`plot` — graphics library

Table of contents

Notes

The library `plot` can be categorized into three parts:

1. So-called "graphical primitives", such as a point, a polygon, a two-dimensional graph of a (real) function, a graph of a two-dimensional curve, and more.
2. So-called "graphical structs", such as lines, vectors, and more. Graphical structs ease the use of more complex graphical objects. They are build from graphical primitives.
3. Functions for creating complex graphics, such as vectorfields, ode plots, graphs of implicit functions, and more.

For example, to plot the graph of the function $\sin(x)$ for $x \in [0, 2\pi]$, we enter:

```
>> f := plot::Function2d(sin(x), x = 0..2*PI)
```

```
                plot::Function2d(sin(x), x = 0..2 PI)
```

What happens here? We create a graphical primitive representing the graph of $\sin(x)$ in the specified interval and stored the result in the variable `f`. The result is an object of the domain `plot::Function2d`.

To plot the graph on the screen, use the function `plot`:

```
>> plot(f)
```

In general, to plot a graphical scene consisting of the graphical primitives `o1, o2, ...`, call `plot(o1, o2, ...)`.

One may first create a graphical scene with `scene:= plot::Scene(o1, o2, ...)`, and then call `plot(scene)`. We give an example:

```
>> scene := plot::Scene(f);
   plot(scene)
```

```
                              plot::Scene()
```

See the following help pages for further examples.

`plot::Curve2d` – graphical primitive for a two-dimensional curve

`plot::Curve2d([x, y], t = a..b)` represents a plot of the curve defined by $t \mapsto \big(x(t); y(t)\big)$ with $t \in [a, b]$.

Creating Elements:

- `plot::Curve2d([x, y], t = a..b<, option1, option2, ...>)`

Parameters:

`x, y`	—	arithmetical expressions in `t`
`t`	—	identifier
`a, b`	—	arithmetical expressions
`option1, option2, ...`	—	plot option(s) of the form `OptionName = value`

Related Domains: `plot::Curve3d`, `plot::Function2d`, `RGB`

Related Functions: `plot`, `plot2d`, `plot::copy`

Details:

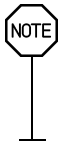
- Objects generated by `plot::Curve2d` represent graphical primitives for two-dimensional curves that can be displayed via `plot(...)`, or used with other graphical primitives of the `plot` library.
- An object of `plot::Curve2d` has the type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

`OptionName`	admissible values	default value
`Color`	`[`*`Flat`*`]`, `[`*`Flat`*`, [r,g,b]]`, `[`*`Height`*`]`, `[`*`Height`*`, [r,g,b], [R,G,B]]`, `[`*`Function`*`, f]`	`[`*`Height`*`]`
`Discont`	`TRUE`, `FALSE`	`TRUE`
`Grid`	`[n]`	`[100]`
`LineStyle`	*`SolidLines`*, *`DashedLines`*	*`SolidLines`*
`LineWidth`	positive integers	`1`
`PointStyle`	*`Circles`*, *`FilledCircles`*, *`FilledSquares`*, *`Squares`*	*`FilledSquares`*
`PointWidth`	positive integers	`30`

OptionName	admissible values	default value
RealValuesOnly	TRUE, FALSE	TRUE
Smoothness	[n]	[0]
Style	[*Points*], [*Lines*], [*LinesPoints*], [*Impulses*]	[*Lines*]
Title	strings	""
TitlePosition	[x, y]	

See `plot2d` for further details on each option, except for *`Discont`* and *`RealValuesOnly`*, which are described in detail below.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the call of `plot`, or use `plot::Scene` to create an object representing a graphical scene. Cf. example 1.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

  The following attributes are available for a curve primitive:

attribute	meaning	properties
`options`	A table of plot options of the curve primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for curve primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write

attribute	meaning	properties
`plotdata`	List of the plot data of the curve primitive in a `plot2d` conforming syntax (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read
`range`	The parameter of the curve and its range. The initial value is the parameter `t = a..b`.	read/write
`refreshPlotdata`	A boolean value which signals whether the plot data of the curve primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the curve primitive is stored in the attribute `plotdata`. The initial value is `TRUE`. Cf. example 5.	read/write
`term`	The term of the curve. Its initial value is the parameter `[x, y]`.	read/write

Example 5 illustrates how to read and write such attributes.

Option <*`Discont`* `= value`>:

- This option determines, whether the parametrization `[x(t), y(t)]` of the curve is checked for discontinuities. Admissible values are `TRUE` and `FALSE`; the default is *`Discont`* = `TRUE`.

  - *`Discont`* = `TRUE` enables symbolic checking of discontinuities. If found, unwanted graphical effects such as spurious lines at the discontinuities are eliminated.
  - *`Discont`* = `FALSE` disables the check.

  See example 3.

Option <*`RealValuesOnly`* `= value`>:

- If the parametrization `[x(t), y(t)]` of the curve produces a complex value during the evaluation of the plot, then an error occurs. Specifying

RealValuesOnly = TRUE, such errors are trapped. Only those parts of the curve producing real values are plotted.

With *RealValuesOnly* = FALSE no internal check is performed. The renderer produces an error, when it encounters a complex value. The default is *RealValuesOnly* = TRUE.

See example 4.

Operands: An object of `plot::Curve2d` consists of two operands. The first operand is the term of the curve specified as the list `[x, y]`. The second one is the parameter of the curve and its range in the form `t = a..b`.

Important Operations:

- Operands of a curve primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attributes* described above. For example, if `curve` is such an object, then the calls `op(curve,1)`, `curve[1]` and `curve::term` return the list `[x, y]`.

  Via `curve[1] := [new_x, new_y]` or `curve::term := [new_x, new_y]`, the term of a curve primitive can be changed.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `curve` is such an object, then `curve::Color := RGB::Red` changes the color of the curve primitive `curve` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Curve2d` returns itself.

Function Call: Calling an object of `plot::Curve2d` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for curve primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Curve2d` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters

`option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot2d` which is used to plot the object. See the table of plot options above, which gives a summary of the available plot options for curve primitives and their default values. See example 2.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of two-dimensional curves.

Access Methods

Method `_index`: indexed access to the operands of a curve primitive

```
_index(dom curve, positive integer i)
```

- Returns the `i`th operand of `curve`. See "Operands" above for a description of the operands of `curve`. If `i` is greater than 2, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `curve[i]`, or in functional notation `_index(curve, i)`.

Method `dimension`: dimension of a curve primitive

```
dimension(dom curve)
```

- Returns the integer 2.

Method `getPlotdata`: create the plot data of a curve primitive

```
getPlotdata(dom curve)
```

- Returns a list of an inner list, where the inner list is a plot description of `curve` in a `plot2d` conforming syntax, i.e., it has the form `[Mode = Curve, [...], ...]`.
  For example, with `s := plot::Curve2d::getPlotdata(curve)` the call `plot2d(s[1])` gives a plot of `curve`.

- Only those plot options will be included in the plot data of the curve, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` for curves is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `curve`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `curve` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a curve primitive

```
nops(dom curve)
```

- Returns the integer 2.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(curve)`.

Method `op`: extract operands of a curve primitive

```
op(dom curve, positive integer i)
```

- Returns the `i`th operand of `curve`. See "Operands" above for a description of the operands of `curve`. If `i` is greater than 2, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(curve, i)`.

Method `set_index`: set operands of a curve primitive

```
set_index(dom curve, positive integer i, any val)
```

- Replaces the `i`th operand of `curve` by the value `val`. See "Operands" above for a description of the operands of `curve`.
- If `i` is greater than 2, or if `val` is not an admissible value for the `i`th operand, then a warning message is issued. In this case the call of this method has no effect on the object `curve`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `curve` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `curve, `*`string`* `slotname)`

- Reads the value of the slot `slotname` of `curve`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `curve::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(curve, slotname)`.

`slot(`*`dom`* `curve, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `curve::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(curve, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `curve` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot option

`checkOption(`*`equation`* `OptionName = value)`

- This method checks whether `OptionName` is an available plot option for curve primitives (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.

- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a curve primitive

```
copy(dom curve)
```

- Returns a copy of the object `curve`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of a curve primitive

```
modify(dom curve, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `curve` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Curve2d`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `curve` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a curve primitive

```
print(dom curve)
```

- This method returns an unevaluated expression of the form `plot::Curve2d([x, y], t = a..b)`. It is used to print objects of `plot::Curve2d` to the screen.
- See the system function `print` for details.

Example 1. The following call returns an object representing the graph of the unit circle:

```
>> c := plot::Curve2d([sin(t), cos(t)], t = 0..2*PI)

          plot::Curve2d([sin(t), cos(t)], t = 0..2 PI)
```

To plot this curve in a graphical scene, call `plot`:

```
>> plot(c)
```

Plot options of the curve can be given as additional parameters in the creating call, such as increasing the width of the lines of a graph and plotting the graph in red color:

```
>> c2 := plot::Curve2d([sin(t), cos(t)], t = 0..2*PI,
     Color = RGB::Red, LineWidth = 50
   )

          plot::Curve2d([sin(t), cos(t)], t = 0..2 PI)

>> plot(c2)
```

To change default values of some scene options, pass the scene options to the function `plot` as additional arguments. For example, change the scaling of the plot and increase the number of ticks on both axes:

```
>> plot(c2, Scaling = Constrained, Ticks = [10,10])
```

See the help page of `plot::Scene` for available scene options.

Example 2. If a curve primitive is created, values of some plot options of the created object can be read, or replaced by new values.

To illustrate this, we create the following curve:

```
>> c1 := plot::Curve2d([t*cos(t), t*sin(t)], t = 0..4*PI)

        plot::Curve2d([t cos(t), t sin(t)], t = 0..4 PI)
```

We create a copy of this curve, change some plot options of the copied object, and plot both objects in a graphical scene:

```
>> c2 := plot::copy(c1):
   c2::Style := [Points]: c2::Grid := [30]:
   plot(c1, c2)
```

Plot options, which are explicitely set for a curve primitive, are stored under the attribute `options` and can be read with the slot operator `::`. The plot options for the first created object are:

```
>> c1::options
```

```
                table(
                  RealValuesOnly = TRUE,
                  Discont = TRUE,
                  Grid = [100],
                  Color = [Flat, [1.0, 0.0, 0.0]]
                )
```

These are the default values of some plot options for two-dimensional curve primitives, defined by the entry `"defaultOptions"` of the domain `plot::Curve2d`:

```
>> plot::Curve2d::defaultOptions

                table(
                  RealValuesOnly = TRUE,
                  Discont = TRUE,
                  Grid = [100],
                  Color = [Flat, [1.0, 0.0, 0.0]]
                )
```

When the plot data of a curve primitive is created (calling the method `"getPlotdata"`), only those plot options are used that are contained in the table `c1::options`. Here these are:

```
>> plot::Curve2d::getPlotdata(c1)

 [[Mode = Curve, [t cos(t), t sin(t)], t = [0.0, 12.56637061],

    Grid = [100], Color = [Flat, [1.0, 0.0, 0.0]]]]
```

This means that for any other available plot option not contained in the table `c1::options`, the default value is either set by `plot::Scene`, if the option also exists as a scene option, or it is internally set by the function `plot2d` when plotting the object.

You might wonder why the options *`RealValuesOnly`* and *`Discont`* are not contained in the plot structure returned by the method `"getPlotdata"`. The options are special options for objects of the domain `plot::Curve2d`. They are used to determine the plot data of such an object. They are not accepted as valid options for the mode `Curve` of the `plot2d` command.

The curve primitive `c2` contains the following options:

```
>> c2::options

                table(
                  Style = [Points],
                  RealValuesOnly = TRUE,
                  Discont = TRUE,
                  Grid = [30],
                  Color = [Flat, [1.0, 0.0, 0.0]]
                )
```

As you see, the option *Style* was added to this table, and the default value of the option *Grid* was replaced by the new value [30]. Use delete to remove plot options:

```
>> delete c2::options[Style]: c2::options

                table(
                  RealValuesOnly = TRUE,
                  Discont = TRUE,
                  Grid = [30],
                  Color = [Flat, [1.0, 0.0, 0.0]]
                )
```

Example 3. In order to illustrate the effect of the option *Discont*, we create the following curve primitive:

```
>> c := plot::Curve2d([tan(t), t], t = 0..4*PI)

             plot::Curve2d([tan(t), t], t = 0..4 PI)
```

The tangens has singularities at multiplicities of $\pi/2$, which is determined by plot::Curve2d, if the option *Discont* is set to TRUE. This is the default behaviour of plot::Curve2d:

```
>> plot(c)
```

Setting this option to FALSE causes spurious lines at the singularities:

```
>> c::Discont := FALSE: plot(c)
```

Example 4. If the option *RealValuesOnly* is disabled, complex arguments produced by the parametrization of the curve lead to runtime errors during the evaluation of the plot:

```
>> c := plot::Curve2d([sqrt(t), -sqrt(t)], t = -5..5,
     RealValuesOnly = FALSE
   ):
   plot(c)

 Error: Non-real values detected (try option RealValuesOnly = \
 TRUE) [plotlib::clip2d_Curve]
```

Example 5. This example illustrates how to read and write attributes of curve primitives (see the table of available attributes in "Details" above).

In example 2, we already used the attribute `options`, which stores plot options defined individually for a curve primitive and which overrides the corresponding default values set by MuPAD.

The attribute `term` holds the term of a curve primitive. For example, if `curve` is an object of `plot::Curve2d` such as:

```
>> c := plot::Curve2d([tan(t), t], t = 0..4*PI, Color = RGB::Blue)

              plot::Curve2d([tan(t), t], t = 0..4 PI)
```

then `c::term` returns the list:

```
>> c::term

                              [tan(t), t]
```

We plot this curve:

```
>> plot(c)
```

Because the attribute `term` has the "write" property, you can change the value of this attribute as follows:

```
>> c::term := [1/cos(t), t]: plot(c)
```

The value of `term` must be a list of two arithmetical expressions, otherwise an error message is issued.

An example of a "read-only" attribute is the attribute `plotdata`. It stores the plot data of a curve primitive in a `plot2d` conforming syntax. However, the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE`.

For example, if we create a new curve primitive, then the value of `plotdata` is the empty list:

```
>> c2 := plot::Curve2d([t^2, t^2], t = -5..5):
   c2::plotdata

                                    []
```

and `refreshPlotdata` signals that the plot data must be created:

```
>> c2::refreshPlotdata

                                  TRUE
```

A call of the method `"getPlotdata"`, which is caused by plotting the curve, for example, creates the plot data of a curve primitive:

```
>> plot::Curve2d::getPlotdata(c2)
```

```
                    2   2
 [[Mode = Curve, [t , t ], t = [-5.0, 5.0], Grid = [100],

    Color = [Flat, [1.0, 0.0, 0.0]]]]
```

This plot data is the new value of the attribute `plotdata`, and `refreshPlotdata` was set to `FALSE`. Thus, an unnecessary rebuilding of the plot data of this object can be avoided by reading the value of `plotdata`:

```
>> c2::refreshPlotdata(c2), c2::plotdata

                            2   2
 FALSE, [[Mode = Curve, [t , t ], t = [-5.0, 5.0],

    Grid = [100], Color = [Flat, [1.0, 0.0, 0.0]]]]
```

Any change of a plot option or an attribute of the curve primitive sets the value of `refreshPlotdata` to `TRUE`:

```
>> c2::Color:= RGB::Black: c2::refreshPlotdata(c2)

                                TRUE
```

This means that the value of `plotdata` is not longer valid, and the method `"getPlotdata"` must be called to rebuild the plot data of `c2`.

Changes:

- `plot::Curve2d` is a new function.

`plot::Curve3d` – graphical primitive for a three-dimensional curve

`plot::Curve3d([x, y, z], t = a..b)` represents a plot of the curve defined by $t \mapsto \big(x(t); y(t); z(t)\big)$ with $t \in [a, b]$.

Creating Elements:

- `plot::Curve3d([x, y, z], t = a..b<, option1, option2, ...>)`

Parameters:

`x, y, z`	—	arithmetical expressions in `t`
`t`	—	identifier
`a, b`	—	arithmetical expressions
`option1, option2, ...`	—	plot option(s) of the form `OptionName = value`

Related Domains: `plot::Curve2d`, `plot::Function3d`, `plot::Surface3d`, `RGB`

Related Functions: `plot`, `plot3d`, `plot::copy`

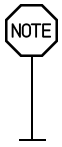
Details:

- Objects generated by `plot::Curve3d` represent graphical primitives for three-dimensional curves that can be displayed via `plot(...)`, or used with other graphical primitives of the `plot` library.
- An object of `plot::Curve3d` has the type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

`OptionName`	admissible values	default value
`Color`	`[`*`Flat`*`]`, `[`*`Flat`*`, [r,g,b]]`, `[`*`Height`*`]`, `[`*`Height`*`, [r,g,b], [R,G,B]]`, `[`*`Function`*`, f]`	`[`*`Height`*`]`
`Grid`	[integer]	`[100]`
`LineStyle`	*`SolidLines`*, *`DashedLines`*	*`SolidLines`*
`LineWidth`	positive integers	`1`
`PointStyle`	*`Circles`*, *`FilledCircles`*, *`FilledSquares`*, *`Squares`*	*`FilledSquares`*
`PointWidth`	positive integers	`30`
`Smoothness`	[integer]	`[0]`
`Style`	`[`*`Points`*`]`, `[`*`Lines`*`]`, `[`*`LinesPoints`*`]`, `[`*`Impulses`*`]`	`[`*`Lines`*`]`
`Title`	strings	`""`
`TitlePosition`	`[x, y]`	

  See `plot3d` for further details on each option.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the call of `plot`, or use `plot::Scene` to create an object representing a graphical scene. Cf. example 1.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also

has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

The following attributes are available for a curve primitive:

attribute	meaning	properties
`options`	A table of plot options of the curve primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for curve primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the curve primitive in a `plot3d` conforming syntax (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read
`range`	The parameter of the curve and its range. The initial value is the parameter `t = a..b`.	read/write
`refreshPlotdata`	A boolean value which signals whether the plot data of the curve primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the curve primitive is stored in the attribute `plotdata`. The initial value is `TRUE`. See the help page of `plot::Curve2d` for an example.	read/write

attribute	meaning	properties
`term`	The term of the curve. The initial value is the parameter `[x, y, z]`.	read/write

See example 3.

Operands: An object of `plot::Curve3d` consists of two operands. The first operand is the term of the curve specified as the list `[x, y, z]`. The second one is the parameter of the curve and its range in the form `t = a..b`.

Important Operations:

- Operands of a curve primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attributes* described above. For example, if `curve` is such an object, then the calls `op(curve,1)`, `curve[1]` and `curve::term` return the list `[x, y, z]`.

  Via `curve[1] := [new_x, new_y, new_z]` or `curve::term := [new_x, new_y, new_z]`, the term of a curve primitive can be changed.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `curve` is such an object, then `curve::Color := RGB::Red` changes the color of the curve primitive `curve` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Curve3d` returns itself.

Function Call: Calling an object of `plot::Curve3d` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for curve primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Curve3d` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters

`option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot3d` which is used to plot the object. See the table of plot options above, which gives a summary of the available plot options for curve primitives and their default values. See example 2.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of three-dimensional curves.

Access Methods

Method `_index`: indexed access to the operands of a curve primitive

`_index(`*`dom`* `curve, `*`positive integer`* `i)`

- Returns the `i`th operand of `curve`. See "Operands" above for a description of the operands of `curve`. If `i` is greater than 2, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `curve[i]`, or in functional notation `_index(curve, i)`.

Method `dimension`: dimension of a curve primitive

`dimension(`*`dom`* `curve)`

- Returns the integer 3.

Method `getPlotdata`: create the plot data of a curve primitive

`getPlotdata(`*`dom`* `curve)`

- Returns a list of an inner list, where the inner list is a plot description of `curve` in a `plot3d` conforming syntax, i.e., it has the form `[Mode = Curve, [...], ...]`.
  For example, with `s := plot::Curve3d::getPlotdata(curve)` the call `plot3d(s[1])` gives a plot of `curve`.

- Only those plot options will be included in the plot data of the curve, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot3d` for curves is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `curve`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `curve` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a curve primitive

```
nops(dom curve)
```

- Returns the integer 2.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(curve)`.

Method `op`: extract operands of a curve primitive

```
op(dom curve, positive integer i)
```

- Returns the `i`th operand of `curve`. See "Operands" above for a description of the operands of `curve`. If `i` is greater than 2, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(curve, i)`.

Method `set_index`: set operands of a curve primitive

```
set_index(dom curve, positive integer i, any val)
```

- Replaces the `i`th operand of `curve` by the value `val`. See "Operands" above for a description of the operands of `curve`.
- If `i` is greater than 2, or if `val` is not an admissible value for the `i`th operand, then a warning message is issued. In this case the call of this method has no effect on the object `curve`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `curve` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `curve, `*`string`* `slotname)`

- Reads the value of the slot `slotname` of `curve`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `curve::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(curve, slotname)`.

`slot(`*`dom`* `curve, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `curve::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(curve, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `curve` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot option

`checkOption(`*`equation`* `OptionName = value)`

- This method checks whether `OptionName` is an available plot option for curve primitives (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.

- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a curve primitive

```
copy(dom curve)
```

- Returns a copy of the object `curve`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of a curve primitive

```
modify(dom curve, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `curve` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Curve3d`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `curve` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a curve primitive

```
print(dom curve)
```

- This method returns an unevaluated expression of the form `plot::Curve3d([x, y, z], t = a..b)`. It is used to print objects of `plot::Curve3d` to the screen.
- See the system function `print` for details.

Example 1. The following call returns an object representing the graph of a spiral:

```
>> c1 := plot::Curve3d([sin(t), cos(t), t], t = 0..2*PI)

         plot::Curve3d([sin(t), cos(t), t], t = 0..2 PI)
```

To plot this curve in a graphical scene, call `plot`:

```
>> plot(c1)
```

Plot options of the curve can be given as additional parameters in the creating call, such as increasing the width of the lines of a graph and plotting the graph in green color:

```
>> c2 := plot::Curve3d([sin(t), cos(t), t], t = 0..2*PI,
     Color = RGB::Green, LineWidth = 50
   )

         plot::Curve3d([sin(t), cos(t), t], t = 0..2 PI)

>> plot(c2)
```

To change default values of some scene options, pass the scene options to the function `plot` as additional arguments. For example, change the scaling of the plot and change the number of ticks on the axes:

```
>> plot(c2, Scaling = Constrained, Ticks = [3, 3, 10])
```

See the help page of `plot::Scene` for available scene options.

Example 2. If a curve primitive is created, values of some plot options of the created object can be read, or replaced by new values.

To illustrate this, we create the following curve:

```
>> c1 := plot::Curve3d([t*sin(t), t*cos(t), t], t = 0..4*PI)

       plot::Curve3d([t sin(t), t cos(t), t], t = 0..4 PI)
```

We create a copy of this curve, change some plot options of the copied object, and plot both objects in a graphical scene:

```
>> c2 := plot::copy(c1):
   c2::Style := [Points]: c2::Grid := [30]:
   plot(c1, c2)
```

Plot options, which are explicitely set for a curve primitive, are stored under the attribute `options` and can be read with the slot operator `::`. The plot options for the first created object are:

```
>> c1::options
```

```
table(
  Grid = [100]
)
```

These are default values of some plot options of two-dimensional curve primitives, defined by the entry `"defaultOptions"` of the domain `plot::Curve3d`:

```
>> plot::Curve3d::defaultOptions
```

```
table(
  Grid = [100]
)
```

When the plot data of a curve primitive is created (calling the method `"getPlotdata"`), only those plot options are used that are contained in the table `c1::options`. Here these are:

```
>> plot::Curve3d::getPlotdata(c1)
```

```
 [[Mode = Curve, [t sin(t), t cos(t), t],

    t = [0.0, 12.56637061], Grid = [100]]]
```

This means that for any other available plot option not contained in the table `c1::options`, the default value is either set by `plot::Scene`, if the option also exists as a scene option, or internally set by the function `plot2d` when plotting the object.

The curve primitive `c2` contains the following options:

```
>> c2::options
```

```
table(
  Style = [Points],
  Grid = [30]
)
```

As you see, the option *`Style`* was added to this table, and the default value of the option *`Grid`* was replaced by the new value `[30]`. Use `delete` to remove plot options:

```
>> delete c2::options[Style]: c2::options
```

```
table(
  Grid = [30]
)
```

Example 3. This example illustrates how to read and write attributes of curve primitives (see the table of available attributes in "Details" above).

In the last example, we already used the attribute `options`, which stores plot options defined individually for a curve primitive and which overrides the corresponding default values set by MuPAD.

The attribute `term` holds the term of a curve primitive. For example, if `curve` is an object of `plot::Curve3d` such as:

```
>> c := plot::Curve3d([cos(t)*sin(t), t, t], t = -5..5, Color = RGB::Blue)

         plot::Curve3d([cos(t) sin(t), t, t], t = -5..5)
```

then `curve::term` returns the list:

```
>> c::term

                      [cos(t) sin(t), t, t]
```

We plot this curve:

```
>> plot(c)
```

Because the attribute `term` has the "write" property, you can change the value of this attribute as follows:

```
>> c::term:= [t, t, sin(t)*cos(t)]: plot(c)
```

The value of `term` must be a list of three arithmetical expressions, otherwise an error message is issued.

Changes:

- `plot::Curve3d` is a new function.

`plot::Ellipse2d` – graphical primitive for a two-dimensional ellipse

`plot::Ellipse2d(p, l1, l2)` represents a plot of a two-dimensional ellipse with center point $p = (p_x; p_y)$ and semi-axes of lengths l_1 and l_2.

Creating Elements:

- `plot::Ellipse2d(m, l1, l2<, option1, option2, ...>)`
- `plot::Ellipse2d(p, l1, l2<, option1, option2, ...>)`

Parameters:

`m`	— a list of two arithmetical expressions
`p`	— a two-dimensional point, i.e., an object of the domain `plot::Point` or `DOM_POINT`
`l1, l2`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `OptionName = value`

Related Domains: `plot::Point`, `plot::Rectangle2d`, `plot::Scene`, `RGB`

Related Functions: `plot`, `plot2d`, `plot::copy`

Details:

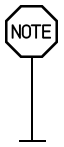
- Objects generated by `plot::Ellipse2d` represent graphical primitives for two-dimensional ellipses that can be displayed via the call `plot`, or used with other graphical primitives of the `plot` library. See example 1.
- If the first parameter of the call of `plot::Ellipse2d` is a point `p`, then it is converted into a list containing the two coordinates of `p`. Specified plot options for `p` are ignored! (Cf. example 2.)
- An object of `plot::Ellipse2d` has the type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

`OptionName`	admissible values	default value
`Color`	`[`*`Flat`*`]`, `[`*`Flat`*`, [r,g,b]]`, `[`*`Height`*`]`, `[`*`Height`*`, [r,g,b], [R,G,B]]`, `[`*`Function`*`, f]`	`[`*`Flat`*`, RGB::Red]`
`Filled`	`TRUE`, `FALSE`	`FALSE`
`Grid`	`[n]`	`[100]`
`LineStyle`	*`SolidLines`*, *`DashedLines`*	*`SolidLines`*
`LineWidth`	positive integers	`1`
`PointStyle`	*`Circles`*, *`FilledCircles`*, *`FilledSquares`*, *`Squares`*	*`FilledSquares`*
`PointWidth`	positive integers	`30`
`Smoothness`	`[n]`	`[0]`
`Style`	`[`*`Points`*`]`, `[`*`Lines`*`]`, `[`*`LinesPoints`*`]`, `[`*`Impulses`*`]`	`[`*`Lines`*`]`

OptionName	admissible values	default value
Title	strings	""
TitlePosition	[x, y]	

See plot2d for further details on each option, except of the option *Filled*, which is described in detail below.

- Scene options for the parameters option1, option2, ... are not allowed! One may pass scene options to the call of plot, or use plot::Scene to create an object representing a graphical scene. Cf. example 1.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator ::.

  Each attribute has the property "read", i.e., the value of an attribute attr of a graphical primitive o can be read with o::attr. If the attribute also has the "write" property, then the value of the attribute can be changed with o::attr := new_value.

  The following attributes are available for an ellipse primitive:

attribute	meaning	properties
center	A list of two arithmetical expressions describing the center point of the ellipse. The initial value is the parameter m, or the list of the coordinates of p, respectively.	read/write
options	A table of plot options of the ellipse primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for ellipse primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry "defaultOptions", where such options are replaced and added, respectively, which are given with the parameters option1, option2, ... of the creating call.	read/write

attribute	meaning	properties
`plotdata`	List of the plot data of the ellipse primitive in a `plot2d` conforming syntax (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read
`radius1`	The length of the first semi-axis (an arithmetical expression). The initial value is the parameter `l1`.	read/write
`radius2`	The length of the second semi-axis (an arithmetical expression). The initial value is the parameter `l2`.	read/write
`range`	A range of the form `a..b` specifying the range of the parameter of the ellipse in parametrized form. `a` and `b` must be arithmetical expressions. The initial value is `0..2*PI`. Cf. example 3.	read/write
`refreshPlotdata`	A boolean value which signals whether the plot data of the ellipse primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the ellipse is stored in the attribute `plotdata`. The initial value is `TRUE`.	read/write

See the examples 2 and 3.

Option <*Filled* = value>:

- With *Filled* = `TRUE` the ellipse is filled with the color specified with the option *Color*.

  In this case, the ellipse is approximated by a (filled) polygon. The number of the vertices of the polygon is the value `n` of the option *Grid*(see the table of options above).

Note that drawing a filled polygon with more than three vertices is quite time consuming in MuPAD!

The default is *Filled* = FALSE.

Operands: An object of `plot::Ellipse2d` consists of the three operands `m`, `l1` and `l2`.

Important Operations:

- Operands of an ellipse primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using some *attributes* described above. For example, if `ellipse` is such an object, then the calls `op(ellipse,1)`, `ellipse[1]` and `ellipse::center` return the center point of the ellipse.

  Via `ellipse[1] := new_point` or `ellipse::center := new_point`, the center point of an ellipse can be changed.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `ellipse` is such an object, then `ellipse::Color := RGB::Red` changes the color of `ellipse` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Ellipse2d` returns itself.

Function Call: Calling an object of `plot::Ellipse2d` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for ellipse primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Ellipse2d` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot2d` which is used to plot the object. See the table of plot options above, which gives a summary of the available plot options for curve primitives and their default values.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of two-dimensional ellipses.

Access Methods

Method `_index`: indexed access to the operands of an ellipse primitive

```
_index(dom ellipse, positive integer i)
```

- Returns the `i`th operand of `ellipse`. See "Operands" above for a description of the operands of `ellipse`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `ellipse[i]`, or in functional notation `_index(ellipse, i)`.

Method `dimension`: dimension of an ellipse primitive

```
dimension(dom ellipse)
```

- Returns the integer 2.

Method `getPlotdata`: create the plot data of an ellipse primitive

```
getPlotdata(dom ellipse)
```

- Returns a list of an inner list, where the inner list is a plot description of `ellipse` in a `plot2d` conforming syntax. If the option *`Filled`* of `ellipse` is set to `FALSE`, then the plot description has the form `[Mode = Curve, [...], ...]`. Otherwise, it has the form `[Mode = List, [polygon(...)], ...]`.
  For example, with `s := plot::Ellipse2d::getPlotdata(ellipse)` the call `plot2d(s[1])` gives a plot of `ellipse`.

- Only those plot options will be included in the plot data of the ellipse, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` for curves is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `ellipse`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `ellipse` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of an ellipse primitive

```
nops(dom ellipse)
```

- Returns the integer 3.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(ellipse)`.

Method `op`: extract operands of an ellipse primitive

```
op(dom ellipse, positive integer i)
```

- Returns the `i`th operand of `ellipse`. See "Operands" above for a description of the operands of `ellipse`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(ellipse, i)`.

Method `set_index`: set operands of an ellipse primitive

```
set_index(dom ellipse, positive integer i, any val)
```

- Reset the `i`th operand of `ellipse` to the value `val`. See "Operands" above for a description of the operands of `ellipse`.
- If `i` is greater than 3, or if `val` is not an admissible value for the `i`th operand, then a warning message is issued. In this case the call of this method has no effect on the object `ellipse`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `ellipse` to `TRUE`.

Method `slot`: read and write attributes and plot options

```
slot(dom ellipse, string slotname)
```

- Reads the value of the slot `slotname` of `ellipse`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `ellipse::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(ellipse, slotname)`.

```
slot(dom ellipse, string slotname, any val)
```

- Changes the value of the attribute or option with the name `slotname` to the value `val`. See the Details above for existing attributes and options of an ellipse primitive.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `ellipse::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(ellipse, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `ellipse` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot option

```
checkOption(equation OptionName = value)
```

- This method checks whether `OptionName` is an available plot option for ellipse primitives (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.

- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of an ellipse primitive

```
copy(dom ellipse)
```

- Returns a copy of the object `ellipse`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of an ellipse primitive

```
modify(dom ellipse, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `ellipse` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Ellipse2d`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `ellipse` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print an ellipse primitive

```
print(dom ellipse)
```

- This method returns an unevaluated expression of the form `plot::Ellipse2d(p, l1, l2`. It is used to print objects of `plot::Ellipse2d` to the screen.
- See the system function `print` for details.

Example 1. We create a plot of an ellipse with center point (1;2) and semi-axes of length 3 and 4:

```
>> ellipse := plot::Ellipse2d([1, 2], 3, 4)

                  plot::Ellipse2d([1, 2], 3, 4)
```

The center point can also be an object of the domain `plot::Point`. The following example creates the unit circle around the point $(-1;-1)$, filled with green color:

```
>> circle := plot::Ellipse2d(
     plot::Point(-1, -1), 1, 1, Filled = TRUE, Color = RGB::Green
   )

                 plot::Ellipse2d([-1, -1], 1, 1)
```

We plot these two objects in a graphical scene, where the scaling of the plot is changed to be constrained:

```
>> plot(ellipse, circle, Scaling = Constrained)
```

Example 2. The attribute `center`, which specifies the center point of the ellipse, is a list of two arithmetical expressions. This is also the case if the center point of the created object was given as an object of the domain `plot::Point` or `DOM_POINT`:

```
>> c := plot::Point([-1, 1]):
   ellipse := plot::Ellipse2d(c, 2, -2):
   ellipse::center

                              [-1, 1]
```

If you replace the value of the attribute `center`, then the point must be given as a list of two arithmetical expressions, otherwise a warning message is issued:

```
>> ellipse::center:= point(0, 0)

 Warning: attribute 'center': expecting a list of two arithmeti\
 cal expressions; assignment ignored [plot::Ellipse2d::slot]

                           point(0, 0)

>> ellipse::center

                              [-1, 1]
```

Note that if you specify an object of the domain `plot::Point` or `DOM_POINT` as the center point of the ellipse, then plot options of the point are ignored. For example, if we change the color of the point `c` created above to blue and create a new ellipse:

```
>> c::Color := RGB::Blue: ellipse := plot::Ellipse2d(c, 1, 1):
   plot(ellipse)
```

then the ellipse is still drawn in red color (the default color of objects of the domain `plot::Ellipse2d`). You must use the color option of the object `ellipse` to change the color of the object:

```
>> ellipse::Color := RGB::Blue: plot(ellipse)
```

Example 3. The value of the attribute `range` specifies the range of the parameter of the ellipse in parametrized form. For example, if we create a plot of the unit circle arount the point $(0;0)$:

```
>> circle := plot::Ellipse2d([0, 0], 1, 1)

                  plot::Ellipse2d([0, 0], 1, 1)
```

and restrict the parameter of the circle to the interval $[0, \pi]$, we get the following plot:

```
>> circle::range := 0..PI: plot(circle)
```

See the help page of `plot::Curve2d` for more examples for working with attributes of graphical primitives.

Changes:

- `plot::Ellipse2d` is a new function.

`plot::Function2d` – graphical primitive for a two-dimensional graph of a function

`plot::Function2d(f, x = a..b)` represents a plot of the function $f(x)$ with $x \in [a, b]$.

Creating Elements:

- `plot::Function2d(f, x = a..b<, option1, option2, ...>)`
- `plot::Function2d(f, x = a..b, y = ymin..ymax<, option1, option2, ...>)`

Parameters:

`f`	— arithmetical expression in `x`
`x, y`	— identifiers
`a, b, ymin, ymax`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `OptionName = value`

Related Domains: `plot::Curve2d`, `plot::Function3d`, `RGB`

Related Functions: `plot`, `plot2d`, `plotfunc2d`, `plot::copy`

Details:

- Objects generated by `plot::Function2d` represent graphical primitives for two-dimensional graphs of functions which can be displayed via `plot(...)`, or used with other graphical primitives of the `plot` library.

- `plot::Function2d(f, x = a..b, y = ymin..ymax, ...)` clips the graph of `f` to the rectangle with lower left corner $(a; y_{min})$ and upper right corner (b, y_{max}).

  The clipping of the graph to the given rectangle is implemented by `plot::Function2d` as follows: It tries to determine subintervals of $[a, b]$, where $f(x)$ lies in the interval $[y_{min}, y_{max}]$. This process does not work in general, and thus can produce graph plots outside the specified rectangle. It also depends on the value of the option *`Grid`*, and increasing the value of calculated grid points can be necessary. This is the case, for example, if `f` strongly oscillates in the interval $[a, b]$. See example 3.

- `plot::Function2d` automatically attempts to determine the locations of discontinuities before plotting. If `f` has discontinuities that can be determined, then the result of `plot::Function2d` consists of $n + 1$ subgraphs, where n is the number of discontinuities of `f`.

  Use the option *`Discont`* to disable the determination of discontinuities.

- Note that `plotfunc2d` is also used to plot two-dimensional graphs of functions. But in contrast to `plot::Function2d`, it does not return the graph in form of a graphical object but displays the graph immediatly after executing the command.

  The main advantage of using `plot::Function2d` is, that you get the representation of the graph as an object. It can be manipulated afterwards or combined easily with other graphical primitives, such as polygons, curves or lists of points, to a common graphical scene.

  See example 2 below.

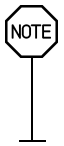
- An object of `plot::Function2d` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.

- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

OptionName	admissible values	default value
Color	[*Flat*], [*Flat*, [r,g,b]], [*Height*], [*Height*, [r,g,b], [R,G,B]], [*Function*, f]	[*Flat*,[1,0,0]]
Discont	TRUE, FALSE	TRUE
Grid	[n]	[100]
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
RealValuesOnly	TRUE, FALSE	TRUE
Smoothness	[n]	[0]
Style	[*Points*], [*Lines*], [*LinesPoints*], [*Impulses*]	[*Lines*]
Title	strings	""
TitlePosition	[x, y]	

  See `plot2d` for further details on each option, except for *Discont* and *RealValuesOnly*, which are described in detail below.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the function `plot`, or use `plot::Scene` to create an object representing a graphical scene. Cf. example 1.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`. See example 5.

  The following attributes are available for a function primitive:

attribute	meaning	properties
`objects`	If `f` has discontinuities and if the value of the option *`Discont`* is `TRUE`, then the graph of the function is splitted into a left and a right part around each determined discontinuity. The attribute `objects` is then the list of these parts, where each part is an object of `plot::Function2d`. If `f` has no discontinuities, or if the value of the option *`Discont`* is `FALSE`, then this list only consists of one entry, the (whole) graph. Note that if you extract an object of this list and do some changes to this object, then you must set the value of the attribute `refreshPlotdata` to `TRUE` in order to force a rebuild of the plot data of the graph.	read
`options`	A table of plot options of the function primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for function primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the function primitive in a `plot2d` conforming syntax (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read
`range`	The variable of the function and its range in the form `ident = a..b`. The initial value is the parameter `x = a..b`.	read/write
`refreshPlotdata`	A boolean value which signals whether the plot data of the function primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the function primitive is stored in the attribute `plotdata`. The initial value is `TRUE`. See the help page of `plot::Curve2d` for an example.	read/write

attribute	meaning	properties
`term`	The term of the function (an arithmetical expression). The initial value is the parameter `f`.	read/write
`yrange`	The "y-range" of the graph (see above). It is either of the form `ymin..ymax`, or the value `Automatic`. The initial value is the parameter `ymin..ymax`, if the parameter was given in the function call, or the value `Automatic` otherwise.	read/write

---

Option <*Discont* = value>:

- This option determines, whether the function `f(x)` is checked for discontinuities. Admissible values are TRUE and FALSE; the default is *Discont* = TRUE.

  - *Discont* = TRUE enables symbolic checking of discontinuities of `f`. If found, unwanted graphical effects such as spurious lines at the discontinuities are eliminated.
  - *Discont* = FALSE disables the check.

  See example 4.

Option <*RealValuesOnly* = value>:

- If the function `f(x)` produces a complex value during the evaluation of the plot, then an error occurs. Specifying *RealValuesOnly* = TRUE, such errors are trapped. Only those parts of the function producing real values are plotted. E.g., with this option the function `sqrt(x)` can be plotted over the interval $x \in [-1, 1]$: the plot only displays the real function values for $x \geq 0$.

  With *RealValuesOnly* = FALSE no internal check is performed. The renderer produces an error, when it encounters a complex value. The default is *RealValuesOnly* = TRUE.

Operands: An object of `plot::Function2d` consists of three operands. The first operand is the term `f` of the function. The second one is the variable `x` of the term and its range in the form `x = a..b`. The third one is the value `Automatic`, or the range `ymin..ymin` if the parameter `y = ymin..ymax` was given.

Important Operations:

- Operands of a function primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attributes* described above. For example, if `function` is such an object, then the calls `op(function,1)`, `function[1]` and `function::term` return the term `f`.

  Via `function[1] := f` or `function::term := f`, the term of a function primitive can be changed.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `function` is such an object, then `function::Color := RGB::Red` changes the color of the function primitive `function` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Function2d` returns itself.

Function Call: Calling an object of `plot::Function2d` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for function primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Function2d` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot2d` which is used to plot the object. See the table of plot options above, which gives a summary of the available plot options for function primitives and their default values. See example 4.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of two-dimensional graphs of functions.

Access Methods

Method `_index`: indexed access to the operands of a function primitive

`_index(`*`dom`* `function, `*`positive integer`* `i)`

- Returns the `i`th operand of `function`.See "Operands" above for a description of the operands of `function`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `function[i]`, or in functional notation `_index(function, i)`.

Method `dimension`: dimension of a function primitive

`dimension(`*`dom`* `function)`

- Returns the integer 2.

Method `getPlotdata`: the plot data of a function primitive

`getPlotdata(`*`dom`* `function)`

- Returns a list of inner lists, where each inner list is a plot description of a two-dimensional curve in a `plot2d` conforming syntax, i.e., an inner list has the form `[Mode = Curve, [...], ...]`.
  For example, with `s := plot::Function2d::getPlotdata(function)` the call `plot2d(op(s))` gives a plot of `function`.
- Only those plot options will be included in the plot data of the function, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` for curves is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `function`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `function` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a function primitive

`nops(`*`dom`* `function)`

- Returns the integer 3.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(function)`.

Method `op`: extract operands of a function primitive

`op(`*`dom`* `function,` *`positive integer`* `i)`

- Returns the `i`th operand of `function`. See "Operands" above for a description of the operands of `function`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(function, i)`.

Method `set_index`: set operands of a function primitive

`set_index(`*`dom`* `function,` *`positive integer`* `i,` *`any`* `val)`

- Replaces the `i`th operand of `function` by the value `val`. See "Operands" above for a description of the operands of `function`.
- If `i` is greater than 3, or if `val` is not an admissible value for the `i`th operand, then a warning message is issued. In this case the call of this method has no effect on the object `function`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `function` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `function,` *`string`* `slotname)`

- Reads the value of the slot `slotname` of `function`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `function::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(function, slotname)`.

```
slot(dom function, string slotname, any val)
```

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `function::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(function, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `function` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot options

```
checkOption(equation OptionName = value)
```

- This method checks whether `OptionName` is a known plot option for function primitives (see the table of available plot options above), and `value` is an admissible for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a function plot

```
copy(dom function)
```

- Returns a copy of the object `function`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of a function plot

```
modify(dom function, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `function` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Function2d`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `function` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a function primitive

```
print(dom function)
```

- This method returns an unevaluated expression of the form `plot::Function2d(f, x = a..b)`. It is used to print objects of `plot::Function2d` to the screen.
- See the system function `print` for details.

Example 1. The following call returns an object representing the graph of the sine function in the interval $[0, 2\pi]$:

```
>> f := plot::Function2d(sin(x), x = 0..2*PI)

              plot::Function2d(sin(x), x = 0..2 PI)
```

To plot the graph in a graphical scene, call `plot`:

```
>> plot(f)
```

To restrict the y-range of the graph, for example, to the interval $[0, 1]$, specify a second range in the call of `plot::Function2d`:

```
>> f2 := plot::Function2d(sin(x), x = 0..2*PI, y = 0..1):
   plot(f)
```

The variable of the second range can be any identifier, that, of course, differs from the variable of the function term (here: x).

Plot options of the curve can be given as additional parameters in the creating call, such as plotting the graph in green color and changing its style so that it is drawn as impulses:

```
>> f3 := plot::Function2d(sin(x), x = 0..2*PI,
     Color = RGB::Green, Style = [Impulses]
   )

              plot::Function2d(sin(x), x = 0..2 PI)

>> plot(f3)
```

To change default values of some scene options, pass the scene options to the call of `plot` as additional arguments. For example, to draw grid lines in the background of the plot, call:

```
>> plot(f, GridLines = Automatic)
```

See the help page of `plot::Scene` for available scene options.

Example 2. We want to plot a graph of the sequence $n \mapsto \frac{\sin(n)}{n}$ in the interval $[1, 50]$, enclosed by the graphs of the functions $x \mapsto \frac{1}{x}$ and $x \mapsto -\frac{1}{x}$. We start by creating the three different graphical primitives as follows:

```
>> f1 := plot::Function2d(1/x, x = 1..50);
   f2 := plot::Function2d(-1/x, x = 1..50);
   a  := plot::Pointlist([n, sin(n)/n] $ n = 1..50, Color = RGB::Blue)

                                / 1            \
                  plot::Function2d| -, x = 1..50 |
                                \ x            /

                               /   1            \
                  plot::Function2d| - -, x = 1..50 |
                               \   x            /

                          plot::Pointlist()
```

To plot the scene of these objects, pass them as parameters to the function `plot`:

```
>> plot(f1, f2, a)
```

Example 3. The process of the clipping the graph of a function plot can fail and produce plots outside the specified rectangle. This is the case, for example, if the function strongly oscillates in the given interval.

Consider the function $\sin(e^x)$ for $x \in [-5, 5]$. We are interested only in the positive part of the graph, and thus enter:

```
>> f := plot::Function2d(sin(exp(x)), x = -5..5, y = 0..1):
   plot(f)
```

The clipping of the graph fails in this example (see the note in the "Details" above). It can be helpful in such cases to increase the value of the option *Grid*:

```
>> f::Grid:= [500]: plot(f)
```

Example 4. If a function primitive is created, values of some plot options of the created object can be read, or replaced by new values.

To illustrate this, we create the following function:

```
>> f1 := plot::Function2d(1/x^2, x = -5..5, Color = RGB::Blue)
```

$$\mathrm{plot::Function2d}\left(\frac{1}{x^2},\ x = -5..5\right)$$

We create a copy of the graph, change some plot options of the copied object, and plot both objects in a graphical scene:

```
>> f2 := plot::copy(f1):
   f2::Style := [Impulses]: f2::Grid := [20]:
   f2::Color := RGB::Red:
   plot(f1, f2)
```

Plot options, which are explicitely set for a function primitive, are stored under the attribute `options` and can be read with the slot operator `::`. The plot options for the first created object are:

```
>> f1::options
```

```
table(
  RealValuesOnly = TRUE,
  Discont = TRUE,
  Grid = [100],
  Color = [Flat, [0.0, 0.0, 1.0]]
)
```

These are default values of some plot options of two-dimensional function primitives, defined by the entry `"defaultOptions"` of the domain `plot::Function2d`:

```
>> plot::Function2d::defaultOptions
```

```
table(
  RealValuesOnly = TRUE,
  Discont = TRUE,
  Grid = [100],
  Color = [Flat, [1.0, 0.0, 0.0]]
)
```

When the plot data of a function primitive is created (calling the method `"getPlotdata"`), only those plot options are used that are contained in the table stored under the attribute `options` (here this is the table `f1::options`):

```
>> plot::Function2d::getPlotdata(f1)

 -- --                --    1  --
|  |  Mode = Curve, |  x, --  |, x = [-5.0, -0.1413648217],
|  |                |      2  |
 -- --                --    x  --

                                                        --
    Grid = [100], Color = [Flat, [0.0, 0.0, 1.0]]  |,
                                                        |
                                                        --

    --                --    1  --
    |  Mode = Curve, |  x, --  |, x = [0.1413648217, 5.0],
    |                |      2  |
    --                --    x  --

                                                                -
-
    Title = "", Grid = [100], Color = [Flat, [0.0, 0.0, 1.0]]  |
                                                                |
                                                                -
-

    --
     |
     |
    --
```

This means that for any other available plot option not contained in the table `f1::options`, the default value is either set by `plot::Scene`, if the option also exists as a scene option, or it is internally set by the function `plot2d` when plotting the object.

This example also illustrates that `plot::Function2d` automatically determines discontinuities and splits the graph into two subgraphs around each discontinuity (these subgraphs are stored in the attribute `objects`, see above).

Here, the graph of $\frac{1}{x^2}$ was splitted around 0 into two subgraphs. Thus, the plot structure in a `plot2d` conforming syntax consists of two objects of the mode `Curve`.

The determination of discontinuities can be controlled with the option *`Discont`*. If we set this option to `FALSE`, discontinuities are not determined. The plot structure of the function primitive then only consists of one object:

```
>> f1::Discont:= FALSE: plot::Function2d::getPlotdata(f1)
```

```
-- --                --    1  --
|  |  Mode = Curve, |  x, --  |, x = [-5.0, 5.0],
|  |                |      2  |
-- --                --    x  --

                                                        -- --
   Grid = [100], Color = [Flat, [0.0, 0.0, 1.0]]         |  |
                                                         |  |
                                                        -- --
```

Computation problems during the evaluation of the plot is the consequence here:

```
>> plot(f1)
```

The function primitive `f2` contains the following options:

```
>> f2::options
```

```
table(
  Style = [Impulses],
  RealValuesOnly = TRUE,
  Discont = TRUE,
  Grid = [20],
  Color = [Flat, [1.0, 0.0, 0.0]]
)
```

As you see, the option *`Style`* was added to this table, and the default value of the option *`Grid`* was replaced by the new value `[20]`. Use `delete` to remove plot options set for a curve primitive:

```
>> delete f2::options[Style]: f2::options
```

```
table(
  RealValuesOnly = TRUE,
  Discont = TRUE,
  Grid = [20],
  Color = [Flat, [1.0, 0.0, 0.0]]
)
```

You might wonder why the options *`RealValuesOnly`* and *`Discont`* are not contained in the plot structure returned by the method `"getPlotdata"`. They are special options for objects of the domain `plot::Function2d`, used to determine the plot data of such an object. They are not accepted as valid options for curves plotted directly with `plot2d`.

Example 5. This example illustrates how to read and write attributes of function primitives (see the table of available attributes in "Details" above).

In the last example, we already used the attribute `options`, which stores plot options defined individually for a function primitive to override the corresponding default values set by MuPAD.

The attribute `yrange` holds the "y-range" of the function graph. Its value is either a range of the form `y = ymin..ymax`, or the identifier `Automatic`.

For example, if `function` is an object of `plot::Function2d` such as:

```
>> f := plot::Function2d(ln(x), x = 0..10)
```

```
                plot::Function2d(ln(x), x = 0..10)
```

then `f::yrange` returns the default value of this attribute:

```
>> f::yrange
```

```
                                  Automatic
```

We plot this curve:

```
>> plot(f)
```

Because the attribute `yrange` has the "write" property, you can change the value of this attribute as follows:

```
>> f::yrange := 0..1: plot(f)
```

Changes:

- `plot::Function2d` is a new function.

`plot::Function3d` – graphical primitive for a three-dimensional graph of a function

`plot::Function3d(f, x = a..b, y = c..d)` represents a plot of the function $f(x, y)$ with $(x, y) \in [a, b] \times [c, d]$.

Creating Elements:

- `plot::Function3d(f, x = a..b, y = c..d<, option1, option2, ...>)`

Parameters:

`f`	— arithmetical expression in `x` and `y`
`x, y`	— identifiers
`a, b, c, d`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `OptionName = value`

Related Domains: `plot::Curve2d`, `plot::Function2d`, `plot::Surface3d`, `RGB`

Related Functions: `plot`, `plot3d`, `plotfunc3d`, `plot::copy`

Details:

- Objects generated by `plot::Function3d` represent graphical primitives for two-dimensional graphs of functions that can be displayed via `plot(...)`, or used with other graphical primitives of the `plot` library.
- Note that `plotfunc3d` is also used to plot three-dimensional graphs of functions. But in contrast to `plot::Function3d`, it does not return the graph in form of a graphical object but displays the graph immediatly after executing the command.

  The main advantage of using `plot::Function3d` is, that you get the representation of the graph as an object. It can be manipulated afterwards or combined easily with other graphical primitives, such as polygons, curves or surfaces, to a common graphical scene.

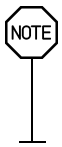
  See example 2 below.
- An object of `plot::Function3d` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

OptionName	admissible values	default value
Color	[*Flat*],[*Flat*, [r,g,b]], [*Height*], [*Height*, [r,g,b], [R,G,B]], [*Function*, f]	[*Height*]
Grid	[integer]	[20,20]
LineStyle	*SolidLines*,*DashedLines*	*SolidLines*
LineWidth	positive integers	1
PointStyle	*Circles*,*FilledCircles*, *FilledSquares*,*Squares*	*FilledSquares*
PointWidth	positive integers	30
Smoothness	[integer]	[0]
Style	[*Points*] [*WireFrame*, *Mesh*] [*WireFrame*, *ULine*] [*WireFrame*, *VLine*] [*HiddenLine*, *Mesh*] [*HiddenLine*, *ULine*] [*HiddenLine*, *VLine*]	[*ColorPatches*, *AndMesh*]

OptionName	admissible values	default value
	[*ColorPatches*, *Only*] [*ColorPatches*, *AndMesh*] [*ColorPatches*, *AndU-Line*] [*ColorPatches*, *AndV-Line*] [*Transparent*, *Only*] [*Transparent*, *AndMesh*] [*Transparent*, *AndULine*] [*Transparent*, *AndVLine*]	
Title	strings	""
TitlePosition	[x, y]	

See `plot3d` for further details on each option.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the function `plot`, or use `plot::Scene` to create an object representing a graphical scene. Cf. example 1.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`. See example 4.

  The following attributes are available for a function primitive:

attribute	meaning	properties
`options`	A table of plot options of the function primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for function primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the function primitive in a `plot3d` conforming syntax (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read
`range1`	The first variable of the function and its range in the form `ident1 = a..b`. The initial value is the parameter `x = a..b`.	read/write
`range2`	The second variable of the function and its range in the form `ident2 = c..d`. The initial value is the parameter `y = c..d`.	read/write

attribute	meaning	properties
`refreshPlotdata`	A boolean value which signals whether the plot data of the function primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the function primitive is stored in the attribute `plotdata`. The initial value is `TRUE`. See the help page of `plot::Curve2d` for an example.	read/write
`term`	The term of the function. The initial value is the parameter `f`.	read/write

Operands: An object of `plot::Function3d` consists of three operands. The first operand is the term `f` of the graph. The second operand is the first variable of the function and its range in the form `x = a..b`, and the third one is the second variable of `f` and its range in the form `y = c..d`.

Important Operations:

- Operands of a function primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attributes* described above. For example, if `function` is such an object, then the calls `op(function,1)`, `function[1]` and `function::term` return the expression `f`.

  Via `function[1] := g` or `function::term := g`, the term of a function plot can be changed to the value `g`.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `function` is such an object, then `function::Color := RGB::Red` changes the color of the function primitive `function` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Function3d` returns itself.

Function Call: Calling an object of `plot::Function3d` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for function primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Function3d` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot3d` which is used to plot the object. See the table of plot options above, which gives a summary of the available plot options for function primitives and their default values. See example 3.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of three-dimensional graphs of functions.

Access Methods

Method `_index`: indexed access to the operands of a function primitive

`_index(`*`dom`* `function, `*`positive integer`* `i)`

- Returns the `i`th operand of `function`. See "Operands" above for a description of the operands of `function`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `function[i]`, or in functional notation `_index(function, i)`.

Method `dimension`: dimension of a function primitive

`dimension(`*`dom`* `function)`

- Returns the integer 3.

Method `getPlotdata`: the plot data of a function primitive

```
getPlotdata(dom function)
```

- Returns a list of an inner list, where the inner list is a plot description of `function` in a `plot3d` conforming syntax, i.e., it has the form `[Mode = Surface, [...], ...]`.
  For example, with `s := plot::Function3d::getPlotdata(function)` the call `plot3d(s[1])` gives a plot of `function`.
- Only those plot options will be included in the plot data of the function, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot3d` for surfaces is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `function`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `function` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a function primitive

```
nops(dom function)
```

- Returns the integer 3.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(function)`.

Method `op`: extract operands of a function primitive

```
op(dom function, positive integer i)
```

- Returns the `i`th operand of `function`. See "Operands" above for a description of the operands of `function`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(function, i)`.

Method `set_index`: set operands of a function primitive

```
set_index(dom function, positive integer i, any val)
```

- Replaces the `i`th operand of `function` by the value `val`. See "Operands" above for a description of the operands of `function`.

- If `i` is greater than 3, or if `val` is not an admissible value for the `i`th operand, then a warning message is issued. In this case the call of this method has no effect on the object `function`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `function` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `function, `*`string`* `slotname)`

- Reads the value of the slot `slotname` of `function`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.
  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `function::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(function, slotname)`.

`slot(`*`dom`* `function, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `function::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(function, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `function` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot option

`checkOption(`*`equation`* `OptionName = value)`

- This method checks whether `OptionName` is a known plot option for function primitives (see the table of available plot options above), and `value` is an admissible value for this option.

- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a function primitive

`copy(`*`dom`* `function)`

- Returns a copy of the object `function`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of a function plot

`modify(`*`dom`* `function,` *`equation(s)`* `Name1 = value1, ...)`

- Creates a copy of the object `function` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Function3d`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `function` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a function primitive

`print(`*`dom`* `function)`

- This method returns an unevaluated expression of the form `plot::Function3d(f, x = a..b, y = c..d)`. It is used to print objects of `plot::Function3d` to the screen.
- See the system function `print` for details.

Example 1. The following call returns an object representing the graph of the function $(x, y) \mapsto \sin(xy)$ for $x \in [0, 2\pi]$ and $y \in [-\pi, \pi]$:

```
>> f1 := plot::Function3d(sin(x*y), x = 0..2*PI, y = -PI..PI)

      plot::Function3d(sin(x y), x = 0..2 PI, y = -PI..PI)
```

To plot this function in a graphical scene, call:

```
>> plot(f1)
```

Plot options of the surface can be given as additional parameters in the creating call, such as plotting the graph in blue color:

```
>> f2 := plot::Function3d(sin(x*y), x = 0..2*PI, y = -PI..PI,
     Color = [Flat, RGB::Blue])

      plot::Function3d(sin(x y), x = 0..2 PI, y = -PI..PI)

>> plot(f2)
```

To change default values of some scene options, pass the scene options to the call of `plot` as additional arguments. For example, to change the style of the axes:

```
>> plot(f1, Axes = Corner)
```

See the help page of `plot::Scene` for available scene options.

Example 2. We want to display a graph of the function $(x, y) \mapsto \sin(x)$ and the curve defined by $t \mapsto (t, \sin(t), \sin(t))$. We start by creating the two different graphical primitives as follows:

```
>> f := plot::Function3d(sin(x), x = -PI..PI, y = -5..5);
   c := plot::Curve3d(
     [t, sin(t), sin(t)], t = -PI..PI,
     Color = RGB::Blue, LineWidth = 20
   )

        plot::Function3d(sin(x), x = -PI..PI, y = -5..5)

         plot::Curve3d([t, sin(t), sin(t)], t = -PI..PI)
```

To plot the scene of these objects, pass them as parameters to the function `plot`:

```
>> plot(f, c)
```

Example 3. If a function primitive is created, values of some plot options of the created object can be read, or replaced by new values.

To illustrate this, we create the following function:

```
>> f := plot::Function3d(sin(x)*cos(y), x = 0..2*PI, y = 0..2*PI)

    plot::Function3d(cos(y) sin(x), x = 0..2 PI, y = 0..2 PI)
```

and plot the graph:

```
>> plot(f)
```

The graph is drawn in color patches, together with the parameter lines of the two parameters of the surface. To change the style of the graph, use the option *Style*. For example, to display the graph as an opaque object together with the parameter lines, call:

```
>> f::Style:= [HiddenLine, Mesh]: plot(f)
```

and plot the graph:

```
>> plot(f)
```

The options that are set for an object are stored under the attribute `options`. You can read the value of this attribute as follows:

```
>> f::options

                  table(
                    Style = [HiddenLine, Mesh],
                    Grid = [20, 20]
                  )
```

If an option is not contained in this table, then its value is set either by `plot::Scene`, if the option also exists as a scene option, or internally set by the function `plot3d` when plotting the surface.

For example, the option *Color* is not contained in this table. If you try to read its value, you get the following result:

```
>> f::Color

                                FAIL
```

The default value of this option is the list [*Height*], set by the function `plot3d` (see the table of options above). The default colors are taken from the preferences of the MuPAD's graphic tool VCam.

To override the default value of this option for the object `f`, enter:

```
>> f::Color:= [Flat, RGB::Red]: plot(f)
```

If we now take a look at the table stored under the attribute `options`, we get:

```
>> f::options
```

```
               table(
                 Color = [Flat, [1.0, 0.0, 0.0]],
                 Style = [HiddenLine, Mesh],
                 Grid = [20, 20]
               )
```

To delete some plot options set for a graphical primitive, call:

```
>> delete f::options[Style]: f::options
```

```
               table(
                 Color = [Flat, [1.0, 0.0, 0.0]],
                 Grid = [20, 20]
               )
```

If we now plot the object `f`, the default value of the option *`Style`* is used, which is the list `[ColorPatches, AndMesh]`:

```
>> plot(f)
```

Example 4. This example illustrates how to read and write attributes of function primitives (see the table of available attributes in "Details" above).

In the previous example we already introduced the attribute `options`. The attributes `range1` and `range2`, for example, hold the ranges for the variables of the function graph. We give an example:

```
>> f := plot::Function3d(x^2 + y^2, x = -5..5,y = -5..5)
```

```
                            2    2
         plot::Function3d(x  + y , x = -5..5, y = -5..5)
```

We can read the ranges of the variables:

```
>> f::range1, f::range2
```

```
                      x = -5..5, y = -5..5
```

Because these attributes have the "write" property, we can also change their values as follows:

```
>> f::range1:= x = -10..10: f::range2:= y = -10..10:
   plot(f)
```

Changes:

- `plot::Function3d` is a new function.

`plot::Group` – a group of graphical primitives

`plot::Group(object1, object2, ...)` groups the graphical primitives `object1, object2, ...` into a single graphical primitives.

Creating Elements:

- `plot::Group(object1<, object2, ...><, option1, option2, ...>)`

Parameters:

`object1, object2`	—	either two- or three-dimensional graphical primitives, i.e., objects of type `"graphprim"` of the same dimension
`option1, option2, ...`	—	plot option(s) of the form `OptionName = value`

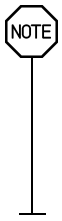
Related Domains: `plot::Scene`, `RGB`

Related Functions: `plot`, `plot2d`, `plot3d`, `plot::copy`

Details:

- Objects generated by `plot::Group` represent groups of graphical primitives that can be displayed via the call `plot(...)`, or used with other graphical primitives of the `plot` library.
- The given graphical primitives must either be two-dimensional or three-dimensional, otherwise an error message is issued.
- Note that the call `plot::Group(object1)` returns an object of the domain `plot::Group`, i.e., a group can consists of only one graphical primitive.
- An object of `plot::Group` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- The plot options `option1, option2, ...` must be valid plot options for two- and three-dimensional graphical primitives, respectively. Available options and their default values can be found on the corresponding help pages for the objects `object1, object2, ...`.

The plot options `option1, option2, ...` are passed to each graphical primitive `object_i`. This means, a plot option should be a valid plot option for *every* given graphical primitive `object_i`! If an invalid option is given for some primitives of the group, then a warning message is issued, and the setting of this option has no effect on the corresponding graphical primitives.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

  The following attributes are available for a group of graphical objects:

attribute	values	properties
`dimension`	The dimension of the group, that is the dimension of the graphical primitives `object1, object2, ...` where the dimension of `object1, object2, ...` must be equal. This attribute exists in order to have an efficient access to the dimension of a group primitive. Because not every graphical primitive has this attribute, one should call the method `"dimension"` (see below) instead in order to determine the dimension of graphical primitives.	read
`objects`	A list of the graphical primitives of the group. A graphical primitive is an object of type `"graphprim"`. The initial value is the list `[object1, object2, ...]` of the parameters `object1, object2, ...`	read/write

attribute	values	properties
`plotdata`	List of the plot data of the group primitive in a `plot2d` and `plot3d` conforming syntax, respectively. The value of this attribute can be read if the method `"getPlotdata"` (see below) was called before. The initial value is the empty list `[]`.	read

See the examples of the help page of `plot::Scene` about working with attributes.

Operands: The operands of an object of `plot::Group` are the parameters `object1, object2, ...` (in that order).

Important Operations:

- Operands of a group primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attribute* `objects` described above. For example, if `group` is such an object, then the calls `op(group,1)`, `group[1]` and `group::objects[1]` return the first graphical object `object1` of the group.

  Via `group[i] := g` or `group::objects[i] := g`, the *i*th object of the group is replaced by the graphical primitive `g` (which must be an object of type `"graphprim"`).

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of the grouped graphical primitives. For example, if `group` is such an object, then `group::Color := RGB::Red` changes the color of *each* graphical primitive of `group` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Group` returns itself.

Function Call: Calling an object of `plot::Group` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Access Methods

Method `_index`: indexed access to the operands of a group primitive

`_index(`*`dom`* `group, `*`positive integer`* `i)`

- Returns the `i`th graphical primitive of `group`. If `i` is greater than the number of graphical primitive of `group`, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `function[i]`, or in functional notation `_index(group, i)`.

Method `dimension`: dimension of a group primitive

`dimension(`*`dom`* `group)`

- Returns the value of the attribute `dimension`, i.e., the integer 2 or 3 (see the table of attributes above).

Method `getPlotdata`: create the plot data of a group primitive

`getPlotdata(`*`dom`* `group)`

- Returns a list of inner lists, where each inner list is a plot description of a graphical primitive of `group` in a `plot2d` and `plot3d` conforming syntax, respectively, i.e., an inner list has the form `[Mode = ..., ...]`.
  For example, with `s := plot::Group::getPlotdata(group)` the call `plot2d(op(s))` and `plot3d(op(s))`, respectively, gives a plot of `group`.
- The result is stored as the value of the attribute `plotdata` of `group`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a group primitive

`nops(`*`dom`* `group)`

- Returns the number of graphical primitives of `group`.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(group)`.

Method `op`: extract operands of a group primitive

`op(`*`dom`* `group,` *`positive integer`* `i)`

- Returns the `i`th graphical primitive of `group`. If `i` is greater than the number of graphical primitives of `group`, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(group, i)`.

Method `set_index`: set operands of a group primitive

```
set_index(dom group, positive integer i, graphprim object)
```

- Replaces the `i`th graphical primitive of `group` by the graphical primitive `object`.
- If `i` is greater than the number of graphical primitives of `group`, or if `object` is not of type `"graphprim"`, then a warning message is issued. In this case the call of this method has no effect on the object `group`.

Method `slot`: read attributes, and write attributes and plot options

```
slot(dom group, string slotname)
```

- Reads the value of the slot `slotname` of `group`. `slotname` must be the name of an attribute of a group primitive (see the table of attributes above), otherwise an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `group::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(group, slotname)`.

```
slot(dom group, string slotname, any val)
```

- If `slotname` is an attribute of a group primitive, then this method changes the value of this attribute to `val` (see the table above for the attributes of a group primitive).

  Otherwise the slot `slotname` of each primitive of the group is changed to the value `val`. If `slotname` is an invalid attribute or plot option of some primitives of the group, then a warning message is issued that the change is ignored for the corresponding primitive.
- This method overloads the system function `slot`, i.e., one may use it in the form `group::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(group, slotname, val)`.

Technical Methods

Method `copy`: create a copy of a group primitive

```
copy(dom group)
```

- Returns a copy of the object `group`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `expose`: expose the definition of a group

```
expose(dom group)
```

- This method returns a sequence of the graphical primitives of `group`.
- This method overloads the system function `expose`, i.e., one may use it in the form `expose(group)`.

Method `modify`: modify a copy of a group primitive

```
modify(dom group, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `group` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes of a group primitive or plot options of graphical primitives of `group`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available attributes above.
- This method is called from the function `plot::modify`.

Method `print`: printing a group primitive

```
print(dom group)
```

- This method returns an unevaluated expression of the form `plot::Group()`. It is used to print objects of `plot::Group` to the screen.
- Call `expose(group)` to expose the graphical primitives of the group.
- See the system function `print` for details.

Example 1. We create a group consisting of a graph of function and two vertical dashed lines from some points on the x-axis to the corresponding points on the graph:

```
>> g := plot::Group(
     plot::Function2d(4 - x^2, x = -2..2),
     plot::line([-1, 0], [-1, 3], LineStyle = DashedLines),
     plot::line([1, 0], [1, 3], LineStyle = DashedLines)
   )
```

```
plot::Group()
```

To plot the object in a graphical scene, enter:

```
>> plot(g)
```

Plot options can either be specified as additional arguments to the call `plot::Group(...)`, or set via the slot operator `::` as follows:

```
>> g::Color := RGB::Blue: plot(g)
```

If a plot option is invalid for some primitives of the group, then a warning message is issued. For example, the plot option *`Grid`* only exists for graphs of functions but not for polygons (the two lines here), and thus specifying this option only has an effect on the first primitive of the group `g`:

```
>> g::Grid := [200]:

 Warning: unknown option name 'Grid'; assignment ignored [plot:\
 :Polygon::slot]
 Warning: unknown option name 'Grid'; assignment ignored [plot:\
 :Polygon::slot]
```

Example 2. This example illustrates how primitives of the group can be extracted and manipulated seperately. We take the group of the previous example:

```
>> g := plot::Group(
     plot::Function2d(4 - x^2, x = -2..2),
     plot::line([-1, 0], [-1, 3], LineStyle = DashedLines),
     plot::line([1, 0], [1, 3], LineStyle = DashedLines)
   )

                          plot::Group()
```

With `expose` one can see the definition of the group `g` and the definition of its graphical primitives:

```
>> expose(g)

                                    2
 plot::Group(plot::Function2d(- x  + 4, x = -2..2),

    plot::Polygon(plot::Point(-1, 0), plot::Point(-1, 3)),

    plot::Polygon(plot::Point(1, 0), plot::Point(1, 3)))
```

In order to change the color of the graph, which is the first graphical primitive of the group into blue, we enter:

```
>> (g[1])::Color := RGB::Blue: plot(g)
```

Changes:

- `plot::Group` is a new function.

`plot::Lsys` – graphical primitive for a Lindenmayer system

`plot::Lsys(deg, start, rule, ...)` represents a Lindenmayer system with turning degree `deg`, starting word `start` and rule set `rule...`.

Creating Elements:

- `plot::Lsys(deg, start, rule, ...)`

Parameters:

`deg`	—	the degree by which the turtle turns left or right, an arithmetical expression
`start`	—	the starting word of the system, a string
`rule`	—	a rule of the system, an equation

Related Domains: `plot::Scene`, `plot::Turtle`

Related Functions: `plot`

Details:

- Objects generated by `plot::Lsys` represent Lindenmayer systems. Lindenmayer systems are a means to describe the growth of plants and may be used to create beautiful recursive drawings.

- A *Lindenmayer system* (A, R, s) is quite similar to a usual context-free grammar. One has a finite set A of characters (the *alphabet*), a map $R : A \to A^\star$ and a non-empty *starting word* s, an element of $A^\star$. ($A^\star$ are the words with characters from A.) For each $a \in A$ the pair $(a, R(a))$ is called a *rule* and is written as $a \to b_1 b_2 \dots b_n$ where $R(a) = b_1 b_2 \dots b_n \in A^\star$. a is the left hand side and $b_1 b_2 \dots b_n$ the right hand side of the rule.

  A Lindenmayer system describes a language L, a subset of $A^\star$. The language is defined as follows:

  - s is an element of L.
  - Let w be an element of L, let $\tilde{w}$ be the word where each character a of w has been replaced by $R(a)$. Then $\tilde{w}$ is in L.

  The language L can be created as follows: From the starting word $s = s_0$ the word s_1 is created (by replacing all characters by right hand sides of the corresponding rules). From s_1 the word s_2 is created, from that s_3 and so on. Call s_i the i-th *generation* of the starting word s.

- The interesting part of the story is the interpretation of the words s_i of the language L. Here a creature called *turtle* enters the picture, it helps to visualize the words of the language. A turtle is a drawing device which understands few simple commands. Given a word of the language L each character of the word is interpreted as a command for the turtle. The word turns into a picture with the help of the turtle.

  A turtle has a position in the plane, a direction and a colour. It understands only few commands: Move forward and draw a line, move without drawing, turn left, turn right, change your colour. Further a turtle may remember its current state (position, direction and colour) by pushing it onto a stack and change its state to a former one by popping it off from the stack. For each character of the alphabet A one of these turtle commands may be defined. A character may also have no command, causing the turtle to do nothing.

- `plot::Lsys` allows the definition of Lindenmayer systems and the plotting of words defined by the system. The domain `plot::Turtle` creates turtles, it may be useful for other purposes than plotting words of Lindenmayer systems.

- Regarding the definition of a Lindenmayer system:

  - The starting word `start` is interpreted as the starting word of the Lindenmayer system, each character in the string represents a character of the system.
  - Each rule must be of the form `lhs = rhs` where the left hand side `lhs` must be a string of lenght 1. The right hand side `rhs` must be a string, a turtle command or a colour value.

    A left hand side is interpreted as single character of the Lindenmayer system. A right hand side which is a string is interpreted as the word which replaces the left hand character. Each character in the string represents a character of the system.

    A right hand side which is a turtle command or a colour value desribe how a word generated by the Lindenmayer system is to be interpreted by the turtle. A turtle command or a colour value changes the actual state of the turtle and moves it around.

    A turtle command may be one of the identifiers `Move`, `Line`, `Left`, `Right`, `Push` or `Pop`. These commands cause the turtle to move without drawing, to draw a line, to turn left or right and to push or pop its current state. A colour value must be a list of three numbers `[r,g,b]` defining new red-, green- and blue colour values of the turtle. The colour values must be in the range between 0 and 1 like the colour values in the other plot commands.

  The following default rules for the turtle commands exist:

  - `"F" = Line`

- `"f" = Move`
- `"+" = Left`
- `"-" = Right`
- `"[" = Push`
- `"]" = Pop`

These rules are used if no other rules for the turtle commands are defined.

- In order to do a turtle plot defined by a Lindenmayer system one further needs to specify which generation of the starting word of the system is to be plotted. The default generation is the fifth one. One may use the systems `"generations"` attribute to change the generation, see below.
- A certain generation of a Lindenmayer system `l` can be displayed via the call `plot(l)`, or used with other graphical primitives of the `plot` library.
- An object of `plot::Lsys` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

  The following attributes are available for a Lindenmayer system:

attribute	meaning	properties
`generations`	The generation of the system which is to be plotted, a positive integer.	read/write

  See the help page of `plot::Curve2d` for examples for working with attributes of graphical primitives.

Result of Evaluation: Evaluating an object of the domain type `plot::Lsys` returns itself.

Access Methods

Method `getPlotdata`: create the plot data of a Lindenmayer system

`getPlotdata(`*`dom`* `l)`

- Returns a list of an inner list, where the inner list is a plot description of `l` in a `plot2d` and `plot3d` conforming syntax, respectively, i.e., it has the form `[Mode = List, [polygon(...)], ...]`.
- The plot data returned are the plot data of the turtle defined by the actual generation of the system.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `slot`: read and write attributes and plot options

```
slot(dom l, string slotname)
```

- Reads the value of the slot `slotname` of `l`. `slotname` may be the name of an attribute. See the tables of available plot attributes above.
- If `slotname` is an invalid attribute, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `l::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(l, slotname)`.

```
slot(dom l, string slotname, any val)
```

- Changes the value of the attribute with the name `slotname` to the value `val`.
- If there is no attribute with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `l::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(l, slotname, val)`.

Technical Methods

Method `print`: print a Lindenmayer system

```
print(dom l)
```

- This method returns an unevaluated expression of the form `plot::Lsys(deg, start, rule...)`. It is used to print objects of `plot::Lsys` to the screen.
- See the system function `print` for details.

Example 1. We simply list some examples without any interpretation:

```
>> L := plot::Lsys(90, "F-F-F-F", "F"="F-F+F+FF-F-F+F"):
   L::generations := 4:
   plot(L, Axes = None)
```

```
>> L := plot::Lsys(90, "F-F-F-F", "F"="FF-F--F-F"):
   L::generations := 4:
   plot(L, Axes = None)
```

```
>> L := plot::Lsys(90, "F-F-F-F", "F"="FF-F+F-F-FF"):
   L::generations := 4:
   plot(L, Axes = None)
```

```
>> L := plot::Lsys(90, "L", "L"="L+R+", "R"="-L-R", "L"=Line, "R"=Line):
   L::generations := 10:
   plot(L, Axes = None)
```

```
>> L := plot::Lsys(60, "R", "L"="R+L+R", "R"="L-R-L", "L"=Line, "R"=Line):
   L::generations := 7:
   plot(L, Axes = None)
```

```
>> L := plot::Lsys(20, "L", "L"="R[+L]R[-L]+L", "R"="RR", "L"=Line, "R"=Lin
   L::generations := 6:
   plot(L, Axes = None)
```

```
>> L := plot::Lsys(20, "L", "L"="BR[+HL]BR[-GL]+HL", "R"="RR",
                   "L"=Line, "R"=Line,
                   "B"=RGB::Brown, "H"=RGB::ForestGreen,
                   "G"=RGB::SpringGreen):
   L::generations := 6:
   plot(L, Axes = None)
```

```
>> L := plot::Lsys(60, "F++F++F", "F"="F-F++F-F"):
   L::generations := 5:
   plot(L, Axes = None)
```

Changes:

- `plot::Lsys` used to be `Lsys`.
- Adapted to comply with the `plot` library.

`plot::Point` – graphical primitive for a point

`plot::Point(x, y)` represents a plot of a two-dimensional point with the coordinates $(x; y)$.

`plot::Point(x, y, z)` represents a plot of a three-dimensional point with the coordinates $(x; y; z)$.

Creating Elements:

- `plot::Point(x, y<, option1, option2, ...>)`
- `plot::Point(x, y, z<, option1, option2, ...>)`

Parameters:

`x, y, z`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `OptionName = value`

Related Domains: `DOM_POINT`, `plot::Pointlist`, `plot::Scene`, `RGB`

Related Functions: `plot`, `plot2d`, `plot3d`, `plot::copy`, `point`

Details:

- Objects generated by `plot::Point` represent graphical primitives for two- or three-dimensional points that can be displayed via the call `plot(...)`, or used with other graphical primitives of the `plot` library.
- Note that `plot::Point`, in difference to the standard graphical primitive `point`, allows arbitrary arithmetical expressions. These expressions must evaluate to numbers at the time where you plot the points.
- An object of `plot::Point` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

`OptionName`	admissible values	default value
`Color`	`[`*`Flat`*`]`, `[`*`Flat`*`, [r,g,b]]`, `[`*`Height`*`]`, `[`*`Height`*`, [r,g,b], [R,G,B]]`, `[`*`Function`*`, f]`	`[`*`Height`*`]`
`LineStyle`	*`SolidLines`*, *`DashedLines`*	*`SolidLines`*
`LineWidth`	positive integers	`1`
`PointStyle`	*`Circles`*, *`FilledCircles`*, *`FilledSquares`*, *`Squares`*	*`FilledSquares`*
`PointWidth`	positive integers	`30`
`Title`	strings	`""`
`TitlePosition`	`[x, y]`	

See `plot2d` and `plot3d`, respectively, for further details on each option.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the call of `plot`, or use `plot::Scene` to create an object representing a graphical scene.

- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

  The following attributes are available for a point primitive:

attribute	meaning	properties
`coords`	The list of the coordinates of the point (list of two or three arithmetical expressions). The initial value is the list `[x, y]` and `x, y, z]`, respectively.	read/write
`options`	A table of plot options of the point primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for point primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the point primitive in a `plot2d` and `plot3d` conforming syntax, respectively (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read

attribute	meaning	properties
`refreshPlotdata`	A boolean value which signals whether the plot data of the point primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the point primitive is stored in the attribute `plotdata`. The initial value is `TRUE`.	read/write

See the help page of `plot::Curve2d` for examples for working with attributes of graphical primitives.

Operands: An object of `plot::Point` has either two or three operands, namely the coordinates `x`, `y` and `z`, respectively.

Important Operations:

- Operands of a point primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attribute* `coords` described above. For example, if `point` is such an object, then the calls `op(point,1)`, `point[1]` and `point::coords[1]` return the first coordinate of `point`.

  Via `point[1] := x_new` or `point::coords[1] := x_new`, the first coordinate of `point` is replaced by `x_new`.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `point` is such an object, then `point::Color := RGB::Red` changes the color of the point to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Point` returns itself.

Function Call: Calling an object of `plot::Point` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for point primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Point` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot2d` and `plot3d`, respectively, which are used to plot the object. See the table of plot options above, which gives a summary of the available plot options for point primitives and their default values.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of points.

Access Methods

Method `_index`: indexed access to the operands of a point primitive

`_index(`*`dom`* `point, `*`positive integer`* `i)`

- Returns the `i`th coordinate of `point`. If `i` is greater than the number of coordinates of `point`, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `point[i]`, or in functional notation `_index(point, i)`.

Method `dimension`: dimension of a point primitive

`dimension(`*`dom`* `point)`

- Returns the number of coordinates of `point`.

Method `getPlotdata`: create the plot data of a point primitive

`getPlotdata(`*`dom`* `point)`

- Returns a list of an inner list, where the inner list is a plot description of `point` in a `plot2d` and `plot3d` conforming syntax, respectively, i.e., it has the form `[Mode = List, [point(...)], ...]`.
  For example, with `s := plot::Point::getPlotdata(point)` the call `plot2d(s[1])` and `plot3d(s[1])`, respectively, gives a plot of `point`.
- Only those plot options will be included in the plot data of the point, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` and `plot3d`, respectively, for lists of primitives is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `point`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `point` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a point primitive

`nops(`*`dom`* `point)`

- Returns the number of coordinates of `point`, i.e., the integer 2 or 3.
- This method overloads the system function `nops`, i.e., onemay use it in the form `nops(point)`.

Method `op`: extract operands of a point primitive

`op(`*`dom`* `point,` *`positive integer`* `i)`

- Returns the `i`th coordinate of `point`. If `i` is greater than the number of coordinates of `point`, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(point, i)`.

Method `set_index`: set operands of a point primitive

`set_index(`*`dom`* `point,` *`positive integer`* `i,` *`arithm. expr.`* `x)`

- Replaces the `i`th coordinate of `point` to the value `x`.
- If `i` is greater than the number of coordinates of `point`, or if `x` is not an arithmetical expression, then a warning message is issued. In this case the call of this method has no effect on the object `point`.

- A call of this method sets the value of the attribute `refreshPlotdata` of `point` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `point, `*`string`* `slotname)`

- Reads the value of the slot `slotname` of `point`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `point::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(point, slotname)`.

`slot(`*`dom`* `point, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `point::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(point, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `point` is set to `TRUE`.

Conversion Methods

Method `convert`: conversion of objects into a point primitive

`convert(`*`any`* `p)`

- This method tries to convert `p` into an object of the domain `plot::Point`. If this is not possible, then `FAIL` is returned.
- Currently this method handles objects of the following two domain types:

- `p` is an object of the domain `DOM_POINT`. If `p` does not have a color specification, then the default color for objects of `plot::Point` is used (see "Details" above).
- `p` is a list of two or three arithmetical expressions.

Method `convert_to`: conversion of a point primitive

```
convert_to(dom point, domain T)
```

- This method tries to convert `point` into an object of the domain `T`. If this is not possible, then `FAIL` is returned.
- Currently this method handles the following domains:
  - `T` is the domain `DOM_LIST`. Then the result is a list of the coordinates of `point`.
  - `T` is the domain `DOM_POINT`. Then the result is a system point primitive, i.e., an object of the domain type `DOM_POINT`.
    Note that plot options can not be stored in such an object, except of the option *`Color`*. However, the color specification is used only if it is contained in the attribute `options` of the point primitive `point` (see the table of attributes above). Otherwise the default color specification for system point primitives is used (see the help page of `DOM_POINT`).

Method `expr`: conversion into a system point primitive

```
expr(dom point)
```

- This method converts `point` into an object of the domain type `DOM_POINT`. See the method `"convert_to"` above for details.

Technical Methods

Method `checkOption`: check a plot option

```
checkOption(equation OptionName = value)
```

- This method checks whether `OptionName` is a known plot option for point primitives (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a point primitive

```
copy(dom point)
```

- Returns a copy of the object `point`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of a point primitive

```
modify(dom point, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `point` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Point`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored. See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `point` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a point primitive

```
print(dom point)
```

- This method returns an unevaluated expression of the form `plot::Point([x, y]` and `plot::Point([x, y, z]`, respectively. It is used to print objects of `plot::Point` to the screen.
- See the system function `print` for details.

Example 1. We create the points $(1;2)$ and $(3;-1)$, setting the color of the second point to green and its size to 50:

```
>> p1 := plot::Point([1, 2]);
   p2 := plot::Point([3, 1], Color = RGB::Blue, PointWidth = 50)

                        plot::Point(1, 2)

                        plot::Point(3, 1)
```

To plot these two points in a graphical scene, enter:

```
>> plot(p1, p2)
```

Scene options may be given to the call of `plot`, such as changing the style of the axes and drawing grid lines in the background of the plot:

```
>> plot(p1, p2, Axes = Box, GridLines = Automatic)
```

Example 2. Objects of the domain `plot::Point`, and objects of the basic domain `DOM_POINT` are graphical primitives for two- or three dimensional points. The main difference between objects of these two domains is, that objects of `plot::Point` can be used together with other graphical primitives of the library `plot` such as function graphs, surface plots, point-list plots, and more.

To ease the use of such different objects, you can easily convert objects of one domain into the other. For example, an object of the domain `plot::Point` such as:

```
>> p := plot::Point([1, 2])

                          plot::Point(1, 2)
```

can be converted into the domain `DOM_POINT` as follows:

```
>> plot::Point::convert_to(p, DOM_POINT)

                           point(1.0, 2.0)
```

Note that because objects of the domain `DOM_POINT` only know the plot option *`Color`*, any other plot option set for the object `p` is lost by this conversion.

With the method `"convert"`, objects can be converted into the domain `plot::Point`. For example, we convert the list `[1, 2, 3]` into the point (1;2;3) as an object of the domain `plot::Point`:

```
>> l := [1, 2, 3]: p:= plot::Point::convert(l)

                       plot::Point(1, 2, 3)
```

One may now override default values of some plot options for the object `p` as follows:

```
>> p::Color := RGB::Blue: p::PointWidth := 50:
   plot(p, Axes = None)
```

Changes:

- `plot::Point` is a new function.

`plot::Pointlist` – graphical primitive for a list of points

`plot::Pointlist(p1, p2, ...)` represents a plot of either two- or three-dimensional points. Several drawing options exist such as drawing a line from point to point, or drawing a horizontal or vertical line from the axes to each point.

Creating Elements:

- `plot::Pointlist(p1, p2, ...<, option1, option2, ...>)`

Parameters:

`p1, p2`	—	plot points, i.e., objects that can be converted into the domain `plot::Point`; the points must have the same dimension
`option1, option2, ...`	—	plot option(s) of the form `OptionName = value`

Related Domains: `plot::Point`, `plot::Scene`

Related Functions: `plot`, `plot2d`, `plot3d`, `plot::copy`

Details:

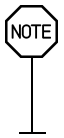
- Objects generated by `plot::Pointlist` represent graphical primitives for lists of two- or three-dimensional points that can be displayed via `plot`, or used with other graphical primitives of the `plot` library.
- Points `p1, p2, ...` of the represented list which are not given as objects of the domain `plot::Point`, are converted into objects of this domain. See the help page of this domain for information about the default plot options used in this case.
- Use the option *`DrawMode`* to connect every two points of the list with a line, or to draw a line from the x-axis and y-axis, respectively, to each point of the list.
- An object of `plot::Pointlist` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

OptionName	admissible values	default value
Color	[*Flat*], [*Flat*, [r,g,b]], [*Height*], [*Height*, [r,g,b], [R,G,B]], [*Function*, f]	[*Height*]
DrawMode	*None*, *Connected*, *Horizontal*, *Vertical*, set of these values	*None*
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
Title	strings	""
TitlePosition	[x, y]	

See `plot2d` for further details on each option, except of *DrawMode* that is explained in detail below.

- Plot options acts on every point of the list as well as on the connecting lines (see option *DrawMode*). One can specify plot options for each point individually by creating the list of points with objects of the domain `plot::Point`. See example 2.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

  The following attributes are available for a point-list primitive:

attribute	meaning	properties
`objects`	A list of objects of the domain `plot::Point`, the points of the point-list primitive. The initial value is the list `[p1, p2, ...]` of the parameters `p1, p2, ...`, where each parameter was converted into an object of the domain `plot::Point`. Note that if you extract an object of this list and do some changes on this object, then you must set the value of the attribute `refreshPlotdata` to `TRUE` in order to force a rebuild of the plot data of the point list.	read/write
`options`	A table of plot options of the point-list primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for point-list primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the point-list primitive in a `plot2d` and `plot3d` conforming syntax, respectively (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read

attribute	meaning	properties
`refreshPlotdata`	A boolean value which signals whether the plot data of the point-list primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the point-list primitive is stored in the attribute `plotdata`. The initial value is `TRUE`.	read/write

See the help page of `plot::Curve2d` for examples for working with attributes of graphical primitives.

Option <*DrawMode* = value>:

- With this option lines between points and axes can be drawn. Admissible values are *None*, *Connected*, *Horizontal*, *Vertical*, or a combination of *Connected*, *Horizontal* and *Vertical* given in form of a set; the default is *DrawMode* = *None*.
  - *DrawMode* = *None* disables lines between the points.
  - *DrawMode* = *Connected* enables lines from point to point.
  - *DrawMode* = *Horizontal* enables lines from the y-axis to each point.
  - *DrawMode* = *Vertical* enables lines from the x-axis to each point.
  - *DrawMode* = `modes` enables the modes given in the set `modes`. For example, with *DrawMode*= {*Horizontal*, *Vertical*}, a line from the x-axis to each point and a line from the y-axis to each point is drawn.
- Each line is a polygon with two vertices. Use the plot options of the point-list object to change their color or width, for example. See example 2.
- This option is only available for lists of *two-dimensional* points.

Operands: The operands of an object of `plot::Pointlist` are the parameters `p1, p2, ...` (in this order).

Important Operations:

- Operands of a point-list primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attribute* `objects` described above. For example, if `pointlist` is such an object, then the calls `op(pointlist,1)`, `pointlist[1]` and `pointlist::objects[1]` return the first point of `pointlist`.

  Via `pointlist[1] := new_point` or `pointlist::objects[1] := new_point`, the first point of `pointlist` is replaced by `new_point`.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `pointlist` is such an object, then `pointlist::Color := RGB::Red` changes the color of each point of `pointlist` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Pointlist` returns itself.

Function Call: Calling an object of `plot::Pointlist` primitive as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for point-list primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Pointlist` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot2d` and `plot3d`, respectively, which are used to plot the object. See the table of plot options above, which gives a summary of

the available plot options for point-list primitives and their default values.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of point-lists.

Access Methods

Method `_index`: indexed access to the operands of a point-list primitive

`_index(`*`dom`* `pointlist,` *`positive integer`* `i)`

- Returns the `i`th point of `pointlist`. If `i` is greater than the number of points of `pointlist`, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `pointlist[i]`, or in functional notation `_index(pointlist, i)`.

Method `dimension`: dimension of a point-list primitive

`dimension(`*`dom`* `pointlist)`

- Returns the integer 2 or 3.

Method `getPlotdata`: create the plot data of a point-list primitive

`getPlotdata(`*`dom`* `pointlist)`

- Returns a list of an inner list, where the inner list is a plot description of `pointlist` in a `plot2d` and `plot3d` conforming syntax, respectively, i.e., it has the form `[Mode = List, [point(...), ...]]`.
  For example, with `s := plot::Pointlist::getPlotdata(pointlist)` the call `plot2d(s[1])` and `plot3d(s[1])`, respectively, gives a plot of `pointlist`.
- Only those plot options will be included in the plot data of the point-list, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` and `plot3d`, respectively, for lists of primitives is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `pointlist`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `pointlist` to `FALSE`.

- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a point-list primitive

`nops(`*`dom`* `pointlist)`

- Returns the number of points of `pointlist`.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(pointlist)`.

Method `op`: extract operands of a point-list primitive

`op(`*`dom`* `pointlist,` *`positive integer`* `i)`

- Returns the `i`th point of `pointlist`. If `i` is greater than the number of points of `pointlist`, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(pointlist, i)`.

Method `set_index`: set operands of a point-list primitive

`set_index(`*`dom`* `pointlist,` *`positive integer`* `i,` *`plot::Point`* `p)`

- Replaces the `i`th point of `pointlist` by the point `p`.
- If `i` is greater than the number of coordinates of `pointlist`, or if `p` not an object of the domain `plot::Point`, then a warning message is issued. In this case the call of this method has no effect on the object `pointlist`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `pointlist` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `pointlist,` *`string`* `slotname)`

- Reads the value of the slot `slotname` of `pointlist`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.

- This method overloads the system function `slot`, i.e., one may use it in the form `pointlist::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(pointlist, slotname)`.

`slot(`*`dom`* `pointlist, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `pointlist::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(pointlist, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `point` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot option

`checkOption(`*`equation`* `OptionName = value)`

- This method checks whether `OptionName` is a known plot option for point-list primitives (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a point-list primitive

`copy(`*`dom`* `pointlist)`

- Returns a copy of the object `pointlist`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `expose`: expose the definition of a point-list

```
expose(dom pointlist)
```

- This method returns a sequence of the points of `pointlist`.
- This method overloads the system function `expose`, i.e., one may use it in the form `expose(pointlist)`.

Method `modify`: modify a copy of a point-list primitive

```
modify(dom pointlist, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `pointlist` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Pointlist`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `pointlist` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a point-list primitive

```
print(dom pointlist)
```

- This method returns an unevaluated expression of the form `plot::Pointlist()`. It is used to print objects of `plot::Pointlist` to the screen.
- Call `expose(pointlist)` to expose the points of `pointlist`.
- See the system function `print` for details.

Example 1. This example illustrates the different modes for plotting lists of points. We start with the default mode, that is plotting each point of the list in the order as specified:

```
>> p1 := plot::Pointlist(
     plot::Point(i, exp(i)) $ i = 1..5
   )

                          plot::Pointlist()
```

To plot the list of points created, call:

```
>> plot(p1)
```

If you want to connect every two points by a line, then use the option *DrawMode=Connected*:

```
>> p2 := plot::Pointlist(
     plot::Point(i, exp(i)) $ i = 1..5, DrawMode = Connected
   ):
   plot(p2)
```

Set the value of the option *DrawMode* to *Vertical* or *Horizontal* in order to draw a line from the x-axis and the y-axis, respectively, to each point of the list:

```
>> p2::DrawMode := Vertical:
   plot(p2)
```

```
>> p2::DrawMode := Horizontal:
   plot(p2)
```

Here, we used the slot operator `::` to change the value of some plot options of the graphical primitive `p2`.

One can also combine plot modes for lists of points, such as plotting vertical and horizontal lines to each point:

```
>> p2::DrawMode := {Horizontal, Vertical}:
   plot(p2)
```

Example 2. One can specify plot options for each point of the list. For example, to plot the points of a list in different sizes, enter:

```
>> p := plot::Pointlist(
     plot::Point(i, sin(i), PointWidth = 10*i) $ i = 1..10
   ):
   plot(p)
```

A plot option of the object `p` acts on every point of the list, except of the points of the list, for which a plot option is set explicitely.

For example, if we change the value of the option *Color* of the object `p` to blue, then every point is drawn in blue color:

```
>> p::Color := RGB::Blue:
   plot(p)
```

To set the color option for some points explicitely, such as changing the color of the last point to red, one may enter:

```
>> (p[10])::Color := RGB::Red:
   plot(p)
```

The plot options of a plot-list object (here: `p`) also act on the connecting lines, if the option *`DrawMode`* is used.

For example, if we create a point-list and set the color of the points explicitely to blue, then connecting lines are still drawn in the default color of objects of the domain `plot::Pointlist`, i.e., in red color:

```
>> p2 := plot::Pointlist(
     plot::Point(i, i^2, Color = RGB::Blue) $ i = -5..5,
     DrawMode = Connected
   ):
   plot(p2)
```

Changes:

- `plot::Pointlist` is a new function.

`plot::Polygon` – graphical primitive for a polygon

`plot::Polygon(p1, p2, ...)` represents a plot of a polygon build of the points $p_1, p_2, \ldots$, where the points must either be of dimension two or three.

Creating Elements:

- `plot::Polygon(p1, p2, ...<, option1, option2, ...>)`

Parameters:

`p1, p2`	—	plot points, i.e., objects that can be converted into the domain `plot::Point`; the points must have the same dimension
`option1, option2, ...`	—	plot option(s) of the form `OptionName = value`

Related Domains: `DOM_POLYGON`, `plot::Point`, `plot::Rectangle2d`, `plot::Scene`, `RGB`

Related Functions: `plot`, `plot2d`, `plot3d`, `plot::copy`, `plot::line`, `polygon`

Details:

- Objects generated by `plot::Polygon` represent graphical primitives for two- or three-dimensional polygons that can be displayed via the call `plot(...)`, or used with other graphical primitives of the `plot` library.
- An object of `plot::Polygon` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Note that `plot::Polygon`, in difference to the standard graphical object `polygon`, allows to build a polygon of points of arbitrary arithmetical expressions. These expressions must evaluate to numbers at the time where you plot the polygon.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

OptionName	admissible values	default value
Closed	TRUE, FALSE	FALSE
Color	[*Flat*], [*Flat*, [r,g,b]], [*Height*], [*Height*, [r,g,b], [R,G,B]], [*Function*, f]	[*Height*]
Filled	TRUE, FALSE	FALSE
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
Title	strings	""
TitlePosition	[x, y]	

  See `plot2d` and `polygon` for further details on each option.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the call of `plot`, or use `plot::Scene` to create an object representing a graphical scene.

- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

  The following attributes are available for a polygon primitive:

attribute	meaning	properties
`objects`	A list of the points of the polygon primitive. The initial value is the list `[p1, p2, ...]` of the parameters `p1, p2, ...`, where each parameter was converted into an object of the domain `plot::Point`. Note that if you extract an object of this list and do some changes on this object, then you must set the value of the attribute `refreshPlotdata` to `TRUE` in order to force a rebuild of the plot data of the polygon.	read/write
`options`	A table of plot options of the polygon primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for polygon primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the polygon primitive in a `plot2d` and `plot3d` conforming syntax, respectively (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read

attribute	meaning	properties
`refreshPlotdata`	A boolean value which signals whether the plot data of the polygon primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the polygon primitive is stored in the attribute `plotdata`. The initial value is `TRUE`.	read/write

See the help page of `plot::Curve2d` for examples for working with attributes of graphical primitives.

Operands: The operands of an object of `plot::Polygon` are the points `p1, p2, ...` (in this order).

Important Operations:

- Operands of a polygon primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attribute* `objects` described above. For example, if `polygon` is such an object, then the calls `op(polygon,1)`, `polygon[1]` and `polygon::objects[1]` return the first point of `polygon`.

  Via `polygon[1] := new_point` or `polygon::objects[1] := new_point`, the first point of `polygon` is replaced by `new_point`.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `polygon` is such an object, then `polygon::Color := RGB::Red` changes the color of each point of `polygon` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Polygon` returns itself.

Function Call: Calling an object of `plot::Polygon` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for polygon primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Polygon` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot2d` and `plot3d`, respectively, which are used to plot the object. See the table of plot options above, which gives a summary of the available plot options for polygon primitives and their default values.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of polygons.

Access Methods

Method `_index`: indexed access to the operands of a polygon primitive

`_index(`*`dom`* `polygon,` *`positive integer`* `i)`

- Returns the `i`th point of `polygon`. If `i` is greater than the number of points of `polygon`, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `polygon[i]`, or in functional notation `_index(polygon, i)`.

Method `dimension`: dimension of a polygon primitive

`dimension(`*`dom`* `polygon)`

- Returns the integer 2 or 3.

Method `getPlotdata`: create the plot data of a polygon primitive

```
getPlotdata(dom polygon)
```

- Returns a list of an inner list, where the inner list is a plot description of `polygon` in a `plot2d` and `plot3d` conforming syntax, respectively, i.e., it has the form `[Mode = List, [polygon(...)], ...]`.
  For example, with `s := plot::Polygon::getPlotdata(polygon)` the call `plot2d(s[1])` and `plot3d(s[1])`, respectively, gives a plot of `polygon`.
- Only those plot options will be included in the plot data of the point, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` and `plot3d`, respectively, for lists of primitives is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `polygon`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `polygon` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a polygon primitive

```
nops(dom polygon)
```

- Returns the number of points of `polygon`.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(polygon)`.

Method `op`: extract operands of a polygon primitive

```
op(dom polygon, positive integer i)
```

- Returns the `i`th point of `polygon`. If `i` is greater than the number of points of `polygon`, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(polygon, i)`.

Method `set_index`: set operands of a polygon primitive

`set_index(`*`dom`* `polygon, `*`positive integer`* `i, `*`plot::Point`* `p)`

- Replaces the `i`th point of `polygon` by the point `p`.
- If `i` is greater than the number of points of `polygon`, or if `p` is not an object of the domain `plot::Point`, then a warning message is issued. In this case the call of this method has no effect on the object `polygon`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `point` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `polygon, `*`string`* `slotname)`

- Reads the value of the slot `slotname` of `polygon`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `polygon::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(polygon, slotname)`.

`slot(`*`dom`* `polygon, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `polygon::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(polygon, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `polygon` is set to `TRUE`.

Conversion Methods

Method `convert`: conversion of objects into a polygon primitive

```
convert(any p)
```

- This method tries to convert `p` into an object of the domain `plot::Polygon`. If this is not possible, then `FAIL` is returned.
- Currently this method only handles objects of the domain type `DOM_POLYGON`. If `p` does not have a color specification, then the default color for objects of `plot::Polygon` is used (see "Details" above).

Method `convert_to`: conversion of a polygon primitive

```
convert_to(dom polygon, domain T)
```

- This method tries to convert `polygon` into an object of the domain `T`. If this is not possible, then `FAIL` is returned.
- Currently this method only implements the case where `T` is the domain `DOM_POLYGON`. The result is a system polygon primitive, i.e., an object of the domain `DOM_POLYGON`.

  Note that plot options can not be stored in such an object, except of the options *`Color`*, *`Filled`* and *`Closed`*. However, the color specifiaction is used only if it is contained in the attribute `options` of the polygon primitive `polygon` (see the table of attributes above). Otherwise the default color specification for system polygon primitives is used (see the help page of `DOM_POLYGON`).

Method `expr`: conversion into a system polygon primitive

```
expr(dom polygon)
```

- This method converts `polygon` into an object of the domain type `DOM_POLYGON`. See the method `"convert_to"` for details.

Technical Methods

Method `checkOption`: check a plot option

```
checkOption(equation OptionName = value)
```

- This method checks whether `OptionName` is a known plot option for polygon primitives (see the table of available plot options above), and `value` is an admissible value for this option.

- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a polygon primitive

`copy(`*`dom`* `polygon)`

- Returns a copy of the object `polygon`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `expose`: expose the definition of a polygon

`expose(`*`dom`* `polygon)`

- This method returns a sequence of the points of `polygon`.
- This method overloads the system function `expose`, i.e., you may use it in the form `expose(polygon)`.

Method `modify`: modify a copy of a polygon primitive

`modify(`*`dom`* `polygon,` *`equation(s)`* `Name1 = value1, ...)`

- Creates a copy of the object `polygon` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Polygon`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `polygon` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a polygon primitive

```
print(dom polygon)
```

- This method returns an unevaluated expression of the form `plot::Polygon()`. It is used to print objects of `plot::Polygon` to the screen.
- Call `expose(polygon)` to expose the points of `polygon`.
- See the system function `print` for details.

Example 1. We create a polygon build of the points (1;2), (10;2) and (10;10):

```
>> p := plot::Polygon(
     plot::Point(1, 2), plot::Point(10, 2), plot::Point(10, 10)
   )
```

```
                          plot::Polygon()
```

and plot the polygon in a graphical scene:

```
>> plot(p)
```

Plot options may be specified as additional arguments. For example, to close the polygon created above, set the option *`Closed`* to the value `TRUE`:

```
>> p := plot::Polygon(
     plot::Point(1, 2), plot::Point(10, 2), plot::Point(10, 10),
     Closed = TRUE
   ):
   plot(p)
```

You can also set plot options of an object of the domain `plot::Polygon` via the slot operator `::`. For example, to change the color of the polygon `p` into green and increase the width of the lines of the polygon, e.g., to the value 50, we enter:

```
>> p::Color := RGB::Red: p::LineWidth:= 50:
   plot(p)
```

Scene options may be given to the call of `plot`, such as removing axes from the plot:

```
>> plot(p, Axes = None)
```

Example 2. Objects of the domain `plot::Polygon`, and objects of the basic domain `DOM_POLYGON` are graphical primitives for two- or three dimensional polygons. The main difference between objects of these two domains is, that objects of `plot::Polygon` can be used together with other graphical primitives of the library `plot` such as function graphs, surface plots, point-list plots, and more.

To ease the use of such different objects, you can easily convert objects of one domain into the other. For example, an object of the domain `plot::Polygon` such as:

```
>> p := plot::Polygon([1, 2], [2, 3], [0, 3], [-1,2])
```

```
                          plot::Polygon()
```

can be converted into the domain `DOM_POLYGON` as follows:

```
>> plot::Polygon::convert_to(p, DOM_POLYGON)
```

```
 polygon(point(1.0, 2.0), point(2.0, 3.0), point(0.0, 3.0), poi\
 nt(-1.0, 2.0))
```

Note that because objects of the domain `DOM_POLYGON` only know the plot options *`Color`*, *`Filled`* and *`Closed`*, any other plot option set for the object `p` is lost by this conversion.

With the method `"convert"`, objects can be converted into the domain `plot::Polygon`. For example, we convert the polygon `polygon(point(1, 2, 3), point(2, 4, 6))` into an object of the domain `plot::Polygon`:

```
>> p := polygon(point(1, 2, 3), point(2, 4, 6)):
   q := plot::Polygon::convert(p)
```

```
                          plot::Polygon()
```

One may now override default values of some plot options for the object `q` as follows:

```
>> q::Color := RGB::Blue: q::LineWidth := 50:
   plot(q, Axes = None)
```

Changes:

- `plot::Polygon` is a new function.

`plot::Rectangle2d` – graphical primitive for a two-dimensional rectangle

`plot::Rectangle2d(p, w, h)` represents a plot of a two-dimensional rectangle with lower left corner $p = (p_x; p_y)$, width w and height h.

Creating Elements:

- `plot::Rectangle2d(c, w, h<, option1, option2, ...>)`
- `plot::Rectangle2d(p, w, h<, option1, option2, ...>)`

Parameters:

`c`	— a list of two arithmetical expressions
`p`	— a two-dimensional point, i.e., an object of the domain `plot::Point` or `DOM_POINT`
`w, h`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `OptionName = value`

Related Domains: `plot::Ellipse2d`, `plot::Point`, `plot::Polygon`, `plot::Scene`, `RGB`

Related Functions: `plot`, `plot2d`, `plot::copy`, `plot::line`

Details:

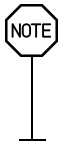
- Objects generated by `plot::Rectangle2d` represent graphical primitives for two-dimensional rectangles that can be displayed via the clal `plot(...)`, or used with other graphical primitives of the `plot` library. See example 1.
- If the first parameter of the call of `plot::Rectangle2d` is a point `p`, then it is converted into a list of the two coordinates of `p`. Specified plot options for `p` are ignored! (Cf. example 2.)
- The values for the width `w` and height `h` can also be negative numbers. The rectangle drawn consists of the four points $p = (p_x; p_y)$, $(p_x; p_y + h)$, $(p_x + w; p_y + h)$ and $(p_x + w; p_y)$.
- An object of `plot::Rectangle2d` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

`OptionName`	admissible values	default value
`Color`	`[`*`Flat`*`]`, `[`*`Flat`*`, [r,g,b]]`, `[`*`Height`*`]`, `[`*`Height`*`, [r,g,b], [R,G,B]]`, `[`*`Function`*`, f]`	`[`*`Flat`*`, RGB::Red]`

OptionName	admissible values	default value
Filled	TRUE, FALSE	FALSE
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
Title	strings	""
TitlePosition	[x, y]	

See plot2d and polygon for further details on each option.

- Scene options for the parameters option1, option2, ... are not allowed! One may pass scene options to the call of plot, or use plot::Scene to create an object representing a graphical scene. Cf. example 1.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator ::.

  Each attribute has the property "read", i.e., the value of an attribute attr of a graphical primitive o can be read with o::attr. If the attribute also has the "write" property, then the value of the attribute can be changed with o::attr := new_value.

  The following attributes are available for a rectangle primitive:

attribute	meaning	properties
corner	a list of two arithmetical expressions being the lower left corner of the rectangle. The initial value is the parameter c, or the list of the coordinates of p, respectively.	read/write
dimension	The dimension of the rectangle primitive, i.e., the integer 2.	read
height	The height of the rectangle (an arithmetical expression). The initial value is the parameter h.	read/write

attribute	meaning	properties
`options`	A table of plot options of the rectangle primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for rectangle primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the rectangle primitive in a `plot2d` and `plot3d` conforming syntax, respectively (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read
`refreshPlotdata`	A boolean value which signals whether the plot data of the rectangle primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the rectangle primitive is stored in the attribute `plotdata`. The initial value is `TRUE`.	read/write
`width`	The width of the rectangle (an arithmetical expression). The initial value is the parameter `w`.	read/write

See example 2.

Operands: An object of `plot::Rectangle2d` consists of the three operands `c`, `w` and `h`.

Important Operations:

- Operands of a rectangle primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attributes* `corner`, `height` and `width` described above. For example, if `rectangle` is such an object, then the calls `op(rectangle,1)`, `rectangle[1]` and `rectangle::corner` return the list representing the lower left corner of `rectangle`.

  Via `rectangle[1] := new_point` or `rectangle::corner := new_point`, the lower left corner of `rectangle` is replaced by the list `new_point`.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `rectangle` is such an object, then `rectangle::Color := RGB::Red` changes the color of of `rectangle` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Rectangle2d` returns itself.

Function Call: Calling an object of `plot::Rectangle2d` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for rectangle primitives and their default values. Each entry has the form `OptionName = default`.

When an object of the domain `plot::Rectangle2d` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in this table, will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

`optionNames` is a set of the available options for plots of two-dimensional rectangles.

Access Methods

Method `_index`: indexed access to the operands of a rectangle primitive

```
_index(dom rectangle, positive integer i)
```

- Returns the `i`th operand of `rectangle`. See "Operands" above for a description of the operands of `curve`. If `i` is greater than 2, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `rectangle[i]`, or in functional notation `_index(rectangle, i)`.

Method `dimension`: dimension of a rectangle primitive

```
dimension(dom rectangle)
```

- Returns the integer 2.

Method `getPlotdata`: create the plot data of a rectangle primitive

```
getPlotdata(dom rectangle)
```

- Returns a list of an inner list, where the inner list is a plot description of `rectangle` in a `plot2d` conforming syntax, i.e., it has the form `[Mode = List, [polygon(...)], ...]`.
  For example, with `s := plot::Rectangle2d::getPlotdata(rectangle)` the call `plot2d(s[1])` gives a plot of `rectangle`.
- Only those plot options will be included in the plot data of the rectangle, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` for curves is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `rectangle`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `rectangle` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a rectangle primitive

`nops(`*`dom`* `rectangle)`

- Returns the integer 3.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(rectangle)`.

Method `op`: extract operands of a rectangle primitive

`op(`*`dom`* `rectangle,` *`positive integer`* `i)`

- Returns the `i`th operand of `rectangle`. See "Operands" above for a description of the operands of `rectangle`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(rectangle, i)`.

Method `set_index`: set operands of a rectangle primitive

`set_index(`*`dom`* `rectangle,` *`positive integer`* `i,` *`any`* `val)`

- Replaces the `i`th operand of `rectangle` by the value `val`. See "Operands" above for a description of the operands of `rectangle`.
- If `i` is greater than 3, or if `val` is not an admissible value for the `i`th operand, then a warning message is issued. In this case the call of this method has no effect on the object `rectangle`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `rectangle` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `rectangle,` *`string`* `slotname)`

- Reads the value of the slot `slotname` of `rectangle`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `rectangle::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(rectangle, slotname)`.

```
slot(dom rectangle, string slotname, any val)
```

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `rectangle::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(rectangle, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `rectangle` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot option

```
checkOption(equation OptionName = value)
```

- This method checks whether `OptionName` is a known plot option for rectangle primitives (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a rectangle primitive

```
copy(dom rectangle)
```

- Returns a copy of the object `rectangle`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of a rectangle primitive

```
modify(dom rectangle, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `rectangle` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Rectangle2d`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `rectangle` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a rectangle primitive

```
print(dom rectangle)
```

- This method returns an unevaluated expression of the form `plot::Rectangle2d(p, w, h)`. It is used to print objects of `plot::Rectangle2d` to the screen.
- See the system function `print` for details.

Example 1. We define a rectangle with lower left corner at point (1;2), width 3 and height 4, and a square of length one with lower left corner at the point (0;0):

```
>> r := plot::Rectangle2d([1, 2], 3, 4);
   s := plot::Rectangle2d([0, 0], 1, 1, Filled = TRUE)

                plot::Rectangle2d([1, 2], 3, 4)

                plot::Rectangle2d([0, 0], 1, 1)
```

The area of the square is filled in the color of the border of the rectangle (which is red by default). We plot these two objects in a graphical scene, without showing axes:

```
>> plot(r, s, Axes = None)
```

Example 2. The attribute `corner`, which specifies the lower left point of the rectangle, is a list of two arithmetical expressions also if the corner point of the object created was given as an object of the domain `plot::Point` or `DOM_POINT`:

```
>> c := plot::Point([-1, 1]):
   r := plot::Rectangle2d(c, 2, -2):
   r::corner
```

```
                              [-1, 1]
```

If you replace the value of the attribute `corner`, then the point must be given as a list of two arithmetical expressions, otherwise a warning message is issued, saying that the assignment is ignored:

```
>> r::corner:= point(0, 0)
```

```
 Warning: 3rd argument: expecting a list of two arithmeti-
cal ex\
 pressions; assignment ignored [plot::Rectangle2d::slot]
```

```
                           point(0, 0)
```

```
>> r::corner
```

```
                              [-1, 1]
```

Note that if you specify an object of the domain `plot::Point` or `DOM_POINT` as the corner of the rectangle, then plot options of the point are ignored. For example, if we change the color of the point `c` created above to blue and create a new rectangle:

```
>> c::Color := RGB::Blue: r := plot::Rectangle2d(c, 1, 1):
   plot(r)
```

then the rectangle is still drawn in red color (the default color of objects of the domain `plot::Rectangle2d`). You must use the color option of the object `r` to change the color of the rectangle:

```
>> r::Color := RGB::Blue: plot(r)
```

See the help page of `plot::Curve2d` for more examples for working with attributes of graphical primitives.

Changes:

- `plot::Rectangle2d` is a new function.

`plot::Scene` – a graphical scene

`plot::Scene(object1, object2, ...)` combines the graphical objects `object1`, `object2` etc. to a graphical scene.

Creating Elements:

- `plot::Scene(object1 <, object2, ...> <, option1, option2, ...>)`

Parameters:

`scene`	— a graphical scene: an object of domain type `plot::Scene`
`object1, object2, ...`	— 2D or 3D graphical objects
`option1, option2, ...`	— scene options of the form `OptionName = value`

Related Domains: `RGB`

Related Functions: `plot`, `plot2d`, `plot3d`, `plot::copy`

Details:

- Objects generated by `plot::Scene` represent two- or three-dimensional graphical scenes that can be displayed via `plot(...)`.

  The parameters `object1`, `object2` etc. must be graphical objects generated by routines of the library `plot`. Graphical primitives include graphs of functions (of domain type `plot::Function2d` and `plot::Function3d`), points and polygons (of domain type `plot::Point` and `plot::Polygon`, respectively), and surfaces (of domain type `plot::Surface3d`).

  See examples 1 and 2.

- Scene options `option1`, `option2` etc. are specified by equations `OptionName = value`. The following tables give an overview of the available options for two- and three-dimensional scenes.

  This table contains options and their default values for two-dimensional graphical scenes:

`OptionName` (2D)	admissible values	default value
`Arrows`	`TRUE`, `FALSE`	`FALSE`
`Axes`	*`Box`*, *`Corner`*, *`None`*, *`Origin`*	*`Origin`*
`AxesOrigin`	*`Automatic`*, `[x0, y0]`	*`Automatic`*
`AxesScaling`	[*`Lin`*/*`Log`*, *`Lin`*/*`Log`*]	[*`Lin`*, *`Lin`*]

OptionName (2D)	admissible values	default value
BackGround	[r, g, b]	RGB::White
Discont	TRUE, FALSE	FALSE
FontFamily	"helvetica", "lucida", ..	"helvetica"
FontSize	positive integers	8
FontStyle	"bold", ..	"bold"
ForeGround	[r, g, b]	RBG::Black
GridLines	*Automatic*, *None* or [xValue, yValue]. Admissible values for xValue, yValue are *Automatic*, integers, *Steps* = d or *Steps* = [d, n].	*None*
GridLinesColor	[r, g, b]	RGB::Gray
GridLinesWidth	positive integers	1
GridLinesStyle	*SolidLines*, *DashedLines*	*DashedLines*
Labeling	TRUE, FALSE	TRUE
Labels	[string, string]	["x", "y"]
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PlotDevice	*Screen*, "filename", ["filename", *Ascii*], ["filename", *Binary*]	*Screen*
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
RealValuesOnly	TRUE, FALSE	FALSE
Scaling	*Constrained*, *UnConstrained*	*UnConstrained*
Ticks	*Automatic*, *None*, an integer or [xValue, yValue]. Admissible values for xValue, yValue are *Automatic*, an integer, *Steps* = d, *Steps* = [d, n] or a list of user defined ticks.	*Automatic*
Title	strings	""
TitlePosition	*Above*, *Below*, [x, y]	*Above*
ViewingBox	*Automatic* or [xValue, yValue]. Admissible values for xValue, yValue are *Automatic* or a range a..b.	*Automatic*

See plotOptions2d for further details on each option.

- For three-dimensional graphical scenes the following options are available:

OptionName (3D)	admissible values	default value
Arrows	TRUE, FALSE	FALSE
Axes	*Box*, *Corner*, *None*, *Origin*	*Box*
AxesOrigin	*Automatic*, [x0, y0, z0]	*Automatic*
AxesScaling	[*Lin*/*Log*, *Lin*/*Log*, *Lin*/*Log*]	[*Lin*, *Lin*, *Lin*]
BackGround	[r, g, b]	RGB::White
CameraPoint	*Automatic*, [x, y, z]	*Automatic*
FocalPoint	*Automatic*, [x, y, z]	*Automatic*
FontFamily	"helvetica", "lucida", ..	"helvetica"
FontSize	positive integers	8
FontStyle	"bold", ..	"bold"
ForeGround	[r, g, b]	RGB::Black
Labeling	TRUE, FALSE	TRUE
Labels	[string, string, string]	["x","y","z"]
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PlotDevice	*Screen*, "filename", ["filename", *Ascii*], ["filename", *Binary*]	*Screen*
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
Scaling	*Constrained*, *UnConstrained*	*UnConstrained*
Ticks	*Automatic*, *None* or integers	*Automatic*
Title	strings	""
TitlePosition	*Above*, *Below*, [x, y]	*Above*
ViewingBox	*Automatic*	*Automatic*

See plotOptions3d for further details on each option.

- A so-called *attribute* of an object of the domain plot::Scene is a named entry of the object which can be accessed via the slot operator ::.

  Each attribute has the property "read", i.e., the value of an attribute attr of a graphical primitive o can be read with o::attr. If the attribute also has the "write" property, then the value of the attribute can be changed with o::attr := new_value. See example 3.

  The following attributes are available for an object representing a graphical scene:

attribute	meaning	properties
dimension	The dimension of the scene, i.e., the integer 2 or 3.	read

attribute	meaning	properties
`objects`	A list of the graphical primitives of the scene. A graphical primitive is an object of type `"graphprim"`. The initial value is the list `[object1, object2, ...]` of the parameters `object1, object2, ...`	read/write
`options`	A table of plot options of the graphical scene. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for graphical scenes. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions2d"` and `"defaultOptions3d"`, respectively, where such options are replaced and added, respectively, which are given with the parameters `option1, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the graphical scene in a `plot2d` and `plot3d` conforming syntax, respectively. The value of this attribute can be read if the method `"getPlotdata"` (see below) was called before. The initial value is the empty list `[]`.	read

- The graphical objects `object1`, `object2` etc. must have the same dimension. A mix of two- and three-dimensional primitives in a single scene is not supported!

Operands: The operands of an object of `plot::Scene` are the parameters `object1, object2, ...` (in this order).

Important Operations:

- Operands of a graphical scene can be accessed either using the system

function `op`, the index operator `[ ]`, or using the *attribute* `objects` described above. For example, if `scene` is such an object, then the calls `op(scene,1)`, `scene[1]` and `scene::objects[1]` return the parameter `object1`.

Via `scene[1] := g`, the first object of the scene is replaced by the graphical primitive `g` (that must be of the type `"graphprim"`).

See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `scene` is such an object, then `scene::Axes := None` removes the axes in the graphical scene.

Result of Evaluation: Evaluating an object of the domain type `plot::Scene` returns itself.

Function Call: Calling an object of `plot::Scene` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions2d` is a table of plot options for two-dimensional graphical scenes and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Scene` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value internally set by the function `plot2d` is used when plotting the object. See the table of plot options above, which gives a summary of the available plot options for two-dimensional graphical scenes and their default values.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames2d` is a set of the available option names for plots of two-dimensional graphical scenes.

`defaultOptions3d` is a table of plot options for three-dimensional graphical scenes and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Scene` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value internally set by the function `plot3d` is used when plotting the object. See the table of plot options above, which gives a summary of the available plot options for three-dimensional graphical scenes and their default values.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames3d` is a set of the available option names for plots of three-dimensional graphical scenes.

Access Methods

Method `_index`: indexed access to the operands of a graphical scene

```
_index(dom scene, positive integer i)
```

- Returns the `i`th graphical primitive of `scene`. If `i` is greater than the number of graphical primitives of `scene`, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `scene[i]`, or in functional notation `_index(scene, i)`.

Method `dimension`: dimension of a graphical scene

```
dimension(dom scene)
```

- Returns the value of the attribute `dimension`, i.e., the integer 2 or 3 (see the table of attributes above).

Method `getPlotdata`: create the plot data of a graphical scene

```
getPlotdata(dom scene)
```

- Returns a list in the form `[SceneOptions, [Mode = ...], ...]`, i.e., a plot description of `scene` in a `plot2d` and `plot3d` conforming syntax, respectively.
  For example, with `s := plot::Scene::getPlotdata(scene)` the call `plot2d(op(s)` and `plot3d(op(s)`, respectively, gives a plot of `scene`.
- Only those plot options will be included in the plot data of the point, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot2d` and `plot3d`, respectively, for graphical scenes is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `scene`.

Method `nops`: number of operands of a graphical scene

```
nops(dom scene)
```

- Returns the number of graphical primitives of `scene`.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(scene)`.

Method `op`: extract operands of a graphical scene

```
op(dom scene, positive integer i)
```

- Returns the `i`th graphical primitive of `scene`. If `i` is greater than the number of graphical primitives of `scene`, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(scene, i)`.

Method `set_index`: set operands of a graphical scene

```
set_index(dom scene, positive integer i, graphprim o)
```

- Replaces the `i`th graphical primitive of `scene` by `o`.
- If `i` is greater than the number of graphical primitives of `scene`, or if `g` is not an object of type `"graphprim"`, then a warning message is issued. In this case the call of this method has no effect on the object `scene`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `scene, `*`string`* `slotname)`

- Reads the value of the slot `slotname` of `scene`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.
  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `scene::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(scene, slotname)`.

`slot(`*`dom`* `scene, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `scene::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(scene, slotname, val)`.

Technical Methods

Method `checkOption2d`: check a plot option

`checkOption2d(`*`equation`* `OptionName = value)`

- This method checks whether `OptionName` is a known plot option for a two-dimensional scene (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `checkOption3d`: check on plot options

```
checkOption3d(equation OptionName = value)
```

- This method checks whether `OptionName` is a known plot option for a three-dimensional scene (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to throw an error message to the user.

Method `copy`: create a copy of a graphical scene

```
copy(dom scene)
```

- Returns a copy of the object `scene`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `expose`: expose the definition of a scene

```
expose(dom scene)
```

- This method returns a sequence of the graphical primitives of `scene`.
- This method overloads the system function `expose`, i.e., one may use it in the form `expose(scene)`.

Method `modify`: modify a copy of a graphical scene

```
modify(dom scene, equation ident = val, ...)
```

- Creates a copy of the object `scene` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Scene`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored. See the tables of available options and attributes above.
- This method is called from the function `plot::modify`.

Method `print`: print a graphical scene

```
print(dom scene)
```

- This method returns an unevaluated expression of the form `plot::Scene()`. It is used to print objects of `plot::Scene` to the screen.
- Call `expose(scene)` to expose the graphical primitives of the scene.
- See the system function `print` for details.

Example 1. The following calls return objects representing the graphs of the sinus and cosinus function in the interval $[0, 2\pi]$:

```
>> f1 := plot::Function2d(sin(x), x = 0..2*PI);
   f2 := plot::Function2d(
     cos(x), x = 0..2*PI, Color = RGB::Blue
   )
```

```
plot::Function2d(sin(x), x = 0..2 PI)
```

```
plot::Function2d(cos(x), x = 0..2 PI)
```

The call `plot(f1, f2)` displays these graphs `f1` and `f2`. In fact, the function `plot` first creates a graphical scene consisting of these two graphs as follows:

```
>> s := plot::Scene(f1, f2)
```

```
plot::Scene()
```

and then displays this scene:

```
>> plot(s)
```

To change default values of some scene options, pass the scene options to the call of `plot::Scene` as additional arguments, or, if an object of `plot::Scene` is already created, change the value of the corresponding option with the slot operator `::` and call `plot` again to display the changed object.

For example, to draw grid lines in the background of the plot of the graphical scene `s` created in the previous input, we enter:

```
>> s::GridLines := Automatic:
   plot(s)
```

Example 2. We create a graphical scene consisting of a graph of the sequence $n \mapsto \frac{\sin(n)}{n}$ in the interval $[1, 50]$, enclosed by the graphs of the functions $x \mapsto 1/x$ and $x \mapsto -1/x$:

```
>> s := plot::Scene(
     plot::Function2d(1/x, x = 1..50),
     plot::Pointlist([n, sin(n)/n] $ n = 1..50, Color = RGB::Blue),
     plot::Function2d(-1/x, x = 1..50)
   )
```

```
plot::Scene()
```

We plot the scene:

```
>> plot(s)
```

One can access the graphical primitives of a graphical scene using the index operator `[]`. For example, the sequence created above is the second operand of the scene `s`:

```
>> pl := s[2]
```

```
plot::Pointlist()
```

Any change of the object `pl`, for example, a change of some plot options, reacts on the plot of the scene `s`:

```
>> pl::PointWidth := 15: plot(s)
```

This is due to the reference effect for domains. If changes of an object should not reflect the scene in that the object is contained, one must first explicitly create a copy of the corresponding object using the function `plot::copy`:

```
>> pl2 := plot::copy(pl)
```

```
plot::Pointlist()
```

Now changes on the object `pl2` do not react on the object `pl` of the graphical scene `s`. For example, if we change the draw mode of the point list `pl2` in order to connect every two points of the point list by a line, the plot of the scene `s` remains unchanged:

```
>> pl2::DrawMode := Connected: plot(s)
```

whereas the copied object `pl2` is displayed as follows:

```
>> plot(pl2)
```

Example 3. This example illustrates how to read and write attributes of graphical scenes (see the table of available attributes in "Details" above).

Plot options, which are explicitly set for a graphical scene, are stored under the attribute `options` and can be read with the slot operator `::`:

```
>> s := plot::Scene(
     plot::Curve2d([sin(x), cos(x)], x = 0..2*PI),
     Axes = Box
   ):
   s::options

        table(
          TitlePosition = Above,
          PointWidth = 30,
          LineStyle = SolidLines,
          AxesScaling = [Lin, Lin],
          Axes = Box,
          BackGround = [1.0, 1.0, 1.0],
          Ticks = Automatic,
          ViewingBox = Automatic,
          RealValuesOnly = FALSE,
          Labeling = TRUE,
          Discont = FALSE,
          GridLinesStyle = DashedLines,
          ForeGround = [0.0, 0.0, 0.0],
          AxesOrigin = Automatic,
          LineWidth = 1,
          PointStyle = FilledSquares,
          GridLines = None,
          Arrows = FALSE,
          GridLinesWidth = 1,
          Scaling = UnConstrained,
          GridLinesColor = [0.752907, 0.752907, 0.752907]
        )
```

These are default values of some options of two-dimensional graphical scenes, defined by the entry `"defaultOptions2d"` of the domain `plot::Scene`:

```
>> plot::Scene::defaultOptions2d

        table(
          TitlePosition = Above,
          PointWidth = 30,
          LineStyle = SolidLines,
          AxesScaling = [Lin, Lin],
          Axes = Origin,
          BackGround = [1.0, 1.0, 1.0],
          Ticks = Automatic,
```

```
      ViewingBox = Automatic,
      RealValuesOnly = FALSE,
      Labeling = TRUE,
      Discont = FALSE,
      GridLinesStyle = DashedLines,
      ForeGround = [0.0, 0.0, 0.0],
      AxesOrigin = Automatic,
      LineWidth = 1,
      PointStyle = FilledSquares,
      GridLines = None,
      Arrows = FALSE,
      GridLinesWidth = 1,
      Scaling = UnConstrained,
      GridLinesColor = [0.752907, 0.752907, 0.752907]
    )
```

When the plot data of a graphical scene is created (calling the method `"getPlotdata"`), only those plot options are used that are contained in the table `s::options`:

```
>> plot::Scene::getPlotdata(s)

 [TitlePosition = Above, PointWidth = 30,

    LineStyle = SolidLines, AxesScaling = [Lin, Lin],

    Axes = Box, BackGround = [1.0, 1.0, 1.0],

    Ticks = Automatic, ViewingBox = Automatic,

    RealValuesOnly = FALSE, Labeling = TRUE, Discont = FALSE,

    GridLinesStyle = DashedLines, ForeGround = [0.0, 0.0, 0.0],

    AxesOrigin = Automatic, LineWidth = 1,

    PointStyle = FilledSquares, GridLines = None,

    Arrows = FALSE, GridLinesWidth = 1,

    Scaling = UnConstrained, GridLinesColor =

    [0.752907, 0.752907, 0.752907],

    [Mode = Curve, [sin(x), cos(x)], x = [0.0, 6.283185307],

    Grid = [100], Color = [Flat, [1.0, 0.0, 0.0]]]]
```

This means that for any other available scene option not contained in the table `s::options` (e.g., the option *`Title`*), the default value is internally set by the function `plot2d` when plotting the scene.

Use `delete` to remove plot options set for a graphical scene:

```
>> delete s::options[Axes]: s::options
```

```
        table(
          TitlePosition = Above,
          PointWidth = 30,
          LineStyle = SolidLines,
          AxesScaling = [Lin, Lin],
          BackGround = [1.0, 1.0, 1.0],
          Ticks = Automatic,
          ViewingBox = Automatic,
          RealValuesOnly = FALSE,
          Labeling = TRUE,
          Discont = FALSE,
          GridLinesStyle = DashedLines,
          ForeGround = [0.0, 0.0, 0.0],
          AxesOrigin = Automatic,
          LineWidth = 1,
          PointStyle = FilledSquares,
          GridLines = None,
          Arrows = FALSE,
          GridLinesWidth = 1,
          Scaling = UnConstrained,
          GridLinesColor = [0.752907, 0.752907, 0.752907]
        )
```

Now the value of the option *`Axes`* is the default value set by `plot2d` (which is the value `Origin`), or read from the preferences of the MuPAD's graphic tool VCam:

```
>> plot(s)
```

Changes:

- `plot::Scene` is a new function.

`plot::Surface3d` – graphical primitive for a three-dimensional surface plot

`plot::Surface3d([x, y, z], u = a..b, v = c..d)` represents a plot of the surface defined by $(u, v) \mapsto \big(x(u,v); y(u,v); z(u,v)\big)$ with $(u, v) \in [a, b] \times [c, d]$.

Creating Elements:

- `plot::Surface3d([x, y, z], u = a..b, v = c..d<, option1, option2, ...>)`

Parameters:

`x, y, z`	— arithmetical expressions in `u` and `v`
`u, v`	— identifiers
`a, b, c, d`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `OptionName = value`

Related Domains: `plot::Curve3d`, `plot::Function3d`, `RGB`

Related Functions: `plot`, `plot3d`, `plot::copy`

Details:

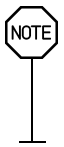
- Objects generated by `plot::Surface3d` represent graphical primitives for three-dimensional surfaces that can be displayed via `plot(...)`, or used with other graphical primitives of the `plot` library.
- An object of `plot::Surface3d` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.
- Options `option1, option2, ...` are specified by equations `OptionName = value`. The following table gives an overview of the available options:

`OptionName`	admissible values	default value
`Color`	`[`*`Flat`*`]`, `[`*`Flat`*`, [r,g,b]]`, `[`*`Height`*`]`, `[`*`Height`*`, [r,g,b], [R,G,B]]`, `[`*`Function`*`, f]`	`[`*`Height`*`]`
`Grid`	[integer]	`[20,20]`
`LineStyle`	*`SolidLines`*, *`DashedLines`*	*`SolidLines`*
`LineWidth`	positive integers	`1`
`PointStyle`	*`Circles`*, *`FilledCircles`*, *`FilledSquares`*, *`Squares`*	*`FilledSquares`*
`PointWidth`	positive integers	`30`
`Smoothness`	[integer]	`[0]`
`Style`	`[`*`Points`*`]` `[`*`WireFrame`*`, `*`Mesh`*`]` `[`*`WireFrame`*`, `*`ULine`*`]` `[`*`WireFrame`*`, `*`VLine`*`]`	`[`*`ColorPatches`*`, `*`AndMesh`*`]`

OptionName	admissible values	default value
	[*HiddenLine*, *Mesh*] [*HiddenLine*, *ULine*] [*HiddenLine*, *VLine*] [*ColorPatches*, *Only*] [*ColorPatches*, *AndMesh*] [*ColorPatches*, *AndU-Line*] [*ColorPatches*, *AndV-Line*] [*Transparent*, *Only*] [*Transparent*, *AndMesh*] [*Transparent*, *AndULine*] [*Transparent*, *AndVLine*]	
Title	strings	""
TitlePosition	[x, y]	

See `plot3d` for further details on each option.

- Scene options for the parameters `option1, option2, ...` are not allowed! One may pass scene options to the call of `plot`, or use `plot::Scene` to create an object representing a graphical scene. Cf. example 1.



- A so-called *attribute* of a graphical primitive is a named entry of the object which can be accessed via the slot operator `::`.

  Each attribute has the property "read", i.e., the value of an attribute `attr` of a graphical primitive `o` can be read with `o::attr`. If the attribute also has the "write" property, then the value of the attribute can be changed with `o::attr := new_value`.

  The following attributes are available for a surface primitive:

attribute	values	properties
`options`	A table of plot options of the surface primitive. Note that if you change the value of this attribute, the entries of the assigned table are not checked to be valid plot options for surface primitives. Invalid entries lead to runtime errors. The initial value of this attribute is the table stored under the domain entry `"defaultOptions"`, where such options are replaced and added, respectively, which are given with the parameters `option1, option2, ...` of the creating call.	read/write
`plotdata`	List of the plot data of the surface primitive in a `plot3d` conforming syntax (see the method `"getPlotdata"` below). Note that the value of this attribute should only be used if the attribute `refreshPlotdata` has the value `FALSE` (see below).	read
`range1`	The first parameter of the surface and its range in the form `ident1 = a..b`. The initial value is the parameter `u = a..b`.	read/write
`range2`	The second parameter of the surface and its range in the form `ident2 = c..d`. The initial value is the parameter `v = c..d`.	read/write

attribute	values	properties
`refreshPlotdata`	A boolean value which signals whether the plot data of the surface primitive must be (re-)build with the method `"getPlotdata"` (see below). If its value is `FALSE`, then the plot data of the surface primitive is stored in the attribute `plotdata`. The initial value is `TRUE`. See the help page of `plot::Curve2d` for an example.	read/write
`term`	The term of the surface in form of a list of three arithmetical expressions. The initial value is the parameter `[x, y, z]`.	read/write

See the examples of the help page of `plot::Function3d` about working with attributes.

Operands: An object of `plot::Surface3d` consists of three operands. The first operand is the list `[x, y, z]`. The second operand is the first parameter of the surface and its range in the form `u = a..b`, and the third one is the second parameter of the surface and its range in the form `v = c..d`.

Important Operations:

- Operands of a surface primitive can be accessed either using the system function `op`, the index operator `[ ]`, or using the *attributes* described above. For example, if `surface` is such an object, then the calls `op(surface,1)`, `surface[1]` and `surface::term` return the list `[x, y, z]`.

  Via `surface[1] := [x_new, y_new, z_new]` or `surface::term := [x_new, y_new, z_new]`, the parametrization of a surface plot can be changed.

  See the methods `"op"`, `"_index"`, `"set_index"` and `"slot"` below.

- Use the slot operator `::` to get or set plot options of such objects afterwards, i.e., when they have been created. For example, if `surface` is such an object, then `surface::Color := RGB::Red` changes the color of the surface primitive `surface` to red.

Result of Evaluation: Evaluating an object of the domain type `plot::Surface3d` returns itself.

Function Call: Calling an object of `plot::Surface3d` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Entries:

`defaultOptions` is a table of plot options for surface primitives and their default values. Each entry has the form `OptionName = default_value`.

When an object of the domain `plot::Surface3d` is created, then a copy of this table is stored under the attribute `options` (see the table of attributes above), where those options are added and replaced, respectively, which are given by the (optional) parameters `option1, option2, ...` of the creating call (see "Creating Elements" above).

Plot options, which are not contained in the table stored under the attribute `options` will not be included in the plot data of the object created by the method `"getPlotdata"` (see below).

For those options, the corresponding default value either is set by a graphical scene, if the option also exists as a scene option (such as the option *`PointWidth`*), or it is internally set by the function `plot3d` which is used to plot the object. See the table of plot options above, which gives a summary of the available plot options for function primitives and their default values.

See the examples of the help page of `plot::Function3d`.

To change the default value of some plot options, the option name and its default value may be added to the table `"defaultOptions"`, or replaced by a new value, respectively.

`optionNames` is a set of the available option names for plots of three-dimensional surfaces.

Access Methods

Method `_index`: indexed access to the operands of a surface primitive

`_index(`*`dom`* `surface,` *`positive integer`* `i)`

- Returns the `i`th operand of `surface`. See "Operands" above for a description of the operands of `surface`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `_index`, i.e., one may use it in the form `surface[i]`, or in functional notation `_index(surface, i)`.

Method `dimension`: dimension of a surface primitive

```
dimension(dom surface)
```

- Returns the integer 3.

Method `getPlotdata`: the plot data of a surface primitive

```
getPlotdata(dom surface)
```

- Returns a list of an inner list, where the inner list is a plot description of `surface` in a `plot3d` conforming syntax, i.e., it has the form `[Mode = Surface, [...], ...]`.
  For example, with `s := plot::Function3d::getPlotdata(surface)` the call `plot3d(s[1])` gives a plot of `surface`.
- Only those plot options will be included in the plot data of the surface, that are contained in the table stored under the attribute `options` (see the table of attributes above). For any other plot option not contained in this table, the corresponding default value set by the function `plot3d` for surfaces is used when plotting the object.
- The result is stored as the value of the attribute `plotdata` of `surface`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `surface` to `FALSE`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `nops`: number of operands of a surface primitive

```
nops(dom surface)
```

- Returns the integer 3.
- This method overloads the system function `nops`, i.e., one may use it in the form `nops(surface)`.

Method `op`: extract operands of a surface primitive

```
op(dom surface, positive integer i)
```

- Returns the `i`th operand of `function`. See "Operands" above for a description of the operands of `surface`. If `i` is greater than 3, then `FAIL` is returned.
- This method overloads the system function `op`, i.e., one may use it in the form `op(surface, i)`.

Method `set_index`: set operands of a surface primitive

`set_index(`*`dom`* `surface, `*`positive integer`* `i, `*`any`* `val)`

- Replaces the `i`th operand of `surface` by the value `val`. See "Operands" above for a description of the operands of `surface`.
- If `i` is greater than 3, or if `val` is not an admissible value for the `i`th operand, then a warning message is issued. In this case the call of this method has no effect on the object `surface`.
- A call of this method sets the value of the attribute `refreshPlotdata` of `surface` to `TRUE`.

Method `slot`: read and write attributes and plot options

`slot(`*`dom`* `surface, `*`string`* `slotname)`

- Reads the value of the slot `slotname` of `surface`. `slotname` may either be the name of an attribute or the name of a plot option. See the tables of available plot options and attributes above.
- If `slotname` is the name of a plot option, but the option is not contained in the table stored under the attribute `options`, then `FAIL` is returned.

  If `slotname` is an invalid attribute or option, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `surface::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(surface, slotname)`.

`slot(`*`dom`* `surface, `*`string`* `slotname, `*`any`* `val)`

- Changes the value of the attribute or option with the name `slotname` to the value `val`.
- If there is no attribute or option with the name `slotname`, or if `val` is not an admissible value for `slotname`, then a warning message is issued. In this case, the value of `slotname` remains unchanged.
- This method overloads the system function `slot`, i.e., one may use it in the form `surface::slotname_id := val` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(surface, slotname, val)`.
- The value of the attribute `refreshPlotdata` of `surface` is set to `TRUE`.

Technical Methods

Method `checkOption`: check a plot option

```
checkOption(equation OptionName = value)
```

- This method checks whether `OptionName` is a known plot option for surface primitives (see the table of available plot options above), and `value` is an admissible value for this option.
- If both is correct, then the list `[TRUE, OptionName, newValue]` is returned. Note that the value of the option could have been converted into an admissible format. Thus, `newValue` must be used as the value of the option `OptionName` instead of `value`.
- Otherwise, the list `[FALSE, error_msg]` is returned. The string `error_msg` is a description of the located problem, which can be passed, for example, to the system function `error` to raise a user-specified exception.

Method `copy`: create a copy of a surface primitive

```
copy(dom surface)
```

- Returns a copy of the object `surface`.
- This method is called from the function `plot::copy`. See its help page for details.

Method `modify`: modify a copy of a surface primitive

```
modify(dom surface, equation(s) Name1 = value1, ...)
```

- Creates a copy of the object `surface` and changes the slots `Name1, ...` of this copy to the new values `value1, ...`
- The identifiers `Name1, ...` must be names of attributes or plot options of the domain `plot::Surface3d`. Otherwise a warning message is issued, and the slot remains unchanged. Also, if one of the values `value1, ...` is not an admissible value for the corresponding attribute or plot option, respectively, the change of the slot is ignored.

  See the tables of available options and attributes above.
- A call of this method sets the value of the attribute `refreshPlotdata` of the copy of `surface` to `TRUE`.
- This method is called from the function `plot::modify`.

Method `print`: print a surface primitive

```
print(dom surface)
```

- This method returns an unevaluated expression of the form `plot::Surface3d([x, y, z], u = a..b, v = c..d)`. It is used to print objects of `plot::Surface3d` to the screen.
- See the system function `print` for details.

Example 1. The following call returns an object representing a plot of the surface defined by $(u, v) \mapsto (u, \sin(v), \cos(v))$ with $(u, v) \in [0, 2\pi] \times [-1, 1]$:

```
>> s1 := plot::Surface3d(
     [u, sin(v), cos(v)], u = 0..2*PI, v = -1..1
   )

  plot::Surface3d([u, sin(v), cos(v)], u = 0..2 PI, v = -1..1)
```

To plot this surface in a graphical scene, call:

```
>> plot(s1)
```

Plot options of the surface can be given as additional parameters in the creating call, such as displaying the graph as an opaque object together with the parameter lines:

```
>> s2 := plot::Surface3d(
     [u, sin(v), cos(v)], u = 0..2*PI, v = -1..1,
     Style = [HiddenLine, Mesh]
   )

  plot::Surface3d([u, sin(v), cos(v)], u = 0..2 PI, v = -1..1)

>> plot(s2)
```

To change default values of some scene options, pass the scene options to the call of `plot` as additional arguments. For example, to change the style of the axes:

```
>> plot(s2, Axes = Corner)
```

See the help page of `plot::Scene` for available scene options.

Please refer to the examples of the help page of `plot::Function3d` about working with options and attributes.

Changes:

- `plot::Surface3d` is a new function.

`plot::Turtle` – graphical primitive for turtle graphics

`plot::Turtle()` creates a *turtle*, which is a drawing device which understands few simple commands.

Creating Elements:

- `plot::Turtle()`

Related Domains: `plot::Lsys`, `plot::Scene`

Related Functions: `plot`

Details:

- Objects generated by `plot::Turtle` represent a simple drawing device called a *turtle*. A turtle may be used to create 2-dimensional line drawings.
- A turtle has a position in the plane, a direction and a colour. It understands only few commands: Move forward and draw a line, move without drawing, turn left, turn right, change your colour. Further a turtle may remember its current state (position, direction and colour) by pushing it onto a stack and change its state to a former one by popping it off from the stack.
- One may display the path which a turtle has been taken since its creation using the function `plot`. One may also insert a turtle into a graphical scene of the `plot` library like any other graphical object.
- The state of a turtle may be changed by procedures which are called *methods* of the turtle. These methods can be accessed via the slot operator `::`. One method of a turtle for example has the name `line`. With the call `t::line(1)` one causes the turtle `t` to draw a line of length 1 in its current direction.

  The following methods are available for a turtle `t`:

- `t::color(c)` changes the actual colour of the turtle `t` to `c`. The colour must be given as a list of 3 real-valued numbers in the range between 0 and 1, similar to the other colour values of the plot library.

  The method returns the turtle.

- `t::left(deg)` changes the direction of the turtle `t`. It rotates left for `deg` degrees.

  The method returns the turtle.

- `t::right(deg)` changes the direction of the turtle `t`. It rotates right for `deg` degrees.

  The method returns the turtle.

- `t::line(len)` causes the turtle `t` to move forward in its current direction, drawing a line in its current colour. The length of the line is `len`.

  The method returns the turtle.

- `t::move(len)` causes the turtle `t` to move forward in its current direction without drawing a line. The length of the move is given by `len`.

  The method returns the turtle.

- `t::push()` causes the turtle `t` to save its current state, i.e., its position, direction and colour. The state is pushed onto a stack.

  The method returns the turtle.

- `t::pop()` restores the state of the turtle to the one which is currently stored on the stack. The top element of the stack is popped off the stack.

  The method returns the turtle.

- The initial state of a turtle after creation is the following: It is located at the origin, heading north in the direction of the point (0,1). Its colour is green.

- An object of `plot::Turtle` has type `"graphprim"`, i.e., if `o` is such an object, then the result of `type(o)` is the string `"graphprim"`.

Result of Evaluation: Evaluating an object of the domain type `plot::Turtle` returns itself.

Access Methods

Method `getPlotdata`: create the plot data of a turtle

```
getPlotdata(dom t)
```

- Returns a list of an inner list, where the inner list is a plot description of `t` in a `plot2d` and `plot3d` conforming syntax, respectively, i.e., it has the form `[Mode = List, [polygon(...)], ...]`.
- This method is called from `plot::Scene` to build the plot data of the graphical scene.

Method `slot`: read turtle methods

```
slot(dom t, string slotname)
```

- Reads the value of the slot `slotname` of `t`. `slotname` must be the name of a method of the turtle, see above.
- If `slotname` is an invalid method name, then an error message is issued.
- This method overloads the system function `slot`, i.e., one may use it in the form `t::slotname_id` (here, `slotname_id` must be the identifier corresponding to the string `slotname`), or in functional notation `slot(t, slotname)`.
- One may not change a turtles method or state using the slot function.

Technical Methods

Method `print`: print a turtle

```
print(dom t)
```

- This method returns a string representing the turtle. This string shows the turtles current position and direction, but not the other aspects of its state, like the path which has been taken by the turtle in the past. It is used to print objects of `plot::Turtle` to the screen.
- See the system function `print` for details.

Example 1. We create a turtle, let it draw a triangle and then show its path:

```
>> T := plot::Turtle():
   T::right(90): T::line(1):
   T::left(120): T::line(1):
   T::left(120): T::line(1):
   plot(T, Axes = None)
```

Example 2. We draw a star-like object:

```
>> T := plot::Turtle():
   for i from 1 to 36 do
       T::right(170); T::line(1)
   end_for:
   plot(T, Axes = None)
```

Changes:

- `plot::Turtle` used to be `Turtle`.
- Adapted to comply with the `plot` library.

`plot::contour` – generate contour and implicit plots

`plot::contour([x, y, z], u = a..b, v = c..d)` returns a contour plot of the surface defined by $(u,v) \mapsto \big(x(u,v); y(u,v); z(u,v)\big)$ with $(u,v) \in [a,b] \times [c,d]$.

Call(s):

- `plot::contour([x, y, z], u = a..b, v = c..d<, option1, option2, ...>)`

Parameters:

`x, y, z`	—	arithmetical expressions in `u` and `v`
`u, v`	—	identifiers
`option1, option2, ...`	—	plot option(s) of the form `option = value`, including the special plot options *`Colors`* and *`Contours`* (see below)

Return Value: an object of the domain type `plot::Group`.

Options:

`Colors`	—	either the list `[Flat<, color>]`, `[Height<, fromColor, toColor>]` or `[Curve,color1, ...]`, where `color`, `fromColor`, `toColor` and `color1, ...` are RGB color specifications, i.e., lists of three real numerical values between 0 and 1.
`Contours`	—	either an integer greater than two, or a list of the form $[r_1, \ldots, r_n]$ of real numerical values.

Related Functions: `plot`, `plot2d`, `plot::density`, `plot::implicit`

Details:

- Call `plot(...)` to display the contour plot created on the screen.
- The contour lines are drawn on the bottom of the viewing box by default. The result of `plot::contour` is a two-dimensional object, and the plot options `option1, option2, ...` must be valid plot options for two-dimensional graphical objects. See `plot2d` for details.

- With option `Style = Attached`, the contour lines are drawn with respect to the height of the surface which results in a graphical object of dimension three.

  Here, the plot options `option1, option2, ...` must be valid plot options for three-dimensional graphical objects. See `plot3d` for details.

- Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Option <*Colors*>:

- The default value for `color` is red, the default values for `fromColor` and `toColor` are yellow and red, respectively.

Option <*Contours*>:

- Giving in integer specifies the number of contour lines to be drawn. This is the default case using eight contour lines.

Example 1. The following call returns an object representing a contour plot of the surface defined by $(u, v) \mapsto (u, v, e^{uv})$ with $(u, v) \in [-1, 1] \times [-1, 1]$:

```
>> c:= plot::contour([x, y, exp(x*y)], x = -1..1, y = -1..1)

                                plot::Group()
```

To plot this object on the screen, call `plot`:

```
>> plot(c)
```

With the option `Style = Attached`, we get the following three-dimensional contour plot of the same surface:

```
>> plot(plot::contour(
     [x, y, exp(x*y)], x = -1..1, y = -1..1, Style = Attached
   ))
```

If you want to color the contour plot with respect to the height of the surface, you may enter:

```
>> plot(plot::contour(
     [x, y, exp(x*y)], x = -1..1, y = -1..1, Colors=[Height]
   ))
```

Here, the default color values from red to yellow are used.

Example 2. If you want to plot multiple contour plots in a single graphical scene, first create the desired contour plots, such as:

```
>> c1:= plot::contour(
       [x, y, sin(x*y)], x = -PI..PI, y = -PI..PI, Grid = [20,20]
   ):
   c2:= plot::contour(
       [x, y, x + 2*y], x= -PI..PI, y = -PI..PI, Colors=[Flat,RGB::Blue]
   ):
```

and collect them into a single graphical scene:

```
>> plot(c1, c2)
```

Example 3. We plot the implicit function defined by $\left(x^2+y^2\right)^3-\left(x^2-y^2\right)^2=0$:

```
>> plot(plot::contour(
     [x, y, (x^2 + y^2)^3 - (x^2 - y^2)^2], x = -1..1, y = -
1..1,
     Contours=[0], Grid=[20,20]
   ))
```

Anyway, you may prefer the function `plot::implicit` that is used to plot graphs of implicit functions and therefore usually yields better results:

```
>> plot(plot::implicit(
     (x^2 + y^2)^3 - (x^2 - y^2)^2, x = -1..1, y = -1..1
   ))
```

Changes:

- `plot::contour` used to be `plotlib::contourplot`.
- `plot::contour` is now part of the new plot library `plot`, and its calling syntax and the return value were changed.

`plot::copy` – create a copy of a graphical primitive

`plot::copy(o)` returns a copy of the graphical object `o`.

Call(s):

- `plot::copy(o)`

Parameters:

`o` — graphical object, i.e., an object of type `"graphprim"`

Return Value: an object of the same domain type as `o`.

Related Functions: `plot::modify`

Details:

- If a plot option of `o` is changed via the slot operator `::`, e.g., the color of `o` by calling `o::Color:= rgbvalue`, then the object `o` (and possibly the objects of that `o` consists) is changed due to the reference effect of domains (see Example 2). With `plot::copy` you can explicitly create a copy of `o` first, before changing plot options of this copy.

Example 1. We create an object representing a two-dimensional function plot:

```
>> f:= plot::Function2d(sin(x), x = 0..2*PI):
   plot(f)
```

If we want to add another graph to the same graphical scene, built of `f` by changing its term to the cosine function and its color to blue, we must first create a copy of `f` and then change the term attribute `term` and the options `Color` and `Title` as desired:

```
>> g:= plot::copy(f): g::term:= cos(x):
   g::Title := "cos(x)": g::Color:= RGB::Blue:
   plot(f, g)
```

Example 2. This example illustrate the reference effect for graphical objects. Let us create a scene consisting of three graphical objects:

```
>> s:= plot::Scene(
     plot::Function2d(1/x, x = 1..50),
     plot::Pointlist([n, sin(n)/n] $ n = 1..50, Color = RGB::Blue),
     plot::Function2d(-1/x, x = 1..50)
   ):
   plot(s)
```

If we want to increase the size of the points of the graph of the sequence $n \mapsto \frac{\sin(n)}{n}$, we may extract the corresponding graphical object of that scene:

```
>> p:= s[2]
```

```
plot::Pointlist()
```

and set the corresponding plot option `PointWidth` to the value 50:

```
>> p::PointWidth:= 50: plot(s)
```

Changes on the object `p` reflects changes on every object that consists of `p`, such as the graphical scene `s` in this example. This is called the "reference effect".

Changes:

- `plot::copy` is a new function.

`plot::cylindrical` – generate plots in cylindrical coordinates

`plot::cylindrical([rho, phi, z], u = a..b, v = c..d)` represents a plot of the surface defined by $(u,v) \mapsto (\rho(u,v);\phi(u,v);z(u,v))$ with $(u,v) \in [a,b] \times [u,v]$ in the cylindrical coordinates ρ, ϕ, z.

Call(s):

- `plot::cylindrical([rho, phi, z], u = a..b, v = c..d<, option1, option2>, ...)`

Parameters:

`rho, phi, z`	— arithmetical expressions in `u` and `v`
`u, v`	— identifiers
`a, b, c, d`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) for three-dimensional graphical objects

Related Domains: `plot::Surface3d`

Related Functions: `plot`, `plot3d`, `plot::spherical`, `plot::polar`

Return Value: a graphical object of the domain type `plot::Surface3d`.

Details:

- Call `plot(...)` to display the result on the screen.
- The following relationship between cylindrical coordinates and Cartesian coordinates holds:

$$x = \rho\cos\phi,\ y = \rho\sin\phi,\ z = z.$$

- The plot options `option1, option2, ...` must be valid plot options for three-dimensional graphical objects. See `plot::Surface3d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Example 1. We define a three-dimensional surface in cylindrical coordinates:

```
>> s:= plot::cylindrical(
     [1, u, v], u = -PI..PI,  v = -1..1, Grid = [20, 20]
   )

  plot::Surface3d([cos(u), sin(u), v], u = -PI..PI, v = -1..1)
```

and plot it on the screen:

```
>> plot(s, Axes = Box)
```

Example 2. Here we illustrate how to combine multiple cylindrical plots into a single graphical scene. We start by creating the two objects representing the cylindrical plots:

```
>> s1:= plot::cylindrical(
     [u, u, v], u = -PI..PI, v = -PI..PI,
     Grid = [30, 30], Color = [Height]
   );
   s2:= plot::cylindrical(
     [-u, u, v], u = -PI..PI, v = -2..2,
     Grid = [30, 30], Color = [Height]
   )

 plot::Surface3d([u cos(u), u sin(u), v], u = -PI..PI,

    v = -PI..PI)

 plot::Surface3d([-u cos(u), -u sin(u), v], u = -PI..PI,

    v = -2..2)
```

Then the next call plots these two objects in one graphical scene and sets the style of the axes to the value `Box`:

```
>> plot(s1, s2, Axes = Box)
```

Changes:

- `plot::cylindrical` used to be `plotlib::cylindricalplot`.
- `plot::cylindrical` is now part of the new plot library `plot`, and thus its calling syntax and the return value were changed.

`plot::data` – create two- and three-dimensional plots of data

`plot::data(format, datalist)` plots data specified in `datalist` in different formats, such as points, columns, beams or pie-charts.

Call(s):

- `plot::data(format, datalist<, option1, option2, ...>)`

Parameters:

`format`	— either *`Beam`*, *`Column`*, *`Curve`*, *`Line`*, *`Piechart2d`*, *`Piechart3d`*, *`Plain`*, *`Points`*, *`LinesPoints`*, *`CurvesPoints`* or *`Surface`*
`datalist`	— either list of numerical values, or a list of lists, where each inner list is a list of numerical values
`option1, option2, ...`	— plot option(s) of the form `option = value`, including the special plot option *`Colors`*

Options:

`Colors` — list of RGB color specifications, i.e., list of three real numerical values between 0 and 1.

Return Value: an object of the domain type `plot::Scene`.

Related Functions: `plot2d`, `plot3d`, `plot::Pointlist`, `plot::Polygon`

Details:

- Call `plot(...)` to display the plot of the data on the screen.
- Note that `plot::data` does not return a graphical object but a graphical scene!

- The plot options `option1, option2, ...` must be valid plot options for two or three-dimensional graphical scenes, respectively. See `plot2d` and `plot3d` for details.

Option <*Piechart2d* and *Piechart3d*>:

- A two- or three-dimensional piechart is drawn, respectively.
- You can specify a color for every object with the plot option *Colors*.

Option <*Plain* and *Surface*>:

- These plotting types are used to create a 3-d surface plotting from data. `datalist` must be a list of lists of numeric values. Each of the values in the list `[z_i_1,...,z_i_n]` are interpreted as the z-value corresponding to the y-value i and x-values `1,...,n`, i.e., the points `(1,i,z_1),...,(n,i,z_n)`. Consists the list of `m` lists `i=1,...,m`.
- *Surface* computes the 2-dimensional Lagrange interpolation polynomial to plot the surface. `datalist` is a list `[d_1,...,d_n]`.
- You can specify a color for every object with the plot option *Colors*.

Option <*Points*, *LinesPoints* and *CurvesPoints*>:

- With *Points* a point for each entry of `datalist` is drawn.
- With *LinesPoints*, these points are connected by lines.
- With *CurvesPoints*, these points are connected by cubic splines.
- You can specify a color for every object with the plot option *Colors*.

Option <*Line*, *Curve*, *Column* and *Beam*>:

- These plotting types are used to create a 2-d plotting from data. `datalist` must be a list of lists of numeric values. Each of the values in the list `[y_1,...,y_n]` are interpreted as the y-value corresponding to the x-values `1,...,n`, i.e., the points `(1,y_1),...,(n,y_n)` are plotted. The values of one list defines an object.
- *Curve* computes the Lagrange interpolation polynomial to plot the curve.
- You can specify a color for every object with the plot option *Colors*.

Example 1.

```
>> plot(plot::data(
       Piechart2d, [5,12,38,14,25]
   ))
>> plot(plot::data(
       Piechart3d, [5,12,38],
       Colors = [RGB::RoyalBlue, RGB::VioletRed, RGB::GreenPale]
   ))
```

Example 2.

```
>> plot(plot::data(
       Lines, [[5,10,24,-3], [6,5,2,18], [19,45,12,-10]],
       Colors = [RGB::Red, RGB::Green, RGB::Blue]
   ))
```

Example 3.

```
>> plot(plot::data(
       Columns, [[5,10,24,-3,6,5,2,18]]
   ))
>> plot(plot::data(
       Columns, [[5,10,24,-3], [6,5,2,18], [19,45,12,-10]],
       Colors = [RGB::Red, RGB::Green, RGB::Blue]
   ))
```

Example 4.

```
>> plot(plot::data(
       Beams, [[5,10,24,-3], [6,5,2,18], [19,45,12,-10]],
       Colors = [RGB::Red, RGB::Green, RGB::Blue],
       Axes = Box
   ))
```

Example 5.

```
>> plot(plot::data(
       Curves, [[5,10,24,-3], [6,5,2,18], [19,45,12,-10]],
       Colors = [RGB::Red, RGB::Green, RGB::Blue]
   ))
```

Example 6.

```
>> plot(plot::data(
       Surface, [[5,10,24,-3], [6,5,2,18], [19,45,12,-10]]
   ))
```

Example 7.

```
>> plot(plot::data(Plain,
       [[5,10,24,-3], [6,5,2,18], [19,45,12,-10]],
       Colors = [RGB::Red, RGB::Red, RGB::Green]
   ))
```

Changes:

- `plot::data` used to be `plotlib::dataplot`.
- `plot::data` is now part of the new plot library `plot`, and its calling syntax and the return value were changed.
- new plot formats: `Points`, `LinesPoints` and `LinesPoints`.

`plot::density` – generate two-dimensional density plots

`plot::density([x, y, z], u = a..b, v = c..d)` returns a density plot of the surface defined by $(u,v) \mapsto \big(x(u,v); y(u,v); z(u,v)\big)$ with $(u,v) \in [a,b] \times [c,d]$.

Call(s):

- `plot::density([x, y, z], u = a..b, v = c..d<, option1, option2, ...>)`

Parameters:

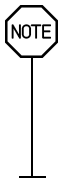
`x, y, z`	—	arithmetical expressions in `u` and `v`
`u, v`	—	identifiers
`option1, option2, ...`	—	plot option(s) of the form `option = value`

Return Value: an object of the domain type `plot::Scene`.

Related Functions: `plot`, `plot3d`, `plot::contour`

Details:

- Call `plot(...)` to display the density plot created on the screen.
- Note that `plot::density` does not return a graphical object but a graphical scene consisting of the surface as defined!
  Moreover, the scene returned is of dimension three, but the focal point and the camera point is set such that the plot looks like a two-dimensional plot.



- The plot options `option1, option2, ...` must be valid plot options for three-dimensional graphical scenes. See `plot3d` for details.

Example 1. The following call returns an object representing a density plot of the surface defined by $(u, v) \mapsto \left(u, v, \frac{1}{2}\left(\sin(u^2 + v^2)\right)\right)$ with $(u, v) \in [0, \pi] \times [0, \pi]$:

```
>> d:= plot::density([u, v, 1/2*sin(u^2 + v^2)], u = [0,PI], v = [0, PI] )
                                plot::Scene()
```

To plot this surface on the screen, call `plot`:

```
>> plot(d)
```

Changes:

- `plot::density` used to be `plotlib::densityplot`.
- `plot::density` is now part of the new plot library `plot`, and its calling syntax and the return value were changed.

`plot::HOrbital` – visualize the electron orbitals of a hydrogen atom

`plot::HOrbital(n, l, m)` yields a visualization of the hydrogen electron orbital with quantum numbers `n`, `l`, `m`.

Call(s):

- `plot::HOrbital(n, l, m, Option1, Option2, ...)`

Parameters:

`n` — the principal (energy) quantum number: a positive integer
`l` — the angular momentum quantum number: an integer between `0` and `n - 1`
`m` — the magnetic quantum number: an integer between `-l` and `l`

Options:

`Option1, Option2, ...` — options allowed in `plot::Surface3d`

Return Value: an object of type `plot::Surface3d`.

Details:

- See the 'Background' below for a physical interpretation of the surface generated by `plot::HOrbital`.
- `plot::HOrbital` generates a surface of type `plot::Surface3d`. It uses the default settings with one exception: a specialized color scheme is used to depict information on the radial part of the electron orbit (see "Background"). Using the `Color` option of `plot::Surface3d`, this internal color scheme may be overridden by a user-defined color.
- The surface generated by `plot::HOrbital` may be passed to the function `plot::Scene` to create a graphical scene. In the call to `plot::Scene`, you may specify scene options. Call `plot(...)` to display the scene.

  Alternatively, you can pass the surface directly to `plot` together with scene options.

Example 1. The following call yields a symbolic surface object:

```
>> orbit := plot::HOrbital(3, 2, 0)

 plot::Surface3d([cos(phi1) sin(theta1)

                     2       2
    (1.5 cos(theta1)  - 0.5) , sin(phi1) sin(theta1)

                     2       2
    (1.5 cos(theta1)  - 0.5) , cos(theta1)

                     2       2
    (1.5 cos(theta1)  - 0.5) ], theta1 = 0..PI, phi1 = 0..2 PI)
```

We pass this object to `plot` to render the object:

```
>> plot(orbit, Ticks = None)
```

With the *Grid*-Option of `plot::Surface3d`, a smoother surface is generated. The scene option `Axis = None` is used in `plot` to switch off the default box around the graphical scene:

```
>> orbit := plot::HOrbital(3, 2, 0, Grid = [30, 30],
                           Title = "quantum numbers: 3, 2, 0"):
   plot(orbit, Axes = None)
```

The internal coloring is replaced by a new coloring scheme:

```
>> orbit := plot::HOrbital(3, 2, 0, Grid = [30, 30],
                            Color = [Height]):
   plot(orbit)

>> delete orbit:
```

Background:

- The probability density of an electron is the absolute square of the quantum mechanical wave function $\psi(x, y, z)$. With the usual polar coordinates

$$\begin{aligned} x(r, \phi, \theta) &= r \cos(\phi) \sin(\theta) , \\ y(r, \phi, \theta) &= r \sin(\phi) \sin(\theta) , \\ z(r, \phi, \theta) &= r \cos(\theta) , \end{aligned}$$

  the wave function has the form $\psi(x, y, y) = R(r)\, Y(\phi, \theta)$. The surface is a visualization of the real part of the function $Y(\phi, \theta)$: the surface is defined by the parameterization

$$\begin{aligned} x(\phi, \theta) &= Re(Y(\phi, \theta))^2 \cos(\phi) \sin(\theta) , \\ y(\phi, \theta) &= Re(Y(\phi, \theta))^2 \sin(\phi) \sin(\theta) , \\ z(\phi, \theta) &= Re(Y(\phi, \theta))^2 \cos(\theta) . \end{aligned}$$

  The distance of a surface point to the origin is $Re(Y(\phi, \theta))^2$. Hence, the real part of the electron density is reflected by the shape of the surface: the "bulges" indicate high probabilities.

  The radial part $R(r)$ of the wave function $\psi(x, y, z) = R(r)\, Y(\phi, \theta)$ is only used in the coloring scheme of the surface: high values of $R(r)^2$ yield bright colors, small values yield dark colors. Red colors indicate points where the real part of the wave function $Re(\psi)$ is positive. Blue colors indicate negative values.

- The orbits corresponding to an energy index `n` are referred to as a "shell". Traditionally, the following shell symbols are used:

`n`	1	2	3	4	...
shell symbol	K	L	M	N	...

  The following symbols are associated with the angular momentum:

`l`	0	1	2	3	4	5	...
symbol	s	p	d	f	g	h	...

Changes:

- `plot::HOrbital` is a new function.

`plot::implicit` – implicit plot of smooth functions

`plot::implicit` is used to get a plot of $f = 0$ for a smooth f from $\mathbb{R}^2 \to \mathbb{R}$. f must be regular almost everywhere on this curve.

Call(s):

- `plot::implicit(expr, x=a..b, y=c..d<, options>)`
- `plot::implicit([expr, ...], x=a..b, y=c..d<, options>)`

Parameters:

`expr`	—	function(s) to plot, given as arithmetical expression(s) in two identifiers
`x, y`	—	identifiers used in `expr`
`a..b, c..d`	—	ranges to plot

Options:

`Grid` `= gridval`	—	grid division to use for finding starting points
`Colors` `= [col1, ...]`	—	colors used for plotting the components.
`MinStepsize` `= hmin`	—	minimum step-size for tracing a contour
`MaxStepsize` `= hmax`	—	maximum step-size for tracing a contour
`StartingStepsize` `= hstart`	—	step-size the iteration starts with
`Precision` `= eps`	—	precision of the Newton iteration
`Contours` `= [c1, ...]`	—	contours to plot
`Splines` `= Boolean`	—	If set to `TRUE`, the contours will be plotted with cubic splines; otherwise, straight lines will be used. Default: `FALSE`.
`Factor` `= Boolean`	—	If set to `TRUE`, each function will be factored prior to iterating. This may improve the results. Default: `FALSE`.

Return Value: a graphical object of the domain type `plot::Group`.

Related Functions: `plot::contour`, `plotfunc2d`, `plot2d`

Details:

- `plot::implicit` plots $f = 0$ by a curve tracking method. For this, it first generates start points with a Newton iteration starting at grid points (the number of grid points can be controlled with the option *`Grid`*) and then iteratively applies the implicit function lemma to get a local approximation to the curve. This approximation is then improved with another Newton step. On hitting a point where f is not regular, the iteration stops.

Option <*`Grid`* `= gridval`>:

- `gridval` must either be a list of two positive integers, the number of divisions in the two coordinates, or a single integer, which is equivalent to repeating this integer twice.
- Increase this number if you suspect not all components are found. The default is `[5, 5]`.

Option <*`Colors`* `= [col1, ...]`>:

- The colors used for plotting the components. A list is expected, each element of which must be a valid argument for the `Color=[Flat,...]` option of `plot2d`.
- Default: `[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0], [1.0, 1.0, 0.0], [1.0, 0.0, 1.0], [0.0, 1.0, 1.0], [1.0, 1.0, 1.0]]`.

Option <*`MinStepsize`* `= hmin`>:

- The minimum step-size for the iteration following a contour. This is a lower limit for the adaptive iteration to avoid spending a lot of time without real progress.
- The default is 1/1.000.000 of the minimum of width or height of the area under consideration.
- Increase this number if you think the algorithm gets stuck in a place unimportant to your application.

Option <`MaxStepsize = hmax`>:

- `MaxStepsize` is complementary to `MinStepsize` in that it defines the maximum step-width for the iteration.
- If this value is set too high, the algorithm may skip over details of the curves; if it is set lower than needed, the computation takes longer.
- The default is $1/100$ of the shorter of width and height.

Option <`StartingStepsize = hstart`>:

- The step-size the iteration starts with. The default is $1/1.000$ of the shorter side of the area.

Option <`Precision = eps`>:

- This floating-point value indicates the relative precision which should be achieved in the Newton iteration.
- Lower values may lead to more accurate results, but slow down the computation. Values smaller than $10.0^{-DIGITS}$ may cause the function not to return.
- The default is $10.0^{2-DIGITS}$.

Option <`Contours = [c1, ...]`>:

- A list of values for which the functions implicitly defined by $f(x, y) = c_i$ should be plotted.
- Default: `[0]`.

Example 1. Let's have a look at elliptic curves:

```
>> plot(
     plot::implicit((x^3 + x + 2) - y^2,
                    x = -5..5, y = -5..5),
     Scaling=Constrained
   )
```

Example 2. To demonstrate how to plot multiple implicit functions, we plot $y = x^2$, $y = x$ and $x = y^2$:

```
>> s:= plot::implicit(
     [x^2 - y, x - y, x - y^2], x = -4..4, y = -4..4
   ):
   plot(s)
```

Example 3. We plot the family $x = y^2 + c$ for $c \in [-5, 5]$:

```
>> p:= plot::implicit(
     y^2 - x, x = -1..25, y = -5..5, Contours = [$-5..5]
   ):
   plot(p, Axes = Origin)
```

Example 4. `plot::implicit` handles quite complex expressions. In the following example, the circle around the origin is left out by many similar tools:

```
>> plot(
     plot::implicit((1-0.99*exp(x^2+y^2))*(x^10-1-y),
                    x=-1.25..1.25,y=-1.1..2)
   ):

>> F2 := (x,y) -> x^4*y^4+sin(x)*cos(y)-(x-1)*(y-2)*exp(-x^2):
   plot(plot::implicit(F2(x,y),x=-10..10,y=-10..10)):

>> delete F2:
```

Example 5. In some cases, `DIGITS` must be increased to get a correct result. In the following example, problems occur around the origin with the default setting of `DIGITS` when a small region is to be displayed. First, we display the whole picture:

```
>> F3 := (x,y) -> y*(3*x^2-y^2)-(x^2+y^2)^2:
   plot(plot::implicit(F3(x, y), x = -1..1, y = -1.3..0.7)):
```

Near the origin, numeric cancellation occurs. If you try to depict a small area around the origin of the above curve, you need to increase `DIGITS`:

```
>> delete DIGITS:
   plot(
     plot::implicit(F3(x, y), x = -0.005..0.005, y = -0.005..0.005)
   ):
```

```
>> DIGITS := 15:
   plot(
     plot::implicit(F3(x, y), x = -0.005..0.005, y = -0.005..0.005)
   ):
   delete DIGITS:
```

Example 6. We plot $sin(5 * sin(x) * y) = 0$. This is an example where multiple implicit functions are found. With a low setting of `DIGITS`, strange artefacts occur:

```
>> DIGITS := 50:
   plot(
     plot::implicit(sin(5*sin(x)*y), x = -5..5, y = -5..5)
   ):
   delete DIGITS:
```

Background:

- Curve Tracking algorithms are usually found in numerics to track stable and instable manifolds of dynamic systems and in homotopy methods for finding roots of highly complicated functions.

Changes:

- `plot::implicit` is a new function.

`plot::inequality` – generate a 2D plot of inequalities

`plot::inequality([f1, f2,...], left..right, bottom..top)` serves for displaying points (x, y) in the rectangle $Q = [left, right] \times [bottom, top]$ satisfying the inequalities

$$f_1(x, y) \geq 0 \quad \text{and} \quad f_2(x, y) \geq 0 \quad \text{and} \quad \ldots \qquad (1)$$

Call(s):

- `plot::inequality([f1, f2, ...], left..right, bottom..top <, n> <Colors = [c1, c2, c3]>)`

Parameters:

`f1, f2, ...`	— real valued functions of two variables: procedures
`left, right, bottom, top`	— real numerical values
`n`	— a nonnegative integer determining the mesh size. The default value is 6.

Options:

`Colors` `= [c1, c2, c3]`	— each of the colors `c1`, `c2`, `c3` must be an RGB specification, i.e., a list of three real numerical values between 0 and 1. The default colors are `c1 = RGB::Green`, `c2 = RGB::Yellow`, `c3 = RGB::Red`.

Return Value: an object of the domain type `plot::Group`.

Details:

- The rectangle Q is divided into $2^n \times 2^n$ subrectangles. With the default value `n = 6`, the drawing area is divided into 64×64 subrectangles. This default produces a rather "discretized" plot. "Smoother" plots are generated by larger values of `n`. Note, however, that increasing `n` by 1 may increase the run time by a factor of 4.
- A subrectangle is displayed with the color `c1` if all its points (x, y) satisfy $f_1(x, y) > 0$ and $f_2(x, y) > 0$ etc. Consequently, all points of this color are guaranteed to satisfy (1).
- A subrectangle is displayed with the color `c3` if there is at least one function f_i such that all points in the subrectangle satisfy $f_i(x, y) < 0$. Consequently, all points of this color are guaranteed to violate (1).
- The remaining subrectangles are displayed with the color `c2`. They cover the boundary of the region defined by the inequalities (1).
- An object generated by `plot::inequality` may be passed to the function `plot::Scene` to create a graphical scene. In the call to `plot::Scene`, you may specify scene options. Call `plot(...)` to display the scene.

  Alternatively, if the scene consists of only one "inequality object", you can pass this object directly to `plot` together with scene options.
- Interval arithmetic is used to check the inequalities. The input functions must be suitable for this kind of arithmetic, i.e., the calls `f1(Dom::Interval(left..right), Dom::Interval(bottom..top))` etc. must produce valid intervals. In MuPAD, interval implementations exist for most of the elementary functions such as `sin`, `exp`, `ln` etc. However, special functions such as Bessel functions, polylogarithms etc. must not turn up in `f1, f2, ...`.

Example 1.

```
>> f1:= (x,y) -> x^2 + y^2 - 1:
   p1:= plot::inequality([f1], -1..1, -1..1, 5)

                          plot::Group()

>> plot(p1, Scaling = Constrained, Axes = Box)

>> f2:= (x,y) -> cos(x) - y: f3:= (x,y) -> cos(x) + y:
   p23:= plot::inequality([f2, f3], -PI..PI, -2..2, 5)

                          plot::Group()

>> plot(p23, Scaling = Constrained, Axes = Box)

>> p123:= plot::inequality(
            [f1, f2, f3], -2..2, -1..1, 5,
            Colors = [RGB::Red, RGB::Black, RGB::White])

                          plot::Group()

>> plot(p123, Scaling = Constrained, Axes = Box)
```

Changes:

- `plot::inequality` is a new function.

`plot::line` – graphical object for lines

`plot::line(p1, p2)` returns a polygon connecting the point p_1 with the endpoint p_2.

Call(s):

- `plot::line(p1, p2<, option1, option2, ...>)`

Parameters:

`p1, p2`	—	points, i.e., objects of domain type `plot::Point` or lists of two or three arithmetical expressions, respectively
`option1, option2, ...`	—	plot option(s) of the form `option = value`

Return Value: a graphical object of the domain type `plot::Polygon`.

Related Functions: `plot`, `plot2d`, `plot3d`, `plot::Polygon`

Details:

- Use the function `plot` to display the result on the screen.
- The plot options `option1, option2, ...` must be valid plot options for two- or three-dimensional graphical objects, respectively. See `plot2d` and `plot3d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Example 1. We define a line starting at point (1;2) and ending at point (3;−1):

```
>> l := plot::line([1, 2], [3, -1])
                          plot::Polygon()
```

To plot this line, call `plot`:

```
>> plot(l)
```

Use `expose` to expose the points of a line:

```
>> expose(l)
      plot::Polygon(plot::Point(1, 2), plot::Point(3, -1))
```

Example 2. To change the color of the line from the previous example, enter:

```
>> l::Color := RGB::Green: plot(l, Axes = Box)
```

Here the scene option `Axes = Box` was given.

Changes:

- `plot::line` is a new function.

`plot::modify` – create modified copies of graphical objects

`plot::modify(o, option1, option2, ...)` creates a copy of the graphical objects `o` and sets the plot options given in `option1, option2, ...` to this copy.

Call(s):

- `plot::modify(o<, option1, option2>, ...)`

Parameters:

`o`	— graphical scene, or a graphical object (i.e., an object of type `"graphprim"`)
`option1, option2, ...`	— plot option(s) of the form `option = value`

Return Value: object of the domain type as `o`.

Related Functions: `plot::copy`

Details:

- If a plot option of `o` is changed via the slot operator `::`, e.g., the color of `o` by calling `o::Color:= rgbvalue`, then the object `o` (and possibly the objects of that `o` consists) is changed due to the reference effect of domains.

  With `plot::modify` you can directly change plot options of a copy of `o`.

- The plot options `option1, option2, ...` must be valid plot options for two- or three-dimensional graphical scenes, respectively. See `plot2d` and `plot3d` for details.

  Note that plot options for graphical objects are not allowed! You may give such options as optional arguments to the corresponding function call creating the object `oi` ($i = 1, 2, \ldots$).

Example 1. We create an object representing a graph of a two-dimensional function:

```
>> f:= plot::Function2d(sin(x), x = 0..2)

                plot::Function2d(sin(x), x = 0..2)
```

Then the following call creates a copy of the object `f` and changes its color to blue and the line width to the value 12:

```
>> g:= plot::modify(f, Color = RGB::Blue, LineWidth = 12)

                plot::Function2d(sin(x), x = 0..2)

>> f::Color, f::LineWidth, g::Color, g::LineWidth

   [Flat, [1.0, 0.0, 0.0]], FAIL, [Flat, [0.0, 0.0, 1.0]], 12
```

Changes:

- `plot::modify` is a new function.

`plot::ode` – plot the numerical solution of an ordinary differential equation

`plot::ode([t0, t1, ...], f, Y0, [G])` computes a mesh of numerical sample points $Y(t_0), Y(t_1), \ldots$ representing the solution $Y(t)$ of the first order differential equation (dynamical system)

$$\frac{dY}{dt} = f(t, Y)\,, \quad Y(t_0) = Y_0\,, \quad t_0, t \in R\,, \quad Y_0, Y(t) \in C^n\,.$$

The procedure

$$G:\ (t, Y) \to [x(t, Y), y(t, Y)] \quad \text{or} \quad G:\ (t, Y) \to [x(t, Y), y(t, Y), z(t, Y)]$$

maps these solution points $(t_i, Y(t_i))$ in $R \times C^n$ to a mesh of 2D plot points $[x_i, y_i]$ or 3D plot points $[x_i, y_i, z_i]$, respectively,

Call(s):

```
plot::ode([t0, t1, ...], f, Y0, <method,>
          <RelativeError = tol,> <Stepsize = h,>
          [G1 <, Style = style1> <, Color = c1>],
          [G2 <, Style = style2> <, Color = c2>],
          ...  )
```

Parameters:

`t0, t1, ...`	— the time mesh: real numerical values. If data are displayed with `Style = Splines`, these values must be in ascending order.
`f`	— the vector field of the ODE: a procedure. See `numeric::odesolve`.
`Y0`	— the initial condition of the ODE: a list or a 1-dimensional array. See `numeric::odesolve`.
`G1, G2, ...`	— "generators of plot data": procedures mapping a solution point `(t, Y(t))` to a list `[x, y]` or `[x, y, z]` representing a plot point in 2D or 3D, respectively.

Options:

`method`	— use a specific numerical scheme (see `numeric::odesolve`)
`RelativeError` `= tol`	— sets a numerical discretization tolerance (see `numeric::odesolve`)
`Stepsize` `= h`	— sets a constant stepsize (see `numeric::odesolve`)
`Style` `= style`	— sets the style in which the plot data are displayed. The following styles are available: *`Points`*, *`Lines`*, *`Splines`*. The default style is *`Lines`*.
`Color` `= c`	— sets the RGB color `c` in which the plot data are displayed. The default color is `RGB::Black`.

Return Value: a graphical object of domain type `plot::Group`.

Related Functions: `numeric::odesolve`, `plot`, `plot::Group`, `plot::Scene`

Details:

- Internally, a sequence of numerical sample points

  `Y1 := numeric::odesolve(t0..t1, f, Y0 <, Options>)`,

  `Y2 := numeric::odesolve(t1..t2, f, Y1 <, Options>)` etc.

  is computed where `Options` is some combination of `method`, *`RelativeError`*`= tol`, and *`Stepsize`*`= h`. See `numeric::odesolve` for details on the vector field procedure `f`, the initial condition `Y0`, and the options.

- Each of the "generators of plot data" `G1`, `G2` etc. creates a graphical solution curve from the sample points `Y0`, `Y1` etc. Each generator `G`, say, is internally called in the form `G(t0, Y0)`, `G(t1, Y1)`, `...` to produce a sequence of plot points in 2D or 3D.

  The solver `numeric::odesolve` returns the solution points `Y0`, `Y1` etc. as lists or 1-dimensional arrays (the actual type is determined by the initial value `Y0`). Consequently, each generator `G` must accept two arguments `(t, Y)`: `t` is a real parameter, `Y` is a "vector" (either a list or a 1-dimensional array).

  Each generator must return a list with 2 or 3 elements representing the (x, y) or (x, y, z) coordinates of the graphical point associated with a solution point `(t, Y)` of the ODE. All generators must produce graphical data of the same dimension, i.e., either 2D data as lists with 2 elements, or 3D data as lists with 3 elements.

  Some examples:

`G := (t, Y) -> [t, Y[1]]` creates a 2D plot of the first component of the solution vector along the y-axis, plotted against the time variable `t` along the x-axis

`G := (t, Y) -> [Y[1], Y[2]]` creates a 2D phase plot, plotting the first component of the solution along the x-axis and the second component along the y-axis. The result is a solution curve in phase space (parametrized by the time `t`).

`G := (t, Y) -> [Y[1], Y[2], Y[3]]` creates a 3D phase plot of the first three components of the solution curve.

- Note that arbitrary values associated with the solution curve may be displayed graphically by an appropriate generator `G`. Cf. example 2.
- Several generators `G1`, `G2` etc. can be specified to generate several curves associated with the same numerical mesh `Y0`, `Y1`, `...`. E.g., one can use a generator twice to plot the data with the options *`Style = Splines`* and *`Style = Points`*. This allows to display a smooth solution curve together with the sample points. Cf. examples 1, 2, and 3.
- `plot::ode` returns a graphical object of type `plot::Group` containing all graphical solution curves specified by the generators `G1`, `G2` etc. This object can be combined with other plot objects to a graphical scene via `plot::Scene`. Cf. example 3. A final call to `plot` calls the renderer to display the scene.

Option <*`Color`* `= c`>:

- This option sets the color in which the graphical data are displayed. The RGB color `c` must be a list of three numerical real values `[r, g, b]` between `0` and `1` representing the red, green and blue contributions according to the RGB color model. A variety of colors is provided by MuPAD's `RGB` data structure.

Option <*`Style`* `= style`>:

- The graphical data produced by each of the generators `G1`, `G2` etc. consists of a sequence of mesh points in 2D or 3D, respectively.

  With *`Style = Points`*, the graphical data are displayed as a discrete set of points.

  With *`Style = Lines`*, the graphical data points are displayed as a curve consisting of straight line segments between the sample points. The points themselves are not displayed.

  With *`Style = Splines`*, the graphical data points are displayed as a smooth spline curve connecting the sample points. The points themselves are not displayed.

Example 1. The following procedure `f` together with the initial value `Y0` represent the initial value problem $dY/dt = f(t, Y) = tY - Y^2$, $Y(t_0) = 1$. In MuPAD, the function Y is represented by a list with one element:

```
>> f := (t, Y) -> [t*Y[1] - Y[1]^2]: Y0 := [1]:
```

The solution is to be plotted as a function of the time `t`. We define an appropriate generator for the plot data:

```
>> G := (t, Y) -> [t, Y[1]]:
```

The numerical solution is to consist of sample points over the time mesh $t_i = i$, $i = 0, 1, \ldots, 10$. The generator `G` is used twice with different *`Style`* options. This generates the sample points together with a smooth spline curve connecting these points:

```
>> p := plot::ode([i $ i = 0..10], f, Y0,
                  [G, Style = Points, Color = RGB::Blue],
                  [G, Style = Splines, Color = RGB::Red])

                          plot::Group()
```

The point size of the point objects inside `p` is increased:

```
>> p:= plot::modify(p, PointWidth = 50):
```

The object `p` is converted to a graphical scene with various scene options:

```
>> s := plot::Scene(p, Labels = ["t", "y"],
          Ticks = [Steps = [1.0, 1], Steps = [1.0, 1]],
          GridLines = Automatic)

                          plot::Scene()
```

Finally, the scene is rendered by a call to `plot`:

```
>> plot(s)
```

```
>> delete f, Y0, G, p, s:
```

Example 2. We consider the nonlinear oscillator $y'' + y^3 = \sin(t)$, $y(0) = 0$, $y'(0) = 0.5$. As a dynamical system for $Y = [y, y']$, we have to solve the following initial value problem $dY/dt = f(t, Y)$, $Y(0) = Y_0$:

```
>> f := (t, Y) -> [Y[2], sin(t) - Y[1]^3]: Y0 := [0, 0.5]:
```

The following generator produces a phase plot in the (x, y) plane, embedded in a 3D plot:

```
>> G1 := (t, Y) -> [Y[1], Y[2], 0]:
```

Further, we use the z coordinate of the 3D plot to display the value of the "energy" function $E = y^2/2 + y'^2/2$ over the phase curve:

```
>> G2 := (t, Y) -> [Y[1], Y[2], (Y[1]^2 + Y[2]^2)/2]:
```

The phase curve in the (x, y) plane is combined with the graph of the energy function:

```
>> p := plot::ode([i/5 $ i = 0..100], f, Y0,
                  [G1, Style = Splines, Color = RGB::Red],
                  [G2, Style = Points, Color = RGB::Black],
                  [G2, Style = Lines, Color = RGB::Blue]):
```

We increase the size of the points used in the representation of the energy:

```
>> p:= plot::modify(p, PointWidth = 40):
```

The renderer is called:

```
>> plot(p, Axes = Box, Labels = ["y", "y'", "E"],
          CameraPoint = [10, -15, 5]):

>> delete f, Y0, G1, G2, p:
```

Example 3. We consider the initial value problem $y' = f(t, y) = t \sin(t + y^2)$, $y(0) = 1$.

```
>> f := (t, y) -> t*sin(t + y^2) - y:
```

The following vector field is tangent to the solution curves:

```
>> p1 := plot::vectorfield([1, f(t, y)], t = 1..3, y = 0..1,
                           Grid = [10, 5], Color = RGB::Black):
```

The following object represents the plot of the solution as a function of `t`:

```
>> p2 := plot::ode(
      [i/3 $ i=0..12], (t,Y) -> [f(t, Y[1])], [0],
      [(t, Y) -> [t, Y[1]], Style = Points, Color = RGB::Red],
      [(t, Y) -> [t, Y[1]], Style = Splines, Color = RGB::Blue]):
```

We increase the point size:

```
>> p2:= plot::modify(p2, PointWidth = 40):
```

Finally, we combine the vector field and the ODE plot to a scene and call the renderer:

```
>> plot(p1, p2, Scaling = Constrained, Labels = ["t", "y"],
           GridLines = [Steps = 0.25, Steps = 0.25],
           Ticks = [Steps = 1.0, Steps = 0.5]):

>> delete f, p1, p2:
```

Example 4. The Lorenz ODE is the system

$$\frac{d}{dt}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} p(y-x) \\ -xz+rx-y \\ xy-bz \end{pmatrix}$$

with fixed parameters p, r, b. As a dynamical system for $Y = [x, y, z]$, we have to solve the ODE $dY/dt = f(t, Y)$ with the following vector field:

```
>> f := proc(t, Y)
       local x, y, z;
       begin
          [x, y, z] := Y:
          [p*(y - x), -x*z + r*x - y, x*y - b*z]
       end_proc:
```

We consider the following parameters and the following initial condition `Y0`:

```
>> p := 10: r := 28: b := 1: Y0 := [1, 1, 1]:
```

The following generator produces a 3D phase plot:

```
>> Gxyz :=  (t, Y) -> Y:
```

The following generator projects the solution curve to the (y, z) plane with $x = -20$:

```
>> Gyz := (t, Y) -> [-20, Y[2], Y[3]]:
```

The following generator projects the solution curve to the (x, z) plane with $y = 30$:

```
>> Gxz := (t, Y) -> [Y[1], 30, Y[3]]:
```

The following generator projects the solution curve to the (x, y) plane with $z = 0$:

```
>> Gxy := (t, Y) -> [Y[1], Y[2], 0]:
```

With these generators, we create a 3D plot object consisting of the phase curve and its projections. The following call is somewhat time consuming, because it calls the numerical integrator to produce the graphical data:

```
>> obj := plot::ode([i/10 $ i=1..100], f, Y0,
           [Gxyz, Style = Splines, Color = RGB::Red],
           [Gyz, Style = Splines, Color = RGB::LightGrey],
           [Gxz, Style = Splines, Color = RGB::LightGrey],
           [Gxy, Style = Splines, Color = RGB::LightGrey]):
```

Finally, the plot is rendered. Again, this call is somewhat time consuming, because internally, several thousand calls to spline functions occur:

```
>> plot(obj, Axes = Box, Ticks = 4,
           CameraPoint = [400, -800, 1200]):
```

```
>> delete f, p, r, b, Y0, Gxyz, Gyz, Gxz, Gxy, obj:
```

Changes:

- `plot::ode` is a new function.

`plot::polar` – generate plots in polar coordinates

`plot::polar([f1, f2], phi = pmin..pmax)` represents a plot of the curve defined by $\phi \mapsto \big(f_1(\phi); f_2(\phi)\big)$ with $\phi \in [\phi_{\min}, \phi_{\max}]$ in polar coordinates.

Call(s):

- `plot::polar([f1, f2], phi = pmin..pmax<, option1, option2>, ...)`

Parameters:

`f1, f2`	—	arithmetical expressions in `phi`
`phi`	—	identifier (the angle)
`pmin, pmax`	—	arithmetical expressions
`option1, option2, ...`	—	plot option(s) for two-dimensional graphical objects

Return Value: a graphical object of the domain type `plot::Curve2d`.

Related Functions: `plot2d`, `plot::spherical`, `plot::cylindrical`

Details:

- Call `plot(...)` to display the result on the screen.

- The plot options `option1, option2, ...` must be valid plot options for two-dimensional graphical objects. See `plot2d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Example 1.

```
>> delete phi:
   c:= plot::polar([phi, phi], phi = [0, 4*PI], Grid = [100])

   plot::Curve2d([phi cos(phi), phi sin(phi)], phi = 0..4 PI)

>> plot(c, Axes = None)
```

Changes:

- `plot::polar` used to be `plotlib::polarplot`.
- `plot::polar` is now part of the new plot library `plot`, and thus its calling syntax and the return value were changed.

`plot::spherical` – generate plots in spherical coordinates

`plot::spherical([theta, phi, r], u = a..b, v = c..d)` represents a plot of the surface defined by $(u,v) \mapsto (\theta(u,v); \phi(u,v); r(u,v))$ with $(u,v) \in [a,b] \times [c,d]$ in the spherical coordinates θ, ϕ, r.

Call(s):

- `plot::spherical([theta, phi, r], u = a..b, v = c..d))`

Parameters:

`theta, phi, r`	—	arithmetical expressions in `u` and `v`
`u, v`	—	identifiers
`a, b, c, d`	—	arithmetical expressions
`option1, option2, ...`	—	plot option(s) for three-dimensional graphical objects

Return Value: Call `plot(...)` to display the result on the screen.
a graphical object of the domain type `plot::Surface3d`.

Related Domains: `plot::Surface3d`

Related Functions: `plot`, `plot2d`, `plot::cylindrical`, `plot::polar`

Details:

- The following relationship between spherical coordinates and Cartesian coordinates holds:

$$x = r\cos\phi\sin\theta,\ y = r\sin\phi\sin\theta,\ z = r\cos\theta.$$

- The plot options `option1, option2, ...` must be valid plot options for three-dimensional graphical objects. See `plot::Surface3d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Example 1. We define a three-dimensional surface in spherical coordinates:

```
>> s:= plot::spherical(
     [1, u, z], u = -PI..PI,  z = -1..1,  Grid = [20, 20]
   )

 plot::Surface3d([cos(u) sin(z), sin(u) sin(z), cos(z)],

    u = -PI..PI, z = -1..1)

>> plot(s, Axes = Box)
```

Example 2. We plot the surface $(\phi, \theta) \mapsto (1, \phi, \theta)$ in $[-\pi, \pi] \times [0, \pi]$ (setting the number of surface points to 20):

```
>> delete phi, theta:
   plot(plot::spherical(
     [1, phi, theta], phi = -PI..PI,  theta = 0..PI, Grid = [20, 20]
   ), Axes = Box)
```

Changes:

- `plot::spherical` used to be `plotlib::sphericalplot`.
- `plot::spherical` is now part of the new plot library `plot`, and thus its calling syntax and the return value were changed.

`plot::vector` – graphical object for vectors

`plot::vector(p1, p2)` returns a polygon consisting of three lines. Theses lines are connecting the points p_1 with p_2 and the endpoints of the arrow with the point p_2.

Call(s):

- `plot::vector(p1, p2, angle, l<, option1, option2, ...>)`

Parameters:

`p1, p2`	—	points with real valued coordinates, i.e., objects of the domain type `plot::Point` or lists of two or three real numerical values, respectively
`angle, l`	—	real numerical values
`option1, option2, ...`	—	plot option(s) of the form `option = value`

Return Value: a graphical object of the domain type `plot::Polygon`.

Related Functions: `plot`, `plot2d`, `plot3d`, `plot::Polygon`

Details:

- `angle` specifies the angle (in radian measure) between the line and the arrow.
- `l` specifies the length of the arrow.
- Use the function `plot` to display the result on the screen.
- The plot options `option1, option2, ...` must be valid plot options for two- or three-dimensional graphical objects, respectively. See `plot2d` and `plot3d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Example 1. We define a vector starting at $(1;2)$ and ending at $(3;-1)$:

```
>> v := plot::vector([1, 2], [3, -1], 0.2, 0.1)
                                    plot::Polygon()
```

The lenght of the arrow is 0.1, and the angle between the line and the arrow is 0.2 (in radian measure).

To plot this vector, call `plot`:

```
>> plot(v)
```

To change the color of the vector from the previous example, enter:

```
>> v::Color := RGB::Blue: plot(v, Axes = None)
```

Here the scene option `Axes = None` was given.

Changes:

- `plot::vector` is a new function.

`plot::vectorfield` – generate plots of two-dimensional vector fields

`plot::vectorfield([v1, v2], x = a..b, y = c..d)` represents a plot of the vectorfield defined by $(x, y) \mapsto \big(v_1(x, y); v_2(x, y)\big)$ with $(x, y) \in [a, b] \times [c, d]$.

Call(s):

- `plot::vectorfield([v1, v2], x = a..b, y = c..d <, option1, option2, ...>)`

Parameters:

`v1, v2`	—	arithmetical expressions in `x` and `y` (the x- and y-component of the vectorfield)
`x, y`	—	identifiers
`a, b, c, d`	—	real numerical values
`option1, option2, ...`	—	plot option(s) for two-dimensional graphical objects

Related Functions: `plot`, `plot2d`

Details:

- Call `plot(...)` to display the result on the screen.
- The plot options `option1, option2, ...` must be valid plot options for two-dimensional graphical objects. See `plot2d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

- The form of the arrows used for displaying the vectors is built for the scene option `Scaling = Constrained`.

  For `Scaling = UnConstrained`, you can resize the plot window and change the scaling.

- The directions of the vectors change in the correct way. However, the tips of the arrows will be distorted and may look strange (this is unavoidable)

  So the user is advised to use `Scaling = Constrained` for a nice graphical appearance of the arrows, whenever this is appropriate.

  However, if the scale ratio $\frac{y_{max}-y_{min}}{x_{max}-x_{min}}$ of the viewing box is not close to one, then `Scaling = Constrained` is usually not appropriate. If you do not like the arrows, then you must re-parameterize the vector field such that the scale ratio in the new $x-y$ parameters is close to one.

Example 1. We demonstrate a plot of the vector field $v(x,y) = (1, \sin(x) + \cos(y))$.

```
>> DIGITS:=5:
   field:= plot::vectorfield(
     [1,sin(x)+cos(y)], x = 0..3, y = 1..2,
     Grid=[30,20], Color = [Flat, RGB::Red]
   )

                              plot::Group()
```

It is the directional field associated with the ode $y'(x) = \sin(x) + \cos(x)$. We insert a curve representing the numerical solution of this ode into this plot. We compute the numerical solution of $y'(x) = \sin(x) + \cos(y)$, $y(1) = 1.2$ via `numeric::odesolve`. `DIGITS` is increased to get rich sample of points:

```
>> DIGITS:=20:
   f:= (x,y) -> [sin(x)+cos(y[1])]:
   data:= numeric::odesolve(1..2, f, [1.2], Alldata):
```

```
>> curve:= plot::Polygon(
     (
       [data[i][1], data[i][2][1]], [data[i+1][1], data[i+1][2][1]]
     ) $ i=1..nops(data)-1,
     Color = RGB::Blue
   )
```

```
                        plot::Polygon()
```

Define a circle with center $(1.5; 1.5)$ and radius $\frac{1}{2}$:

```
>> circle:= plot::Ellipse2d([1.5,1.5], 1/2, 1/2 ):
```

We plot the three objects in a single graphical scene:

```
>> plot(field, circle, curve,
       Axes = Box, Labeling = TRUE, Scaling = Constrained
   )
```

Compare the plot when setting `Scaling = UnConstrained`:

```
>> plot(field, circle, curve,
     Axes = Box, Labeling = TRUE, Scaling = UnConstrained
   )
```

Changes:

- `plot::vectorfield` used to be `plotlib::fieldplot`.
- `plot::vectorfield` is now part of the new plot library `plot`, and its calling syntax and the return value were changed.

`plot::xrotate` – generate plots of surface of revolution (x-axis)

`plot::xrotate(f, x = a..b)` returns the surface of revolution defined by the function $f(x)$ around the x-axis in the interval $[a, b]$. The rotation angle ranges from 0 to 2π.

`plot::xrotate(f, x = a..b, Angle = r1..r2)` returns the surface of revolution defined by the function $f(x)$ around the x-axis in the interval $[a, b]$. The rotation angle ranges from r_1 to r_2.

Call(s):

- `plot::xrotate(f, x = a..b<, option1, option2>, ...)`
- `plot::xrotate(f, x = a..b, Angle = r1..r2<, option1, option2>, ...)`

Parameters:

`f`	— arithmetical expression in `x`
`x`	— identifier
`a, b, r1, r2`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `option = value`

Return Value: an object of the domain type `plot::Surface3d`.

Related Functions: `plot`, `plot::Surface3d`, `plot::yrotate`

Details:

- The result of `plot::xrotate` is the surface $(x, \alpha) \mapsto \big(x, f(x)\cos(\alpha), f(x)\sin(\alpha)\big)$, where x ranges from a to b and α from 0 to 2π or r_1 to r_2, respectively. It is an object of the domain type `plot::Surface3d` in two variables, namely the identifier `x` and an identifier built of the name `"angle"`.
- Use `plot` to display the revolution created on the screen.
- The plot options `option1, option2, ...` must be valid plot options for three-dimensional graphical objects. See `plot3d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Example 1. Let us revolve the sinus function around the x-axis in the interval $x \in [0, \pi]$:

```
>> r:= plot::xrotate(sin(x), x = 1..3)

 plot::Surface3d([x, sin(x) cos(angle1), sin(x) sin(angle1)],

    x = 1..3, angle1 = 0..2 PI)
```

The result is a graphical object of the domain type `plot::Surface3d`. To display the surface on the screen, call `plot`:

```
>> plot(r)
```

Here you can give scene options, for example, to change the style of the axis:

```
>> plot(r, Axes = Box)
```

Example 2. We can restrict the rotation angle like in the following call:

```
>> r2:= plot::xrotate(sin(x), x = 1..3, Angle = 0..PI):
   plot(r2)
```

Plot objects for the surface can be given together with the call of `plot::xrotate`, like in:

```
>> r2:= plot::xrotate(sin(x), x = 1..3, Color = RGB::Blue):
   plot(r2)
```

Or use the slot operator `::` to get or set afterwards plot options of such graphical objects. For example, the rotation angle of the revolution `r2` is the y-variable of the surface kept in the attribute `range2`:

```
>> angle:= r2::range2

                          angle3 = 0..2 PI
```

Hence, to restrict the rotation angle to the interval $[0, \frac{\pi}{2}]$, we enter:

```
>> r2::range2:= lhs(angle) = 0 .. PI/2:
   plot(r2)
```

Changes:

- `plot::xrotate` used to be `plotlib::xrotate`.
- `plot::xrotate` is now part of the new plot library `plot`, and thus its calling syntax and the return value were changed.

`plot::yrotate` – generate plots of surface of revolution (y-axis)

`plot::yrotate(f, x = a..b)` returns the surface of revolution defined by the function $f(x)$ around the y-axis in the interval $[a, b]$. The rotation angle ranges from 0 to 2π.

`plot::yrotate(f, x = a..b, Angle = r1..r2)` returns the surface of revolution defined by the function $f(x)$ around the y-axis in the interval $[a, b]$. The rotation angle ranges from r_1 to r_2.

Call(s):

- `plot::yrotate(f, x = a..b<, option1, option2>, ...)`
- `plot::yrotate(f, x = a..b, Angle = r1..r2<, option1, option2>, ...)`

Parameters:

`f`	— arithmetical expression in `x`
`x`	— identifier
`a, b, r1, r2`	— arithmetical expressions
`option1, option2, ...`	— plot option(s) of the form `option = value`

Return Value: an object of the domain type `plot::Surface3d`.

Related Functions: `plot`, `plot::Surface3d`, `plot::xrotate`

Details:

- The result of `plot::yrotate` is the surface $(x, \alpha) \mapsto \big(x\cos(\alpha), x\sin(\alpha), f(x)\big)$, where x ranges from a to b and α from 0 to 2π or r_1 to r_2, respectively. It is an object of the domain type `plot::Surface3d` in two variables, namely the identifier `x` and an identifier built of the name `"angle"`.
- Use the function `plot` to display the revolution created on the screen.
- The plot options `option1, option2, ...` must be valid plot options for three-dimensional graphical objects. See `plot3d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

Example 1. Let us revolve the sinus function around the y-axis in the interval $x \in [0, \pi]$:

```
>> r:= plot::yrotate(sin(x), x = 1..3, Title = "")

 plot::Surface3d([x cos(angle1), x sin(angle1), sin(x)],

    x = 1..3, angle1 = 0..2 PI)
```

The result is a graphical object of the domain type `plot::Surface3d`. To display the surface on the screen, call `plot`:

```
>> plot(r)
```

Here you can give scene options, for example, to change the style of the axis:

```
>> plot(r, Axes = Box)
```

Example 2. We can restrict the rotation angle like in the following call:

```
>> r2:= plot::yrotate(sin(x), x = 1..3, Angle = 0..PI):
   plot(r2)
```

Plot objects for the surface can be given together with the call of `plot::yrotate`, like in:

```
>> r2:= plot::yrotate(sin(x), x = 1..3, Color = RGB::Blue):
   plot(r2)
```

Or use the slot operator `::` to get or set afterwards plot options of such graphical objects. For example, the rotation angle of the revolution `r2` is the y-variable of the surface kept in the attribute `range2`:

```
>> angle:= r2::range2
```

```
                          angle3 = 0..2 PI
```

Hence, to restrict the rotation angle to the interval $[0, \frac{\pi}{2}]$, we enter:

```
>> r2::range2:= lhs(angle) = 0..PI/2:
   plot(r2)
```

Changes:

- `plot::yrotate` used to be `plotlib::yrotate`.
- `plot::yrotate` is now part of the new plot library `plot`, and thus its calling syntax and the return value were changed.