1 Introduction

`sollya` is an interactive tool to handle numerical functions and working with arbitrary precision. It can evaluate functions accurately, compute polynomial approximations of functions, plot functions, compute infinite norms, etc. Let us begin this manual with an example. `sollya` does not allow command line edition; since it may quickly become uncomfortable, we highly suggest to use the software `rlwrap` with `sollya`:

```
 /%rlwrap ./tools
>
```

`sollya` manipulates only with univariate functions. The first time that an unbound variable is used, it fixes the name that will be used to refer to the free variable. For instance, try

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

Now, the name x cannot be used to refer to anything but the free variable:

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

If you really want to unbound 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
>
```

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

```
> evaluatef(-2);
Warning:  rounding has happened.  The value displayed is a faithful rounding of
the true result.
0.4546487134128408476980099329558724213511274857239 41
```

The printed value is generally a faithful rounding of the exact value at the working precision. 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
the true result.
0.45464871341284084769800993295587242135112748572394513418948652
```

Sometimes, a faithful rounding cannot be easily computed. In such a case, an approximated value is printed:

```
> sin(pi);
Warning:  rounding has happened.  The value displayed is not a faithful rounding
of the true result.
0.23702821088022102782921822379162058842068273168003320319597978e-15412
```

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);
> max=0; for t in derivativeZeros do
{
 r = evaluate(abs(p-f), t);
 if r > max then max=r; argmax=t;;
};
> print("The infinite norm of", p-f, "is", max, "and is reached at", argmax);
The infinite norm of (-0.41626557294429078481812212) + x * ((-0.1798067204872539
9037039096583612263e1) + x * (-0.389710683640474564448652472492540 26e-1)) - cos(
2 * exp(x)) is 0.86306625059183635084725239e-3 and is reached at -0.580167296300
62879863317e-1
>
```

In this example, we define a function f, an interval d and we compute the best degree-4 polynomial approximation of f on d with respect to the infinite norm. In other words, $\max_{x \in d}\{|p(x) - f(x)|\}$ is minimal amongst polynomials with degree not greater than 4. 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 (the output was too long to fit in this document. That is why we used `[...]` to shortcut the actual output).

Note that you do not really need to use such a script for computing infinite norm; as we will see, the command `dirtyinfnorm` does this for you.

2 General principles

The first goal of `sollya` is to help people using numerical functions and numerical algorithm in a safe way. It is first designed to be used interactively but it can also be used in script files.

One of the originalities of `sollya` is to work with a multiprecision arithmetic (it uses the library `MPFR`). For safety purpose `sollya` knows how to use interval arithmetic. It uses the interval arithmetic to produce tight and safe results with the precision required by the user.

The general philosophy of `sollya` is: *When you can make a computation exactly, do it; when you cannot, do not, unless you have been explicitely asked for.*

The precision of the tools is set by the global variable `prec`. It indicates the number of bits used to represent the constants in `sollya`. In general, the variable `prec` determines the precision of the outputs of commands: more precisely, the command will internally determine what precision should be used during the computations in order to ensure that the output is a faithful result with `prec` bits.

For decidability and efficiency reasons, this general principle cannot be applied everytime, so be careful. Moreover certain commands are known to be unsafe: they give in general excellent results and gives almost `prec` correct bits in output for everyday examples. However they are just heuristic 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 its result.

A second principle (that comes together with the first one) is: *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 set to 0, `sollya` becomes completely silent on stdout and prints only very important messages on stderr. Increase `verbosity` if you want more informations about what `sollya` is doing. Note that when you affect a value to a global variable, a message is always printed even if `verbosity` is set to 0. To affect silently 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 he manipulates. But since constant 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 avaliable in `sollya`. The global variable `display` defines the way constants are displayed. Here is an example of the four avalaible modes:

```
> prec=30!;
> a = 17.25;
> display=decimal;
Display mode is decimal numbers.
> a;
0.1725e2
> 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
```

The symbol `e` means $\times 10^{\square}$ as always. In the same spirit, the symbol `b` means $\times 2^{\square}$. The suffix `_2` indicates to `sollya` that the previous number has been written in binary. `sollya` understands these notations even if you are not in the corresponding `display` mode, so you can always use them.

You can also use the standard hexadecimal notation frequently used to represent `double` numbers. Since this notation cannot represent exactly numbers with arbitrary precision, there is no corresponding `display` mode. However, the command `printhexa` rounds the value to the nearest `double` and displays it in hexadecimal:

```
> printhexa(a);
0x4031400000000000
```

3 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 type error. Let us see the available data types.

3.1 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 be used only between two numbers.

The comparison operators `==` and `!=` are polymorphic. You can use it to compare two strings, two intervals, etc. Note 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. Example:

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

3.2 Numbers

As seen above, `sollya` represents numbers as floating-point values with the current precision `prec`. A number in an expression is immediately rounded to the precision `prec` and the expression is then evaluated:

```
> prec=12!;
> 4097;
Warning:  Rounding occured when converting the constant "4097" to floating-point
with 12 bits.
If safe computation is needed, try to increase the precision.
4096
> 4098;
4098
> 4097+1;
Warning:  Rounding occured when converting the constant "4097" to floating-point
with 12 bits.
If safe computation is needed, try to increase the precision.
Warning:  rounding has happened.  The value displayed is a faithful rounding of
the true result.
4096
```

Note that each variable has its own precision that corresponds to the value of `prec` when the variable was set. Thus you can work with variables having a precision bigger thant the current precision.

In the same spirit, 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 refer to some variable, the precision of the variable is kept, independently of the current precision:

```
> prec=50!;
> a = 4097;
> prec=12!;
> f = x+a;
> g = x+4097;
Warning:  Rounding occured when converting the constant "4097" to floating-point
with 12 bits.
If safe computation is needed, try to increase the precision.
> prec=50!;
> f;
4097 + x
> g;
4096 + x
```

3.3 Intervals

Intervals are composed of two numbers representing the lower and the upper bound. These values are separated either by commas or semi-colons:

```
> d=[1;2];
> d2=[1,2];
> d==d2;
true
```

The constant are stored with the precision used during the affectation. If you refer to a variable when defining an interval, the precision of the variable is used:

```
> prec=30!;
> a=4097;
> prec=12!;
> d=[4096; a];
> prec=30!;
> d;
[4096;4097]
```

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

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

3.4 Functions

`sollya` knows only 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`, `atanh`
- `exp`, `expm1` (defined as $\mathrm{expm1}(x) = \exp(x) - 1$)
- `log` (neperian logarithm), `log2` (binary logarithm), `log10` (decimal logarithm), `log1p` (defined as $\mathrm{log1p}(x) = \log(1 + x)$)
- `erf`, `erfc`

The constant π is available as a 0-ary function: its behavior is exactly the same as if it were a constant with an infinite precision:

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

3.5 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 above). There is no char 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];
!
```

3.6 Particular values

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

- `on`, `off`
- `dyadic`, `powers`, `binary`, `decimal`
- `file`, `postscript`, `postscriptfile`
- `RU`, `RD`, `RN`, `RZ`
- `absolute`, `relative`
- `double`, `doubleextended`, `doubledouble`, `tripledouble`
- `D`, `DE`, `DD`, `TD`

- `perturb`
- `honorcoeffprec`
- `default`
- `error`

3.7 Lists

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

You can add an element to the left or the right of a list using `::`. The following example illustrates some features:

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

4 Tests and loops

5 Commands

5.1 absolute

Name: **perturb**
indicates an absolute error for **externalplot**

Usage:

perturb : absolute—relative

Description:

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

Example 1:

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

See also: **externalplot**, **relative**, **bashexecute**

5.2 abs

Name: **abs**
the absolute value.

Description:

- **abs** is the absolute value function. **abs**(x)=x if x¿0 and -x otherwise.

5.3 accurateinfnorm

Name: **accurateinfnorm**
computes a faithful rounding of the infinite 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 infinite norm is to be computed
range represents the infinite 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 infinite norm of function *function* in *range*. This upper bound is the least floating-point number greater than the value of the infinite 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 infinite 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 infinite norm of the function *function* is not to be considered on.

Example 1:

```
> p = remez(exp(x), 5, [-1;1]);
> accurateinfnorm(p - exp(x), [-1;1], 20);
0.45205641072243e-4
> accurateinfnorm(p - exp(x), [-1;1], 30);
0.45205621802324458e-4
> accurateinfnorm(p - exp(x), [-1;1], 40);
0.4520562176940634557 8e-4
```

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);
0.4520562176940634557 8e-4
```

See also: **infnorm**, **dirtyinfnorm**, **checkinfnorm**, **remez**, **diam**

5.4 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**

5.5 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**

5.6 and

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: ||, **!**

5.7 append

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 be in particular 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: **.:**, **@**

5.8 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. If Sollya is connected to a terminal, the size of the plot is determined by the size of the terminal. If not, the plot is of fixed size. 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, a x-axis is displayed. If zero is in range, an 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]);
                                                                    xx
                                                                  xx
                                                                xx
                                                              xx
                                                            xx
                                                          xxx
                                                        xx
                                                     xxx
                                                   xx
                                                 xxx
                                               xxx
                                             xxx
                                           xxx
                                         xxxx
                                       xxx
                                     xxxx
                                   xxxx
                                 xxxx
                              xxxx
                           xxxxx
                        xxxxx
                     xxxxx
                  xxx
```

Example 2:

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

Example 3:

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

Example 4:

```
> asciiplot(exp(x),[1;1]);
0.271828182845904523536028747135266249775724709369998e1
```

See also: **plot**

5.9 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**

5.10 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**

5.11 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**

5.12 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**

5.13 autosimplify

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

Usage:

autosymplify = *activation value* : on—off → void
autosymplify = *activation value* ! : on—off → void
autosymplify = ? : void → on—off

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

Description:

- An assignment **autosymplify** = *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 commans. When the automatic simplification of expressions is activated, Sollya automatically performs a safe (not value changing) simplification process on such expression. The automatic generation of subexpressions can be annoying, in particular if it takes too much time for not enough usage. 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 **autosymplify** = *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.
- The expression **autosymplify** = ? evaluates to a variable of type on—off, indicating whether or not the automatic simplifications of expressions of functions is activated.

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
```

Example 3:

```
> autosimplify = ?;
on
```

See also: **print**, **prec**, **points**, **diam**, **display**, **verbosity**, **canonical**, **taylorrecursions**, **timing**, **fullparentheses**, **midpointmode**, **hopitalrecursions**, **remez**, **rationalapprox**, **taylor**

5.14 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("ls /");
bin
boot
cdrom
dev
etc
home
lib
lib64
lost+found
media
mnt
opt
proc
root
sbin
srv
sys
tmp
usr
var
```

See also: **execute**, **readfile**, **parse**

5.15 binary

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

See also: **decimal**, **dyadic**, **powers**, **hexadecimal**

5.16 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**, **constant**, **function**, **integer**, **list of**, **range**, **string**

5.17 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
canonical = ? : void → on—off

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 yield 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.
- The expression **canonical** = ? evaluates to a variable of type on—off, indicating whether or not the automatic printing of subexpressions in canonical form is activated. If automatic printing in canonical form is deactivated, automatic printing yield to displaying polynomial subexpressions in Horner form.

Example 1:

```
> print(canonical(1 + x * (x + 3 * x^2));
> 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 + 160 * x^2 + 160
```

Example 3:

```
> canonical = ?;
off
> (x + 2)^9;
512 + x * (2304 + x * (4608 + x * (5376 + x * (4032 + x * (2016 + x * (672 + x * (144 + x * (1
> 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 + 1
> canonical = ?;
on
> canonical = off!;
> (x + 2)^9;
512 + x * (2304 + x * (4608 + x * (5376 + x * (4032 + x * (2016 + x * (672 + x * (144 + x * (1
```

See also: **horner**, **print**

5.18 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**

5.19 checkinfnorm

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

Usage:

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

Parameters: *function* represents the function whose infinite norm is to be checked
range represents the infinite norm is to be considered on
constant represents the upper bound the infinite norm is to be checked to

Description:

- The command **checkinfnorm** checks whether the infinite 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 infinite 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 infinite norm of the function or the bound is greater than the infinite 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 infinite 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 infinite norm of the function *function* in range *range*. In contrast, with **diam** being zero, **checkinfnorm** evaluates to **true** iff the infinite 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],
false
> 1/2);
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, 20, 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**, **dirtyinfnorm**, **accurateinfnorm**, **remez**, **diam**

5.20 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**

5.21 concat

Name: **@**
concatenates two lists or strings.

Usage:

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

Parameters: *L1* and *L2* are two lists.
string1 and *string2* are two strings.

Description:

- **@** concatenates two lists or strings.

Example 1:

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

Example 2:

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

See also: **.:**, **:.**

5.22 constant

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

Usage:

constant : type type

Description:

- **constant** represents the constant type for declarations of external procedures by means of **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**, **boolean**, **function**, **integer**, **list of**, **range**, **string**

5.23 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**, **sinh**, **tanh**, **exp**

5.24 cos

Name: **cos**
the cosine function.

Description:

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

See also: **acos**, **sin**, **tan**

5.25 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**, **powers**, **hexadecimal**, **binary**

5.26 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 uninterresting 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.
```

5.27 degree

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

Usage:

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

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**

5.28 denominator

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

Usage:

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

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**

5.29 diam

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

Description:

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

See also: **infnorm**, **checkinfnorm**, **accurateinfnorm**, **integral**, **findzeros**

5.30 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**

5.31 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 is the following: 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 precision 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]);
[|-0.31415926535897932384626433832795028841971693993750801e1, 0, 0.31415926535897932384626433
```

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**, **points**, **findzeros**

5.32 dirtyinfnorm

Name: **dirtyinfnorm**
computes a numerical approximation of the infinite 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 infinite 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 infinite 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]);
0.145855712891e1
> prec=40!;
> dirtyinfnorm(exp(cos(x))*sin(x),[0;5]);
0.145852853713586227 97e1
> prec=100!;
> dirtyinfnorm(exp(cos(x))*sin(x),[0;5]);
0.14585285371362376444381474550245100 15e1
> prec=200!;
> dirtyinfnorm(exp(cos(x))*sin(x),[0;5]);
0.14585285371362376444381474550238417182992140879936823740941534981883e1
```

Example 3:

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

See also: **prec**, **points**, **infnorm**, **checkinfnorm**

5.33 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 known to be currently unsatisfactory. If you really need to compute integrals, think of using an other tool or report a feature request to sylvain.chevillard@ens-lyon.fr.

Example 1:

```
> sin(10);
-0.544021110889369813404747661851377281683643012916219
> dirtyintegral(cos(x),[0;10]);
-0.544003049051526298224480588824753820365362983562818375241
> points=2000!;
> dirtyintegral(cos(x),[0;10]);
-0.544019977511583219722226973125831990359958379268927638295
```

See also: **prec**, **points**, **integral**

5.34 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 = ? : void → 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/ IEEE754R notation. All output notations can be reparsed 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.

- The expression **display** = ? evaluates to a variable of type decimal—binary—dyadic—powers—hexadecimal, indicating the current notation used.

Example 1:

```
> display = decimal;
Display mode is decimal numbers.
> a = evaluate(sin(pi * x), 0.25);
> a;
0.707106781186547524400844362104849039284835937688470740971063
> display = binary;
Display mode is binary numbers.
> a;
1.01101010000010011110011001100111111100111011110011001001000010001011001011111011000100110110
> 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**, **write**, **decimal**, **dyadic**, **powers**, **binary**, **hexadecimal**, **prec**

5.35 divide

Name: /
division function

Usage:

function1 / *function2* : (function, function) → function

Parameters: *function1* and *function2* represent functions

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.

Example 1:

```
> 5 / 2;
0.25e1
```

Example 2:

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

Example 3:

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

Example 4:

```
> 3 / 0;
@Inf@
```

Example 5:

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

See also: +, −, ∗, ^

5.36 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 xh = **double**(x) and let xl = **double**(x-xh). Return the number xh+xl. Note that if the current precision is not sufficient to represent exactly xh+xl, 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 **roundcoefficients** and **implementpoly**. See the corresponding help pages for examples.

Example 1:

```
> verbosity=1!;
> a = 1+ 2^(-100);
> DD(a);
0.10000000000000000000000000000078886090522101180541173e1
> prec=50!;
> DD(a);
Warning: double rounding occurred on invoking the double-double rounding operator.
Try to increase the working precision.
1
```

See also: **double**, **doubleextended**, **tripledouble**, **roundcoefficients**, **implementpoly**

5.37 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 functions of Sollya such as **exp**, **sin**, etc.
- It does not handle subnormal numbers. The range of possible exponents is the range used for all numbers represented in Sollya (e.g. basically the range used in the library MPFR).
- 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).
- Be aware that **doubleextended** cannot be used as a constant to represent a format in the commands **roundcoefficients** and **implementpoly** (contrary to **D**, **DD**,and **TD**).

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^(-2000));
0.87098098162172166755761954947788722958591037427053886 2e-602
```

Example 3:

```
> verbosity=1!;
> f = sin(DE(x));
> f(pi);
Warning: rounding has happened. The value displayed is a faithful rounding of the true result
-0.501655761266833202355732708033075701383156167025495e-19
> g = sin(round(x,64,RN));
Warning: at least one of the given expressions or a subexpression is not correctly typed
or its evaluation has failed because of some error on a side-effect.
```

See also: **double**, **doubledouble**, **tripledouble**, **round**

5.38 double

Names: **double**, **D**
rounding to the nearest IEEE double.

Description:

- **double** is both a function and a constant.

- As a function, it rounds its argument to the nearest double precision number. Subnormal numbers are supported as well as standard numbers: it is the real ounding described in the standard.
- As a constant, it symoblizes the 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:

```
> 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: **doubleextended**, **doubledouble**, **tripledouble**, **roundcoefficients**, **implementpoly**

5.39 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**, **powers**, **hexadecimal**, **binary**

5.40 equal

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 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.

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:

```
> prec = 12;
The precision has been set to 12 bits.
> 16384 == 16385;
true
```

Example 4:

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

See also: **!**=, >, >=, <=, <, **!**, &&, ||, **error**, **prec**

5.41 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**

5.42 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**, **exp**

5.43 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 input 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 == error || correct != error);
true
> (incorrect == error || incorrect != error);
false
```

See also: **void**, **parse**

5.44 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 pre-image domain *range* is surely enclosed in the returned range.
- If its second argument is a function *function2* that is not a constant, **evaluate** replaces all occurences of the free variable in function *function* by function *function2*.

Example 1:

```
> print(evaluate(sin(pi * x), 2.25));
0.707106781186547524400844362104849039284835937688470741
```

Example 2:

```
> print(evaluate(sin(pi * x), 2));
[-0.172986452514381269516508615031098129542836767991679e-12714;0.75941198201187963145069564314
```

Example 3:

```
> print(evaluate(sin(pi * x), [2, 2.25]));
[-0.514339027267725463004699891996191240734922416542101e-49;0.70710678118654752440084436210484
```

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@
```

See also: **isevaluable**

5.45 execute

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

Usage:

execute(*filename*) : string → 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 interpretation but it
> a;
1
```

See also: **parse**, **readfile**, **write**, **print**, **bashexecute**

5.46 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* is 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
```

See also: **simplify**, **simplifysafe**, **horner**

5.47 expm1

Name: **expm1**
translated exponential function.

Description:

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

See also: **exp**

5.48 exponent

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

Usage:

$$\textbf{exponent}(x) : \mathsf{constant} \to \mathsf{integer}$$

Parameters: x is a dyadic number.

Description:

- **exponent**(x) is by definition 0 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. **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**, **precision**

5.49 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**, **log**

5.50 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 quotiented 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 an evaluates to **perturb**, each of these floating-point numbers is 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 -lmpfr");
> externalplot("./externalplotexample",relative,exp(x),[-1/2;1/2],12,perturb);
```

See also: **plot**, **asciiplot**, **perturb**, **absolute**, **relative**, **file**, **postscript**, **postscriptfile**, **bashexecute**, **externalproc**, **library**

5.51 externalproc

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

Usage:

externalproc(*identifier*, *filename*, *argumenttype -¿ resulttype*) : (identifier type, string, type type, type type) → 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 iternal 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 MPFI 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 MPFI 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 as 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 correctly typed
or its evaluation has failed because of some error on a side-effect.
error
> a = foo;
> a(5,6);
11
```

See also: **library**, **externalplot**, **bashexecute**, **void**, **constant**, **function**, **range**, **integer**, **string**, **boolean**, **list of**

5.52 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**, &&, ||

5.53 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**, **plot**, **postscript**, **postscriptfile**

5.54 findzeros

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

Usage:

findzeros(*f*,*I*) : (function, range) → list

Parameters: *f* is a function.
I is an interval.

Description:

- **findzeros**(*f*,*I*) returns a list of intervals *I1*, ... ,*In* such that, for every zero z of f, there exists some k such that $z \in I_k$.
- The list may contain intervals *Ik* that do not contain any zero of *f*. An interval *Ik* may contain many zeros of *f*.
- This command is ment 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]);
[|[-0.314208984375e1;-0.3140869140625e1], [-0.1220703125e-2;0.1220703125e-2], [0.3140869140625
> diam=1e-10!;
> findzeros(sin(x),[-5;5]);
[|[-0.3141592653701081871986389160156 25e1;-0.3141592652536928653717041015625e1], [-0.11641532
```

See also: **dirtyfindzeros**, **infnorm**, **prec**, **diam**, **taylorrecursions**, **hopitalrecursions**

5.55 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**

5.56 fullparentheses

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

Usage:

fullparentheses = *activation value* : on—off → void
fullparentheses = *activation value* ! : on—off → void
fullparentheses = ? : void → on—off

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 parenthezing. In full parentheszing mode, Sollya commands like **print**, **write** and the implicit command when an expression is given at the prompt will output expressions with parentheses at all places where it is necessary for expressions containing infix operators to be reparsed 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.
- The expression **fullparentheses** = ? evaluates to a variable of type on—off, indicating whether or not the full parenthesized output of expressions is activated or not.

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**, **write**, **autosymplify**

5.57 function

Name: **function**
keyword representing a function type

Usage:

function : type type

Description:

- **function** 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.

See also: **externalproc**, **boolean**, **constant**, **integer**, **list of**, **range**, **string**

5.58 ge

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 >= 16385;
true
```

See also: ==, **!**=, >, <=, <, **!**, &&, ||, **prec**

5.59 gt

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.
> 16384 > 16385;
false
```

See also: ==, **!**=, >=, <=, <, **!**, &&, ||, **prec**

5.60 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*) : (function, range, constant, function) → 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.

Description:

- **guessdegree** tries to find the minimal degree needed to approximate f on I by a polynomial with an infinite error 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 (e.g. 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*.

Example 1:

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

Example 2:

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

See also: **dirtyinfnorm**, **remez**

5.61 head

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

Usage:

head(L) : list → 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**

5.62 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 754R notation. As any value it can be affected to a variable and stored in lists.

See also: **decimal**, **dyadic**, **powers**, **binary**

5.63 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 (heuristical) 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","implementation.c"
Warning: at least one of the coefficients of the given polynomial has been rounded in a way
that the target precision can be achieved at lower cost. Nevertheless, the implemented polynor
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","implementation.c"
Warning: the infered precision of the 2th coefficient of the polynomial is greater than
the necessary precision computed for this step. This may make the automatic determination
of precisions useless.
> printexpansion(r);
0x3ff0000000000000 + x^2 * (0xbfc5555555555555 + 0xbc65555555555555 + 0xb905555555555555)
```

See also: **implementpoly**, **printexpansion**

5.64 hopitalrecursions

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

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 hold 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 one time ; setting it to 1 makes Sollya use the rule two times, 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]);
[-0.252258872223978123766892848583270627230200053744108e1;0.67725887222397812376689284858327
```

5.65 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**, **print**

5.66 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 eightth 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 renormalizes double-double and triple-double precision coefficients to a round-to-nearest expansion, the polynomial return 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 (heuristical) 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 propgation with regard to their coefficients' values. Indicating **honorcoeffprec** prevents the **implementpoly** command from rounding the coefficients and altering by many orders of magnitude 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 writeable 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","implementation.c");
1 + x^2 * ((-0.1666666666666666574148081281236954964697360992431640625) + x^2 * 0.83333333333333
> readfile("implementation.c");
#define p_coeff_0h 1.0000000000000000000000000000000000000000000000000000000000000000000000000
#define p_coeff_2h -1.666666666666666574148081281236954964697360992431640625000000000000000000
#define p_coeff_4h 8.333333333333333217685101601546193705871701240539550781250000000000000000


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","implementation.c","in
1 + x^2 * ((-0.1666666666666666574148081281236954964697360992431640625) + x^2 * 0.833333333333333
> readfile("implementation.gappa");
# The polynomial to implement is: (1b0 - ((x^(1b1)) / 3b1)) + ((x^(1b2)) / 15b3)
# The polynomial implemented is: 1b0 + (x * (0 + (x * ((-6004799503160661b-55) + (x * (0 + (x * 4
# The domain is [-1b-10;1b-10]
# The free variable x is a double precision number, the result p_res* is stored on a double preci
# The code produces 10 intermediate and final arithmetical approximations.

# Double precision rounding operator:
@double = float<ieee_64,ne>;

# Disable some annoying warnings:
#@-Wno-dichotomy-failure

# Helper definitions for decomposing the free variable
xh = x;

# Transcription of the C code
p_coeff_0h = double(1.00000000000000000000000000000000000000000000000000000000000000000000000

p_coeff_2h = double(-1.666666666666666574148081281236954964697360992431640625000000000000000000

p_coeff_4h = double(8.333333333333333217685101601546193705871701240539550781250000000000000000

p_x_0_pow2h = double(xh * xh);
p_t_1_0h = p_coeff_4h;

p_t_2_0h = double(p_t_1_0h * p_x_0_pow2h);
p_t_3_0h = double(p_coeff_2h + p_t_2_0h);
p_t_4_0h = double(p_t_3_0h * p_x_0_pow2h);
p_t_5_0h = double(p_coeff_0h + p_t_4_0h);
p_resh = p_t_5_0h;


# Mathematical equivalents
Mx = x;
Mp_coeff_0 = p_coeff_0h;
Mp_coeff_2 = p_coeff_2h;
Mp_coeff_4 = p_coeff_4h;
Mp_x_0_pow2 = Mx * Mx;
Mp_t_1_0 = Mp_coeff_4;
Mp_t_2_0 = Mp_t_1_0 * Mp_x_0_pow2;
Mp_t_3_0 = Mp_coeff_2 + Mp_t_2_0;
Mp_t_4_0 = Mp_t_3_0 * Mp_x_0_pow2;
Mp_t_5_0 = Mp_coeff_0 + Mp_t_4_0;
Mp_res = Mp_t_5_0;

# Definition of the relative arithmetical error
epsilon = (p_resh - Mp_res) / Mp_res;

# Implication to prove
{((
   x in [-1b-10,-1b-410]
) \/ (
   x in [1b-410,1b-10]
))
->
(
   epsilon in ?
)}

# Hints and Meta-Hints for expansion decomposition
```

Example 3:

```
> verbosity = 1!;
> q = implementpoly(1 - simplify(TD(1/6)) * x^2,[-1b-10;1b-10],1b-60,DD,"p","implementation.c");
Warning: at least one of the coefficients of the given polynomial has been rounded in a way
that the target precision can be achieved at lower cost. Nevertheless, the implemented 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","implementation.c",ho
Warning: the infered precision of the 2th coefficient of the polynomial is greater than
the necessary precision computed for this step. This may make the automatic determination
of precisions useless.
> printexpansion(r);
0x3ff0000000000000 + x^2 * (0xbfc5555555555555 + 0xbc65555555555555 + 0xb905555555555555)
```

Example 4:

```
> p = 0x3ff0000000000000 + x * (0x3ff0000000000000 + x * (0x3fe0000000000000 + x * (0x3fc5555555
> q = implementpoly(p,[-1/2;1/2],1b-60,D,"p","implementation.c",honorcoeffprec,"implementation.ga
> if (q != p) then print("During implementation, rounding has happened.") else print("Polynomial
Polynomial implemented as given.
```

See also: **honorcoeffprec**, **roundcoefficients**, **D**, **DD**, **TD**, **double**, **doubledouble**, **tripledouble**, **readfile**, **printexpansion**, **error**

5.67 infnorm

Name: **infnorm**
computes an interval bounding the infinite 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 infinite 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 *I1*, ... ,*In* is given, the infinite norm will be computed on I ($I_1 \cup \ldots \cup I_n$).
- 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 true but perfectly useless. **infnorm** is not ment 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 big 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 applied recursively 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. 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 infinite norm for the implementation of elementary functions LIP Research Report number RR2007-26 http://prunel.ccsd.cnrs.fr/ensl-00119810

Example 1:

```
> infnorm(exp(x),[-2;3]);
[0.20085536923187667740928529654581717896987907838553 7e2;0.20085536923187667740928529654581717
```

Example 2:

```
> infnorm(exp(x),[-2;3],"proof.txt");
[0.20085536923187667740928529654581717896987907838553 7e2;0.20085536923187667740928529654581717
```

Example 3:

```
> infnorm(exp(x),[-2;3],[| [0;1], [2;2.5]|]);
[0.20085536923187667740928529654581717896987907838553 7e2;0.20085536923187667740928529654581717
```

Example 4:

```
> infnorm(exp(x),[-2;3],"proof.txt", [| [0;1], [2;2.5]|]);
[0.20085536923187667740928529654581717896987907838553 7e2;0.20085536923187667740928529654581717
```

Example 5:

```
> infnorm(exp(x),[1;1]);
[0.27182818284590452353602874713526624977572470936998 9e1;0.27182818284590452353602874713526624
```

Example 6:

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

See also: **prec**, **diam**, **hopitalrecursions**, **dirtyinfnorm**, **checkinfnorm**

5.68 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**, **sup**

5.69 integer

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

Usage:

$$\mathbf{integer} : \mathsf{type}\ \mathsf{type}$$

Description:

- **integer** represents the machine integer type for declarations of external procedures by means of **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**, **boolean**, **constant**, **function**, **list of**, **range**, **string**

5.70 integral

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

Usage:

integral(f,I) : (function, range) → 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 unefficient. 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 9
> integral(cos(x),[0;10]);
[-0.54710197983579690224097637163525943075698599257332 9;-0.54094015130013183848150540881373370
> diam=1e-5!;
> integral(cos(x),[0;10]);
[-0.54432915685955427101857780295936956775293876382777 2;-0.54371306401249969508039644221927489
```

See also: **points**, **dirtyintegral**

5.71 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 globlal 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**

5.72 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. I.e. **isevaluable** returns false iff **evaluate** will return NaN.

Example 1:

```
> isevaluable(sin(pi * 1/x), 0.75);
true
> print(evaluate(sin(pi * 1/x), 0.75));
-0.866025403784438646763723170752936183471402626905185165
```

Example 2:

```
> isevaluable(sin(pi * 1/x), 0.5);
true
> print(evaluate(sin(pi * 1/x), 0.5));
[-0.172986452514381269516508615031098129542836767991679e-12714;0.759411982011879631450695643141
```

Example 3:

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

See also: **evaluate**

5.73 length

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

Usage:

length(L) : list → integer
length(s) : string → 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 has 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@
```

5.74 le

Name: <=
less-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 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 <= 16384;
true
```

See also: ==, **!**=, >=, >, <, **!**, &&, ||, **prec**

5.75 library

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

Usage:

library(*path*) : string → function

Description:

- The command **library** lets you extends the set of mathematical functions known by 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 manipulates 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 with. This implementation is a C file containing a function of the form:

  ```
  int my_ident(mpfi_t result, 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 returned value has no meaning currently.
- You must not provide a non trivial implementation for any `n`. Most functions of Sollya needs a relevant implementation of 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 MPFI standards in your implementation: `result` has its own precision and you should perform the intermediate computations so that `result` is as tighter as possible.
- You can include sollya.h in your implementation and use library functionnalities of Sollya for your implementation.
- 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).

Example 1:

```
> bashexecute("gcc -fPIC -Wall -c libraryexample.c");
> bashexecute("gcc -shared -o libraryexample libraryexample.o -lgmp -lmpfr");
> myownlog = library("./libraryexample");
> evaluate(log(x), 2);
0.693147180559945309417232121458176568075500134360248314207
> evaluate(myownlog(x), 2);
0.693147180559945309417232121458176568075500134360248314207
```

See also: **bashexecute**, **externalproc**, **externalplot**

5.76 listof

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**, **boolean**, **constant**, **function**, **integer**, **range**, **string**

5.77 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**, **log2**

5.78 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**

5.79 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**, **log10**

5.80 log

Name: **log**
neperian logarithm.

Description:

- **log** is the neperian 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**, **log2**, **log10**

5.81 lt

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 < 16385;
   false
```

See also: ==, **!**=, >=, >, <=, **!**, &&, ||, **prec**

5.82 mantissa

Name: **mantissa**
returns the integer mantissa of a number.

Usage:

$$\textbf{mantissa}(x) : \mathsf{constant} \rightarrow \mathsf{integer}$$

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**, **precision**

5.83 midpointmode

Name: **midpointmode**
global variable controlling the way intervals are displayed.

Description:

- **midpointmode** is a global variable. When its value is **off**, intervals are displayed as usual (with 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;
[0.31415926516056060791015625e1;0.31415926553308963775634765625e1]
> midpointmode=on!;
> d;
0.314159265~1/5~e1
```

See also: **on**, **off**

5.84 mid

Name: **mid**
gives the middle of an interval.

Usage:

$$\textbf{mid}(I) : \mathsf{range} \rightarrow \mathsf{constant}$$
$$\textbf{mid}(x) : \mathsf{constant} \rightarrow \mathsf{constant}$$

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 rounded.
- 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**, **sup**

5.85 minus

Name: −
substraction function

Usage:

$$\textit{function1} - \textit{function2} : (\mathsf{function}, \mathsf{function}) \rightarrow \mathsf{function}$$

Parameters: *function1* and *function2* represent functions

Description:

- − represents the substraction (function) on reals. The expression *function1* − *function2* stands for the function composed of the substraction function and the two functions *function1* and *function2*, where *function1* is the subtrahent and *function2* the substractor.

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)
```

See also: +, ∗, /, ^

5.86 mult

Name: ∗
multiplication function

Usage:

$$\textit{function1} * \textit{function2} : (\mathsf{function}, \mathsf{function}) \rightarrow \mathsf{function}$$

Parameters: *function1* and *function2* represent functions

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*.

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)
```

See also: +, −, /, ^

5.87 neq

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 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:

```
> prec = 12;
The precision has been set to 12 bits.
> 16384 != 16385;
false
```

Example 4:

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

See also: ==, >, >=, <=, <, **!**, &&, ||, **error**, **prec**

5.88 nop

Name: **nop**
no operation

Usage:

nop : void → 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 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:

```
> succ = proc(n) { return n + 1; };
> succ;
proc(n)
begin
nop;
return (n) + (1);
end
> succ(5);
6
```

See also: **proc**

5.89 not

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: &&, ||

5.90 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**

5.91 off

Name: **off**
special value for certain global variables.

Description:

- **off** is a special value used to deactivate certain functionnalities of Sollya (namely **canonical**, **timing**, **fullparentheses**, **midpointmode**).
- 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**, **canonical**, **timing**, **fullparentheses**, **midpointmode**

5.92 on

Name: **on**
special value for certain global variables.

Description:

- **on** is a special value used to activate certain functionnalities of Sollya (namely **canonical**, **timing**, **fullparentheses**, **midpointmode**).
- 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**, **canonical**, **timing**, **fullparentheses**, **midpointmode**

5.93 or

Name: ||
boolean OR operator

Usage:

$$expr1 \;||\; expr2 \;:\; (\mathsf{boolean}, \mathsf{boolean}) \rightarrow \mathsf{boolean}$$

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* evaluate to true.

Example 1:

```
> false || false;
false
```

Example 2:

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

See also: &&, **!**

5.94 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.

Example 1:

```
> parse("exp(x)");
exp(x)
```

Example 2:

```
> verbosity = 1!;
> parse("5 + + 3");
Warning: syntax error, unexpected PLUSTOKEN. Will try to continue parsing (expecting ";"). May
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 correctly typed
or its evaluation has failed because of some error on a side-effect.
error
```

See also: **execute**, **readfile**

5.95 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 -lmpfr");
> externalplot("./externalplotexample",relative,exp(x),[-1/2;1/2],12,perturb);
```

See also: **externalplot**, **absolute**, **relative**, **bashexecute**

5.96 pi

Name: **pi**
the constant Pi.

Description:

- **pi** is the constant Pi, defined as half the period of sine and cosine.
- In Sollya, **pi** is considered as 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 Pi, 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 the true result
0.62832e1
> prec=20!;
> a;
Warning: rounding has happened. The value displayed is a faithful rounding of the true result
0.62831879e1
```

Example 2:

```
> prec=12!;
> d = [pi; 5];
> d;
[0.31406e1;5]
> prec=20!;
> d;
[0.31406e1;5]
```

See also: **cos**, **sin**

5.97 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. Functions are plotted on the same graphic 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 big precision. Each point is finally plotted. This avoid numerical artefacts such as critical cancellations.
- You can save the graphic 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 plots datas stored in name.dat.
- If you use argument **postscript** with a string *name*, Sollya will save a postscript file called name.eps representing your graphic.
- 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 graphic. If your terminal is not graphic (typically if you use Sollya by ssh without -X) gnuplot should be able to detect it and produce an ASCII-art version on the standard output. If it is not the case, you can either store the graphic 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 3
```

Example 4:

```
> plot(sin(x), cos(x), [1;1]);
0.84147098480789650665250232163029899962256306079837 3
0.54030230586813971740093660744297660373231042061792 3
```

See also: **externalplot**, **asciiplot**, **file**, **postscript**, **postscriptfile**

5.98 plus

Name: +
addition function

Usage:

$$\textit{function1} + \textit{function2} : (\mathsf{function}, \mathsf{function}) \rightarrow \mathsf{function}$$

Parameters: *function1* and *function2* represent functions

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*.

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)
```

See also: −, ∗, /, ^

5.99 points

Name: **points**
controls the number of points chosen by Sollya in certain commands.

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, 0.31830988618379067153776752674502872406891929148091789|]
> points=100;
The number of points has been set to 100.
> dirtyfindzeros(f, [0;1]);
[|0, 0.24485375860291590118289809749617594159147637806224298 9e-1, 0.353677651315322968375297 25
```

See also: **dirtyinfnorm**, **dirtyintegral**, **dirtyfindzeros**, **plot**

5.100 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**, **plot**, **file**, **postscript**

5.101 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**, **plot**, **file**, **postscriptfile**

5.102 power

Name: ^
power function

Usage:

function1 ^ *function2* : (function, function) → function

Parameters: *function1* and *function2* represent functions

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 exp(y * log(x)).

Example 1:

```
> 5 ^ 2;
25
```

Example 2:

```
> x ^ 2;
x^2
```

Example 3:

```
> 3 ^ (-5);
0.41152263374485596707818930041152263374485596707818 6e-2
```

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)))
```

See also: +, −, ∗, /

5.103 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**, **dyadic**, **hexadecimal**, **binary**

5.104 precision

Name: **precision**
returns the precision necessary to represent a number.

Usage:

$$\textbf{precision}(x) : \mathsf{constant} \rightarrow \mathsf{integer}$$

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 (e.g. $\lceil \log_2(m) \rceil$).

Example 1:

```
> a=round(Pi,20,RN);
> precision(a);
19
> m=mantissa(a);
> ceil(log2(m));
19
```

See also: **mantissa**, **exponent**

5.105 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.
- A lot of commands try to adapt their intern 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]);
1.10110001110011001001011100011010100110111011011_2 * 2^(2)
> prec=100;
The precision has been set to 100 bits.
> dirtyinfnorm(exp(x),[1;2]);
1.10110001110011001001011100011010100110111011010110111001100001100111010001110111010001000
```

5.106 prepend

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

Usage:

x**.:**L : (any type, list) → 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: **:.**, **@**

5.107 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 **printhexa**). 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 * ((0x3fc555555555
```

Example 2:

```
> printexpansion(remez(exp(x),5,[-1;1]));
(0x3ff0002eec887685 + 0x3c691f5f67e9bc02 + 0xb8da4eda3533a5a0) + x * ((0x3ff00028359b7449 + 0x
```

Example 3:

```
> verbosity = 1!;
> prec = 3500!;
> printexpansion(pi);
(0x400921fb54442d18 + 0x3ca1a62633145c07 + 0xb92f1976b7ed8fbc + 0x35c4cf98e804177d + 0x32631d8
Warning: the expansion is not complete because of the limited exponent range of double precisi
Warning: rounding occurred while printing.
```

See also: **printhexa**, **horner**, **print**, **prec**, **remez**, **taylor**, **roundcoefficients**

5.108 printfloat

Name: **printfloat**
prints a constant value as a hexadecimal single precision number

Usage:

printfloat(*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 displayed. A warning is displayed in this case.

Example 1:

```
> printfloat(3);
0x40400000
```

Example 2:

```
> prec=100!;
> verbosity = 1!;
> printfloat(exp(5));
Warning: the given expression is not a constant but an expression to evaluate.
Warning: rounding occurred before printing a value as a simple.
0x431469c5
```

See also: **printhexa**

5.109 printhexa

Name: **printhexa**
prints a constant value as a hexadecimal double precision number

Usage:

printhexa(*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:

```
> printhexa(3);
0x4008000000000000
```

Example 2:

```
> prec=100!;
> verbosity = 1!;
> printhexa(exp(5));
Warning: the given expression is not a constant but an expression to evaluate.
Warning: rounding occurred before printing a value as a double.
0x40628d389970338f
```

See also: **printfloat**, **printexpansion**

5.110 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 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        you"|]
```

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);
0.70710678118654752440084436210484903928483593768847074097 1
> display = binary;
Display mode is binary numbers.
> print(a);
1.0110101000001001111001100110011111110011101111001100100100001000101100101111101100010011011001
> 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);
[0.70710678118654752440084436210484903928483593768844 9;0.70710678118654949743721782517557347782641
> midpointmode = on;
Midpoint mode has been activated.
> print(b);
0.70710678118654~7/9~
> display = dyadic;
Display mode is dyadic numbers.
> print(b);
[206687543694384741333274896801380902250419419582 9b-161;165350034955508254441962370193859364144321
> 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**, **printexpansion**, **printhexa**, **printfloat**, **printxml**, **readfile**, **autosymplify**, **display**, **midpointmode**, **fullparentheses**, **evaluate**

5.111 printxml

Name: **printxml**
prints an expression as an MathML-Content-Tree

Usage:

printxml(*expr*) : function → void
printxml(*expr*) ¿ *filename* : (function, string) → void
printxml(*expr*) ¿¿ *filename* : (function, string) → 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(...);   exemple: printxml(x^2-2*x+5); -->
<?xml-stylesheet type="text/xsl" href="http://perso.ens-lyon.fr/nicolas.jourdan/mathmlc2p-web
<?xml-stylesheet type="text/xsl" href="mathmlc2p-web.xsl"?>
<!-- This stylesheet allows direct web browsing of MathML-c XML files (http:// or 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**, **print**, **write**

5.112 proc

Name: **proc**
defines a Sollya procedure

Usage:

proc(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) **begin** *procedure body* **end** : void → procedure
proc(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) **begin** *procedure body* **return** *expression*; **end** : any type → procedure

Parameters:

- *formal parameter1*, *formal parameter2* through *formal parameter n* represent identifiers used as formal 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* 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, it is checked whether the number of actual parameters corresponds to the number of formal parameters. 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 semantical correctness checks, this check is really performed before interpreting any other statement in body of the procedure.

- By means provided by 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)
begin
nop;
return (n) + (1);
end
```

Example 2:

```
> add = proc(m,n) { var res; res := m + n; return res; };
> add(5,6);
11
> add;
proc(m, n)
begin
var res;
res := (m) + (n);
return res;
end
> verbosity = 1!;
> add(3);
Warning: at least one of the given expressions or a subexpression is not correctly 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 correctly typed
or its evaluation has failed because of some error on a side-effect.
Warning: the given expression or command could not be handled.
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()
begin
print("Hello world.");
return void;
end
```

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)
begin
var res;
if (n) == (0) then
res := 1
else
res := (n) * (fac((n) - (1)));
return res;
end
```

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)
begin
print(n);
quit;
return void;
end
> 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 occured 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("Successor of",n,"is",s
> 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)
begin
nop;
return (n) + (m);
end
> (makeadd(4))(5,6);
n = 4
11
```

See also: **return**, **externalproc**, **void**, **quit**, **restart**, **var**

5.113 procedure

Name: **procedure**
defines and assigns a Sollya procedure

Usage:

procedure *identifier*(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) **begin** *procedure body* **end** : void → void
procedure *identifier*(*formal parameter1*, *formal parameter2*,..., *formal parameter n*) **begin** *procedure body* **return** *expression*; **end** : any type → 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
- *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)
begin
nop;
return (n) + (1);
end
```

See also: **proc**, **var**

5.114 quit

Name: **quit**
quits Sollya

Usage:

quit : void → void

Description:

- The command **quit**, when executed abandons 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 the abandon of the Sollya interpreter, a warning is displayed.

Example 1:

```
> quit;
```

See also: **restart**, **execute**, **plot**, **externalplot**

5.115 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**, **boolean**, **constant**, **function**, **integer**, **list of**, **string**

5.116 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. It 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**, **numerator**, **denominator**

5.117 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**, **RU**, **RN**, **round**

5.118 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 correctly typed
or its evaluation has failed because of some error on a side-effect.
error
```

See also: **parse**, **execute**, **write**, **print**

5.119 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 parseable 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**, **readfile**, **parse**

5.120 relative

Name: **perturb**
indicates a relative error for **externalplot**

Usage:

perturb : absolute—relative

Description:

- The use of **perturb** in the command **externalplot** indicates that during plotting in **externalplot** a relative error is to be considered. See **externalplot** for details.

Example 1:

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

See also: **externalplot**, **absolute**, **bashexecute**

5.121 remez

Name: **remez**
computes the minimax of a function on an interval.

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 monomials that can be used to represent the polynomial that must approximate *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 a polynomial p such that $\|pw - f\|_\infty$ is (almost minimal) among all polynomials p of a certain form. The norm is the infinite 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, the polynomial p is searched among the polynomials with degree not greater than n. If L is given, the polynomial p is searched as a linear combination of monomials X^k where k belongs to L. L may contain ellipses but cannot be end-elliptic.
- The polynomial is obtained by a convergent iteration called Remez' algorithm. 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*.
- Note: the algorithm may not converge in certain cases. Moreover, it may converge towards a polynomial that is not optimal. These cases correspond to the cases when Haar's condition is not fulfilled. See [Cheney - Approximation theory] for details.

Example 1:

```
> p = remez(exp(x),5,[0;1]);
> degree(p);
5
> dirtyinfnorm(p-exp(x),[0;1]);
0.11295699414577782697647458116695164216174683110856685194 6e-5
```

Example 2:

```
> p = remez(1,[|0,2,4,6,8|],[0,Pi/4],1/cos(x));
> canonical=on!;
> p;
0.99999999994393732180541030692769254120351838903168 5 + (-0.49999999571556857755957441358176690047
```

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]);
0.11295699414577782697647458116695164216174683110856685194 6e-5
> dirtyinfnorm(p2-exp(x),[0;1]);
0.11295698022747876310115474830183586181755211237038868201e-5
> dirtyinfnorm(p3-exp(x),[0;1]);
0.11295698022747867369986911458163194541017612706338366850 3e-5
```

See also: **dirtyinfnorm**, **infnorm**

5.122 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** lets one change 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*.

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.
```

Example 4:

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

See also: **isbound**

5.123 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 of 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 function nor equal
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 function nor equal
Will interpret "a" as "x".
x
```

See also: **quit**, **execute**, **while**, **for**

5.124 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 explicited.

Example 1:

```
> succ = proc(n) { var res; res := n + 1; return res; };
> succ(5);
6
> succ;
proc(n)
begin
var res;
res := (n) + (1);
return res;
end
```

Example 2:

```
> hey = proc(s) { print("Hello",s); };
> hey("world");
Hello world
> hey;
proc(s)
begin
print("Hello", s);
return void;
end
```

See also: **proc**, **void**

5.125 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|]
```

5.126 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**, **RU**, **RZ**, **round**

5.127 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**D**, **double**, **DD**, **doubledouble**, **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 last coefficients (that do not have a corresponding element in L) are kept with their own precision. If L contains too much elements, the last 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|]);
1.01011011111100001010100010110001010000101011101101001010100110101010111111011100010101100010
> roundcoefficients(p,[|DD,D...|]);
1.01011011111100001010100010110001010000101011101101001010100110101010111111011100010101100010
```

Example 2:

```
> f=sin(exp(1)*x);
> display=binary!;
> f;
sin(x * 1.01011011111100001010100010110001010000101011101101001010100110101010111111011100010101
> roundcoefficients(f,[|D...|]);
sin(x * 1.01011011111100001010100010110001010000101011101101001010100110101010111111011100010101
```

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 the given polync
Warning: the 0th coefficient of the given polynomial does not evaluate to a floating-point cor
Will evaluate the coefficient in the current precision in floating-point before rounding to th
Warning: the 1th coefficient of the given polynomial does not evaluate to a floating-point cor
Will evaluate the coefficient in the current precision in floating-point before rounding to th
Warning: rounding may have happened.
1.01011011111100001010100010110001010001010111011010010101001101010101111110111000101011000100
```

See also: **double**, **doubledouble**, **tripledouble**

5.128 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 * 2^(0)
> printbinary(roundcorrectly([1.00001_2; 1.001_2]));
1._2 * 2^(0)
```

Example 2:

```
> roundcorrectly([-1; 1]);
@NaN@
```

See also: **round**

5.129 round

Name: **round**
rounds a number to a floating-point format.

Usage:

round(x,n,*mode*) : (constant, integer, **RD**—**RU**—**RN**—**RZ**) → constant

Parameters: x is a constant to be rounded.
n is the precision of the target format.
mode is the desired rounding mode.

Description:

- **round**(*x*,*n*,*mode*) rounds x to a floating-point number with precision n, according to rounding-mode *mode*.
- Subnormal numbers are not handled. The range of possible exponents is the range used for all numbers represented in Sollya (e.g. basically the range used in the library MPFR). Please use the functions **double**, **doubleextended**, **doubledouble** and **tripledouble** for roundings to classical formats with their range of exponents.

Example 1:

```
> display=binary!;
> round(Pi,20,RN);
1.10010010000111111_2 * 2^(1)
```

Example 2:

```
> display=binary!;
> a=2^(-1100);
> round(a,53,RN);
1._2 * 2^(-1100)
> double(a);
0
```

See also: **RN**, **RD**, **RU**, **RZ**, **double**, **doubleextended**, **doubledouble**, **tripledouble**, **roundcoefficients**, **roundcorrectly**

5.130 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**, **RD**, **RN**, **round**

5.131 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 truncate the value to the desired format.

Example 1:

```
> display=binary!;
> round(Pi,20,RZ);
1.10010010000111111_2 * 2^(1)
```

See also: **RD**, **RU**, **RN**, **round**

5.132 searchgal

Name: **searchgal**
searches for a preimage of a function such that the rounding the image commits an error smaller than a constant

Usage:

searchgal(*function*, *start*, *preimage precision*, *steps*, *format*, *error bound*) : (function, constant, integer, integer, D—double—DD—doubledouble—DE—doubleextended—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 (discretisation) for the eligible preimage values
steps represents the log2 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 a 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 raised to *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);
[|0.200000000003849720542348222807049751281738281 25e1|]
```

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^( -0.112106878438809380148206984258358542322113874177832146e3 )
```

Example 3:

```
> searchgal([|sin(x),cos(x)|],1,53,15,[|D,D|],[|1b-62,1b-60|]);
[|0.100000000000159494639717649988597258925437927246093 75e1|]
```

See also: **round**, **D**, **DD**, **TD**, **double**, **doubledouble**, **tripledouble**, **evaluate**, **worstcase**

5.133 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**, **autosymplify**

5.134 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 in using floating-point arithmetic with the global precision **prec**.

Example 1:

```
> print(simplify(sin(pi * x)));
sin(0.31415926535897932384626433832795028841971693993750801e1 * x)
> print(simplify(erf(exp(3) + x * log(4))));
erf(0.20085536923187667740928529654581717896987907838554378 5e2 + x * 0.13862943611198906188344
```

Example 2:

```
> prec = 20!;
> t = erf(0.5);
> s = simplify(erf(0.5));
> prec = 200!;
> t;
0.52049987781304653768274665389196452873645157575796370005880583
> s;
0.52050018310546875
```

See also: **simplifysafe**, **autosymplify**, **prec**, **evaluate**

5.135 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**, **cosh**, **tanh**

5.136 sin

Name: **sin**
the sine function.

Description:

- **sin** is the usual sine function.
- It is defined for every real number x.

See also: **asin**, **cos**, **tan**

5.137 sort

Name: **sort**
sorts a list of real numbers.

Usage:

$$\mathbf{sort}(L) : \mathsf{list} \to \mathsf{list}$$

Parameters: L is a list.

Description:

- If L contains only constant values, **sort**(L) returns the same list, but sorted increasingly.
- 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|]
```

5.138 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 $[0; +\infty]$.

5.139 string

Name: **string**
keyword representing a string type

Usage:

string : type 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**, **boolean**, **constant**, **function**, **integer**, **list of**, **range**

5.140 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 explicitely 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 * (0.99986632946592739219220656843208843699165447057218 8 + x^2 * ((-0.33030478550497113295 06
```

Example 3:

```
> subpoly(exp(x),[|1,2,3|]);
0
```

See also: **roundcoefficients**, **taylor**, **remez**

5.141 substitute

Name: **substitute**
replace the occurences 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. It is absolutely equivalent to writing *F(g)* or *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);
0.84147098480789650665250232163029899962256306079837 3
> f(a);
0.84147098480789650665250232163029899962256306079837 3
```

5.142 sup

Name: **sup**
gives the upper bound of an interval.

Usage:

sup(I) : range → constant
sup(x) : constant → 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**, **mid**

5.143 tail

Name: **tail**
gives the tail of a list.

Usage:

tail(L) : list → any type

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**

5.144 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**, **cosh**, **sinh**

5.145 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**, **cos**, **sin**

5.146 taylorrecursions

Name: **taylorrecursions**
controls the number of recursion steps when applying Taylor's rule.

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 one time; setting it to 1 makes Sollya use the rule two times, 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]);
[-0.468393649043687401639739552931015355858675716815424;0.469477818012697748765376148435266259
> taylorrecursions=1;
The number of recursions for Taylor evaluation has been set to 1.
> evaluate(f-p, [0;1]);
[-0.138131114682060169766463621512015307211349976570287;0.139215283651070516892100217016266210
```

5.147 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

$$\sum_{i=0}^{n} \frac{f^{(i)}}{i!} (x-t)^i$$

  Remark that **taylor** evaluates to 0 if the degree *degree* is negative.

Example 1:

```
> print(taylor(exp(x),5,0));
1 + x * (1 + x * (0.5 + x * (1 / 6 + x * (1 / 24 + x / 120))))
```

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
```

See also: **remez**

5.148 timing

Name: **timing**
global variable controlling timing measures in Sollya.

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 3 ms
Information: Remez: computing the quality of approximation spent 17 ms
Information: computing a minimax approximation spent 24 ms
Information: assignment spent 24 ms
Information: full execution of the last parse chunk spent 24 ms
```

See also: **on**, **off**

5.149 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 xh = **double**(x) and let xl = **doubledouble**(x-xh). Return the number xh+xl. Note that if the current precision is not sufficient to represent exactly xh+xl, 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);
0.10000000000000002775557561562891353466491600711095598e1
> prec=110!;
> TD(a);
Warning: double rounding occurred on invoking the triple-double rounding operator.
Try to increase the working precision.
0.1000000000000000027755575615628913510 6e1
```

See also: **double**, **doubleextended**, **doubledouble**, **roundcoefficients**, **implementpoly**

5.150 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**, &&, ||

5.151 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**

5.152 verbosity

Name: **verbosity**
global variable controlling the quantity of information displayed by commands.

Description:

- **verbosity** accepts any integer value. At level 0, commands do not display anything on standard out. Note that very critical information may however be displayed on standard err.
- Default level is 1. It displays important informations such as warnings when roundings happen.
- For higher levels more informations are 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 correctly 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 correctly 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:
(0.11999999999999999999999999999999999999999999999999e1) + ("toto")
error
```

5.153 void

Name: **void**
the functional result of a side-effect or empty argument resp. the correponding 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 not argument to a procedure or an external procedure when applied. It can be explicited nevertheless. 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.
- **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()
begin
print("Hello world.");
return void;
end
> 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 externalprocvoidexample.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
```

See also: **error**, **proc**, **externalproc**

5.154 worstcase

Name: **worstcase**
searches for hard-to-round

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 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 = 0.199999988e1                f(x) = 0.738905525e1                eps = 0.4599860142344669
x = 2                f(x) = 0.738905621e1                eps = 0.144563608749673018122228379
```

See also: **round**, **searchgal**, **evaluate**

5.155 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 explicitely. 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        you"|]
```

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**, **printexpansion**, **printhexa**, **printfloat**, **printxml**, **readfile**, **autosymplify**, **display**, **midpointmode**, **fullparentheses**, **evaluate**