.155), **RZ** (8.156), **round** (8.151), **nearestint** (8.106)

8.151 round

Name: **round**
rounds a number to a floating-point format.
Usage:

round(*x*,*n*,*mode*) : (constant, integer, RN|RZ|RU|RD) → constant
round(*x*,*format*,*mode*) : (constant,
HP|halfprecision|SG|single|D|double|DE|doubleextended|DD|doubledouble|QD|quad|TD|tripledouble,
RN|RZ|RU|RD) → constant

Parameters:

- *x* is a constant to be rounded.
- *n* is the precision of the target format.
- *format* is the name of a supported floating-point format.
- *mode* is the desired rounding mode.

Description:

- If used with an integer parameter *n*, **round**(*x*,*n*,*mode*) rounds *x* to a floating-point number with precision *n*, according to rounding-mode *mode*.
- If used with a format parameter *format*, **round**(*x*,*format*,*mode*) rounds *x* to a floating-point number in the floating-point format *format*, according to rounding-mode *mode*.
- Subnormal numbers are handled for the case when *format* is one of **halfprecision**, **single**, **double**, **doubleextended**, **doubledouble**, **quad** or **tripledouble**. Otherwise, when *format* is an integer, **round** does not take any exponent range into consideration, i.e. typically uses the full exponent range of the underlying MPFR library.

Example 1:

```
> display=binary!;
> round(Pi,20,RN);
1.10010010000111111_2 * 2^(1)
```

Example 2:

```
> printdouble(round(exp(17),53,RU));
0x417709348c0ea4f9
> printdouble(D(exp(17)));
0x417709348c0ea4f9
```

Example 3:

```
> display=binary!;
> a=2^(-1100);
> round(a,53,RN);
1_2 * 2^(-1100)
> round(a,D,RN);
0
> double(a);
0
```

See also: **RN** (8.150), **RD** (8.141), **RU** (8.155), **RZ** (8.156), **halfprecision** (8.73), **single** (8.162), **double** (8.45), **doubleextended** (8.47), **doubledouble** (8.46), **quad** (8.136), **tripledouble** (8.180), **roundcoefficients** (8.152), **roundcorrectly** (8.153), **printdouble** (8.129), **printsingle** (8.131), **ceil** (8.22), **floor** (8.66), **nearestint** (8.106)

8.152 roundcoefficients

Name: **roundcoefficients**
rounds the coefficients of a polynomial to classical formats.
Usage:

roundcoefficients(p,L) : (function, list) → function

Parameters:

- p is a function. Usually a polynomial.
- L is a list of formats.

Description:

- If p is a polynomial and L a list of floating-point formats, **roundcoefficients**(p,L) rounds each coefficient of p to the corresponding format in L.
- If p is not a polynomial, **roundcoefficients** does not do anything.
- If L contains other elements than **HP**, **halfprecision**, **SG**, **single**, **D**, **double**, **DE**, **doubleextended**, **DD**, **doubledouble**, **QD**, **quad**, **TD** and **tripledouble**, an error occurs.
- The coefficients in p corresponding to X^i is rounded to the format L[i]. If L does not contain enough elements (e.g. if **length**(L) < **degree**(p)+1), a warning is displayed. However, the coefficients corresponding to an element of L are rounded. The trailing coefficients (that do not have a corresponding element in L) are kept with their own precision. If L contains too much elements, the trailing useless elements are ignored. In particular L may be end-elliptic in which case **roundcoefficients** has the natural behavior.

Example 1:

```
> p=exp(1) + x*(exp(2) + x*exp(3));
> display=binary!;
> roundcoefficients(p,[|DD,D,D|]);
1.01011011111100001010100010110001010001010111011010010101001101010101111110111
00101011000100000001001110l_2 * 2^(1) + x * (1.11011000111001100100101110001101
10011011101101011l_2 * 2^(2) + x * (1.01000001010111100101101111111011011111011100
010000011_2 * 2^(4)))
> roundcoefficients(p,[|DD,D...|]);
1.01011011111100001010100010110001010001010111011010010101001101010101111110111
00101011000100000001001110l_2 * 2^(1) + x * (1.11011000111001100100101110001101
10011011101101011l_2 * 2^(2) + x * (1.01000001010111100101101111111011011111011100
010000011_2 * 2^(4)))
```

Example 2:

```
> f=sin(exp(1)*x);
> display=binary!;
> f;
sin(x * (1.01011011111100001010100010110001010001010111011010010101001101010101
11111011100010101100010000000100111001111010011110011111000111011000101110011100001
01100000111101_2 * 2^(1)))
> roundcoefficients(f,[|D...|]);
sin(x * (1.01011011111100001010100010110001010001010111011010010101001101010101
11111011100010101100010000000100111001111010011110011111000111011000101110011100001
01100000111101_2 * 2^(1)))
```

Example 3:

```
> p=exp(1) + x*(exp(2) + x*exp(3));
> verbosity=1!;
> display=binary!;
> roundcoefficients(p,[|DD,D|]);
Warning: the number of the given formats does not correspond to the degree of th
e given polynomial.
Warning: the 0th coefficient of the given polynomial does not evaluate to a floa
ting-point constant without any rounding.
Will evaluate the coefficient in the current precision in floating-point before
rounding to the target format.
Warning: the 1th coefficient of the given polynomial does not evaluate to a floa
ting-point constant without any rounding.
Will evaluate the coefficient in the current precision in floating-point before
rounding to the target format.
Warning: rounding may have happened.
1.01011011111100001010100010110001010001010111011010010101001101010101111110111
00101011000100000000100111101_2 * 2^(1) + x * (1.11011000111001100100101110001101
10011011101101011 1_2 * 2^(2) + x * (1.01000001010111100101101111111011011111011100
01000001011111001011010100101111011111110001010011011101000100110000111010001110
01000001011000010110000011100101110010100 1_2 * 2^(4)))
```

See also: **halfprecision** (8.73), **single** (8.162), **double** (8.45), **doubleextended** (8.47), **doubledouble** (8.46), **quad** (8.136), **tripledouble** (8.180), **fpminimax** (8.67), **remez** (8.145), **implementpoly** (8.81), **subpoly** (8.167)

8.153 roundcorrectly

Name: **roundcorrectly**
rounds an approximation range correctly to some precision
Usage:

roundcorrectly(*range*) : range → constant

Parameters:

- *range* represents a range in which an exact value lies

Description:

- Let *range* be a range of values, determined by some approximation process, safely bounding an unknown value v. The command **roundcorrectly**(*range*) determines a precision such that for this precision, rounding to the nearest any value in *range* yields to the same result, i.e. to the correct rounding of v.

  If no such precision exists, a warning is displayed and **roundcorrectly** evaluates to NaN.

Example 1:

```
> printbinary(roundcorrectly([1.010001_2; 1.0101_2]));
1.01_2
> printbinary(roundcorrectly([1.00001_2; 1.001_2]));
1_2
```

Example 2:

```
> roundcorrectly([-1; 1]);
@NaN@
```

See also: **round** (8.151), **mantissa** (8.99), **exponent** (8.58), **precision** (8.126)

8.154 roundingwarnings

Name: **roundingwarnings**
global variable controlling whether or not a warning is displayed when roundings occur.
Usage:

roundingwarnings = *activation value* : on|off → void
roundingwarnings = *activation value* ! : on|off → void
roundingwarnings : on|off

Parameters:

- *activation value* controls if warnings should be shown or not

Description:

- **roundingwarnings** is a global variable. When its value is **on**, warnings are emitted in appropriate verbosity modes (see **verbosity**) when roundings occur. When its value is **off**, these warnings are suppressed.
- This mode depends on a verbosity of at least 1. See **verbosity** for more details.
- Default is **on** when the standard input is a terminal and **off** when `Sollya` input is read from a file.

Example 1:

```
> verbosity=1!;
> roundingwarnings = on;
Rounding warning mode has been activated.
> exp(0.1);
Warning: Rounding occurred when converting the constant "0.1" to floating-point
with 165 bits.
If safe computation is needed, try to increase the precision.
Warning: rounding has happened. The value displayed is a faithful rounding of th
e true result.
1.10517091807564762481170782649024666822454719473 75
> roundingwarnings = off;
Rounding warning mode has been deactivated.
> exp(0.1);
1.10517091807564762481170782649024666822454719473 75
```

See also: **on** (8.113), **off** (8.112), **verbosity** (8.183), **midpointmode** (8.102), **rationalmode** (8.140)

8.155 RU

Name: **RU**
constant representing rounding-upwards mode.
Description:

- **RU** is used in command **round** to specify that the value x must be rounded to the smallest floating-point number y such that $x \leq y$.

Example 1:

```
> display=binary!;
> round(Pi,20,RU);
1.10010010000111111_2 * 2^(1)
```

See also: **RZ** (8.156), **RD** (8.141), **RN** (8.150), **round** (8.151), **ceil** (8.22)

8.156 RZ

Name: **RZ**
constant representing rounding-to-zero mode.
Description:

- **RZ** is used in command **round** to specify that the value must be rounded to the closest floating-point number towards zero. It just consists in truncating the value to the desired format.

Example 1:

```
> display=binary!;
> round(Pi,20,RZ);
1.10010010000111111101_2 * 2^(1)
```

See also: **RD** (8.141), **RU** (8.155), **RN** (8.150), **round** (8.151), **floor** (8.66), **ceil** (8.22)

8.157 searchgal

Name: **searchgal**
searches for a preimage of a function such that the rounding the image yields an error smaller than a constant
Usage:

searchgal(*function*, *start*, *preimage precision*, *steps*, *format*, *error bound*) : (function, constant, integer, integer, HP|halfprecision|SG|single|D|double|DE|doubleextended|DD|doubledouble|QD|quad|TD|tripledouble, constant) → list
searchgal(*list of functions*, *start*, *preimage precision*, *steps*, *list of format*, *list of error bounds*) : (list, constant, integer, integer, list, list) → list

Parameters:

- *function* represents the function to be considered
- *start* represents a value around which the search is to be performed
- *preimage precision* represents the precision (discretization) for the eligible preimage values
- *steps* represents the binary logarithm ($\log_2$) of the number of search steps to be performed
- *format* represents the format the image of the function is to be rounded to
- *error bound* represents a upper bound on the relative rounding error when rounding the image
- *list of functions* represents the functions to be considered

- *list of formats* represents the respective formats the images of the functions are to be rounded to
- *list of error bounds* represents a upper bound on the relative rounding error when rounding the image

Description:

- The command **searchgal** searches for a preimage z of function *function* or a list of functions *list of functions* such that z is a floating-point number with *preimage precision* significant mantissa bits and the image y of the function, respectively each image y_i of the functions, rounds to format *format* respectively to the corresponding format in *list of format* with a relative rounding error less than *error bound* respectively the corresponding value in *list of error bounds*. During this search, at most 2^{steps} attempts are made. The search starts with a preimage value equal to *start*. This value is then increased and decreased by 1 ulp in precision *preimage precision* until a value is found or the step limit is reached.

  If the search finds an appropriate preimage z, **searchgal** evaluates to a list containing this value. Otherwise, **searchgal** evaluates to an empty list.

Example 1:

```
> searchgal(log(x),2,53,15,DD,1b-112);
[| |]
> searchgal(log(x),2,53,18,DD,1b-112);
[|2.000000000038497205423482228070497512817382812 5|]
```

Example 2:

```
> f = exp(x);
> s = searchgal(f,2,53,18,DD,1b-112);
> if (s != [||]) then {
    v = s[0];
    print("The rounding error is 2^(",evaluate(log2(abs(DD(f)/f - 1)),v),")");
  } else print("No value found");
The rounding error is 2^( -1.121068784388093801482069842583585423221138741778 32e
2 )
```

Example 3:

```
> searchgal([|sin(x),cos(x)|],1,53,15,[|D,D|],[|1b-62,1b-60|]);
[|1.000000000001594946397176499885972589254379272460 94|]
```

See also: **round** (8.151), **double** (8.45), **doubledouble** (8.46), **tripledouble** (8.180), **evaluate** (8.53), **worstcase** (8.185)

8.158 SG

Name: **SG**
short form for **single**
See also: **single** (8.162)

8.159 simplify

Name: **simplify**
simplifies an expression representing a function
Usage:

simplify(*function*) : function → function

Parameters:

- *function* represents the expression to be simplified

Description:

- The command **simplify** simplifies constant subexpressions of the expression given in argument representing the function *function*. Those constant subexpressions are evaluated using floating-point arithmetic with the global precision **prec**.

Example 1:

```
> print(simplify(sin(pi * x)));
sin(3.14159265358979323846264338327950288419716939937508 * x)
> print(simplify(erf(exp(3) + x * log(4))));
erf(2.008553692318766774092852965458171789698790783855 44e1 + x * 1.3862943611198
906188344642429163531361510002687205)
```

Example 2:

```
> prec = 20!;
> t = erf(0.5);
> s = simplify(erf(0.5));
> prec = 200!;
> t;
0.52049987781304653768274665389196452873645157575796370005880 58
> s;
0.52050018310546875
```

See also: **simplifysafe** (8.160), **autosimplify** (8.15), **prec** (8.125), **evaluate** (8.53), **horner** (8.78), **rationalmode** (8.140)

8.160 simplifysafe

Name: **simplifysafe**
simplifies an expression representing a function
Usage:

simplifysafe(*function*) : function → function

Parameters:

- *function* represents the expression to be simplified

Description:

- The command **simplifysafe** simplifies the expression given in argument representing the function *function*. The command **simplifysafe** does not endanger the safety of computations even in `Sollya`'s floating-point environment: the function returned is mathematically equal to the function *function*.

  Remark that the simplification provided by **simplifysafe** is not perfect: they may exist simpler equivalent expressions for expressions returned by **simplifysafe**.

Example 1:

```
> print(simplifysafe((6 + 2) + (5 + exp(0)) * x));
8 + 6 * x
```

Example 2:

```
> print(simplifysafe((log(x - x + 1) + asin(1))));
(pi) / 2
```

Example 3:

```
> print(simplifysafe((log(x - x + 1) + asin(1)) - (atan(1) * 2)));
(pi) / 2 - (pi) / 4 * 2
```

See also: **simplify** (8.159), **autosimplify** (8.15), **rationalmode** (8.140), **horner** (8.78)

8.161 sin

Name: **sin**
the sine function.
Description:

- **sin** is the usual sine function.
- It is defined for every real number x.

See also: **asin** (8.10), **cos** (8.28), **tan** (8.172)

8.162 single

Names: **single**, **SG**
rounding to the nearest IEEE 754 single (binary32).
Description:

- **single** is both a function and a constant.
- As a function, it rounds its argument to the nearest IEEE 754 single precision (i.e. IEEE754-2008 binary32) number. Subnormal numbers are supported as well as standard numbers: it is the real rounding described in the standard.
- As a constant, it symbolizes the single precision format. It is used in contexts when a precision format is necessary, e.g. in the commands **round** and **roundcoefficients**. In is not supported for **implementpoly**. See the corresponding help pages for examples.

Example 1:

```
> display=binary!;
> SG(0.1);
1.10011001100110011001101_2 * 2^(-4)
> SG(4.17);
1.00001010111000010100l_2 * 2^(2)
> SG(1.011_2 * 2^(-1073));
0
```

See also: **halfprecision** (8.73), **double** (8.45), **doubleextended** (8.47), **doubledouble** (8.46), **quad** (8.136), **tripledouble** (8.180), **roundcoefficients** (8.152), **implementpoly** (8.81), **round** (8.151), **printsingle** (8.131)

8.163 sinh

Name: **sinh**
the hyperbolic sine function.
Description:

- **sinh** is the usual hyperbolic sine function: $\sinh(x) = \frac{e^x - e^{-x}}{2}$.
- It is defined for every real number x.

See also: **asinh** (8.11), **cosh** (8.29), **tanh** (8.173)

8.164 sort

Name: **sort**
sorts a list of real numbers.
Usage:

$$\textbf{sort}(L) : \mathsf{list} \to \mathsf{list}$$

Parameters:

- L is a list.

Description:

- If L contains only constant values, **sort**(L) returns the same list, but sorted in increasing order.
- If L contains at least one element that is not a constant, the command fails with a type error.
- If L is an end-elliptic list, **sort** will fail with an error.

Example 1:

```
> sort([| |]);
[| |]
> sort([|2,3,5,2,1,4|]);
[|1, 2, 2, 3, 4, 5|]
```

8.165 sqrt

Name: **sqrt**
square root.
Description:

- **sqrt** is the square root, e.g. the inverse of the function square: $\sqrt{y}$ is the unique positive x such that $x^2 = y$.
- It is defined only for x in $[0; +\infty]$.

8.166 string

Name: **string**
keyword representing a string type
Usage:

$$\textbf{string} : \mathsf{type}\ \mathsf{type}$$

Description:

- **string** represents the string type for declarations of external procedures by means of **externalproc**.

  Remark that in contrast to other indicators, type indicators like **string** cannot be handled outside the **externalproc** context. In particular, they cannot be assigned to variables.

See also: **externalproc** (8.60), **boolean** (8.20), **constant** (8.27), **function** (8.69), **integer** (8.85), **list of** (8.93), **range** (8.138)

8.167 subpoly

Name: **subpoly**
restricts the monomial basis of a polynomial to a list of monomials
Usage:

subpoly(*polynomial*, *list*) : (function, list) → function

Parameters:

- *polynomial* represents the polynomial the coefficients are taken from
- *list* represents the list of monomials to be taken

Description:

- **subpoly** extracts the coefficients of a polynomial *polynomial* and builds up a new polynomial out of those coefficients associated to monomial degrees figuring in the list *list*.

  If *polynomial* represents a function that is not a polynomial, subpoly returns 0.

  If *list* is a list that is end-elliptic, let be j the last value explicitly specified in the list. All coefficients of the polynomial associated to monomials greater or equal to j are taken.

Example 1:

```
> p = taylor(exp(x),5,0);
> s = subpoly(p,[|1,3,5|]);
> print(p);
1 + x * (1 + x * (0.5 + x * (1 / 6 + x * (1 / 24 + x / 120))))
> print(s);
x * (1 + x^2 * (1 / 6 + x^2 / 120))
```

Example 2:

```
> p = remez(atan(x),10,[-1,1]);
> subpoly(p,[|1,3,5...|]);
x * (0.99986632946591986997581285958052433296267358727229 + x^2 * (-0.3303047855
04861260596093435534236137298206064685038 + x^2 * (0.18015929463652346799743775 1
17895903961773054107393 + x * (-1.21704858321866028906175835649390114311877360 2
60197e-14 + x * (-8.51563508337027029965053368037708589181209615 66741e-2 + x * (
1.39681284176342339364451388757935358048374878126733e-14 + x * (2.08451141754345
61643018447784809880955983412532269e-2 + x * (-5.68101310125794362656976224260 11
349460288598691964e-15))))))))
```

Example 3:

```
> subpoly(exp(x),[|1,2,3|]);
0
```

See also: **roundcoefficients** (8.152), **taylor** (8.174), **remez** (8.145), **fpminimax** (8.67), **implementpoly** (8.81)

8.168 substitute

Name: **substitute**
replace the occurrences of the free variable in an expression.
Usage:

substitute(*f*,*g*) : (function, function) → function
substitute(*f*,*t*) : (function, constant) → constant

Parameters:

- f is a function.
- g is a function.
- t is a real number.

Description:

- **substitute**(f, g) produces the function $(f \circ g) : x \mapsto f(g(x))$.
- **substitute**(f, t) is the constant $f(t)$. Note that the constant is represented by its expression until it has been evaluated (exactly the same way as if you type the expression f replacing instances of the free variable by t).
- If f is stored in a variable F, the effect of the commands **substitute**(F,g) or **substitute**(F,t) is absolutely equivalent to writing *F(g)* resp. *F(t)*.

Example 1:

```
> f=sin(x);
> g=cos(x);
> substitute(f,g);
sin(cos(x))
> f(g);
sin(cos(x))
```

Example 2:

```
> a=1;
> f=sin(x);
> substitute(f,a);
0.841470984807896506652502321630298999622563060798 37
> f(a);
0.841470984807896506652502321630298999622563060798 37
```

See also: **evaluate** (8.53), **composepolynomials** (8.25)

8.169 sup

Name: **sup**
gives the upper bound of an interval.
Usage:

$$\mathbf{sup}(I) : \mathsf{range} \to \mathsf{constant}$$
$$\mathbf{sup}(x) : \mathsf{constant} \to \mathsf{constant}$$

Parameters:

- I is an interval.
- x is a real number.

Description:

- Returns the upper bound of the interval *I*. Each bound of an interval has its own precision, so this command is exact, even if the current precision is too small to represent the bound.
- When called on a real number x, **sup** considers it as an interval formed of a single point: $[x, x]$. In other words, **sup** behaves like the identity.

Example 1:

```
> sup([1;3]);
3
> sup(5);
5
```

Example 2:

```
> display=binary!;
> I=[0; 0.111110000011111_2];
> sup(I);
1.11110000011111_2 * 2^(-1)
> prec=12!;
> sup(I);
1.11110000011111_2 * 2^(-1)
```

See also: **inf** (8.83), **mid** (8.101), **max** (8.100), **min** (8.103)

8.170 supnorm

Name: **supnorm**
computes an interval bounding the supremum norm of an approximation error (absolute or relative) between a given polynomial and a function.
Usage:

supnorm(*p*, *f*, *I*, *errorType*, *accuracy*) : (function, function, range, absolute|relative, constant) → range

Parameters:

- *p* is a polynomial.
- *f* is a function.
- *I* is an interval.
- *errorType* is the type of error to be considered: **absolute** or **relative** (see details below).
- *accuracy* is a constant that controls the relative tightness of the interval returned.

Description:

- **supnorm**(*p*, *f*, *I*, *errorType*, *accuracy*) tries to compute an interval bound $r = [\ell, u]$ for the supremum norm of the error function $\varepsilon_{\text{absolute}} = p - f$ (when *errorType* evaluates to **absolute**) or $\varepsilon_{\text{relative}} = p/f - 1$ (when *errorType* evaluates to **relative**), over the interval I, such that $\sup_{x \in I}\{|\varepsilon(x)|\} \subseteq r$ and $0 \leq \left|\frac{u}{\ell} - 1\right| \leq$ *accuracy*. If **supnorm** succeeds in computing a suitable interval r, which it returns, that interval is guaranteed to contain the supremum norm value and to satisfy the required quality. Otherwise, **supnorm** evaluates to **error**, displaying a corresponding error message. These failure cases are rare and basically happen only for functions which are too complicated.
- Roughly speaking, **supnorm** is based on **taylorform** to obtain a higher degree polynomial approximation for *f*. This process is coupled with an a posteriori validation of a potential supremum norm upper bound. The validation is based on showing a certain polynomial the problem gets reduced to does not vanish. In cases when this process alone does not succeed, for instance because **taylorform** is unable to compute a sufficiently good approximation to *f*, **supnorm** falls back to bisecting the working interval until safe supremum norm bounds can be computed with the required accuracy or until the width of the subintervals becomes less than **diam** times the original interval *I*, in which case **supnorm** fails.

- The algorithm used for **supnorm** is quite complex, which makes it impossible to explain it here in further detail. Please find a complete description in the following article:

  Sylvain Chevillard, John Harrison, Mioara Joldes, Christoph Lauter
  Efficient and accurate computation of upper bounds of approximation errors
  Journal of Theoretical Computer Science (TCS), 2010
  LIP Research Report number RR LIP2010-2
  http://prunel.ccsd.cnrs.fr/ensl-00445343/fr/

- In practical cases, **supnorm** should be able to automatically handle removable discontinuities that relative errors might have. This means that usually, if f vanishes at a point x_0 in the interval considered, the approximation polynomial p is designed such that it also vanishes at the same point with a multiplicity large enough. Hence, although f vanishes, $\varepsilon_{\text{relative}} = p/f - 1$ may be defined by continuous extension at such points x_0, called removable discontinuities (see Example 3).

Example 1:

```
> p = remez(exp(x), 5, [-1;1]);
> midpointmode=on!;
> supnorm(p, exp(x), [-1;1], absolute, 2^(-40));
0.452055210438~2/7~e-4
```

Example 2:

```
> prec=200!;
> midpointmode=on!;
> d = [1;2];
> f = exp(cos(x)^2 + 1);
> p = remez(1,15,d,1/f,1e-40);
> theta=1b-60;
> prec=default!;
> mode=relative;
> supnorm(p,f,d,mode,theta);
0.30893006200251428~5/6~e-13
```

Example 3:

```
> midpointmode=on!;
> mode=relative;
> theta=1b-135;
> d = [-1b-2;1b-2];
> f = expm1(x);
> p = x * (1 +  x * ( 2097145 * 2^(-22) + x * ( 349527 * 2^(-21) + x * (87609 *
2^(-21) + x * 4369 * 2^(-19)))));
> theta=1b-40;
> supnorm(p,f,d,mode,theta);
0.98349131972~2/3~e-7
```

See also: **dirtyinfnorm** (8.41), **infnorm** (8.84), **checkinfnorm** (8.23), **absolute** (8.2), **relative** (8.144), **taylorform** (8.175), **autodiff** (8.14), **numberroots** (8.110), **diam** (8.37)

8.171 tail

Name: **tail**
gives the tail of a list.
Usage:

tail(L) : list → list

Parameters:

- L is a list.

Description:

- **tail**(L) returns the list L without its first element.
- If L is empty, the command will fail with an error.
- **tail** can also be used with end-elliptic lists. In this case, the result of **tail** is also an end-elliptic list.

Example 1:

```
> tail([|1,2,3|]);
[|2, 3|]
> tail([|1,2...|]);
[|2...|]
```

See also: **head** (8.74)

8.172 tan

Name: **tan**
the tangent function.
Description:

- **tan** is the tangent function, defined by $\tan(x) = \sin(x)/\cos(x)$.
- It is defined for every real number x that is not of the form $n\pi + \pi/2$ where n is an integer.

See also: **atan** (8.12), **cos** (8.28), **sin** (8.161)

8.173 tanh

Name: **tanh**
the hyperbolic tangent function.
Description:

- **tanh** is the hyperbolic tangent function, defined by $\tanh(x) = \sinh(x)/\cosh(x)$.
- It is defined for every real number x.

See also: **atanh** (8.13), **cosh** (8.29), **sinh** (8.163)

8.174 taylor

Name: **taylor**
computes a Taylor expansion of a function in a point
Usage:

taylor(*function*, *degree*, *point*) : (function, integer, constant) → function

Parameters:

- *function* represents the function to be expanded
- *degree* represents the degree of the expansion to be delivered
- *point* represents the point in which the function is to be developped

Description:

- The command **taylor** returns an expression that is a Taylor expansion of function *function* in point *point* having the degree *degree*.

  Let f be the function *function*, t be the point *point* and n be the degree *degree*. Then, **taylor**(*function*,*degree*,*point*) evaluates to an expression mathematically equal to

  $$\sum_{i=0}^{n} \frac{f^{(i)}(t)}{i!} x^i.$$

  In other words, if $p(x)$ denotes the polynomial returned by **taylor**, $p(x-t)$ is the Taylor polynomial of degree n of f developped at point t.

  Remark that **taylor** evaluates to 0 if the degree *degree* is negative.

Example 1:

```
> print(taylor(exp(x),3,1));
exp(1) + x * (exp(1) + x * (0.5 * exp(1) + x * exp(1) / 6))
```

Example 2:

```
> print(taylor(asin(x),7,0));
x * (1 + x^2 * (1 / 6 + x^2 * (9 / 120 + x^2 * 225 / 5040)))
```

Example 3:

```
> print(taylor(erf(x),6,0));
x * (1 / sqrt((pi) / 4) + x^2 * ((sqrt((pi) / 4) * 4 / (pi) * (-2)) / 6 + x^2 *
(sqrt((pi) / 4) * 4 / (pi) * 12) / 120))
```

See also: **remez** (8.145), **fpminimax** (8.67), **taylorform** (8.175)

8.175 taylorform

Name: **taylorform**
computes a rigorous polynomial approximation (polynomial, interval error bound) for a function, based on Taylor expansions.
Usage:

taylorform(*f*, *n*, x_0, *I*, *errorType*) : (function, integer, constant, range, absolute|relative) → list
taylorform(*f*, *n*, x_0, *I*, *errorType*) : (function, integer, range, range, absolute|relative) → list
taylorform(*f*, *n*, x_0, *errorType*) : (function, integer, constant, absolute|relative) → list
taylorform(*f*, *n*, x_0, *errorType*) : (function, integer, range, absolute|relative) → list

Parameters:

- *f* is the function to be approximated.
- *n* is the degree of the polynomial that must approximate *f*.
- x_0 is the point (it can be a real number or an interval) where the Taylor exansion of the function is to be considered.
- *I* is the interval over which the function is to be approximated. If this parameter is omitted, the behavior is changed (see detailed description below).
- *errorType* (optional) is the type of error to be considered. See the detailed description below. Default is **absolute**.

Description:

- **taylorform** computes an approximation polynomial and an interval error bound for function f. More precisely, it returns a list $L = [p, \text{coeffErrors}, \Delta]$ where:
  - p is an approximation polynomial of degree n which is roughly speaking a numerical Taylor expansion of f at the point x_0.
  - coeffsErrors is a list of $n+1$ intervals. Each interval coeffsErrors$[i]$ contains an enclosure of all the errors accumulated when computing the i-th coefficient of p.
  - Δ is an interval that provides a bound for the approximation error between p and f. Its significance depends on the *errorType* considered.
- The polynomial p and the bound Δ are obtained using Taylor Models principles.
- Please note that x_0 can be an interval. In general, it is meant to be a small interval approximating a non representable value. If x_0 is given as a constant expression, it is first numerically evaluated (leading to a faithful rounding $\tilde{x_0}$ at precision **prec**), and it is then replaced by the (exactly representable) point-interval $[\tilde{x_0}, \tilde{x_0}]$. In particular, it is not the same to call **taylorform** with $x_0 =$ **pi** and with $x_0 =$ [**pi**], for instance. In general, if the point around which one desires to compute the polynomial is not exactly representable, one should preferably use a small interval for x_0.
- More formally, the mathematical property ensured by the algorithm may be stated as follows. For all ξ_0 in x_0, there exist (small) values $\varepsilon_i \in$ coeffsErrors$[i]$ such that:
  If *errorType* is **absolute**, $\forall x \in I, \exists \delta \in \Delta,\ f(x) - p(x - \xi_0) = \sum_{i=0}^{n} \varepsilon_i \, (x - \xi_0)^i + \delta$.
  If *errorType* is **relative**, $\forall x \in I, \exists \delta \in \Delta,\ f(x) - p(x - \xi_0) = \sum_{i=0}^{n} \varepsilon_i \, (x - \xi_0)^i + \delta \, (x - \xi_0)^{n+1}$.
- It is also possible to use a large interval for x_0, though it is not obvious to give an intuitive sense to the result of **taylorform** in that case. A particular case that might be interesting is when $x_0 = I$ in relative mode. In that case, denoting by p_i the coefficient of p of order i, the interval $p_i +$ coeffsError$[i]$ gives an enclosure of $f^{(i)}(I)/i!$. However, the command **autodiff** is more convenient for computing such enclosures.
- When the interval I is not given, the approximated Taylor polynomial is computed but no remainder is produced. In that case the returned list is $L = [p, \text{coeffErrors}]$.
- The relative case is especially useful when functions with removable singularities are considered. In such a case, this routine is able to compute a finite remainder bound, provided that the expansion point given is the problematic removable singularity point.
- The algorithm does not guarantee that by increasing the degree of the approximation, the remainder bound will become smaller. Moreover, it may even become larger due to the dependecy phenomenon present with interval arithmetic. In order to reduce this phenomenon, a possible solution is to split the definition domain I into several smaller intervals.
- The command **taylor** also computes a Taylor polynomial of a function. However it does not provide a bound on the remainder. Besides, **taylor** is a somehow symbolic command: each coefficient of the Taylor polynomial is computed exactly and returned as an expression tree exactly equal to theoretical value. It is henceforth much more inefficient than **taylorform** and **taylorform** should be prefered if only numercial (yet safe) computations are required. The same difference exists between commands **diff** and **autodiff**.

Example 1:

```
> TL=taylorform(sin(x)/x, 10, 0, [-1,1], relative);
> p=TL[0];
> Delta=TL[2];
> errors=TL[1];
> for epsi in errors do epsi;
[0;0]
[0;0]
[0;5.3455294201843912922810729343029637576303937602101e-51]
[0;0]
[-3.3409558876152445576756705839393523485189961001313e-52;3.340955887615244557 67
56705839393523485189961001313e-52]
[0;0]
[-1.04404871487976392427364705748104760891218628129103e-53;1.0440487148797639242
7364705748104760891218628129103e-53]
[0;0]
[-1.63132611699963113167757352731413688892529106451724e-55;1.6313261169996311316
7757352731413688892529106451724e-55]
[0;0]
[-1.91171029335894273243465647732125416670932546623114e-57;1.9117102933589427324
3465647732125416670932546623114e-57]
> p; Delta;
1 + x^2 * (-0.16666666666666666666666666666666666666666666666667 + x^2 * (8.3333
333333333333333333333333333333333333333333333e-3 + x^2 * (-1.9841269841269841269
8412698412698412698412698412698e-4 + x^2 * (2.7557319223985890652557319223985890
6525573192239859e-6 + x^2 * (-2.505210838544171877505210838544171877505210838544
19e-8)))))
[-1.6135797443886066084999806203254010793747502812764e-10;1.61357974438860660849
99806203254010793747502812764e-10]
```

Example 2:

```
> TL=taylorform(exp(x), 10, 0, [-1,1], absolute);
> p=TL[0];
> Delta=TL[2];
> p; Delta;
1 + x * (1 + x * (0.5 + x * (0.1666666666666666666666666666666666666666666666666
7 + x * (4.1666666666666666666666666666666666666666666666667e-2 + x * (8.3333333
333333333333333333333333333333333333333333e-3 + x * (1.3888888888888888888888888
888888888888888888888888889e-3 + x * (1.984126984126984126984126984126984126984126
98412698e-4 + x * (2.4801587301587301587301587301587301587301587301587e-5 + x *
(2.7557319223985890652557319223985890652557319223985 9e-6 + x * 2.755731922398589
0652557319223985890652557319223986e-7))))))))))
[-2.31142719641187619441242534182684745832539555102969e-8;2.73126607556424744202
06278018039434042553645532164e-8]
```

Example 3:

```
> TL1 = taylorform(exp(x), 10, log2(10), [-1,1], absolute);
> TL2 = taylorform(exp(x), 10, [log2(10)], [-1,1], absolute);
> TL1==TL2;
false
```

Example 4:

```
> TL1 = taylorform(exp(x), 3, 0, [0,1], relative);
> TL2 = taylorform(exp(x), 3, 0, relative);
> TL1[0]==TL2[0];
true
> TL1[1]==TL2[1];
true
> length(TL1);
3
> length(TL2);
2
```

Example 5:

```
> f = exp(cos(x)); x0 = 0;
> TL = taylorform(f, 3, x0);
> T1 = TL[0];
> T2 = taylor(f, 3, x0);
> print(coeff(T1, 2));
-1.359140914229522617680143735676331248878623546849 99
> print(coeff(T2, 2));
-(0.5 * exp(1))
```

See also: **diff** (8.39), **autodiff** (8.14), **taylor** (8.174), **remez** (8.145)

8.176 taylorrecursions

Name: **taylorrecursions**
controls the number of recursion steps when applying Taylor's rule.
Usage:

$$\textbf{taylorrecursions} = n : \mathsf{integer} \rightarrow \mathsf{void}$$
$$\textbf{taylorrecursions} = n\ ! : \mathsf{integer} \rightarrow \mathsf{void}$$
$$\textbf{taylorrecursions} : \mathsf{integer}$$

Parameters:

- n represents the number of recursions

Description:

- **taylorrecursions** is a global variable. Its value represents the number of steps of recursion that are used when applying Taylor's rule. This rule is applied by the interval evaluator present in the core of `Sollya` (and particularly visible in commands like **infnorm**).
- To improve the quality of an interval evaluation of a function f, in particular when there are problems of decorrelation), the evaluator of `Sollya` uses Taylor's rule: $f([a,b]) \subseteq f(m) + [a - m,\ b - m] \cdot f'([a,\ b])$ where $m = \frac{a+b}{2}$. This rule can be applied recursively. The number of step in this recursion process is controlled by **taylorrecursions**.
- Setting **taylorrecursions** to 0 makes `Sollya` use this rule only once; setting it to 1 makes `Sollya` use the rule twice, and so on. In particular: the rule is always applied at least once.

Example 1:

```
> f=exp(x);
> p=remez(f,3,[0;1]);
> taylorrecursions=0;
The number of recursions for Taylor evaluation has been set to 0.
> evaluate(f-p, [0;1]);
[-0.468393648163036275229635657547431698623576204877 39;0.4694778175466708649168246499708805444358300351 7779]
> taylorrecursions=1;
The number of recursions for Taylor evaluation has been set to 1.
> evaluate(f-p, [0;1]);
[-0.138131114953879100663379409126970153172186472088 04;0.139215284337513690350568401550418998984440302388 44]
```

See also: **hopitalrecursions** (8.77), **evaluate** (8.53), **infnorm** (8.84)

8.177 TD

Name: **TD**
short form for **tripledouble**
See also: **tripledouble** (8.180)

8.178 time

Name: **time**
procedure for timing `Sollya` code.
Usage:

$$\textbf{time}(\textit{code}) : \mathsf{code} \rightarrow \mathsf{constant}$$

Parameters:

- *code* is the code to be timed.

Description:

- **time** permits timing a `Sollya` instruction, resp. a begin-end block of `Sollya` instructions. The timing value, measured in seconds, is returned as a `Sollya` constant (and not merely displayed as for **timing**). This permits performing computations of the timing measurement value inside `Sollya`.
- The extended **nop** command permits executing a defined number of useless instructions. Taking the ratio of the time needed to execute a certain `Sollya` instruction and the time for executing a **nop** therefore gives a way to abstract from the speed of a particular machine when evaluating an algorithm's performance.

Example 1:

```
> t = time(p=remez(sin(x),10,[-1;1]));
> write(t,"s were spent computing p = ",p,"\n");
0.144180000000000000000277555756156289135105907917023s were spent computing p =
-3.342655029334517190851399512740712219469120005963 9e-17 + x * (0.9999999997362835995537201146471312100344298816769 3 + x * (7.880275187730278668449934379904773249556887381969 3e-16 + x * (-0.166666661386013037032912982196741385680498698107285 + x * (-5.373444491115911218628935513855750483969298722123 3e-15 + x * (8.333303718654853765100213303167507281000932787714 8e-3 + x * (1.337972213892188158841123410055098314293473 2308712 84e-14 + x * (-1.983448630182774164932681551541589244220042902390 26e-4 + x * (-1.378911645128667417053161644191618341759870973281 6e-14 + x * (2.687625949543030468425182202489621096340167226200 5e-6 + x * 5.028237835001021105812838412357880558617378286360 5e-15)))))))))
```

Example 2:

```
> write(time({ p=remez(sin(x),10,[-1;1]); write("The error is 2^(", log2(dirtyin
fnorm(p-sin(x),[-1;1])), ")\n"); }), " s were spent\n");
The error is 2^(log2(2.39602467695631727848641768186659313738474584992648e-11))
0.250836999999999999961912411361453223435091786086559 s were spent
```

Example 3:

```
> t = time(bashexecute("sleep 10"));
> write(~(t-10),"s of execution overhead.\n");
2.56400000000000030664359940146823646500706672668457e-3s of execution overhead.
```

Example 4:

```
> ratio := time(p=remez(sin(x),10,[-1;1]))/time(nop(10));
> write("This ratio = ", ratio, " should somehow be independent of the type of m
achine.\n");
This ratio = 3.848089346899275494701526419954839844767676756853683 should somehow
be independent of the type of machine.
```

See also: **timing** (8.179), **nop** (8.108)

8.179 timing

Name: **timing**
global variable controlling timing measures in `Sollya`.
Usage:

$$\textbf{timing} = \textit{activation value} : \mathsf{on}|\mathsf{off} \to \mathsf{void}$$
$$\textbf{timing} = \textit{activation value}\ ! : \mathsf{on}|\mathsf{off} \to \mathsf{void}$$
$$\textbf{timing} : \mathsf{on}|\mathsf{off}$$

Parameters:

- *activation value* controls if timing should be performed or not

Description:

- **timing** is a global variable. When its value is **on**, the time spent in each command is measured and displayed (for **verbosity** levels higher than 1).

Example 1:

```
> verbosity=1!;
> timing=on;
Timing has been activated.
> p=remez(sin(x),10,[-1;1]);
Information: Remez: computing the matrix spent 2 ms
Information: Remez: computing the quality of approximation spent 11 ms
Information: Remez: computing the matrix spent 1 ms
Information: Remez: computing the quality of approximation spent 7 ms
Information: Remez: computing the matrix spent 1 ms
Information: Remez: computing the quality of approximation spent 7 ms
Information: computing a minimax approximation spent 142 ms
Information: assignment spent 142 ms
Information: full execution of the last parse chunk spent 142 ms
```

See also: **on** (8.113), **off** (8.112), **time** (8.178)

8.180 tripledouble

Names: **tripledouble**, **TD**
represents a number as the sum of three IEEE doubles.
Description:

- **tripledouble** is both a function and a constant.
- As a function, it rounds its argument to the nearest number that can be written as the sum of three double precision numbers.
- The algorithm used to compute **tripledouble**(x) is the following: let x_h = **double**(x), let x_m = **double**($x - x_h$) and let x_l = **double**($x - x_h - x_m$). Return the number $x_h + x_m + x_l$. Note that if the current precision is not sufficient to represent exactly $x_h + x_m + x_l$, a rounding will occur and the result of **tripledouble**(x) will be useless.
- As a constant, it symbolizes the triple-double precision format. It is used in contexts when a precision format is necessary, e.g. in the commands **roundcoefficients** and **implementpoly**. See the corresponding help pages for examples.

Example 1:

```
> verbosity=1!;
> a = 1+ 2^(-55)+2^(-115);
> TD(a);
1.00000000000000002775557561562891353466491600711096
> prec=110!;
> TD(a);
Warning: double rounding occurred on invoking the triple-double rounding operato
r.
Try to increase the working precision.
1.00000000000000002775557561562891 3
```

See also: **halfprecision** (8.73), **single** (8.162), **double** (8.45), **doubleextended** (8.47), **doubledouble** (8.46), **quad** (8.136), **roundcoefficients** (8.152), **implementpoly** (8.81), **fpminimax** (8.67), **printexpansion** (8.130)

8.181 true

Name: **true**
the boolean value representing the truth.
Description:

- **true** is the usual boolean value.

Example 1:

```
> true && false;
false
> 2>1;
true
```

See also: **false** (8.61), && (8.6), || (8.114)

8.182 var

Name: **var**
declaration of a local variable in a scope
Usage:

var *identifier1*, *identifier2*,... , *identifiern* : void

Parameters:

- *identifier1*, *identifier2*,... , *identifiern* represent variable identifiers

Description:

- The keyword **var** allows for the declaration of local variables *identifier1* through *identifiern* in a begin-end-block ({}-block). Once declared as a local variable, an identifier will shadow identifiers declared in higher scopes and undeclared identifiers available at top-level.

  Variable declarations using **var** are only possible in the beginning of a begin-end-block. Several **var** statements can be given. Once another statement is given in a begin-end-block, no more **var** statements can be given.

  Variables declared by **var** statements are dereferenced as **error** until they are assigned a value.

Example 1:

```
> exp(x);
exp(x)
> a = 3;
> {var a, b; a=5; b=3; {var a; var b; b = true; a = 1; a; b;}; a; b; };
1
true
5
3
> a;
3
```

See also: **error** (8.52), **proc** (8.133)

8.183 verbosity

Name: **verbosity**
global variable controlling the amount of information displayed by commands.
Usage:

verbosity = n : integer → void
verbosity = n ! : integer → void
verbosity : integer

Parameters:

- n controls the amount of information to be displayed

Description:

- **verbosity** accepts any integer value. At level 0, commands do not display anything on standard output. Note that very critical information may however be displayed on standard error.
- Default level is 1. It displays important information such as warnings when roundings happen.
- For higher levels more information is displayed depending on the command.

Example 1:

```
> verbosity=0!;
> 1.2+"toto";
error
> verbosity=1!;
> 1.2+"toto";
Warning: Rounding occurred when converting the constant "1.2" to floating-point
with 165 bits.
If safe computation is needed, try to increase the precision.
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
error
> verbosity=2!;
> 1.2+"toto";
Warning: Rounding occurred when converting the constant "1.2" to floating-point
with 165 bits.
If safe computation is needed, try to increase the precision.
Warning: at least one of the given expressions or a subexpression is not correct
ly typed
or its evaluation has failed because of some error on a side-effect.
Information: the expression or a partial evaluation of it has been the following
:
(1.19999999999999999999999999999999999999999999999999) + ("toto")
error
```

See also: **roundingwarnings** (8.154)

8.184 void

Name: **void**
the functional result of a side-effect or empty argument resp. the corresponding type
Usage:

void : void | type type

Description:

- The variable **void** represents the functional result of a side-effect or an empty argument. It is used only in combination with the applications of procedures or identifiers bound through **externalproc** to external procedures.

  The **void** result produced by a procedure or an external procedure is not printed at the prompt. However, it is possible to print it out in a print statement or in complex data types such as lists.

  The **void** argument is implicit when giving no argument to a procedure or an external procedure when applied. It can nevertheless be given explicitly. For example, suppose that foo is a procedure or an external procedure with a void argument. Then foo() and foo(void) are correct calls to foo. Here, a distinction must be made for procedures having an arbitrary number of arguments. In this case, an implicit **void** as the only parameter to a call of such a procedure gets converted into an empty list of arguments, an explicit **void** gets passed as-is in the formal list of parameters the procedure receives.

- **void** is used also as a type identifier for **externalproc**. Typically, an external procedure taking **void** as an argument or returning **void** is bound with a signature **void** $->$ some type or some type $->$ **void**. See **externalproc** for more details.

Example 1:

```
> print(void);
void
> void;
```

Example 2:

```
> hey = proc() { print("Hello world."); };
> hey;
proc()
{
print("Hello world.");
return void;
}
> hey();
Hello world.
> hey(void);
Hello world.
> print(hey());
Hello world.
void
```

Example 3:

```
> bashexecute("gcc -fPIC -Wall -c externalprocvoidexample.c");
> bashexecute("gcc -fPIC -shared -o externalprocvoidexample externalprocvoidexam
ple.o");
> externalproc(foo, "./externalprocvoidexample", void -> void);
> foo;
foo(void) -> void
> foo();
Hello from the external world.
> foo(void);
Hello from the external world.
> print(foo());
Hello from the external world.
void
```

Example 4:

```
> procedure blub(L = ...) { print("Argument list:", L); };
> blub(1);
Argument list: [|1|]
> blub();
Argument list: [| |]
> blub(void);
Argument list: [|void|]
```

See also: **error** (8.52), **proc** (8.133), **externalproc** (8.60)

8.185 worstcase

Name: **worstcase**
searches for hard-to-round cases of a function
Usage:

worstcase(*function*, *preimage precision*, *preimage exponent range*, *image precision*, *error bound*) :
(function, integer, range, integer, constant) → void

worstcase(*function*, *preimage precision*, *preimage exponent range*, *image precision*, *error bound*, *filename*) : (function, integer, range, integer, constant, string) → void

Parameters:

- *function* represents the function to be considered
- *preimage precision* represents the precision of the preimages
- *preimage exponent range* represents the exponents in the preimage format
- *image precision* represents the precision of the format the images are to be rounded to
- *error bound* represents the upper bound for the search w.r.t. the relative rounding error
- *filename* represents a character sequence containing a filename

Description:

- The **worstcase** command is deprecated. It searches for hard-to-round cases of a function. The command **searchgal** has a comparable functionality.

Example 1:

```
> worstcase(exp(x),24,[1,2],24,1b-26);
prec = 165
x = 1.9999998807907104492187 5        f(x) = 7.3890552520751953125        eps = 4
.5998601423446695596184695493764120138001954979037e-9 = 2^(-27.695763)
x = 2        f(x) = 7.38905620574951171875        eps = 1.4456360874967301812222
837939553341787812515058 7072e-8 = 2^(-26.043720)
```

See also: **round** (8.151), **searchgal** (8.157), **evaluate** (8.53)

8.186 write

Name: **write**
prints an expression without separators
Usage:

write(*expr1*,...,*exprn*) : (any type,..., any type) → void
write(*expr1*,...,*exprn*) > *filename* : (any type,..., any type, string) → void
write(*expr1*,...,*exprn*) >> *filename* : (any type,...,any type, string) → void

Parameters:

- *expr* represents an expression
- *filename* represents a character sequence indicating a file name

Description:

- **write**(*expr1*,...,*exprn*) prints the expressions *expr1* through *exprn*. The character sequences corresponding to the expressions are concatenated without any separator. No newline is displayed at the end. In contrast to **print**, **write** expects the user to give all separators and newlines explicitly.

  If a second argument *filename* is given after a single ">", the displaying is not output on the standard output of `Sollya` but if in the file *filename* that get newly created or overwritten. If a double ">>" is given, the output will be appended to the file *filename*.

  The global variables **display**, **midpointmode** and **fullparentheses** have some influence on the formatting of the output (see **display**, **midpointmode** and **fullparentheses**).

Remark that if one of the expressions *expri* given in argument is of type string, the character sequence *expri* evaluates to is displayed. However, if *expri* is of type list and this list contains a variable of type string, the expression for the list is displayed, i.e. all character sequences get displayed surrounded by quotes ("). Nevertheless, escape sequences used upon defining character sequences are interpreted immediately.

Example 1:

```
> write(x + 2 + exp(sin(x)));
> write("Hello\n");
x + 2 + exp(sin(x))Hello
> write("Hello","world\n");
Helloworld
> write("Hello","you", 4 + 3, "other persons.\n");
Helloyou7other persons.
```

Example 2:

```
> write("Hello","\n");
Hello
> write([|"Hello"|],"\n");
[|"Hello"|]
> s = "Hello";
> write(s,[|s|],"\n");
Hello[|"Hello"|]
> t = "Hello\tyou";
> write(t,[|t|],"\n");
Hello    you[|"Hello\tyou"|]
```

Example 3:

```
> write(x + 2 + exp(sin(x))) > "foo.sol";
> readfile("foo.sol");
x + 2 + exp(sin(x))
```

Example 4:

```
> write(x + 2 + exp(sin(x))) >> "foo.sol";
```

See also: **print** (8.128), **printexpansion** (8.130), **printdouble** (8.129), **printsingle** (8.131), **printxml** (8.132), **readfile** (8.142), **autosimplify** (8.15), **display** (8.43), **midpointmode** (8.102), **fullparentheses** (8.68), **evaluate** (8.53), **roundingwarnings** (8.154), **autosimplify** (8.15)

9 Appendix: interval arithmetic philosophy in `Sollya`

Although it is currently based on the MPFI library, `Sollya` has its own way of interpreting interval arithmetic when infinities or NaN occur, or when a function is evaluated on an interval containing points out of its domain, etc. This philosophy may differ from the one applied in MPFI. It is also possible that the behavior of `Sollya` does not correspond to the behavior that one would expect, e.g. as a natural consequence of the IEEE-754 standard.

The topology that we consider is always the usual topology of $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$. For any function, if one of its arguments is empty (respectively NaN), we return empty (respectively NaN).

9.1 Univariate functions

Let f be a univariate basic function and I an interval. We denote by J the result of the interval evaluation of f over I in `Sollya`. If I is completely included in the domain of f, J will usually be the smallest interval (at the current precision) containing the exact image $f(I)$. However, in some cases, it may happen that J is not as small as possible. It is guaranteed however, that J tends to $f(I)$ when the precision of the tool tends to infinity.

When f is not defined at some point x but is defined on a neighborhood of x, we consider that the "value" of f at x is the convex hull of the limit points of f around x. For instance, consider the evaluation of $f = \tan$ on $[0, \pi]$. It is not defined at $\pi/2$ (and only at this point). The limit points of f around $\pi/2$ are $-\infty$ and $+\infty$, so, we return $[-\infty, \infty]$. Another example: $f = \sin$ on $[+\infty]$. The function has no limit at this point, but all points of $[-1, 1]$ are limit points. So, we return $[-1, 1]$.

Finally, if I contains a subinterval on which f is not defined, we return [NaN, NaN] (example: $\sqrt{[-1, 2]}$).

9.2 Bivariate functions

Let f be a bivariate function and I_1 and I_2 be intervals. If $I_1 = [x]$ and $I_2 = [y]$ are both point-intervals, we return the convex hull of the limit points of f around (x, y) if it exists. In particular, if f is defined at (x, y) we return its value (or a small interval around it, if it is not exactly representable). As an example $[1]/[+\infty]$ returns $[0]$. Also, $[1]/[0]$ returns $[-\infty, +\infty]$ (note that `Sollya` does not consider signed zeros). If it is not possible to give a meaning to the expression $f(I_1, I_2)$, we return NaN: for instance $[0]/[0]$ or $[0] * [+\infty]$.

If one and only one of the intervals is a point-interval (say $I_1 = [x]$), we consider the partial function $g : y \mapsto f(x, y)$ and return the value that would be obtained when evaluating g on I_2. For instance, in order to evaluate $[0]/I_2$, we consider the function g defined for every $y \neq 0$ by $g(y) = 0/y = 0$. Hence, $g(I_2) = [0]$ (even if I_2 contains 0, by the argument of limit-points). In particular, please note that $[0]/[-1, 1]$ returns $[0]$ even though $[0]/[0]$ returns NaN. This rule even holds when g can only be defined as limit points: for instance, in the case $I_1/[0]$ we consider $g : x \mapsto x/0$. This function cannot be defined *stricto sensu*, but we can give it a meaning by considering 0 as a limit. Hence g is multivalued and its value is $\{-\infty, +\infty\}$ for every x. Hence, $I_1/[0]$ returns $[-\infty, +\infty]$ when I_1 is not a point-interval.

Finally, if neither I_1 nor I_2 are point-intervals, we try to give a meaning to $f(I_1, I_2)$ by an argument of limit-points when possible. For instance $[1, 2]/[0, 1]$ returns $[1, +\infty]$.

As a special exception to these rules, $[0]^{[0]}$ returns $[1]$.

10 Appendix: the Sollya library

10.1 Introduction

The header file of the Sollya library is sollya.h. Its inclusion may provoke the inclusion of other header files, such as gmp.h, mpfr.h or mpfi.h.

The library provides a virtual Sollya session that is perfectly similar to an interactive session: environment variables such as verbosity, prec, display, midpointmode, etc. are maintained and affect the behavior of the library, warning messages are displayed when something is not exact, etc. Please notice that the Sollya library currently is **not** re-entrant and can only be opened once. A process using the library must hence not be multi-threaded and is limited to one single virtual Sollya session.

In order to get started with the Sollya library, the first thing to do is hence to initialize this virtual session. Accordingly, one should close the session at the end of the program (which has the effect of releasing all the memory used by Sollya). This is hence a minimal file:

```
#include <sollya.h>

int main(void) {
  sollya_lib_init();

    /* Functions of the library can be called here */

  sollya_lib_close();
  return 0;
}
```

Suppose that this code is saved in a file called foo.c. The compilation is performed as usual without forgetting to link against libsollya (since the libraries libgmp, libmpfr and libmpfi are dependencies of Sollya, it might also be necessary to explicitly link against them):

```
~/% cc foo.c -c
~/% cc foo.o -o foo -lsollya -lmpfi -lmpfr -lgmp
```

10.2 Sollya object data-type

The library provides a single data type called sollya_obj_t that can contain any Sollya object (a Sollya object is anything that can be stored in a variable within the interactive tool. See Section 5 of the present documentation for details). Please notice that sollya_obj_t is in fact a pointer type; this has two consequences:

- NULL is a placeholder that can be used as a sollya_obj_t in some contexts. This placeholder is particularly useful as an end marker for functions with a variable number of arguments (see Sections 10.5.4 and 10.15).
- An affectation with the "=" sign does not copy an object but only copies the reference to it. To perform a (deep) copy, the sollya_lib_copy_obj() function is available.

Except for a few functions for which the contrary is explicitly specified, the following conventions are used:

- A function does not touch its arguments. Hence if sollya_lib_foo is a function of the library, a call to sollya_lib_foo(a) leaves the object referenced by a unchanged (the notable exceptions to that rule are the functions containing build in their name, e.g., sollya_lib_build_foo).
- A function that returns a sollya_obj_t creates a new object (this means that memory is dynamically allocated for that object). The memory allocated for that object should manually be cleared when the object is no longer used and all references to it (on the stack) get out of reach, e.g. on a function return: this is performed by the sollya_lib_clear_obj() function.

In general, except if the user perfectly knows what they are doing, the following rules should be applied (here `a` and `b` are C variables of type `sollya_obj_t`, and `sollya_lib_foo` and `sollya_lib_bar` are functions of the library):

- One should never write `a = b`. Instead, use `a = sollya_lib_copy_obj(b)`.
- One should never write `a = sollya_lib_foo(a)` because one loses the reference to the object initially referenced by the variable `a` (which is hence not cleared).
- One should never chain function calls such as, e.g., `a = sollya_lib_foo(sollya_lib_bar(b))` (the reference to the object created by `sollya_lib_bar(b)` would be lost and hence not cleared).
- A variable `a` should never be used twice at the left-hand side of the "=" sign (or as an lvalue in general) without performing `sollya_lib_clear_obj(a)` in-between.
- In an affectation of the form "`a = ...`", the right-hand side should always be a function call (i.e., something like `a = sollya_lib_foo(...);`).

Please notice that `sollya_lib_close()` clears the memory allocated by the virtual `Sollya` session but not the objects that have been created and stored in C variables. All the `sollya_obj_t` created by function calls should be cleared manually.

We can now write a simple Hello World program using the `Sollya` library:

```
#include <sollya.h>

int main(void) {
  sollya_obj_t s1, s2, s;
  sollya_lib_init();

  s1 = sollya_lib_string("Hello ");
  s2 = sollya_lib_string("World!");
  s = sollya_lib_concat(s1, s2);
  sollya_lib_clear_obj(s1);
  sollya_lib_clear_obj(s2);

  sollya_lib_printf("%b\n", s);
  sollya_lib_clear_obj(s);
  sollya_lib_close();
  return 0;
}
```

A universal function allows the user to execute any expression, as if it were given at the prompt of the `Sollya` tool, and to get a `sollya_obj_t` containing the result of the evaluation: this function is `sollya_lib_parse_string("some expression here")`. This is very convenient, and indeed, any script written in the `Sollya` language, could easily be converted into a C program by intensively using `sollya_lib_parse_string`. However, this should not be the preferred way if efficiency is targeted because (as its name suggests) this function uses a parser to decompose its argument, then constructs intermediate data structures to store the abstract interpretation of the expression, etc. Low-level functions are provided for efficiently creating `Sollya` objects; they are detailed in the next Section.

10.3 Conventions in use in the library

The library follows some conventions that it is useful to remember:

- When a function is a direct transposition of a command or function available in the interactive tool, it returns a `sollya_obj_t`. This is true, even when it would sound natural to return, e.g. an `int`. For instance `sollya_lib_get_verbosity()` returns a `sollya_obj_t`, which integer value must then be recovered with `sollya_lib_get_constant_as_int`. This forces the user to declare (and clear afterwards) a temporary `sollya_obj_t` to store the value, but this is the price of homogeneity in the library.

- When a function returns an integer, this integer generally is a boolean in the usual C meaning, i.e. 0 represents false and any non-zero value represents true. In many cases, the integer returned by the function indicates a status of success or failure: the convention is "false means failure" and "true means success". In case of failure, the convention is that the function did not touch any of its arguments.
- When a function would need to return several things, or when a function would need to return something together with a status of failure or success, the convention is that pointers are given as the first arguments of the function. These pointers shall point to valid addresses where the function will store the results. This can sometimes give obscure signatures, when the function morally returns a pointer and actually takes as argument a pointer to a pointer (this typically happens when the function allocates a segment of memory and should return a pointer to that segment of memory).

10.4 Displaying `Sollya` objects and numerical values

Within the interactive tool, the most simplest way of displaying the content of a variable or the value of an expression is to write the name of the variable or the expression, followed by the character ";". As a result, `Sollya` evaluates the expression or the variable and displays the result. Alternatively, a set of objects can be displayed the same way, separating the objects with commas. In library mode, the same behavior can be reproduced using the function `void sollya_lib_autoprint(sollya_obj_t, ...)`. Please notice that this function has a variable number of arguments: they are all displayed, until an argument equal to `NULL` is found. The `NULL` argument is mandatory, even if only one object shall be displayed (the function has no other way to know if other arguments follow or not). So, if only one argument should be displayed, the correct function call is `sollya_lib_autoprint(arg, NULL)`. Accordingly, if two arguments should be displayed, the function call is `sollya_lib_autoprint(arg1, arg2, NULL)`, etc. The function `void sollya_lib_v_autoprint(sollya_obj_t, va_list)` is the same, but it takes a `va_list` argument instead of a variable number of arguments.

Further, there is another way of displaying formatted strings containing `Sollya` objects, using a `printf`-like syntax. Four functions are provided, namely `sollya_lib_printf`, `sollya_lib_v_printf`, `sollya_lib_fprintf`, `sollya_lib_v_fprintf`. Each one of these functions overloads the usual function (respectively, `printf`, `vprintf`, `fprintf` and `vfprintf`). The full syntax of conversions specifiers supported with the usual functions is handled (please note that the style using '`$`' — as in `%3$` or `*3$` — is not handled though. It is not included in the C99 standard anyway). Additionally, the following conversion specifiers are provided:

- `%b`: corresponds to a `sollya_obj_t` argument.
- `%v`: corresponds to a `mpfr_t` argument. An optional precision modifier can be applied (e.g `%.5v`).
- `%w`: corresponds to a `mpfi_t` argument. An optional precision modifier can be applied (e.g `%.5w`).
- `%r`: corresponds to a `mpq_t` argument. There is no precision modifier support.

When one of the above conversion specifiers is used, the corresponding argument is displayed as it would be within the interactive tool: i.e. the way the argument is displayed depends on `Sollya` environment variables, such as `prec`, `display`, `midpointmode`, etc. When a precision modifier n is used, the argument is first rounded to a binary precision of roughly $\log_2(10) \times n$ bits (i.e. roughly equivalent to n decimal digits) before being displayed. As with traditional `printf`, the precision modifier can be replaced with `*` which causes the precision to be determined by an additional `int` argument. Notice that no width modifier is supported for the `%v` or `%w` conversion.

The `sollya_lib_printf` functions return an integer with the same meaning as the traditional `printf` functions. It indicates the number of characters that have been output. Similarly, the conversion specifier `%n` can be used, even together with the `Sollya` conversion specifiers `%b`, `%v`, `%w` and `%r`.

10.5 Creating `Sollya` objects

`Sollya` objects conceptually fall into one of five categories: numerical constants (e.g. 1 or 1.5), functional expressions (they might contain numerical constants, e.g., $\sin(\cos(x + 1.5))$), other simple objects

(intervals, strings, built-in constants such as `dyadic`, etc.), lists of objects (e.g., `[|1, "Hello"|]`) and structures (e.g., `{.a = 1; .b = "Hello"}`).

10.5.1 Numerical constants

Table 1 lists the different functions available to construct numerical constants. A `Sollya` constant is always created without rounding (whatever the value of global variable `prec` is at the moment of the function call): a sufficient precision is always allocated so that the constant is stored exactly. The objects returned by these functions are newly allocated and copies of the argument. For instance, after the instruction `a = sollya_lib_constant(b)`, the user will eventually have to clear `a` (with `sollya_lib_clear(a)`) and `b` (with `mpfr_clear(b)`).

The function `sollya_lib_constant_from_double` (or more conveniently its shortcut `SOLLYA_CONST`) is probably the preferred way for constructing numerical constants. As the name indicates it, its argument is a `double`; however, due to implicit casting in C, it is perfectly possible to give an `int` as argument: it will be converted into a `double` (without rounding if the integer fits on 53 bits) before being passed to `SOLLYA_CONST`. On the contrary, the user should be aware of the fact that if decimal non-integer constants are given, C rules of rounding (to double) are applied, regardless of the setting of the `Sollya` precision variable `prec`.

Table 1: Creating numerical constants (Creates a fresh `sollya_obj_t`. Conversion is always exact)

Type of the argument	Name of the function	Shortcut macro
`double`	`sollya_lib_constant_from_double(x)`	`SOLLYA_CONST(x)`
`uint64_t`	`sollya_lib_constant_from_uint64(x)`	`SOLLYA_CONST_UI64(x)`
`int64_t`	`sollya_lib_constant_from_int64(x)`	`SOLLYA_CONST_SI64(x)`
`int`	`sollya_lib_constant_from_int(x)`	N/A
`mpfr_t`	`sollya_lib_constant(x)`	N/A

10.5.2 Functional expressions

Functional expressions are built by composition of basic functions with constants and the free mathematical variable. Since, it is convenient to build such expressions by chaining function calls, the library provides functions that "eat up" their arguments (actually embedding them in a bigger expression). The convention is that functions that eat up their arguments are prefixed by `sollya_lib_build_`. For the purpose of building expressions, shortcut macros for the corresponding functions exist. They are all listed in Table 2.

It is worth mentioning that, although `SOLLYA_X_` and `SOLLYA_PI` are used without parentheses (as if they denoted constants), they are in fact function calls that create a new object each time they are used. The absence of parentheses is just more convenient for constructing expressions, such as, e.g. `SOLLYA_COS(SOLLYA_X_)`.

For each function of the form `sollya_lib_build_function_foo`, there exists a function called `sollya_lib_foo`. There are two differences between them:

- First, `sollya_lib_foo` does not "eat up" its argument. This can sometimes be useful, e.g., if one has an expression stored in a variable `a` and one wants to build the expression `exp(a)` without loosing the reference to the expression represented by `a`.
- Second, while `sollya_lib_build_function_foo` mechanically constructs an expression, function `sollya_lib_foo` also evaluates it, as far as this is possible without rounding.
  For instance, after the instructions `a = SOLLYA_CONST(0); b = sollya_lib_exp(a);` the variable `b` contains the number 1, whereas it would have contained the expression "`exp(0)`" if it had been created by `b = sollya_lib_build_function(a)`.

Actually, `sollya_lib_foo` has exactly the same behavior as writing an expression at the prompt within the interactive tool. In particular, it is possible to give a range as an argument to `sollya_lib_foo`: the

Table 2: Building functional expressions (Eats up arguments, embedding them in the returned object.)

Name in the interactive tool	Function to build it	Shortcut macro
_x_	sollya_lib_build_function_free_variable()	SOLLYA_X_
pi	sollya_lib_build_function_pi()	SOLLYA_PI
e1 + e2	sollya_lib_build_function_add(e1, e2)	SOLLYA_ADD(e1, e2)
e1 - e2	sollya_lib_build_function_sub(e1, e2)	SOLLYA_SUB(e1, e2)
e1 * e2	sollya_lib_build_function_mul(e1, e2)	SOLLYA_MUL(e1, e2)
e1 / e2	sollya_lib_build_function_div(e1, e2)	SOLLYA_DIV(e1, e2)
pow(e1, e2)	sollya_lib_build_function_pow(e1, e2)	SOLLYA_POW(e1, e2)
-e	sollya_lib_build_function_neg(e)	SOLLYA_NEG(e)
sqrt(e)	sollya_lib_build_function_sqrt(e)	SOLLYA_SQRT(e)
abs(e)	sollya_lib_build_function_abs(e)	SOLLYA_ABS(e)
erf(e)	sollya_lib_build_function_erf(e)	SOLLYA_ERF(e)
erfc(e)	sollya_lib_build_function_erfc(e)	SOLLYA_ERFC(e)
exp(e)	sollya_lib_build_function_exp(e)	SOLLYA_EXP(e)
expm1(e)	sollya_lib_build_function_expm1(e)	SOLLYA_EXPM1(e)
log(e)	sollya_lib_build_function_log(e)	SOLLYA_LOG(e)
log2(e)	sollya_lib_build_function_log2(e)	SOLLYA_LOG2(e)
log10(e)	sollya_lib_build_function_log10(e)	SOLLYA_LOG10(e)
log1p(e)	sollya_lib_build_function_log1p(e)	SOLLYA_LOG1P(e)
sin(e)	sollya_lib_build_function_sin(e)	SOLLYA_SIN(e)
cos(e)	sollya_lib_build_function_cos(e)	SOLLYA_COS(e)
tan(e)	sollya_lib_build_function_tan(e)	SOLLYA_TAN(e)
asin(e)	sollya_lib_build_function_asin(e)	SOLLYA_ASIN(e)
acos(e)	sollya_lib_build_function_acos(e)	SOLLYA_ACOS(e)
atan(e)	sollya_lib_build_function_atan(e)	SOLLYA_ATAN(e)
sinh(e)	sollya_lib_build_function_sinh(e)	SOLLYA_SINH(e)
cosh(e)	sollya_lib_build_function_cosh(e)	SOLLYA_COSH(e)
tanh(e)	sollya_lib_build_function_tanh(e)	SOLLYA_TANH(e)
asinh(e)	sollya_lib_build_function_asinh(e)	SOLLYA_ASINH(e)
acosh(e)	sollya_lib_build_function_acosh(e)	SOLLYA_ACOSH(e)
atanh(e)	sollya_lib_build_function_atanh(e)	SOLLYA_ATANH(e)
D(e), double(e)	sollya_lib_build_function_double(e)	SOLLYA_D(e)
SG(e), single(e)	sollya_lib_build_function_single(e)	SOLLYA_SG(e)
QD(e), quad(e)	sollya_lib_build_function_quad(e)	SOLLYA_QD(e)
HP(e), halfprecision(e)	sollya_lib_build_function_halfprecision(e)	SOLLYA_HP(e)
DD(e), doubledouble(e)	sollya_lib_build_function_double_double(e)	SOLLYA_DD(e)
TD(e), tripledouble(e)	sollya_lib_build_function_triple_double(e)	SOLLYA_TD(e)
DE(e), doubleextended(e)	sollya_lib_build_function_doubleextended(e)	SOLLYA_DE(e)
ceil(e)	sollya_lib_build_function_ceil(e)	SOLLYA_CEIL(e)
floor(e)	sollya_lib_build_function_floor(e)	SOLLYA_FLOOR(e)
nearestint(e)	sollya_lib_build_function_nearestint(e)	SOLLYA_NEARESTINT(e)

returned object will be the result of the evaluation of function `foo` on that range by interval arithmetic. In contrast, trying to use `sollya_lib_build_function_foo` on a range would result in a typing error.

10.5.3 Other simple objects

Other simple objects are created with functions listed in Table 3. The functions with a name of the form `sollya_lib_range_something` follow the same convention as `sollya_lib_constant`: they build a new object from a copy of their argument, and the conversion is always exact, whatever the value of `prec` is.

Please note that in the interactive tool, `D` either denotes the discrete mathematical function that maps a real number to its closest `double` number, or is used as a symbolic constant to indicate that the `double` format must be used (as an argument of `round` for instance). In the library, they are completely distinct objects, the mathematical function being obtained with `sollya_lib_build_function_double` and the symbolic constant with `sollya_lib_double_obj`. The same holds for other formats (`DD`, `SG`, etc.)

Table 3: Creating `Sollya` objects from scratch (Returns a new `sollya_obj_t`)

Name in the interactive tool	Function to create it
on	sollya_lib_on();
off	sollya_lib_off();
dyadic	sollya_lib_dyadic();
powers	sollya_lib_powers();
binary	sollya_lib_binary();
hexadecimal	sollya_lib_hexadecimal();
file	sollya_lib_file();
postscript	sollya_lib_postscript();
postscriptfile	sollya_lib_postscriptfile();
perturb	sollya_lib_perturb();
RD	sollya_lib_round_down();
RU	sollya_lib_round_up();
RZ	sollya_lib_round_towards_zero();
RN	sollya_lib_round_to_nearest();
honorcoeffprec	sollya_lib_honorcoeffprec();
true	sollya_lib_true();
false	sollya_lib_false();
void	sollya_lib_void();
default	sollya_lib_default();
decimal	sollya_lib_decimal();
absolute	sollya_lib_absolute();
relative	sollya_lib_relative();
fixed	sollya_lib_fixed();
floating	sollya_lib_floating();
error	sollya_lib_error();
D, double	sollya_lib_double_obj();
SG, single	sollya_lib_single_obj();
QD, quad	sollya_lib_quad_obj();
HP, halfprecision	sollya_lib_halfprecision_obj();
DE, doubleextended	sollya_lib_doubleextended_obj();
DD, doubledouble	sollya_lib_double_double_obj();
TD, tripledouble	sollya_lib_triple_double_obj();
"Hello"	sollya_lib_string("Hello")
[1, 3.5]	sollya_lib_range_from_interval(a);[a]
[1, 3.5]	sollya_lib_range_from_bounds(b, c);[b]

[a] `a` is a `mpfi_t` containing the interval $[1, 3.5]$. Conversion is always exact.
[b] `b` and `c` are `mpfr_t` respectively containing the numbers 1 and 3.5. Conversion is always exact.

10.5.4 Lists

There are actually two kinds of lists: regular lists (such as, e.g., `[|1, 2, 3|]`) and semi-infinite lists (such as, e.g. `[|1, 2, ...|]`). Withing the interactive tool, the ellipse "`...`" can sometimes be used as a shortcut to define regular lists, e.g. `[|1, 2, ..., 10|]`.

In the library, there is no symbol for the ellipse, and there are two distinct types: one for regular lists and one for semi-infinite lists (called end-elliptic). Defining a regular list with an ellipse is not possible in the library (except of course with `sollya_lib_parse_string`).

Constructing regular lists is achieved through three functions:

- `sollya_obj_t sollya_lib_list(sollya_obj_t[] L, int n)`: this function returns a new object that is a list the elements of which are copies of `L[0]`, ..., `L[n]`.
- `sollya_obj_t sollya_lib_build_list(sollya_obj_t obj1, ...)`: this function accepts a variable number of arguments. The last one **must** be `NULL`. It "eats up" its arguments and returns a list containing the objects given as arguments. Since arguments are eaten up, they may be directly produced by function calls, without being stored in variables. A typical use could be

  ```
  sollya_lib_build_list(SOLLYA_CONST(1), SOLLYA_CONST(2), SOLLYA_CONST(3), NULL);
  ```

- `sollya_obj_t sollya_lib_v_build_list(va_list)`: the same as the previous functions, but with a `va_list`.

Following the same conventions, end-elliptic lists can be constructed with the following functions:

- `sollya_obj_t sollya_lib_end_elliptic_list(sollya_obj_t[] L, int n)`.
- `sollya_obj_t sollya_lib_build_end_elliptic_list(sollya_obj_t obj1, ...)`.
- `sollya_obj_t sollya_lib_v_build_end_elliptic_list(va_list)`.

10.5.5 Structures

`Sollya` structures are also available in library mode as any other `Sollya` object. The support for `Sollya` structures is however minimal and creating them might seem cumbersome[2]. The only function available to create structures is

```
int sollya_lib_create_structure(sollya_obj_t *res, sollya_obj_t s, char *name,
                                sollya_obj_t val).
```

This function returns a boolean integer: false means failure, and true means success. Three cases of success are possible. In all cases, the function creates a new object and stores it at the address referred to by `res`.

- If `s` is `NULL`: `*res` is filled with a structure with only one field. This field is named after the string `name` and contains a copy of the object `val`.
- If `s` is an already existing structure that has a field named after the string `name`: `*res` is filled with a newly created structure. This structure is the same as `s` except that the field corresponding to `name` contains a copy of `val`.
- If `s` is an already existing structure that does **not** have a field named after the string `name`: `*res` is filled with a newly created structure. This structure is the same as `s` except that it has been augmented with a field corresponding to `name` and that contains a copy of `val`.

Please notice that `s` is not changed by this function: the structure stored in `*res` is a new one that does not refer to any of the components of `s`. As a consequence, one should not forget to explicitly clear `s` as well as `*res` when they become useless.

[2] Users are encouraged to make well-founded feature requests if they feel the need for better support of structures.

10.6 Getting the type of an object

Functions are provided that allow the user to test the type of a `Sollya` object. They are listed in Table 4. They all return an `int` interpreted as the boolean result of the test. Please note that from a typing point of view, a mathematical constant and a non-constant functional expression are both functions.

Table 4: Testing the type of a `Sollya` object (Returns non-zero if true, 0 otherwise)

`sollya_lib_obj_is_function(obj)`
`sollya_lib_obj_is_range(obj)`
`sollya_lib_obj_is_string(obj)`
`sollya_lib_obj_is_list(obj)`
`sollya_lib_obj_is_end_elliptic_list(obj)`
`sollya_lib_obj_is_structure(obj)`
`sollya_lib_obj_is_error(obj)`
`sollya_lib_is_on(obj)`
`sollya_lib_is_off(obj)`
`sollya_lib_is_dyadic(obj)`
`sollya_lib_is_powers(obj)`
`sollya_lib_is_binary(obj)`
`sollya_lib_is_hexadecimal(obj)`
`sollya_lib_is_file(obj)`
`sollya_lib_is_postscript(obj)`
`sollya_lib_is_postscriptfile(obj)`
`sollya_lib_is_perturb(obj)`
`sollya_lib_is_round_down(obj)`
`sollya_lib_is_round_up(obj)`
`sollya_lib_is_round_towards_zero(obj)`
`sollya_lib_is_round_to_nearest(obj)`
`sollya_lib_is_honorcoeffprec(obj)`
`sollya_lib_is_true(obj)`
`sollya_lib_is_false(obj)`
`sollya_lib_is_void(obj)`
`sollya_lib_is_default(obj)`
`sollya_lib_is_decimal(obj)`
`sollya_lib_is_absolute(obj)`
`sollya_lib_is_relative(obj)`
`sollya_lib_is_fixed(obj)`
`sollya_lib_is_floating(obj)`
`sollya_lib_is_double_obj(obj)`
`sollya_lib_is_single_obj(obj)`
`sollya_lib_is_quad_obj(obj)`
`sollya_lib_is_halfprecision_obj(obj)`
`sollya_lib_is_doubleextended_obj(obj)`
`sollya_lib_is_double_double_obj(obj)`
`sollya_lib_is_triple_double_obj(obj)`
`sollya_lib_is_pi(obj)`

10.7 Recovering the value of a range

If a `sollya_obj_t` is a range, it is possible to recover the values corresponding to the bounds of the range. The range can be recovered either as a `mpfi_t` or as two `mpfr_t` (one per bound). This is achieved with the following conversion functions:

- `int sollya_lib_get_interval_from_range(mpfi_t res, sollya_obj_t arg)`,
- `int sollya_lib_get_bounds_from_range(mpfr_t res_left, mpfr_t res_right, sollya_obj_t arg)`.

They return a boolean integer: false means failure (i.e., if the `sollya_obj_t` is not a range) and true means success. These functions follow the same conventions as those of the `MPFR` and `MPFI` libraries: the variables `res`, `res_left` and `res_right` must be initialized beforehand, and are used to store the result of the conversion. Also, the functions `sollya_lib_get_something_from_range` **do not change the internal precision** of `res`, `res_left` and `res_right`. If the internal precision is sufficient to perform the conversion without rounding, then it is guaranteed to be exact. If, on the contrary, the internal precision is not sufficient, the actual bounds of the range stored in `arg` will be rounded at the target precision using a rounding mode that ensures that the inclusion property remains valid, i.e. `arg` $\subseteq$ `res` (resp. `arg` $\subseteq$ [`res_left`, `res_right`]).

Function `int sollya_lib_get_prec_of_range(mp_prec_t *prec, sollya_obj_t arg)` stores at `*prec` a precision that is guaranteed to be sufficient to represent the range stored in `arg` without rounding. The returned value of this function is a boolean that follows the same convention as above. In conclusion, this is an example of a completely safe conversion:

```
...
mp_prec_t prec;
mpfr_t a, b;

if (!sollya_lib_get_prec_of_range(&prec, arg)) {
  sollya_lib_printf("Unexpected error: %b is not a range\n", arg);
}
else {
  mpfr_init2(a, prec);
  mpfr_init2(b, prec);
  sollya_lib_get_bounds_from_range(a, b, arg);

  /* Now [a, b] = arg exactly */
}
...
```

10.8 Recovering the value of a numerical constant or a constant expression

From a conceptual point of view, a numerical constant is nothing but a very simple constant functional expression. Hence there is no difference in `Sollya` between the way constants and constant expressions are handled. The functions presented in this Section allow one to recover the value of such constants or constant expressions into usual C data types.

A constant expression being given, three cases are possible:

- When naively evaluated at the current global precision, the expression always leads to provably exact computations (i.e., at each step of the evaluation, no rounding happens). For instance numerical constants or simple expressions such as $(\exp(0) + 5)/16$ fall in this category.
- The constant expressions would be exactly representable at some precision but this is not straightforward from a naive evaluation at the current global precision. An example would be $\sin(\pi/3)/\sqrt{3}$ or even $1 + 2^{-\mathrm{prec}-10}$.
- Finally, a third possibility is that the value of the expression is not exactly representable at any precision on a binary floating-point number. Possible examples are π or $1/10$.

From now on, we suppose that `arg` is a `sollya_obj_t` that contains a constant expression (or, as a particular case, a numerical constant). The general scheme followed by the conversion functions is the following: `Sollya` chooses an initial working precision greater that the target precision. If the value of `arg` is easily proved to be exactly representable at that precision, `Sollya` first computes this exact value

and then rounds it to the nearest number of the target format (tie-to-even). Otherwise, Sollya tries to adapt the working precision automatically in order to ensure that the result of the conversion is one of both floating-point numbers in the target format that are closest the exact value (a faithful rounding). A warning message indicates that the conversion is not exact and that a faithful rounding has been performed. In some cases really hard to evaluate, the algorithm can even fail to find a faithful rounding. In that case, too, a warning message is emitted indicating that the result of the conversion should not be trusted. Let us remark that these messages can be caught instead of being displayed and adapted handling can be provided by the user of the library at each emission of a warning (see Section 10.16).

The conversion functions are the following. They return a boolean integer: false means failure (i.e., arg is not a constant expression) and true means success.

- `int sollya_lib_get_constant_as_double(double *res, sollya_obj_t arg)`
- `int sollya_lib_get_constant_as_int(int *res, sollya_obj_t arg)`
- `int sollya_lib_get_constant_as_int64(int64_t *res, sollya_obj_t arg)`
- `int sollya_lib_get_constant_as_uint64(uint64_t *res, sollya_obj_t arg)`
- `int sollya_lib_get_constant(mpfr_t res, sollya_obj_t arg)`: the result of the conversion is stored in res. Please note that res must be initialized beforehand and that its internal precision is not modified by the algorithm.

Function `int sollya_lib_get_prec_of_constant(mp_prec_t *prec, sollya_obj_t arg)` tries to find a precision that would be sufficient to exactly represent the value of arg without rounding. If it manages to find such a precision, it stores it at *prec and returns true. If it does not manage to find such a precision, or if arg is not a constant expression, it returns false and *prec is left unchanged.

In conclusion, here is an example of use for converting a constant expression to a mpfr_t:

```
...
mp_prec_t prec;
mpfr_t a;
int test = 0;

test = sollya_lib_get_prec_of_constant(&prec, arg);
if (test) {
  mpfr_init2(a, prec);
  sollya_lib_get_constant(a, arg); /* Exact conversion */
}
else {
  mpfr_init2(a, 165); /* Initialization at some default precision */
  test = sollya_lib_get_constant(a, arg);
  if (!test) {
    sollya_lib_printf("Error: %b is not a constant expression\n", arg);
  }
}
...
```

10.9 Converting a string from Sollya to C

If arg is a sollya_obj_t that contains a string, that string can be recovered using

`int sollya_lib_get_string(char **res, sollya_obj_t arg).`

If arg really is a string, this function allocates enough memory on the heap to store the corresponding string, it copies the string at that newly allocated place, and sets *res so that it points to it. The function returns a boolean integer: false means failure (i.e., arg is not a string) and true means success.

Since this function allocates memory on the heap, this memory should manually be cleared by the user with sollya_lib_free once it becomes useless.

10.10 Converting a Sollya list to a C array

The function that allows user to recover a C array of sollya_obj_t objects from a Sollya list arg is:

```
int sollya_lib_get_list_elements(sollya_obj_t **L, int *n, int *end_ell,
                                 sollya_obj_t arg).
```

Three cases are possible:

- If arg is a regular list of length N, the function allocates memory on the heap for N sollya_obj_t, sets *L so that it points to that memory segment, and copies each of the elements N of arg to (*L)[0], ..., (*L)[N-1]. Finally, it sets *n to N, *end_ell to zero and returns true. A particular case is when arg is the empty list: everything is the same except that no memory is allocated and *L is left unchanged.
- If arg is an end-elliptic list containing N elements plus the ellipse. The function allocates memory on the heap for N sollya_obj_t, sets *L so that it points to that memory segment, and copies each of the elements N of arg at (*L)[0], ..., (*L)[N-1]. Finally, it sets *n to N, *end_ell to a non-zero value and returns true. The only difference between a regular list and an end-elliptic list containing the same elements is hence that *end_ell is set to a non-zero value in the latter.
- If arg is neither a regular nor an end-elliptic list, *L, *n and *end_ell are left unchanged and the function returns false.

10.11 Recovering the content of a Sollya structure

If arg is a sollya_obj_t that contains a structure, the content of a given field can be recovered using

```
int sollya_lib_get_element_in_structure(sollya_obj_t *res, char *name,
                                        sollya_obj_t arg).
```

If arg really is a structure and if that structure has a field named after the string name, this function copies the content of that field into the Sollya object *res. The function returns a boolean integer: false means failure (i.e., if arg is not a structure or if it does not have a field named after name) and true means success.

It is also possible to get all the field names and their content. This is achieved through the function

```
int sollya_lib_get_structure_elements(char ***names, sollya_obj_t **objs, int *n,
                                      sollya_obj_t arg).
```

If arg really is a structure, say with N fields called "fieldA", ..., "fieldZ", this functions sets *n to N, allocates and fills an array of N strings and sets *names so that it points to that segment of memory (hence (*names)[0] is the string "fieldA", ..., (*names)[N-1] is the string "fieldZ"). Moreover, it allocates memory for N sollya_obj_t, sets *objs so that it points on that memory segment, and copies the content of each of the N fields at (*objs)[0], ..., (*objs)[N-1]. Finally it returns true. If arg is not a structure, the function simply returns false without doing anything. Please note that since *names and *objs point to memory segments that have been dynamically allocated, they should manually be cleared by the user with sollya_lib_free once they become useless.

10.12 Decomposing a functional expression

If a sollya_obj_t contains a functional expression, one can decompose the expression tree using the following functions. These functions all return a boolean integer: true in case of success (i.e., if the sollya_obj_t argument really contains a functional expression) and false otherwise.

- int sollya_lib_get_function_arity(int *n, sollya_obj_t f): it stores the arity of the head function in f at the address referred to by n. Currently, the mathematical functions handled in Sollya are at most dyadic. Mathematical constants are considered as 0-adic functions.

Table 5: List of values defined in type `sollya_base_function_t`

SOLLYA_BASE_FUNC_COS	SOLLYA_BASE_FUNC_DOUBLE	SOLLYA_BASE_FUNC_LOG
SOLLYA_BASE_FUNC_ACOS	SOLLYA_BASE_FUNC_DOUBLEDOUBLE	SOLLYA_BASE_FUNC_LOG_2
SOLLYA_BASE_FUNC_ACOSH	SOLLYA_BASE_FUNC_DOUBLEEXTENDED	SOLLYA_BASE_FUNC_LOG_10
SOLLYA_BASE_FUNC_COSH	SOLLYA_BASE_FUNC_TRIPLEDOUBLE	SOLLYA_BASE_FUNC_LOG_1P
SOLLYA_BASE_FUNC_SIN	SOLLYA_BASE_FUNC_HALFPRECISION	SOLLYA_BASE_FUNC_EXP
SOLLYA_BASE_FUNC_ASIN	SOLLYA_BASE_FUNC_SINGLE	SOLLYA_BASE_FUNC_EXP_M1
SOLLYA_BASE_FUNC_ASINH	SOLLYA_BASE_FUNC_QUAD	SOLLYA_BASE_FUNC_NEG
SOLLYA_BASE_FUNC_SINH	SOLLYA_BASE_FUNC_FLOOR	SOLLYA_BASE_FUNC_SUB
SOLLYA_BASE_FUNC_TAN	SOLLYA_BASE_FUNC_CEIL	SOLLYA_BASE_FUNC_ADD
SOLLYA_BASE_FUNC_ATAN	SOLLYA_BASE_FUNC_NEARESTINT	SOLLYA_BASE_FUNC_MUL
SOLLYA_BASE_FUNC_ATANH	SOLLYA_BASE_FUNC_LIBRARYCONSTANT	SOLLYA_BASE_FUNC_DIV
SOLLYA_BASE_FUNC_TANH	SOLLYA_BASE_FUNC_LIBRARYFUNCTION	SOLLYA_BASE_FUNC_POW
SOLLYA_BASE_FUNC_ERF	SOLLYA_BASE_FUNC_PROCEDUREFUNCTION	SOLLYA_BASE_FUNC_SQRT
SOLLYA_BASE_FUNC_ERFC	SOLLYA_BASE_FUNC_FREE_VARIABLE	SOLLYA_BASE_FUNC_PI
SOLLYA_BASE_FUNC_ABS	SOLLYA_BASE_FUNC_CONSTANT	

- `int sollya_lib_get_head_function(sollya_base_function_t *type, sollya_obj_t f)`: it stores the type of `f` at the address referred to by `type`. The `sollya_base_function_t` is an enum type listing all possible cases (see Table 5).
- `int sollya_lib_get_subfunctions(sollya_obj_t f, int *n, ...)`: let us denote by `g_1`, ..., `g_k` the arguments following the argument `n`. They must be of type `sollya_obj_t *`. The function stores the arity of `f` at the address referred to by `n`. Suppose that `f` contains an expression of the form $f_0(f_1, \ldots, f_s)$. For each i from 1 to s, the expression corresponding to f_i is stored at the address referred to by `g_i`, unless one of the `g_i` is `NULL` in which case the function returns when encountering it. In practice, it means that the users should always put `NULL` as last argument, in order to prevent the case when they would not provide enough variables `g_i`. They can check afterwards that they provided enough variables by checking the value contained at the address referred to by `n`. If users do not put `NULL` as last argument and do not provide enough variables `g_i`, the algorithm will continue storing arguments at random places in the memory. Please note that the objects that have been stored in variables `g_i` must manually be cleared once they become useless.
- `int sollya_lib_v_get_subfunctions(sollya_obj_t f, int *n, va_list va)`: the same as the previous function, but with a `va_list` argument.
- `int sollya_lib_decompose_function(sollya_obj_t f, sollya_base_function_t *type, int *n, ...)`: this function is a all-in-one function equivalent to using `sollya_lib_get_head_function` and `sollya_lib_get_subfunctions` in only one function call.
- `int sollya_lib_v_decompose_function(sollya_obj_t f, sollya_base_function_t *type, int *n, va_list va)`: the same as the previous function, but with a `va_list`.

As an example of use of these functions, the following code returns 1 if `f` denotes a functional expression made only of constants (i.e., without the free variable), and returns 0 otherwise:

```
#include <sollya.h>

/* Note: we suppose that the library has already been initialized */
int is_made_of_constants(sollya_obj_t f) {
  sollya_obj_t tmp1 = NULL;
  sollya_obj_t tmp2 = NULL;
  int n, r, res;
  sollya_base_function_t type;

  r = sollya_lib_decompose_function(f, &type, &n, &tmp1, &tmp2, NULL);
  if (!r) { sollya_lib_printf("Not a mathematical function\n"); res = 0; }
  else if (n >= 3) {
    sollya_lib_printf("Unexpected error: %b has more than two arguments.\n", f);
    res = 0;
  }
  else {
    switch (type) {
      case SOLLYA_BASE_FUNC_FREE_VARIABLE: res = 0; break;
      case SOLLYA_BASE_FUNC_PI: res = 1; break;
      case SOLLYA_BASE_FUNC_CONSTANT: res = 1; break;
      default:
        res = is_made_of_constants(tmp1);
        if ((res) && tmp2) res = is_made_of_constants(tmp2);
    }
  }

  if (tmp1) sollya_lib_clear_obj(tmp1);
  if (tmp2) sollya_lib_clear_obj(tmp2);

  return res;
}
```

10.13 Faithfully evaluate a functional expression

Let us suppose that `f` is a functional expression and `a` is a numerical value or a constant expression. One of the very convenient features of the interactive tool is that the user can simply write `f(a)` at the prompt: the tool automatically adapts its internal precision in order to compute a value that is a faithful rounding (at the current tool precision) of the true value $f(a)$. Sometimes it does not achieve to find a faithful rounding, but in any case, if the result is not proved to be exact, a warning is displayed explaining how confident one should be with respect to the returned value. This feature is made available within the library with the two following functions:

- ```
  sollya_fp_result_t
    sollya_lib_evaluate_function_at_constant_expression(mpfr_t res, sollya_obj_t f,
                                                        sollya_obj_t a,
                                                        mpfr_t *cutoff),
  ```
- ```
  sollya_fp_result_t
    sollya_lib_evaluate_function_at_point(mpfr_t res, sollya_obj_t f,
                                          mpfr_t a, mpfr_t *cutoff).
  ```

In the former, the argument `a` is any `sollya_obj_t` containing a numerical constant or a constant expression, while in the latter `a` is a constant already stored in a `mpfr_t`. These functions store the result in `res` and return a `sollya_fp_result_t` which is an enum type described in Table 6. In order to understand the role of the `cutoff` parameter and the value returned by the function, it is necessary to describe the algorithm in a nutshell:

Input: a functional expression `f`, a constant expression `a`, a target precision q, a parameter ε.

1. Choose an initial working precision p.
2. Evaluate `a` with interval arithmetic, performing the computations at precision p.
3. Replace the occurrences of the free variable in `f` by the interval obtained at step 2. Evaluate the resulting expression with interval arithmetic, performing the computations at precision p. This yields an interval I.
4. If I contains at most one floating-point number of precision q:
   (a) Then set `res` to that number (or set it to any of the two enclosing numbers, if I does not contain any number of precision q) and return.
   (b) Else if all numbers in I are smaller than ε in absolute value, then set `res` to 0 and return .
   (c) Else if p has already been increased many times, then set `res` to the middle of I and return.
   (d) Else, increase p and go back to step 2.

The target precision q is chosen to be the precision of the `mpfr_t` variable `res`. The parameter ε corresponds to the parameter `cutoff`. The reason why `cutoff` is a pointer is that, most of the time, the user may not want to provide it, and using a pointer makes it possible to pass `NULL` instead. So, if `NULL` is given, ε is set to 0. If `cutoff` is not `NULL`, the absolute value of `*cutoff` is used as value for ε. Using a non-zero value for ε can be useful when one does not care about the precise value of $f(a)$ whenever its absolute value is below a given threshold. Typically, if one wants to compute the maximum of $|f(a_1)|$, ..., $|f(a_n)|$, it is not necessary to spend too much effort on the computation of $|f(a_i)|$ if one already knows that it is smaller than $\varepsilon = \max\{|f(a_1)|, \ldots, |f(a_{i-1})|\}$.

In the interactive tool, it is also possible to write `f(a)` when `a` contains an interval: `Sollya` performs the evaluation using an enhanced interval arithmetic, e.g., using L'Hopital's rule to produce finite (yet valid of course) enclosures even in cases when f exhibits removable singularities (for instance $\sin(x)/x$ over an interval containing 0). This feature is achieved in the library with the function

```
int sollya_lib_evaluate_function_over_interval(mpfi_t res, sollya_obj_t f, mpfi_t a).
```

This function returns a boolean integer: false means failure (i.e., `f` is not a functional expression), in which case `res` is left unchanged, and true means success, in which case `res` contains the result of the evaluation. The function might succeed, and yet `res` might contain something useless such as an unbounded interval or even $[\text{NaN}, \text{NaN}]$ (this happens for instance when `a` contains points that lie in the interior of the complement of the definition domain of `f`). It is the user's responsibility to check afterwards whether the computed interval is bounded, unbounded or NaN.

10.14 Name of the free variable

The default name for the free variable is the same in the library and in the interactive tool: it is `_x_`. In the interactive tool, this name is automatically changed at the first use of an undefined symbol. Accordingly in library mode, if an object is defined by `sollya_lib_parse_string` with an expression containing an undefined symbol, that symbol will become the free variable name if it has not already been changed before. But what if one does not use `sollya_lib_parse_string` (because it is not efficient) but one wants to change the name of the free variable? The name can be changed with `sollya_lib_name_free_variable("some_name")`.

It is possible to get the current name of the free variable with `sollya_lib_get_free_variable_name()`. This function returns a `char *` containing the current name of the free variable. Please note that this `char *` is dynamically allocated on the heap and should be cleared after its use with `sollya_lib_free()` (see below).

Table 6: List of values defined in type `sollya_fp_result_t`

Value	Meaning
SOLLYA_FP_OBJ_NO_FUNCTION	`f` is not a functional expression.
SOLLYA_FP_FAITHFUL_PROVEN_EXACT	The algorithm ended up at step (a) and I was equal to $[x, x]$ where x was a finite floating-point number at precision q.
SOLLYA_FP_FAITHFUL_PROVEN_INEXACT	The algorithm ended up at step (a) with a finite value, and I did not contain any floating-point number at precision q.
SOLLYA_FP_FAITHFUL	The algorithm ended up at step (a) with a finite value. This does not say anything about the exactness of the result, though it might have been proved exact or inexact if the algorithm had been run with a larger precision.
SOLLYA_FP_BELOW_CUTOFF	The algorithm ended up at step (b).
SOLLYA_FP_NOT_FAITHFUL_ZERO_CONTAINED_BELOW_THRESHOLD	The algorithm ended up at step (c) and I was an interval of the form $[-\delta_1, \delta_2]$ where the δ_i are positive and very small (below some threshold of the algorithm). This typically happens when $f(a)$ exactly equals zero, but the algorithm does not manage to prove this exact equality.
SOLLYA_FP_NOT_FAITHFUL_ZERO_CONTAINED_NOT_BELOW_THRESHOLD	The algorithm ended up at step (c) with an interval I containing 0 but too large to fall in the above case. In general, this should be considered as a case of failure and the value stored in `res` might be completely irrelevant.
SOLLYA_FP_NOT_FAITHFUL_ZERO_NOT_CONTAINED	The algorithm ended up at step (c) with an interval I that does not contain 0. In general, this should be considered as a case of failure and the value stored in `res` might be completely irrelevant.
SOLLYA_FP_NOT_FAITHFUL_INFINITY_CONTAINED	The algorithm ended up at step (c) and (at least) one of the bounds of I was infinite. This typically happens when the limit of $f(x)$ when x goes to a is infinite.
SOLLYA_FP_INFINITY	The algorithm ended up at step (a) with I of the form $[+\infty, +\infty]$ or $[-\infty, -\infty]$. Hence $f(a)$ is proved to be an exact infinity.
SOLLYA_FP_FAILURE	The algorithm ended up at step (c) and I contained NaN. This typically happens when a is not in the definition domain of f.
SOLLYA_FP_CUTOFF_IS_NAN	`cutoff` was not `NULL` and the value of `*cutoff` is NaN.
SOLLYA_FP_EXPRESSION_NOT_CONSTANT	`a` is not a constant expression.

10.15 Commands and functions

Besides some exceptions, every command and every function available in the `Sollya` interactive tool has its equivalent (with a very close syntax) in the library. Section 8 of the present documentation gives the library syntax as well as the interactive tool syntax of each commands and functions. The same information is available within the interactive tool by typing `help some_command`. So if one knows the name of a command or function in the interactive tool, it is easy to recover its library name and signature.

A particular point is worth mentioning: some functions of the tool such as `remez` for instance have a variable number of arguments. For instance, one might call `remez(exp(x), 4, [0,1])` or `remez(1, 4, [0,1], 1/exp(x))`. This feature is rendered in the C library by the use of variadic functions (functions with an arbitrary number of arguments), as they are permitted by the C standard. The notable difference is that there must **always be an explicit NULL argument** at the end of the function call. Hence one can write `sollya_lib_remez(a, b, c, NULL)` or `sollya_lib_remez(a, b, c, d, NULL)`. It is very easy to forget the `NULL` argument and use for instance `sollya_lib_remez(a, b, c)`. This is **completely wrong** because the memory will be read until a `NULL` pointer is found. In the best case, this will lead to an error or a result obviously wrong, but it could also lead to subtle, not-easy-to-debug errors. The user is advised to be particularly careful with respect to this point.

Each command or function accepting a variable number of arguments comes in a `sollya_lib_v_` version accepting a `va_list` parameter containing the list of optional arguments. For instance, one might write a function that takes as arguments a function f, an interval I, optionally a weight function w, optionally a quality parameter q. That function would display the minimax obtained when approximating f over I (possibly with weight w and quality q) by polynomials of degree $n = 2$ to 20. So, that function would get a variable number of arguments (i.e. a `va_list` in fact) and pass them straight to remez. In that case, one needs to use the `v_remez` version, as the following code shows:

```
#include <sollya.h>
#include <stdarg.h>

/* Note: we suppose that the library has already been initialized */
void my_function(sollya_obj_t f, sollya_obj_t I, ...) {
  sollya_obj_t n, res;
  int i;
  va_list va;

  for(i=2;i<=20;i++) {
    n = SOLLYA_CONST(i);
    va_start(va, I);
    res = sollya_lib_v_remez(f, n, I, va);
    sollya_lib_printf("Approximation of degree %b is %b\n", n, res);
    va_end(va);
    sollya_lib_clear_obj(n);
    sollya_lib_clear_obj(res);
  }

  return;
}
```

10.16 Warning messages in library mode

The philosophy of `Sollya` is "whenever something is not exact, explicitly warn about that". This is a nice feature since this ensures that the user always perfectly knows the degree of confidence they can have in a result (is it exact? or only faithful? or even purely numerical, without any warranty?) However, it is sometimes desirable to hide some (or all) of these messages. This is especially true in library mode where messages coming from `Sollya` are intermingled with the messages of the main program. The library

hence provides a specific mechanism to catch all messages emitted by the `Sollya` core and handle each of them specifically: installation of a callback for messages.

Before describing the principle of the message callback, it seems appropriate to recall that several mechanisms are available in the interactive tool to filter the messages emitted by `Sollya`. These mechanisms are also available in library mode for completeness. When a message is emitted, it has two characteristics: a verbosity level and an id (a number uniquely identifying the message). After it has been emitted, it passes through the following steps where it can be filtered. If it has not been filtered (and only in this case) it is displayed.

1. If the verbosity level of the message if greater than the value of the environment variable `verbosity`, it is filtered.

2. If the environment variable `roundingwarnings` is set to `off` and if the message informs the user that a rounding occurred, it is filtered.

3. If the id of the message has been registered with the `suppressmessage` command, the message is filtered.

4. If a message callback has been installed and if the message has not been previously filtered, it is handled by the callback, which decides to filter it or display it.

A message callback is a function of the form `int my_callback(sollya_msg_t msg)`. It receives as input an object representing the message, performs whatever treatment seems appropriate and returns an integer interpreted as a boolean. If the returned value is false, the message is not displayed. If, on the contrary, the returned value is true, the message is displayed as usual. By default, no callback is installed and all messages are displayed. To install a callback, use `sollya_lib_install_msg_callback(my_callback)`. Please remember that, if a message is filtered because of one of the three other mechanisms, it will never be transmitted to the callback. Hence, in library mode, if one wants to catch every single message through the callback, one should set the value of `verbosity` to `MAX_INT`, set `roundingwarnings` to `on` (this is the default anyway) and one should not use the `suppressmessage` mechanism.

It is possible to come back to the default behavior, using `sollya_lib_uninstall_msg_callback()`. In particular, if a callback is installed and one wants to install another one, one should always uninstall the first callback and, only then, install the new one.

Both `sollya_lib_install_msg_callback` and `sollya_lib_uninstall_msg_callback` return an integer interpreted as a boolean: false means failure and true means success.

It is possible to get the currently installed callback using `sollya_lib_get_msg_callback()`. This function returns a function pointer to the current callback — i.e., a `int (*)(sollya_msg_t)` — if a callback is installed, or `NULL` if no callback is currently installed.

Currently the type `sollya_msg_t` has only two accessors:

- `int sollya_lib_get_msg_id(sollya_msg_t msg)` returns an integer that identifies the type of the message. The message types are listed in the file `sollya-messages.h`. Please note that this file not only lists the possible identifiers but only defines meaningful names to each possible message number (e.g., `SOLLYA_MSG_UNDEFINED_ERROR` is an alias for the number 2 but is more meaningful to understand what the message is about). It is recommended to use these names instead of numerical values.

- `char *sollya_lib_msg_to_text(sollya_msg_t msg)` returns a generic string briefly summarizing the content of the message. Please note that this `char *` is dynamically allocated on the heap and should manually be cleared with `sollya_lib_free` when it becomes useless.

In the future, other accessors could be added (to get the verbosity level at which the message has been emitted, to get data associated with the message, etc.) The developers of `Sollya` are open to suggestions and feature requests on this subject.

As an illustration let us give a few examples of possible use of callbacks:

Example 1: A callback that filters everything.

```
int hide_everything(sollya_msg_t msg) {
  return 0;
}
```

Example 2: filter everything but the messages indicating that a comparison is uncertain.

```
int keep_comparison_warnings(sollya_msg_t msg) {
  switch(sollya_lib_get_msg_id(msg)) {
    case SOLLYA_MSG_TEST_RELIES_ON_FP_RESULT_THAT_IS_NOT_FAITHFUL:
    case SOLLYA_MSG_TEST_RELIES_ON_FP_RESULT:
    case SOLLYA_MSG_TEST_RELIES_ON_FP_RESULT_FAITHFUL_BUT_UNDECIDED:
    case SOLLYA_MSG_TEST_RELIES_ON_FP_RESULT_FAITHFUL_BUT_NOT_REAL:
      return 1;
    default:
      return 0;
  }
}
```

Example 3: ensuring perfect silence for a particular function call (uses the callback defined in Example 1).

```
...
int (*)(sollya_msg_t) old_callback = sollya_lib_get_msg_callback();
sollya_lib_uninstall_msg_callback();
sollya_lib_install_msg_callback(hide_everything);
  /* Here takes place the function call that must be completely silent */
sollya_lib_uninstall_msg_callback();
if (old_callback) sollya_lib_install_msg_callback(old_callback);
...
```

Example 4: changing the value of some flag when some message is emitted.

```
int flag_double_rounding = 0;

int set_flag_on_problem(sollya_msg_t msg) {
  switch(sollya_lib_get_msg_id(msg)) {
    case SOLLYA_MSG_DOUBLE_ROUNDING_ON_CONVERSION:
      flag  = 1;
  }
  return 1;
}

...

int main() {
  ...
  sollya_lib_init();
  sollya_lib_install_msg_callback(set_flag_on_problem);
  ...
}
```

More involved examples are possible: for instance, instead of setting a flag, it is possible to keep in some variable what the last message was. One may even implement a stack mechanism and store the messages in a stack, in order to handle them later. One may also decide to raise an exception flag on a particular message, etc.