

Die Standardbibliothek

Inhaltsverzeichnis

stdlib

Die meisten der von MuPAD zur Verfügung gestellten Funktionen sind in diverse mathematische Bibliotheken aufgeteilt und müssen mit dem entsprechenden Bibliotheksnamen aufgerufen werden wie z. B. `linalg::eigenvalues` oder `numeric::eigenvalues`.

Einige häufig benutzte Funktionen sind keiner speziellen Bibliothek zugeordnet und können ohne einen Bibliothekspräfix aufgerufen werden. Dieses Dokument beschreibt diese „Basisfunktionen“.

glossary – Glossar

Dieses Glossar erklärt einige in der MuPAD-Dokumentation häufig verwendete Begriffe.

arithmetischer Ausdruck	Syntaktisch handelt es sich hierbei um ein Objekt des Typs <code>Type::Arithmetical</code> . Insbesondere sind darin Zahlen, Bezeichner und Ausdrücke vom Domain-Typ <code>DOM_EXPR</code> eingeschlossen.
ausgleichen	Folgen wie $a := (x, y)$ oder $b := (u, v)$ bestehen aus durch Kommata getrennten Objekten. Mehrere Folgen können zu einer längeren Folge zusammengefügt werden: (a, b) wird zu der vierelementigen Folge (x, y, u, v) „ausgeglichen“. Die meisten Funktionen gleichen ihre Argumente aus, d.h. der Aufruf $f(a, b)$ wird als Aufruf $f(x, y, u, v)$ mit vier Argumenten aufgefasst. Es ist jedoch zu beachten, dass manche Funktionen wie z. B. die Operandenfunktion <code>op</code> ihre Argumente nicht ausgleichen: <code>op(a, 1)</code> ist ein Aufruf mit zwei Argumenten. Dasselbe Konzept ist auch auf einige Funktion wie z. B. <code>max</code> anwendbar, wo es einer Vereinfachung wie z. B. $\max(a, \max(b, c)) = \max(a, b, c)$ entspricht.

Domain	<p>Die Bezeichnung „Domain“ ist synonym zu „Datentyp“. Jedes MuPAD-Objekt hat einen Datentyp, der als sein „Domain“ oder auch „Domain-Typ“ bezeichnet wird und mit der Funktion <code>domtype</code> erfragt werden kann.</p> <p>Es gibt „Basis-Domains“, die vom Systemkern zur Verfügung gestellt werden. Dazu gehören verschiedene Typen von Zahlen, Mengen, Listen, Felder, Tabellen, Ausdrücke, Polynome etc. In der Dokumentation werden solche Datentypen als „Kern-Domains“ bezeichnet. Die Namen der Kern-Domains sind von der Form <code>DOM_XXX</code> (z. B., <code>DOM_INT</code>, <code>DOM_SET</code>, <code>DOM_LIST</code>, <code>DOM_ARRAY</code>, <code>DOM_TABLE</code>, etc.). Details zu den Kern-Datentypen finden sich im Dokument „Basic MuPAD Data Types“.</p> <p>Zusätzlich bietet die Programmiersprache MuPADs die Möglichkeit, mit Hilfe des Schlüsselwortes <code>domain</code> oder der Funktion <code>newDomain</code> neue Datentypen zu definieren. Beispielsweise sind Reihenentwicklungen, Matrizen, stückweise definierte Objekte etc. als Domains in der MuPAD-Sprache implementiert. In der Dokumentation werden solche Datentypen als „Bibliotheks-Domains“ bezeichnet. Insbesondere stellt die Bibliothek <code>Dom</code> eine Vielzahl vordefinierter Datentypen wie z. B. Matrizen, Restklassenringe, Intervalle etc. zur Verfügung.</p> <p>Allgemeine Erklärungen zu Datentypen finden sich auf der Hilfseite <code>DOM_DOMAIN</code> (der Datentyp der Datentypen). Hier sind auch einfache Beispiele angegeben, wie der Benutzer eigene Datentypen definieren kann.</p> <p>Das Dokument „Axioms, Categories and Domains“ liefert eine umfassende technische Beschreibung des Domain-Konzepts.</p>
Domain-Element	<p>Die Aussage „x ist ein Element des Domains d“ ist synonym zu „x ist vom Domain-Typ d“, d. h., „<code>domtype(x) = d</code>“. In vielen Fällen ist auf den Hilfeseiten mit „Domain-Element“ ein Objekt eines Bibliotheks-Domains gemeint, d. h., das entsprechende Domain ist in der MuPAD-Sprache implementiert.</p>
Domain-Typ	<p>Der Domain-Typ eines Objekts ist dessen Datentyp und kann mit <code>domtype</code> erfragt werden.</p>

Funktion	Typischerweise werden Funktionen durch Prozeduren oder Funktionsumgebungen repräsentiert. Aber auch funktionale Ausdrücke wie z.B. <code>sin@exp + id^2</code> : $x \mapsto \sin(\exp(x)) + x^2$ repräsentieren Funktionen. Zahlen können ebenfalls als (konstante) Funktionen angesehen werden. Beispielsweise liefert der Aufruf <code>3(x)</code> für ein beliebiges Argument <code>x</code> die Zahl 3.
numerischer Ausdruck	Dies ist ein Ausdruck, in dem keine symbolischen Variablen außer den speziellen Konstanten <code>PI</code> , <code>E</code> , <code>EULER</code> , und <code>CATALAN</code> vorkommen. Ein numerischer Ausdruck wie z.B. <code>I^(1/3) + sqrt(PI)*exp(17)</code> ist eine exakte Darstellung einer reellen oder komplexen Zahl. Er kann aus Zahlen, Wurzelausdrücken und Aufrufen spezieller Funktionen zusammengesetzt sein. Mittels <code>float</code> kann ein numerischer Ausdruck in eine reelle oder komplexe Gleitpunktzahl konvertiert werden.
Polynom	Syntaktisch ist ein Polynom wie z.B. <code>poly(x^2 + 2, [x])</code> ein Objekt vom Typ <code>DOM_POLY</code> . Es wird durch einen Aufruf der Funktion <code>poly</code> erzeugt. Die meisten Funktionen, die mit solchen Polynomen arbeiten, sind von anderen Polynom-Domains in den MuPAD-Bibliotheken überladen.
polynomialer Ausdruck	Dies ist ein arithmetischer Ausdruck, in dem symbolische Variablen und Kombinationen davon nur in Potenzen mit positiven ganzzahligen Exponenten auftreten. Beispiele dafür sind <code>x^2 + 2</code> oder <code>x*y + (z + 1)^2</code> .
rationaler Ausdruck	Dies ist ein arithmetischer Ausdruck, in dem symbolische Variablen und Kombinationen davon nur in Potenzen mit ganzzahligen Exponenten auftreten. Beispiele dafür sind <code>x^(-2) + x + 2</code> oder <code>x*y + 1/(z + 1)^2</code> . Jeder polynomiale Ausdruck ist ein rationaler Ausdruck, aber die beiden Beispielausdrücke sind keine polynomialen Ausdrücke.
Überladung	Die Hilfeseite einer Systemfunktion dokumentiert lediglich die zulässigen Argumente, die einem vom MuPAD-Kern zur Verfügung gestellten Basistyp angehören. Ist beispielsweise die Systemfunktion <code>f</code> als „überladbar“ gekennzeichnet, dann können Benutzer ihre Funktionalität erweitern, indem sie einen Slot „ <code>f</code> “ für ihre eigenen Domains oder Funktionsumgebungen implementieren. Die Systemfunktion <code>f</code> akzeptiert ein Objekt dieses Domains und ruft dafür die benutzerdefinierte Slot-Funktion auf. Siehe auch die Domain-Dokumentation.

Zahl Eine Zahl ist entweder eine ganze Zahl (vom Typ `DOM_INT`), oder eine rationale Zahl (vom Typ `DOM_RAT`), oder eine reelle Gleitpunktzahl (vom Typ `DOM_FLOAT`) oder eine komplexe Zahl (vom Typ `DOM_COMPLEX`).
Der Datentyp `DOM_COMPLEX` umfasst die gaußschen ganzen Zahlen (z. B. $3 + 7 \cdot i$), die gaußschen rationalen Zahlen (z. B. $3/4 + 7/4 \cdot i$) sowie komplexe Gleitpunktzahlen (z. B. $1.2 + 3.4 \cdot i$).

mathematische Konstanten und Funktionen – ein Überblick

Folgende mathematische Konstanten sind vordefiniert:

`complexInfinity` – der unendlich ferne Punkt der komplexen Ebene
`i` – imaginäre Einheit $\sqrt{-1}$ (siehe `DOM_COMPLEX` zu Details)
`infinity` – reelles positives Unendlich
`undefined` – undefinierter Wert

Die folgenden Konstanten sind symbolische Repräsentationen spezieller reeller Zahlen. Gleitpunktapproximationen auf eine durch `DIGITS` bestimmte Genauigkeit erhält man über die Funktion `float`.

`CATALAN` – catalansche Konstante $\sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)^2} = 0.9159..$
`E` – eulersche Zahl $\exp(1) = 2.718..$ (siehe `exp` zu Details)
`EULER` – Euler-Mascheroni-Konstante $\lim_{n \rightarrow \infty} \left(\sum_{i=1}^n \frac{1}{i} - \ln(n) \right) = 0.5772..$
`PI` – Kreiszahl $\pi = 3.141..$

Folgende mathematische Funktionen sind vordefiniert:

abs	– Betrag einer reellen oder komplexen Zahl
arg	– Polarwinkel einer komplexen Zahl
bernoulli	– Bernoullizahlen und -polynome
besselI	– modifizierte Besselfunktionen der ersten Art
besselJ	– Besselfunktionen der ersten Art
besselK	– modifizierte Besselfunktionen der zweiten Art
besselY	– Besselfunktionen der zweiten Art
beta	– Betafunktion
binomial	– Binomialkoeffizient
ceil	– Aufrundung auf die nächstgrößere ganze Zahl
ci	– Integral-Kosinusfunktion
dilog	– Dilogarithmusfunktion
dirac	– diracsche Deltafunktion
Ei	– Integral-Exponentialfunktion
erf	– Fehlerfunktion
erfc	– komplementäre Fehlerfunktion
exp	– Exponentialfunktion
fact	– Fakultät
frac	– Abschneiden von Vorkommastellen
floor	– Abrundung auf die nächstkleinere ganze Zahl
gamma	– Gammafunktion
heaviside	– Heavisidefunktion
igamma	– unvollständige Gammafunktion
Im	– Imaginärteil einer komplexen Zahl
lambertV	– unterer Zweig der lambertschen W-Funktion
lambertW	– Hauptzweig der lambertschen W-Funktion
log	– Logarithmus zu einer beliebigen Basis
ln	– natürlicher Logarithmus
max	– Maximum reeller Zahlen
min	– Minimum reeller Zahlen
polylog	– Polylogarithmusfunktion
psi	– Digamma- und Polygammafunktion
Re	– Realteil einer komplexen Zahl
round	– Rundung auf die nächste ganze Zahl
Si	– Integral-Sinusfunktion
sign	– Vorzeichen reeller und komplexer Zahlen
sqrt	– Quadratwurzel
trunc	– Rundung auf die nächste ganze Zahl in Richtung 0
zeta	– riemannsche Zetafunktion

Außerdem sind die Winkel- und hyperbolischen Funktionen

cos, cot, csc, sec, sin, tan, cosh, coth, csch, sech, sinh, tanh

und ihre inversen Funktionen

arccos, arccot, arccsc, arcsec, arcsin, arctan, arccosh, arccoth, arccsch, arcsech, arcsinh, artanh

vordefiniert.

Änderungen:

- ☞ Die inversen trigonometrischen und hyperbolischen Funktionen wurden von `asin`, `asinh`, ... in `arcsin`, `arcsinh`, ... umbenannt. Die Integral-Exponentialfunktion wurde von `eint` in `Ei` umbenannt. Der Polarwinkel einer komplexen Zahl wird nun durch die neue Funktion `arg` geliefert und nicht mehr durch den Aufruf von `atan` mit zwei Argumenten.
 - ☞ Die neue Konstante `CATALAN` ersetzt den Funktionsaufruf `catalan()` früherer MuPAD-Versionen.
 - ☞ Die neuen Funktionen `arg`, `besseli`, `besselk`, `bessely`, `Ci`, `Ei`, `lambertV`, `lambertW`, `log` und `polylog` wurden eingeführt.
-

`:=` – Zuweisung von Werten an Variablen

`x := value` weist der Variablen `x` den Wert `value` zu.

`[x1, x2, ...] := [value1, value2, ...]` weist den Variablen `x1`, `x2` etc. die entsprechenden Werte `value1`, `value2` zu.

`f(X1, X2, ...) := value` fügt einen Eintrag in die Remember-Tafel der Prozedur `f` ein.

Aufruf(e):

- ☞ `x := value`
- ☞ `_assign(x, value)`
- ☞ `[x1, x2, ...] := [value1, value2, ...]`
- ☞ `_assign([x1, x2, ...], [value1, value2, ...])`
- ☞ `f(X1, X2, ...) := value`
- ☞ `_assign(f(X1, X2, ...), value)`

Parameter:

- | | |
|---|---|
| <code>x, x1, x2, ...</code> | — Bezeichner oder indizierte Bezeichner |
| <code>value, value1, value2, ...</code> | — beliebige MuPAD-Objekte |
| <code>f</code> | — eine Prozedur oder eine Funktionsumgebung |
| <code>X1, X2, ...</code> | — beliebige MuPAD-Objekte |

Rückgabewert: `value` bzw. `[value1, value2, ...]`.

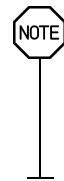
Verwandte Funktionen: `anames`, `assign`, `assignElements`, `delete`, `evalassign`

Details:

☞ `_assign(x, value)` ist äquivalent zu `x := value`.

☞ `_assign([x1, x2, ...], [value1, value2, ...])` ist äquivalent zu `[x1, x2, ...] := [value1, value2, ...]`. Beide Listen müssen dabei die gleiche Anzahl von Elementen haben.

☞ Ist `x` weder eine Liste noch eine Tabelle noch ein Array noch eine Matrix noch ein Element eines Datentyps mit einem "set_index"-Slot, so wandelt eine indizierte Zuweisung wie etwa `x[i] := value` implizit `x` in eine Tabelle mit einem entsprechenden Eintrag um. Siehe Beispiel ??.



☞ Die Zuweisung `f(x1, x2, ...) := value` fügt einen Eintrag in die Remember-Tafel der Funktion `f` ein.

Ist `f` weder eine Prozedur noch eine Funktionsumgebung, so wird implizit eine (triviale) Prozedur mit einem einzigen Eintrag `(x1, x2, ...) = value` in ihrer Remember-Tafel erzeugt und an `f` zugewiesen. Siehe Beispiel ??.



☞ Bezeichner auf der linken Seite von Zuweisungen werden nicht evaluiert (man benutze `evalassign`, wenn dies nicht gewünscht wird). d. h., in `x := value` wird ein eventuell vorhandener Wert von `x` gelöscht und durch den neuen Wert ersetzt. Man beachte jedoch, daß der Index bei einer indizierten Zuweisung evaluiert wird. In `x[i] := value` wird `i` zunächst durch den momentanen Wert von `i` ersetzt, bevor dem entsprechenden Eintrag von `x` der Wert zugewiesen wird. Siehe Beispiel ??.

☞ `_assign` ist eine Funktion des Systemkerns.

Beispiel 1. Der Zuweisungsoperator `:=` kann auf einzelne Bezeichner sowie auf eine Liste von Bezeichnern angewendet werden:

```
>> x := 42: [x1, x2, x3] := [43, 44, 45]: x, x1, x2, x3
      42, 43, 44, 45
```

Im Falle von Listen werden allen Variablen der linken Seite ihre entsprechenden Werte *gleichzeitig* zugewiesen:

```
>> [x1, x2] := [3, 4]: [x1, x2] := [x2, x1]: x1, x2
```

4, 3

Das funktionale Äquivalent des Zuweisungsoperators `:=` ist die Funktion `_assign`:

```
>> _assign(x, 13): _assign([x1, x2], [14, 15]): x, x1, x2
13, 14, 15
```

Zugewiesene Werte werden mittels `delete` gelöscht:

```
>> delete x, x1, x2: x, x1, x2
x, x1, x2
```

Beispiel 2. Bei einer indizierten Zuweisung wird implizit eine neue Tabelle (`table`, `DOM_TABLE`) erzeugt und der Variablen zugewiesen, falls dem Bezeichner nicht vorher eine Liste, eine Tabelle, ein Array oder eine Matrix zugewiesen worden war:

```
>> delete x: x[1] := 7: x
table(
  1 = 7
)
```

Ist `x` eine Liste, eine Tabelle, ein Array oder eine Matrix, so wird durch eine indizierte Zuweisung ein neuer Eintrag eingefügt oder ein existierender Eintrag wird überschrieben:

```
>> x[abc] := 8: x
table(
  abc = 8,
  1 = 7
)
```

```
>> x := [a, b, c, d]: x[3] := new: x
[a, b, new, d]
```

```
>> x := array(1..2, 1..2): x[2, 1] := value: x
```

```
+-          +-
|  ?[1, 1], ?[1, 2] |
|                    |
|  value,  ?[2, 2] |
+-          +-
```

```
>> delete x:
```

Beispiel 3. Folgende einfache Prozedur wird betrachtet:

```
>> f := x -> sin(x)/x: f(0)
```

```
Error: Division by zero;  
during evaluation of 'f'
```

Die folgende Zuweisung fügt einen Eintrag in die Remember-Tafel ein:

```
>> f(0) := 1: f(0)
```

1

Evaluiert f nicht zu einer Funktion, so wird automatisch eine triviale Prozedur mit einem Remember-Eintrag erzeugt:

```
>> delete f: f(x) := x^2: expose(f)
```

```
proc()  
  name f;  
  option remember;  
begin  
  procname(args())  
end_proc
```

Man beachte, daß die Remember-Tafel nur für den speziellen Eingabeparameter x einen Wert liefert:

```
>> f(x), f(1.0*x), f(y)
```

2
x , f(1.0 x), f(y)

```
>> delete f:
```

Beispiel 4. Die linke Seite einer Zuweisung wird nicht evaluiert. In der Zuweisung $x := 3$ erhält x einen neuen Wert, nicht y :

```
>> x := y: x := 3: x, y
```

3, y

Dementsprechend liefert die folgende Zuweisung nicht eine Zuweisung an die Variablen der Liste, sondern nur eine Zuweisung an die Liste L :

```
>> L := [x1, x2]: L := [21, 22]: L, x1, x2
```

[21, 22], x1, x2

Bei indizierten Zuweisungen werden die Indizes jedoch evaluiert:

```
>> i := 2: x[i] := value: x

      table(
        2 = value
      )

>> for i from 1 to 3 do x[i] := i^2 end_for: x

      table(
        3 = 9,
        1 = 1,
        2 = 4
      )

>> delete x, L, i:
```

Beispiel 5. Da eine Zuweisung einen Rückgabewert hat (den zugewiesenen Wert), weist der folgende Befehl mehreren Variablen gleichzeitig Werte zu:

```
>> a := b := c := 42: a, b, c

      42, 42, 42
```

Im nächsten Aufruf muss aus syntaktischen Gründen die innere Zuweisung zusätzlich geklammert werden:

```
>> a := sin((b := 3)): a, b

      sin(3), 3

>> delete a, b, c:
```

Änderungen:

- ☞ In früheren Versionen führte die Zuweisung des Wertes `NIL` zur Freigabe einer Variablen. Nun wird `NIL` wie jeder andere Ausdruck zugewiesen. Die Freigabe von Variablen erfolgt mittels `delete`.
-

. – Verkettung von Objekten

`object1.object2` verkettet zwei Objekte.

`_concat(object1, object2, ...)` verkettet eine beliebige Anzahl von Objekten.

Aufruf(e):

```

⇨ object1 . object2
⇨ _concat(object1, object2, ...)

```

Parameter:

object1 — eine Zeichenkette, ein Bezeichner oder eine Liste
 object2 — eine Zeichenkette, ein Bezeichner, eine ganze Zahl oder eine Liste

Rückgabewert: ein Objekt vom selben Typ wie object1.

Überladbar durch: object1, object2, ...

Verwandte Funktionen: @, append

Details:

⇨ `_concat(object1, object2)` ist äquivalent zu `object1 . object2`.
 Der Aufruf `_concat(object1, object2, object3, ...)` ist äquivalent zu

`((object1.object2).object3).`

`_concat()` liefert das leere Objekt vom Typ `DOM_NULL`.

⇨ Die folgenden Kombinationen sind zulässig:

object1	object2	object1.object2
Zeichenkette	Zeichenkette	Zeichenkette
Zeichenkette	Bezeichner	Zeichenkette
Zeichenkette	ganze Zahl	Zeichenkette
Bezeichner	Zeichenkette	Bezeichner
Bezeichner	Bezeichner	Bezeichner
Bezeichner	ganze Zahl	Bezeichner
Liste	Liste	Liste

Beispielsweise erzeugt `x.1` den Bezeichner `x1`.

⇨ Man beachte, daß die zu verkettenden Objekte vor der Verkettung evaluiert werden, d. h., für `x := y, i := 1` liefert die Verkettung `x.i` den Bezeichner `y1`. Der resultierende Bezeichner `y1` wird jedoch *nicht* vollständig evaluiert! Siehe Beispiel ??

⇨ `_concat` ist eine Funktion des Systemkerns.

Beispiel 1. Alle möglichen Kombinationen von Datentypen werden gezeigt, die verkettet werden können. Ist das erste Objekt eine Zeichenkette, so werden Zeichenketten erzeugt:

```
>> "x"."1", "x".y, "x".1
      "x1", "xy", "x1"
```

Ist das erste Objekt ein Bezeichner, so werden Bezeichner erzeugt:

```
>> x."1", x.y , x.1
      x1, xy, x1
```

Der Verkettungsoperator `.` dient auch zum Aneinanderhängen von Listen:

```
>> [1, 2] . [3, 4]
      [1, 2, 3, 4]

>> L := []: for i from 1 to 10 do L := L . [x.i] end_for: L
      [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]

>> delete L:
```

Beispiel 2. Die Evaluierungsstrategie der Verkettung wird demonstriert. Vor einer Verkettung werden die Objekte evaluiert:

```
>> x := "Val": i := ue: x.i
      "Value"

>> ue := 1: x.i
      "Val1"
```

Ein durch Verkettung entstehender Bezeichner wird nicht vollständig evaluiert:

```
>> delete x: x1 := 17: x.1, eval(x.1)
      x1, 17
```

Der `.`-Operator kann zur dynamischen Erzeugung von Variablen verwendet werden, denen unmittelbar Werte zugewiesen werden können:

```
>> delete x: for i from 1 to 5 do x.i := i^2 end_for:
```

Das Ergebnis der Verkettung wird wiederum nicht unmittelbar evaluiert:

```
>> x.i $ i= 1..5

      x1, x2, x3, x4, x5

>> eval(%)

      1, 4, 9, 16, 25

>> delete i, ue: (delete x.i) $ i = 1..5:
```

Beispiel 3. Die Funktion `_concat` kann beliebig viele Objekte verketteten:

```
>> _concat("an", " ", "ex", "am", "ple")

      "an example"

>> _concat("0", " ".i $ i = 1..15)

      "0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15"

>> _concat([], [x.i] $ i = 1..10)

      [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]
```

Änderungen:

☞ Keine Änderungen.

.. – Bereichsdefinition

`l..r` definiert einen „Bereich“ mit der linken Grenze `l` und der rechten Grenze `r`.

Aufruf(e):

☞ `l .. r`
☞ `_range(l, r)`

Parameter:

`l, r` — beliebige MuPAD-Objekte

Rückgabewert: ein Ausdruck vom Typ `"_range"`.

Überladbar durch: `l, r`

Verwandte Funktionen: `$, Dom::Interval`

Details:

- ⌘ Ein Bereich ist ein technisches Konstrukt zur Angabe von Zahlenbereichen beim Aufruf einiger Systemfunktionen wie z.B. `int`, `array`, `op` oder dem Folgengenerator `$`. Meist stellt `l..r` ein reelles Intervall (z.B., `int(f(x), x = l..r)`) oder eine Folge ganzer Zahlen von `l` bis `r` dar.
 - ⌘ `_range(l, r)` ist äquivalent zu `l..r`.
 - ⌘ Um Intervalle im mathematischen Sinn zu erzeugen und mit ihnen zu rechnen, verwenden Sie bitte den Datentyp `Dom::Interval`.
 - ⌘ `_range` ist eine Funktion des Systemkerns.
-

Beispiel 1. Ein Bereich kann mit dem Operator `..` oder mit der Funktion `_range` definiert werden:

```
>> _range(1, 42), 1..42
1..42, 1..42
```

Im folgenden Aufruf stellt der Bereich ein Integrationsintervall dar:

```
>> int(x, x = 1..r)
      2      2
      r      1
      -- - --
      2      2
```

Bereiche können zum Zugriff auf Operanden von Ausdrücken oder zur Definition der Dimension eines Arrays verwendet werden:

```
>> op(f(a, b, c, d, e), 2..4)
      b, c, d

>> array(1..3, [a1, a2, a3])
      +-          +-
      | a1, a2, a3 |
      +-          +-
```

Bereiche werden auch zum Erzeugen von Ausdrucksfolgen verwendet:

```
>> i^3 $ i = 1..5
1, 8, 27, 64, 125
```


Beispiel 2. Der Bereichsoperator `..` überprüft seine Parameter nicht auf ihre Semantik. Er erzeugt nur einen entsprechenden Bereich, der dann in dem Kontext interpretiert wird, in dem dieser später eingesetzt wird. Alle Grenzen werden akzeptiert:

```
>> float(PI) .. -sqrt(2)/3
```

$$3.141592654 \dots - \frac{\sqrt{2}}{3}$$

Änderungen:

⌘ Keine Änderungen.

`=`, `<>` – Gleichungen und Ungleichungen

`x = y` definiert eine Gleichung.

`x <> y` definiert eine Ungleichung.

Aufruf(e):

⌘ `x = y`

⌘ `_equal(x, y)`

⌘ `x <> y`

⌘ `_unequal(x, y)`

Parameter:

`x`, `y` — beliebige MuPAD Objekte

Rückgabewert: ein Ausdruck vom Typ `"_equal"` bzw. `"_unequal"`.

Verwandte Funktionen: `<`, `<=`, `>`, `>=`, `and`, `bool`, `FALSE`, `if`, `lhs`, `not`, `or`, `repeat`, `rhs`, `solve`, `TRUE`, `while`, `UNKNOWN`

Details:

⌘ `x = y` ist äquivalent zum Funktionsaufruf `_equal(x, y)`.

⌘ `x <> y` ist äquivalent zum Funktionsaufruf `_unequal(x, y)`.

- Die Operatoren `=` und `<>` liefern symbolische Ausdrücke, die Gleichungen bzw. Ungleichungen repräsentieren.

Die resultierenden boolschen Ausdrücke können mit der Funktion `bool` zu `TRUE` oder `FALSE` ausgewertet werden bzw. dienen als Kontrollbedingungen in `if`-, `repeat`- und `while`-Anweisungen. In all diesen Fällen ist der Vergleichstest rein syntaktisch. So liefert beispielsweise `bool(0.5 = 1/2)` den Wert `FALSE`, obwohl beide Zahlen den gleichen numerischen Wert haben. Entsprechend liefert `bool(0.5 <> 1/2)` den Wert `TRUE`.

Weiterhin können boolsche Ausdrücke mittels der Funktion `is` zu `TRUE`, `FALSE` oder `UNKNOWN` ausgewertet werden. Ein Test mit `is` ist semantisch: für den Vergleich von `x` und `y` werden intern mathematische Umformungen angewendet.

- Gleichungen und Ungleichungen haben zwei Operanden: die linke und die rechte Seite (engl.: left hand side, right hand side). Die Funktionen `lhs` und `rhs` extrahieren diese Operanden.
- `not x = y` wird stets zu `x <> y` konvertiert.
- `not x <> y` wird stets zu `x = y` konvertiert.
- `_equal` ist eine Funktion des Systemkerns.
- `_unequal` ist eine Funktion des Systemkerns.

Beispiel 1. Man beachte im Folgenden den Unterschied zwischen syntaktischer und numerischer Gleichheit. Die Zahlen 1.5 und $3/2$ stimmen numerisch überein. Allerdings ist 1.5 eine Gleitpunktzahl vom Datentyp `DOM_FLOAT`, während $3/2$ eine exakte rationale Zahl vom Datentyp `DOM_RAT` ist. Im folgenden syntaktischen Test werden sie unterschieden:

```
>> 1.5 = 3/2; bool(%)
1.5 = 3/2
FALSE
```

Die folgenden Ausdrücke sind syntaktisch gleich:

```
>> _equal(1/x, diff(ln(x),x)); bool(%)
1    1
- = -
x    x
TRUE
```

Der boolsche Operator `not` konvertiert Gleichungen und Ungleichungen:

```
>> not a = b, not a <> b
```

```
a <> b, a = b
```

Beispiel 2. Hier wird gezeigt, wie = und <> mit nicht-mathematischen Objekten und Datenstrukturen umgehen:

```
>> if "text" = "t"."e"."x"."t" then "yes" else "no" end
```

```
"yes"
```

```
>> bool(table(a = PI) <> table(a = E))
```

```
TRUE
```

Beispiel 3. Der Unterschied zwischen dem syntaktischen Test mittels bool und dem semantischen Test mittels is wird demonstriert:

```
>> bool(1 = x/(x + y) + y/(x + y)), is(1 = x/(x + y) + y/(x + y))
```

```
FALSE, TRUE
```

Beispiel 4. Gleichungen und Ungleichungen sind typische Eingabeobjekte für Systemfunktionen wie beispielsweise solve:

```
>> solve(x^2 - 2*x = -1, x)
```

```
{1}
```

```
>> solve(x^2 - 2*x <> -1, x)
```

```
C_ minus {1}
```

Änderungen:

☞ Keine Änderungen.

<, <=, >, >= – Ungleichungen

$x < y$, $x \leq y$, $x > y$ und $x \geq y$ definieren Ungleichungen.

Aufruf(e):

```
# x < y
# _less(x, y)
# x <= y
# _leequal(x, y)
# x > y
# _less(y, x)
# x >= y
# _leequal(y, x)
```

Parameter:

x, y — beliebige MuPAD Objekte

Rückgabewert: ein Ausdruck vom Typ "`_less`" bzw. "`_leequal`".

Überladbar durch: x, y

Verwandte Funktionen: `<`, `=`, `and`, `bool`, `FALSE`, `if`, `lhs`, `not`, `or`, `repeat`, `rhs`, `solve`, `TRUE`, `while`, `UNKNOWN`

Details:

- # $x > y$ und $x \geq y$ werden stets zu $y < x$ bzw. $y \leq x$ konvertiert.
- # $x < y$ ist äquivalent zum Funktionsaufruf `_less(x,y)`. Hierdurch wird der boolsche Ausdruck „ x ist kleiner als y “ dargestellt.
- # $x \leq y$ ist äquivalent zum Funktionsaufruf `_leequal(x,y)`. Hierdurch wird der boolsche Ausdruck „ x ist kleiner gleich y “ dargestellt.
- # Diese Operatoren liefern symbolische boolsche Ausdrücke. Wenn nur reelle Zahlen vom Typ `Type::Real` involviert sind, können diese Ausdrücke mit der Funktion `bool` zu `TRUE` oder `FALSE` ausgewertet werden bzw. können als Kontrollbedingungen in `if`-, `repeat`- und `while`-Anweisungen dienen.
Weiterhin können boolsche Ausdrücke mittels der Funktion `is` zu `TRUE`, `FALSE` oder `UNKNOWN` ausgewertet werden. Ein Test mit `is` kann auch auf konstante symbolische Ausdrücke angewendet werden. Siehe Beispiel ??.
- # `bool` kann auch auf Ungleichungen zwischen Zeichenketten angewendet werden und vergleicht diese dabei bezüglich der lexikographischen Ordnung.

☞ Ungleichungen haben zwei Operanden: die linke und die rechte Seite (engl.: left hand side, right hand side). Die Funktionen `lhs` und `rhs` extrahieren diese Operanden.

☞ `_less` ist eine Funktion des Systemkerns.

☞ `_leequal` ist eine Funktion des Systemkerns.

Beispiel 1. Die Operatoren `<`, `<=`, `>` und `>=` liefern symbolische Ungleichungen, die von der Funktion `bool` zu `TRUE` oder `FALSE` ausgewertet werden können, wenn nur reelle Zahlen vom Typ `Type::Real` (ganze Zahlen, rationale Zahlen und Gleitpunktzahlen) beteiligt sind:

```
>> 1.5 <= 3/2; bool(%)  
  
1.5 <= 3/2  
  
TRUE
```

Man beachte, dass `bool` keine Vergleiche von exakten symbolischen Ausdrücken durchführen kann, selbst wenn diese reelle Zahlen repräsentieren:

```
>> _less(PI, sqrt(2) + 17/10); bool(%)  
  
1/2  
PI < 2 + 17/10  
Error: Can't evaluate to boolean [_less]
```

Beispiel 2. Hier wird gezeigt, wie Zeichenketten verglichen werden können:

```
>> if "text" < "t"."e"."x"."t"."book" then "yes" else "no" end  
  
"yes"  
  
>> bool("aa" >= "b")  
  
FALSE
```

Beispiel 3. Die Funktion `bool` kann nur Zahlen vom Typ `Type::Real` vergleichen, während `is` auch exakte konstante Ausdrücke vergleicht:

```
>> bool(10 < PI^2 + sqrt(2)/10)  
  
Error: Can't evaluate to boolean [_less]  
  
>> is(10 < PI^2 + sqrt(2)/10)  
  
TRUE
```

Beispiel 4. Ungleichungen sind zulässige Eingabeobjekte für die Systemfunktion `solve`:

```
>> solve(x^2 - 2*x < 3, x)

]-1, 3[

>> solve(x^2 - 2*x >= 3, x)

]-infinity, -1] union [3, infinity[
```

Beispiel 5. Die Operatoren `=` und `<=` können durch benutzerdefinierte Domains überladen werden:

```
>> myDom := newDomain("myDom"): myDom::print := x -> extop(x):
```

Ohne Überladung von `_less` bzw. `_leequal` können Elemente dieses Domains nicht verglichen werden:

```
>> x := new(myDom, PI): y := new(myDom, sqrt(10)): bool(x < y)

Error: Can't evaluate to boolean [_less]
```

Nun wird ein Slot `"_less"` definiert. Er wird aufgerufen, wenn eine Ungleichung vom Typ `"_less"` von `bool` ausgewertet wird. Der Slot vergleicht Gleitpunktapproximationen, wenn die Argumente nicht vom Typ `Type::Real` sind:

```
>> myDom::_less := proc(x, y)
  begin
    x := extop(x, 1):
    y := extop(y, 1):
    if not testtype(x, Type::Real) then
      x := float(x):
      if not testtype(x, Type::Real) then
        error("cannot compare")
      end_if
    end_if:
    if not testtype(y, Type::Real) then
      y := float(y):
      if not testtype(y, Type::Real) then
        error("cannot compare")
      end_if
    end_if:
    bool(x < y)
  end_proc:

>> x, y, bool(x < y), bool(x > y)
```

```

                                1/2
                                PI, 10    , TRUE, FALSE

>> bool(new(myDom, I) < new(myDom, PI))

Error: cannot compare [myDom::_less]

>> delete myDom, x, y:

```

Änderungen:

⌘ Keine Änderungen.

+ – Addition von Ausdrücken

$x + y + \dots$ berechnet die Summe von x, y etc.

Aufruf(e):

⌘ $x + y + \dots$
 ⌘ `_plus(x, y, ...)`

Parameter:

x, y, \dots — arithmetische Ausdrücke, Polynome vom Typ
`DOM_POLY` oder Mengen

Rückgabewert: ein arithmetischer Ausdruck, ein Polynom oder eine Menge.

Überladbar durch: x, y, \dots

Verwandte Funktionen: `_invert, _negate, ^, /, *, -, poly, sum,`
`Pref::keepOrder`

Details:

- ⌘ $x + y + \dots$ ist äquivalent zum Funktionsaufruf `_plus(x, y, ...)`.
- ⌘ Alle Summanden, die Zahlen vom Typ `Type::Numeric` sind, werden automatisch zusammengefasst.
- ⌘ Summanden können intern umsortiert werden. Siehe Beispiel ???. Die Reihenfolge kann vom Nutzer durch die Präferenz `Pref::keepOrder` gesteuert werden. Weitere Informationen finden sich auch auf der Hilfe-seite von `print`.

☞ `_plus` akzeptiert eine beliebige Anzahl von Argumenten. In Kombination mit dem Folgenoperator `$` ist diese Funktion das empfohlene Werkzeug zur Berechnung endlicher Summen. Siehe Beispiel ???. Die Funktion `sum` kann ebenfalls zur Berechnung derartiger Summen verwendet werden. Allerdings dient `sum` mehr zur Berechnung von symbolischen und unendlichen Summen und ist langsamer als `_plus`.

☞ $x - y$ wird intern als $x + y * (-1) = \texttt{_plus}(x, \texttt{_mult}(y, -1))$ dargestellt (siehe auch `_subtract`).

☞ Viele Bibliotheksdatentypen überladen die Methode "`_plus`" mit einem geeigneten Slot "`_plus`". Summen, in denen Elemente von Bibliotheksdatentypen auftreten, werden wie folgt behandelt:

In einer Summe $x + y + \dots$ wird von links nach rechts nach Summanden gesucht, die nicht von einem der Basisdatentypen des Kerns (Zahlen, Ausdrücke, etc.) sind. Sei z der erste solche Summand. Wenn das Domain $d = z::\texttt{dom} = \texttt{domtype}(z)$ einen Slot "`_plus`" besitzt, dann wird dieser in der Form $d::\texttt{_plus}(x, y, \dots)$ aufgerufen. Das von $d::\texttt{_plus}$ gelieferte Ergebnis ist dann das Ergebnis von $x + y + \dots$.

Anwender sollten bei der Implementation des Slots $d::\texttt{_plus}$ ihres Domains d folgender Konvention folgen:

- Wenn alle Summanden Elemente von d sind, dann sollte eine entsprechende Summe vom Typ d zurückgegeben werden.
- Wenn mindestens ein Summand nicht in ein Element von d konvertiert werden kann, so sollte der Slot den Wert `FAIL` liefern.
- Mit besonderer Sorgfalt sollte vorgegangen werden, wenn Summanden auftreten, die nicht vom Typ d sind, aber in diesen Typ konvertiert werden können. Solche Summanden sollten nur dann konvertiert werden, wenn die mathematische Semantik für jeden Anwender, der dieses Domain als eine „black box“ verwendet, offensichtlich ist. Beispielsweise können ganze Zahlen aufgrund der natürlichen mathematischen Einbettung als rationale Zahlen aufgefasst werden und bei Bedarf syntaktisch in einen „rationalen Datentyp“ konvertiert werden. Im Zweifelsfall sollte die "`_plus`"-Methode den Wert `FAIL` liefern anstatt eine implizite Konvertierung vorzunehmen. Implizite Konvertierungen müssen dokumentiert werden!

Siehe die Beispiele ??? und ???.

Die meisten Bibliotheksdatentypen der MuPAD-Standardinstallation entsprechen dieser Konvention.

☞ `_plus()` liefert die Zahl 0.

☞ Polynome vom Typ `DOM_POLY` können mittels `+` addiert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben.

☞ Für endliche Mengen X, Y ist $X + Y$ die Menge $\{x + y; x \in X, y \in Y\}$.

☞ `_plus` ist eine Funktion des Systemkerns.

Beispiel 1. Numerische Summanden werden automatisch zusammengefasst:

```
>> 3 + x + y + 2*x + 5*x - 1/2 - sin(4) + 17/4
      8 x + y - sin(4) + 27/4
```

Die Reihenfolge der Terme einer symbolischen Summe stimmt nicht notwendigerweise mit der Reihenfolge der Eingabe überein:

```
>> x + y + z + a + b + c
      a + b + c + x + y + z
```

```
>> 1 + x + x^2 + x^10
      2      10
      x + x  + x  + 1
```

Intern ist diese Summe ein symbolischer Aufruf von `_plus`:

```
>> op(% , 0), type(%)
      _plus, "_plus"
```

Beispiel 2. Das funktionale Äquivalent `_plus` des Operators `+` ist ein nützliches Hilfsmittel zu Berechnung endlicher Summen. Im Folgenden werden die Summanden über den Folgenoperator `$` generiert:

```
>> _plus(i^2 $ i = 1..100)
      338350
```

Beispielsweise können mittels `_plus` leicht alle Elemente einer Menge aufaddiert werden:

```
>> S := {a, b, 1, 2, 27}: _plus(op(S))
      a + b + 30
```

Der folgende Aufruf verknüpft mittels `zip` zwei Listen durch Addition entsprechender Elemente:

```
>> L1 := [a, b, c]: L2 := [1, 2, 3]: zip(L1, L2, _plus)
      [a + 1, b + 2, c + 3]

>> delete S, L1, L2:
```

Beispiel 3. Polynome vom Typ `DOM_POLY` können mittels `+` addiert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben:

```
>> poly(x^2 + 1, [x]) + poly(x^2 + x - 1, [x])
```

$$\text{poly}(2x^2 + x, [x])$$

Bei unterschiedlichen Unbestimmten oder Koeffizientenringen werden durch `+` symbolische Summen erzeugt:

```
>> poly(x, [x]) + poly(x, [x, y])
```

$$\text{poly}(x, [x]) + \text{poly}(x, [x, y])$$

```
>> poly(x, [x]) + poly(x, [x], Dom::Integer)
```

$$\text{poly}(x, [x]) + \text{poly}(x, [x], \text{Dom}::\text{Integer})$$

Beispiel 4. Für endliche Mengen X, Y ist $X + Y$ die Menge $\{x + y; x \in X, y \in Y\}$:

```
>> {a, b, c} + {1, 2}
```

$$\{a + 1, a + 2, b + 1, b + 2, c + 1, c + 2\}$$

Beispiel 5. Viele Bibliotheksdatentypen wie z. B. Matrizen überladen `_plus`:

```
>> x := Dom::Matrix(Dom::Integer)([1, 2]):
    y := Dom::Matrix(Dom::Rational)([2, 3]):
    x + y, y + x
```

$$\begin{array}{cc|cc} + - & - + & + - & - + \\ | & 3 & | & 3 & | \\ | & & | & & | \\ | & 5 & | & 5 & | \\ + - & - + & + - & - + \end{array},$$

Wenn die Summanden in $x + y$ von unterschiedlichem Typ sind, dann versucht der erste Summand x den zweiten Summanden y in den Datentyp von x zu konvertieren. Ist er dabei erfolgreich, so ist die Summe vom gleichen Typ wie x . Im obigen Beispiel haben x und y unterschiedlichen Typ (beide sind Matrizen, aber ihre Komponentenringe sind verschieden). Also unterscheiden sich die Summen $x + y$ und $y + x$ syntaktisch, da sie den Typ des jeweils ersten Summanden erben:

```
>> bool(x + y = y + x)
```

FALSE

```
>> domtype(x + y), domtype(y + x)
```

Dom::Matrix(Dom::Integer), Dom::Matrix(Dom::Rational)

Kann x den Summanden y nicht konvertieren, so wird FAIL zurückgeliefert. Im folgenden Aufruf kann die Komponente $3/2$ nicht in eine ganze Zahl konvertiert werden:

```
>> y := Dom::Matrix(Dom::Rational)([2/3, 3]): x + y
```

FAIL

```
>> delete x, y:
```

Beispiel 6. Das Beispiel demonstriert die Implementation eines Slots "`_plus`" für ein Domain. Das folgende Domain `myString` dient der Darstellung von Zeichenketten. Die Summe dieser Zeichenketten wird durch Hintereinanderhängen definiert.

Die Methode "`new`" verwendet `expr2text` zur Konvertierung eines beliebigen MuPAD-Objektes in eine Zeichenkette. Diese Zeichenkette wird als interne Repräsentation von Elementen des Domains `myString` verwendet. Die Methode "`print`" macht diese Zeichenkette zur Bildschirmausgabe von `myString`-Objekten:

```
>> myString := newDomain("myString"):
  myString::new := proc(x)
  begin
    if args(0) = 0 then x := "": end_if;
    case domtype(x)
    of myString do return(x);
    of DOM_STRING do return(new(dom, x));
    otherwise return(new(dom, expr2text(x)));
    end_case
  end_proc:
  myString::print := x -> extop(x, 1):
```

Ohne eine "`_plus`"-Methode behandelt die Kernfunktion `_plus` die Elemente dieses Domains wie jedes andere symbolische Objekt:

```
>> y := myString(y): z := myString(z): 1 + x + y + z + 3/2
```

$x + y + z + 5/2$

Nun implementieren wir die Methode "`_plus`". Sie prüft alle ihre Argumente. Die Argumente werden konvertiert, wenn sie nicht vom Typ `myString` sind. Im allgemeinen sollten solche impliziten Konvertierungen vermieden werden. In diesem Fall hat jedoch jedes Objekt eine entsprechende Darstellung als Zeichenkette (mittels `expr2text`). Daher wird eine implizite Konvertierung implementiert. Zum Schluss wird die Summe von `myString`-Objekten als die Verkettung ihrer internen Zeichenketten definiert:

```
>> myString::_plus := proc()
  local n, Arguments, i;
  begin
    userinfo(10, "myString::_plus called with the arguments:",
      args()):
    n := args(0):
    Arguments := [args()];
    for i from 1 to n do
      if domtype(Arguments[i]) <> myString then
        // convert the i-th term to myString
        Arguments[i] := myString::new(Arguments[i]):
      end_if;
    end_for;
    myString::new(_concat(extop(Arguments[i], 1) $ i = 1..n))
  end_proc;

  setuserinfo(myString::_plus, 10):
```

Nun können `myString`-Objekte addiert werden:

```
>> myString("This ") + myString("is ") + myString("a string")

Info: myString::_plus called with the arguments:, This , is , \
a string

      This is a string
```

In der folgenden Summe sind `y` und `z` Elemente von `myString`. Der Term `y` ist der erste Summand, der Element eines Bibliotheksdatentyps mit einem "`_plus`"-Slot ist. Dessen "`_plus`"-Methode wird aufgerufen und verkettet alle Summanden zu einer Zeichenkette vom Typ `myString`:

```
>> 1 + x + y + z + 3/2;

Info: myString::_plus called with the arguments:, 1, x, y, z, \
3/2

      1xyz3/2

>> delete myString, y, z:
```

Änderungen:

- ☞ Keine Änderungen.
-

-- Subtraktion von Ausdrücke

$x - y$ berechnet die Differenz von x und y .

Aufruf(e):

- ☞ $x - y$
- ☞ `_subtract(x, y)`

Parameter:

x, y — arithmetische Ausdrücke, Polynome vom Typ `DOM_POLY` oder Mengen

Rückgabewert: ein arithmetischer Ausdruck, ein Polynom oder eine Menge.

Überladbar durch: x, y

Verwandte Funktionen: `_invert, _negate, ^, /, *, +, poly,`
`Pref::keepOrder`

Details:

- ☞ $x - y$ ist äquivalent zum Funktionsaufruf `_subtract(x, y)`.
- ☞ Für Zahlen vom Typ `Type::Numeric` wird die Differenz als Zahl zurückgegeben.
- ☞ Sind weder x noch y von einem Bibliotheksdatentyp mit einer `"_subtract"`-Methode, so wird $x - y$ intern als $x + y*(-1) = \texttt{_plus}(x, \texttt{_mult}(y, -1))$ dargestellt.
- ☞ Wenn x oder y ein Element eines Datentyps ist, das einen Slot `"_subtract"` besitzt, dann wird diese Methode zum Subtrahieren der Objekte aufgerufen.

Viele Bibliotheksdatentypen überladen den Operator `-` mit einem geeigneten Slot `"_subtract"`. Differenzen werden wie folgt behandelt:

In einer Differenz $x - y$ wird von links nach rechts nach Termen gesucht, die nicht von einem der Basisdatentypen des Kerns (Zahlen, Ausdrücke, etc.) sind. Sei z (entweder x oder y) der erste solche Term. Wenn das Domain $d = z : \texttt{dom} = \texttt{domtype}(z)$ einen Slot `"_subtract"` besitzt, dann wird dieser in der Form `d : _subtract(x, y)` aufgerufen.

Das von `d::_subtract` gelieferte Ergebnis ist dann das Ergebnis von $x - y$.

Anwender sollten bei der Implementation des Slots `d::_subtract` ihres Domains `d` folgender Konvention folgen:

- Wenn beide Terme x und y Elemente von `d` sind, dann sollte eine entsprechende Differenz vom Typ `d` zurückgegeben werden.
- Wenn entweder x oder y nicht in ein Element von `d` konvertiert werden kann, so sollte der Slot den Wert `FAIL` liefern.
- Mit besonderer Sorgfalt sollte vorgegangen werden, wenn entweder x oder y nicht vom Typ `d` sind, aber in diesen Typ konvertiert werden können. Der Term sollte nur dann konvertiert werden, wenn die mathematische Semantik für jeden Anwender, der dieses Domain als eine 'black box' verwendet, offensichtlich ist. Beispielsweise können ganze Zahlen aufgrund der natürlichen mathematischen Einbettung als rationale Zahlen aufgefasst werden und bei Bedarf syntaktisch in einen „rationalen Datentyp“ konvertiert werden. Im Zweifelsfall sollte die Methode `_subtract` den Wert `FAIL` liefern anstatt eine implizite Konvertierung vorzunehmen. Implizite Konvertierungen müssen dokumentiert werden!

Siehe Beispiel ?? und ??.

Die meisten Bibliotheksdatentypen der MuPAD-Standardinstallation entsprechen dieser Konvention.

⇒ Polynome vom Typ `DOM_POLY` können mittels `-` subtrahiert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben.

⇒ Für endliche Mengen X, Y ist $X - Y$ die Menge $\{x - y; x \in X, y \in Y\}$.

⇒ `_subtract` ist eine Funktion des Systemkerns.

Beispiel 1. Die Differenz von Zahlen wird zu einer Zahl vereinfacht:

```
>> 1234 - 234, I + x - y - 4*I, 3 + x - y - 29/3
      1000, x - y - 3 I, x - y - 20/3
```

Intern wird eine symbolische Differenz $x - y$ als die Summe $x + y*(-1)$ dargestellt:

```
>> type(x - y), op(x - y, 0), op(x - y, 1), op(x - y, 2)
      "_plus", _plus, x, -y
>> op(op(x - y, 2))
      y, -1
```

Beispiel 2. Polynome vom Typ `DOM_POLY` können mittels `-` subtrahiert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben:

```
>> poly(x^2 + 1, [x]) - poly(x^2 + x - 1, [x])
      poly(- x + 2, [x])
```

Bei unterschiedlichen Unbestimmten oder Koeffizientenringen werden durch `-` symbolische Summen erzeugt:

```
>> poly(x, [x]) - poly(x, [x, y])
      poly(x, [x]) + poly((-1) x, [x, y])
>> poly(x, [x]) - poly(x, [x], Dom::Integer)
      poly(x, [x]) + poly((-1) x, [x], Dom::Integer)
```

Beispiel 3. Für endliche Mengen X, Y ist $X - Y$ die Menge $\{x - y; x \in X, y \in Y\}$:

```
>> {a, b, c} - {1, 2}
      {a - 1, a - 2, b - 1, b - 2, c - 1, c - 2}
```

Beispiel 4. Viele Bibliotheksdatentypen wie Matrizen überladen `_subtract`:

```
>> x := Dom::Matrix(Dom::Integer)([2, 2]):
    y := Dom::Matrix(Dom::Rational)([1, 3]):
    x - y, y - x
```

```

+-      +-      +-      +-
|      1      |      |      -1      | | |
|      |      |      |      |      |
|     -1      |      |      1      |
+-      +-      +-      +-

```

Wenn die Terme in $x - y$ von unterschiedlichem Typ sind, dann versucht der erste Term x den zweiten Term y in den Datentyp von x zu konvertieren. Ist er dabei erfolgreich, so ist die Differenz vom gleichen Typ wie x . Im obigen Beispiel haben x und y unterschiedliche Typen (beide sind Matrizen, aber ihre Komponentenringe sind verschieden). Folglich unterscheiden sich auch die Typen von $x - y$ und $y - x$, da sie den Typ des jeweils ersten Terms erben:

```
>> domtype(x - y), domtype(y - x)
```

```
Dom::Matrix(Dom::Integer), Dom::Matrix(Dom::Rational)
```

Kann x den Term y nicht konvertieren, so wird FAIL zurückgeliefert. Im folgenden Aufruf kann die Komponente $3/2$ nicht in eine ganze Zahl konvertiert werden:

```
>> y := Dom::Matrix(Dom::Rational)([2/3, 3]): x - y
```

```
FAIL
```

Der Matrixdatentyp definiert $x - y$ als $x + (-y)$:

```
>> x::dom::_subtract
```

```
(x, y) -> dom::_plus(x, dom::_negate(y))
```

```
>> delete x, y:
```

Beispiel 5. Das Beispiel demonstriert die Implementation des Slots "`_subtract`" für ein Domain. Das folgende Domain `myString` dient der Darstellung von Zeichenketten. Die Differenz $x - y$ solcher Zeichenketten wird durch das Löschen aller Zeichen von y in x definiert.

Die Methode "`new`" verwendet `expr2text` zur Konvertierung eines beliebigen MuPAD-Objektes in eine Zeichenkette. Diese Zeichenkette wird als interne Repräsentation von Elementen des Domains `myString` verwendet. Die Methode "`print`" macht diese Zeichenkette zur Bildschirmausgabe von `myString`-Objekten:

```
>> myString := newDomain("myString"):
myString::new := proc(x)
begin
  if args(0) = 0 then x := "" end_if;
  case domtype(x)
  of myString do return(x);
  of DOM_STRING do return(new(dom, x));
  otherwise return(new(dom, expr2text(x)));
  end_case
end_proc:
myString::print := x -> extop(x, 1):
```

Ohne die "`_subtract`"-Methode behandelt das System die Elemente dieses Domains wie jedes andere symbolische Objekt:

```
>> x := myString(x): y := myString(y): x - y
```

```
x - y
```


Nun implementieren wir die Methode "_subtract". Sie prüft alle ihre Argumente. Die Argumente werden konvertiert, wenn sie nicht vom Typ myString sind. Im allgemeinen sollten solche impliziten Konvertierungen vermieden werden. In diesem Fall hat jedoch jedes Objekt eine entsprechende Darstellung als Zeichenkette (mittels `expr2text`). Daher wird eine implizite Konvertierung implementiert. Zum Schluss löscht die Differenz $x - y$ zweier myString-Objekte alle Zeichen von y in der Zeichenkette x :

```
>> myString::_subtract := proc(x, y)
  local i, char;
  begin
    userinfo(10, "myString::_subtract called with ".
      "the arguments:", args());
    // Convert all arguments to myString.
    if domtype(x) <> myString then x := myString::new(x) end_if;
    if domtype(y) <> myString then y := myString::new(y) end_if;
    // extract the internal strings
    x := extop(x, 1);
    y := extop(y, 1);
    // convert the strings to a list/set of characters
    x := [x[i] $ i = 0 .. length(x) - 1];
    y := {y[i] $ i = 0 .. length(y) - 1};
    // remove all characters in y from x
    for char in y do
      x := subs(x, char = null());
    end_for;
    // concat the remaining characters in x
    myString::new(_concat(op(x)))
  end_proc;

  setuserinfo(myString::_subtract, 10):
```

Nun können myString-Objekte subtrahiert werden:

```
>> myString("This is a string") - myString("is")

Info: myString::_subtract called with the arguments:, This is \
a string, is

Th a trng
```

Im Folgenden ist y der erste Term, der Element eines Bibliotheksdatentyps mit einem "_subtract"-Slot ist. Dessen "_subtract"-Methode wird aufgerufen, konvertiert xyz in ein myString-Objekt und löscht das Zeichen y :

```
>> xyz - y

Info: myString::_subtract called with the arguments:, xyz, y

xz
```

Die folgende Eingabe $xyz - x - y = (xyz - x) - y$ ruft die `"_subtract"`-Methode zweimal auf:

```
>> xyz - x - y
```

```
Info: myString::_subtract called with the arguments:, xyz, x
```

```
Info: myString::_subtract called with the arguments:, yz, y
```

z

```
>> delete myString, x, y:
```

Änderungen:

⌘ Keine Änderungen.

* – Multiplikation von Ausdrücken

$x * y * \dots$ berechnet das Produkt von x, y etc.

Aufruf(e):

⌘ $x * y * \dots$

⌘ `_mult(x, y, ...)`

Parameter:

x, y, \dots — arithmetische Ausdrücke, Polynome vom Typ
DOM_POLY oder Mengen

Rückgabewert: ein arithmetischer Ausdruck, ein Polynom oder eine Menge.

Überladbar durch: x, y, \dots

Verwandte Funktionen: `_invert`, `_negate`, `product`, `^`, `/`, `+`, `-`, `poly`,
`Pref::timesDot`

Details:

⌘ $x * y * \dots$ ist äquivalent zum Funktionsaufruf `_mult(x, y, ...)`.

⌘ Alle Faktoren, die Zahlen vom Typ `Type::Numeric` sind, werden automatisch zusammengefaßt.

☞ Faktoren eines Produkts können intern umsortiert werden, wenn keiner der Faktoren zu einem Bibliotheksdatentyp gehört, der `_mult` überlädt: aus Effizienzgründen wird die Multiplikation auf Kerndatentypen (Zahlen, Bezeichner, Ausdrücke etc.) als kommutativ vorausgesetzt. Siehe Beispiel ??.

Durch Überladung kann der Nutzer ein nicht-kommutatives Produkt für spezielle Datentypen (Domains) implementieren.

☞ `_mult` akzeptiert eine beliebige Anzahl von Argumenten. In Kombination mit dem Folgenperator `$` ist diese Funktion das empfohlene Werkzeug zur Berechnung endlicher Produkte. Siehe Beispiel ??. Die Funktion `product` kann ebenfalls zur Berechnung derartiger Produkte verwendet werden. Allerdings dient `product` mehr zur Berechnung von symbolischen und unendlichen Produkten und ist langsamer als `_mult`.

☞ Der Quotient x/y wird intern als $x * (1/y) = \text{_mult}(x, \text{_power}(y, -1))$ dargestellt (siehe `_divide`).

☞ Viele Bibliotheksdatentypen überladen die Methode `_mult` mit einem geeigneten Slot "`_mult`". Produkte, in denen Elemente von Bibliotheksdatentypen auftreten, werden wie folgt behandelt:

In einem Produkt $x * y * \dots$ wird von links nach rechts nach Faktoren gesucht, die nicht von einem der Basisdatentypen des Kerns (Zahlen, Ausdrücke, etc.) sind. Sei z der erste solche Faktor. Wenn das Domain $d = z::\text{dom} = \text{domtype}(z)$ einen Slot "`_mult`" besitzt, dann wird dieser in der Form $d::\text{_mult}(x, y, \dots)$ aufgerufen. Das von $d::\text{_mult}$ gelieferte Ergebnis ist dann das Ergebnis von $x * y * \dots$.

Siehe Beispiel ?? und ??.

☞ `_mult()` liefert die Zahl 1.

☞ Polynome vom Typ `DOM_POLY` können mittels `*` multipliziert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben. Mit `multcoeffs` kann ein Polynom mit einem skalaren Faktor multipliziert werden.

☞ Für endliche Mengen X, Y ist $X * Y$ die Menge $\{x y; x \in X, y \in Y\}$.

☞ `_mult` ist eine Funktion des Systemkerns.

Beispiel 1. Numerische Faktoren werden automatisch zusammengefasst:

```
>> 3 * x * y * (1/18) * sin(4) * 4
```

$$\frac{2 \, x \, y \, \sin(4)}{3}$$

Die Reihenfolge der Terme eines symbolischen Produkts stimmt nicht notwendigerweise mit der Reihenfolge der Eingabe überein:

```
>> x * y * 3 * z * a * b * c
      3 a b c x y z
```

Intern ist dieses Produkt ein symbolischer Aufruf von `_mult`:

```
>> op(%), type(%)
      _mult, "_mult"
```

Man beachte, daß die Bildschirmausgabe nicht unbedingt der internen Anordnung der Faktoren eines Produkts entspricht:

```
>> op(%2)
      a, b, c, x, y, z, 3
```

Insbesondere wird ein numerischer Faktor intern als letzter Operand gespeichert. In der Ausgabe steht ein numerischer Faktor vor den restlichen Termen:

```
>> 3 * x * y * 4
      12 x y

>> op(%)
      x, y, 12
```

Beispiel 2. Das funktionale Äquivalent `_mult` des Operators `*` ist ein nützliches Hilfsmittel zu Berechnung endlicher Produkte. Im Folgenden werden die Faktoren über den Folgenoperator `$` generiert:

```
>> _mult(i $ i = 1..20)
      2432902008176640000
```

Beispielsweise können mittels `_mult` leicht alle Elemente einer Menge multipliziert werden:

```
>> S := {a, b, 1, 2, 27}: _mult(op(S))
      54 a b
```

Der folgende Aufruf verknüpft zwei Listen durch Multiplikation entsprechender Elemente:

```
>> L1 := [1, 2, 3]: L2 := [a, b, c]: zip(L1, L2, _mult)
      [a, 2 b, 3 c]

>> delete S, L1, L2:
```

Beispiel 3. Polynome vom Typ DOM_POLY können mittels * multipliziert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben:

```
>> poly(x^2 + 1, [x]) * poly(x^2 + x - 1, [x])
```

$$\text{poly}(x^4 + x^3 + x - 1, [x])$$

Bei unterschiedlichen Unbestimmten oder Koeffizientenringen werden durch * symbolische Produkte erzeugt:

```
>> poly(x, [x]) * poly(x, [x, y])
```

$$\text{poly}(x, [x]) \text{poly}(x, [x, y])$$

```
>> poly(x, [x]) * poly(x, [x], Dom::Integer)
```

$$\text{poly}(x, [x]) \text{poly}(x, [x], \text{Dom}::\text{Integer})$$

Multiplikation von Polynomen mit skalaren Faktoren kann durch * nicht erreicht werden:

```
>> 2 * y * poly(x, [x])
```

$$2 \text{poly}(x, [x]) y$$

Stattdessen kann multcoeffs verwendet werden:

```
>> multcoeffs(poly(x^2 - 2, [x]), 2*y)
```

$$\text{poly}((2 y) x^2 - 4 y, [x])$$

Beispiel 4. Für endliche Mengen X, Y ist $X * Y$ die Menge $\{x y; x \in X, y \in Y\}$:

```
>> {a, b, c} * {1, 2}
```

$$\{a, b, c, 2 a, 2 b, 2 c\}$$

```
>> 2 * {a, b, c} * c
```

$$\{2 a c, 2 b c, 2 c^2\}$$

Beispiel 5. Viele Bibliotheksdatentypen wie z. B. Matrizen überladen `_mult`. Das Matrix-Produkt ist nicht-kommutativ:

```
>> x := Dom::Matrix(Dom::Integer)([[1, 2], [3, 4]]):
    y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 13]]):
    x * y, y * x
```

$$\begin{array}{cc|cc|cc|cc} +- & & -+ & +- & & -+ & & \\ | & 34, & 37 & | & | & 43, & 64 & | \\ | & & & | & | & & & | \\ | & 78, & 85 & | & | & 51, & 76 & | \\ +- & & -+ & +- & & -+ & & \end{array}$$

Wenn die Terme in `x * y` von unterschiedlichem Typ sind, dann versucht der erste Faktor `x`, den zweiten Faktor `y` in den Datentyp von `x` zu konvertieren. Ist er dabei erfolgreich, so ist das Produkt vom gleichen Typ wie `x`. Im vorherigen Beispiel haben `x` und `y` unterschiedlichen Typ (beide sind Matrizen, aber die Komponentenringe sind verschieden). Also haben auch die Produkte `x * y` und `y * x` verschiedene Typen, da sie den Typ des jeweils ersten Faktors erben:

```
>> domtype(x * y), domtype(y * x)

Dom::Matrix(Dom::Integer), Dom::Matrix(Dom::Rational)
```

Kann `x` den Faktor `y` nicht konvertieren, so versucht `y` den Faktor `x` zu konvertieren. Im folgenden Aufruf kann die Komponente `27/2` nicht in eine ganze Zahl konvertiert werden. Dementsprechend muß in `x * y` der Faktor `y` die Konvertierung übernehmen, wodurch ein Ergebnis vom selben Typ wie `y` entsteht:

```
>> y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 27/2]]):
    x * y, y * x
```

$$\begin{array}{cc|cc|cc|cc} +- & & -+ & +- & & -+ & & \\ | & 34, & 38 & | & | & 43, & 64 & | \\ | & & & | & | & & & | \\ | & 78, & 87 & | & | & 105/2, & 78 & | \\ +- & & -+ & +- & & -+ & & \end{array}$$

```
>> domtype(x * y), domtype(y * x)

Dom::Matrix(Dom::Rational), Dom::Matrix(Dom::Rational)
```

```
>> delete x, y:
```

Beispiel 6. Das Beispiel demonstriert die Implementation eines Slots "`_mult`" für ein Domain. Das folgende Domain `myString` dient der Darstellung von Zeichenketten. Durch das Überladen von `_mult` werden ganzzahlige Vielfache einer Zeichenkette durch das Hintereinanderhängen von Kopien der Zeichenkette definiert.

Die Methode "`new`" verwendet `expr2text` zur Konvertierung eines beliebigen MuPAD-Objektes in eine Zeichenkette. Diese Zeichenkette wird als interne Repräsentation von Elementen des Domains `myString` verwendet. Die Methode "`print`" macht diese Zeichenkette zur Bildschirmausgabe von `myString`-Objekten:

```
>> myString := newDomain("myString"):
myString::new := proc(x)
begin
  if args(0) = 0 then x := "": end_if;
  case domtype(x)
  of myString do return(x);
  of DOM_STRING do return(new(dom, x));
  otherwise return(new(dom, expr2text(x)));
  end_case
end_proc:
myString::print := x -> extop(x, 1):
```

Ohne eine "`_mult`"-Methode behandelt die Kernfunktion `_mult` die Elemente dieses Domains wie jedes andere symbolische Objekt:

```
>> y := myString(y): z := myString(z): 4 * x * y * z * 3/2

6 x y z
```

Nun implementieren wir die Methode "`_mult`". Sie verwendet `split`, um alle ganzzahligen Werte aus der Argumentenliste herauszunehmen und zu multiplizieren. Das Ergebnis ist eine ganze Zahl `n`. Wenn genau ein Element in der Argumentenliste verbleibt (dies muss eine Zeichenkette vom Typ `myString` sein), dann wird dieses `n` mal kopiert. Die Verkettung der Kopien wird als Ergebnis zurückgegeben:

```
>> myString::_mult:= proc()
local Arguments, intfactors, others, dummy, n;
begin
  userinfo(10, "myString::_mult called with the arguments:",
    args());
  Arguments := [args()];
  // split the argument list into integers and other factors:
  [intfactors, others, dummy] :=
    split(Arguments, testtype, DOM_INT);
  // multiply all integer factors:
  n := _mult(op(intfactors));
```

```

    if nops(others) <> 1 then
        return(FAIL)
    end_if;
    myString::new(_concat(extop(others[1], 1) $ n))
end_proc:

```

```

setuserinfo(myString::_mult, 10):

```

Nun können ganzzahlige Vielfache von myString-Objekten über den *-Operator erzeugt werden:

```

>> 2 * myString("string") * 3

Info: myString::_mult called with the arguments:, 2, string, 3

stringstringstringstringstringstring

```

Nur Produkte von ganzen Zahlen und myString-Objekten sind möglich:

```

>> 3/2 * myString("a ") * myString("string")

Info: myString::_mult called with the arguments:, 3/2, a , str\
ing

FAIL

```

```

>> delete myString, y, z:

```

Änderungen:

☞ Keine Änderungen.

/ – Division von Ausdrücken

x/y berechnet den Quotienten von x und y .

Aufruf(e):

☞ x/y
☞ `_divide(x, y)`

Parameter:

x, y, \dots — arithmetische Ausdrücke, Polynome vom Typ
DOM_POLY oder Mengen

Rückgabewert: ein arithmetischer Ausdruck, ein Polynom oder eine Menge.

Überladbar durch: x , y

Verwandte Funktionen: `_invert`, `_negate`, `^`, `*`, `+`, `-`, `div`, `divide`, `pdivide`, `poly`

Details:

- ☞ x/y ist äquivalent zum Funktionsaufruf `_divide(x, y)`.
 - ☞ Für Zahlen vom Typ `Type::Numeric` wird der Quotient als Zahl zurückgegeben.
 - ☞ Sind weder x noch y von einem Bibliotheksdatentyp mit einer `"_divide"`-Methode, so wird x/y intern als `x * y^(-1) = _mult(x, _power(y, -1))` dargestellt.
 - ☞ Wenn x oder y ein Element eines Datentyps ist, der einen Slot `"_divide"` besitzt, dann wird diese Methode zur Division der Objekte aufgerufen.
Viele Bibliotheksdatentypen überladen den `/`-Operator mit einem geeigneten Slot `"_divide"`. Quotienten werden wie folgt behandelt:
In einem Quotienten x/y wird von links nach rechts nach Elementen gesucht, die nicht von einem der Basistypen des Kerns (Zahlen, Ausdrücke, etc.) sind. Sei z (entweder x oder y) der erste solche Term. Wenn das Domain $d = z::\text{dom} = \text{domtype}(z)$ einen Slot `"_divide"` besitzt, dann wird dieser in der Form `d::_divide(x, y)` aufgerufen. Das von `d::_divide` gelieferte Ergebnis ist dann das Ergebnis von x/y .
Siehe Beispiel ?? und ??.
 - ☞ Polynome vom Typ `DOM_POLY` können mittels `/` dividiert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben und exakte Division möglich ist. Die Funktion `divide` stellt Polynomdivision mit Rest zur Verfügung.
 - ☞ Für endliche Mengen X, Y ist X/Y die Menge $\{x/y; x \in X, y \in Y\}$.
 - ☞ `_divide` ist eine Funktion des Systemkerns.
-

Beispiel 1. Der Quotient von Zahlen wird zu einer Zahl vereinfacht:

```
>> 1234/234, 7.5/7, 6*I/2
```

```
617/117, 1.071428571, 3 I
```

Intern wird ein symbolischer Quotient x/y als das Produkt `x * y^(-1)` dargestellt:

```
>> type(x/y), op(x/y, 0), op(x/y, 1), op(x/y, 2)

                                     1
                                "_mult", _mult, x, -
                                     y

>> op(op(x/y, 2), 0), op(op(x/y, 2), 1), op(op(x/y, 2), 2)

                                _power, y, -1
```

Beispiel 2. Für endliche Mengen X, Y ist X/Y die Menge $\{x/y; x \in X, y \in Y\}$:

```
>> {a, b, c} / {2, 3}

      { a  a  b  b  c  c }
      { -, -, -, -, -, - }
      { 2  3  2  3  2  3 }
```

Beispiel 3. Polynome vom Typ DOM_POLY können mittels / dividiert werden, wenn sie dieselben Unbestimmten und denselben Koeffizientenring haben und exakte Division möglich ist:

```
>> poly(x^2 - 1, [x]) / poly(x - 1, [x])

      poly(x + 1, [x])

>> poly(x^2 - 1, [x]) / poly(x - 2, [x])

      FAIL
```

Die Funktion divide stellt Division mit Rest zur Verfügung:

```
>> divide(poly(x^2 - 1, [x]), poly(x - 2, [x]))

      poly(x + 2, [x]), poly(3, [x])
```

Die Polynome müssen dieselben Unbestimmten und denselben Koeffizientenring haben:

```
>> poly(x^2 - 1, [x, y]) / poly(x - 1, [x])

      Error: Illegal argument [divide]
```

Beispiel 4. Viele Bibliotheksdatentypen wie z. B. Matrizen überladen `_divide`: Der Matrixdatentyp definiert x/y als $x * (1/y)$, wobei $1/y$ die Inverse von y ist:

```
>> x := Dom::Matrix(Dom::Integer)([[1, 2], [3, 4]]):
    y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 13]]):
    x/y
```

$$\begin{array}{cc} + - & - + \\ | & 11/2, -9/2 \\ | & \\ | & 9/2, -7/2 \\ | & \\ + - & - + \end{array}$$

Die Inverse von x hat rationale Einträge. Daher liefert $1/x$ den Wert `FAIL`, weil der Komponentenring von x (und damit die Arithmetik für x) auf die ganzen Zahlen `Dom::Integer` eingeschränkt ist. Dementsprechend liefert auch y/x den Wert `FAIL`:

```
>> y/x
```

FAIL

```
>> delete x, y:
```

Beispiel 5. Dieses Beispiel demonstriert das Verhalten von `_divide` für benutzerdefinierte Datentypen. Im ersten Fall unten besitzt das Domain keinen `"_divide"`-Slot. Daher wird x/y zu $x * (y^{-1})$ transformiert:

```
>> Do := newDomain("Do"): x := new(Do, 1): y := new(Do, 2):
    x/y; op(x/y, 0..2)
```

$$\begin{array}{c} \text{new(Do, 1)} \\ \text{-----} \\ \text{new(Do, 2)} \end{array}$$

$$\begin{array}{c} 1 \\ \text{_mult, new(Do, 1), -----} \\ \text{new(Do, 2)} \end{array}$$

Nach der Definition des Slots `"_divide"` im Domain `Do`, wird diese Methode zur Division der Elemente verwendet:

```
>> Do::_divide := proc() begin "The Result" end: x/y
```

"The Result"

```
>> delete Do, x, y:
```

Änderungen:

- ☞ Keine Änderungen.
-

\wedge – Potenzieren von Ausdrücken

x^y berechnet die y -te Potenz von x .

Aufruf(e):

- ☞ x^y
- ☞ `_power(x, y)`

Parameter:

x, y — arithmetische Ausdrücke, Polynome vom Typ `DOM_POLY` oder Mengen

Rückgabewert: ein arithmetischer Ausdruck, ein Polynom oder eine Menge.

Überladbar durch: x, y

Verwandte Funktionen: `_invert`, `_negate`, `*`, `/`, `+`, `-`,
`numlib::ispower`, `powermod`

Details:

- ☞ x^y ist äquivalent zum Funktionsaufruf `_power(x, y)`.
- ☞ Der Potenzoperator \wedge ist links-assoziativ: x^y^z wird als $(x^y)^z$ interpretiert. Siehe Beispiel ??.
- ☞ Für Polynome x vom Typ `DOM_POLY` muss y eine nichtnegative ganze Zahl sein.
- ☞ `_power` ist für Matrixdatentypen (`matrix`) überladen. Speziell liefert $x^{(-1)}$ die Inverse der Matrix x .
- ☞ Zur Berechnung modularer Potenzen sollte `powermod` benutzt werden. Siehe Beispiel ??.
- ☞ Mathematisch ist der Aufruf `sqrt(x)` äquivalent zu $x^{(1/2)}$. Die Wurzelfunktion `sqrt` versucht jedoch, zusätzliche Vereinfachungen zu erreichen. Siehe Beispiel ??.

☞ Wenn x oder y ein Element eines Datentyps ist, das einen Slot "`_power`" besitzt, dann wird diese Methode zur Berechnung von x^y aufgerufen.

Viele Bibliotheksdatentypen überladen den Operator $^$ mit einem geeigneten Slot "`_power`". Potenzen werden wie folgt behandelt:

In x^y wird von links nach rechts nach Termen gesucht, die nicht von einem der Basisdatentypen des Kerns (Zahlen, Ausdrücke, etc.) sind. Sei z (entweder x oder y) der erste solche Term. Wenn das Domain $d = z::\text{dom} = \text{domtype}(z)$ einen Slot "`_power`" besitzt, dann wird dieser in der Form $d::_\text{power}(x, y)$ aufgerufen. Das von $d::_\text{power}$ gelieferte Ergebnis ist dann das Ergebnis von x^y .

Siehe die Beispiele ?? und ??.

☞ Für endliche Mengen X, Y ist X^Y die Menge $\{x^y; x \in X, y \in Y\}$.

☞ `_power` ist eine Funktion des Systemkerns.

Beispiel 1. Es werden einige Potenzen berechnet:

```
>> 2^10, I^(-5), 0.3^(1/3), x^(1/2) + y^(-1/2), (x^(-10) + 1)^2
```

$$1024, -I, 0.66943295, x^{1/2} + \frac{1}{y^{1/2}}, \frac{1}{x^{10}} + 1, \sqrt{2}$$

Mit `expand` werden Potenzen von Summen ausmultipliziert:

```
>> (x + y)^2 = expand((x + y)^2)
```

$$(x + y)^2 = 2xy + x^2 + y^2$$

Man beachte, dass Identitäten wie z.B. $(x*y)^z = x^z * y^z$ nur in bestimmten Bereichen der komplexen Ebene gültig sind:

```
>> ((-1)*(-1))^(1/2) <> (-1)^(1/2) * (-1)^(1/2)
```

$$1 <> -1$$

Dementsprechend zieht der folgende `expand`-Aufruf das Argument nicht auseinander:

```
>> expand((x*y)^(1/2))
```

$$(x y)^{1/2}$$

Beispiel 2. Der Potenzoperator \wedge ist links-assoziativ:

```
>> 2^3^4 = (2^3)^4, x^y^z
      4096 = 4096, (x^y)^z
```

Beispiel 3. Modulare Potenzen können direkt mittels \wedge und mod berechnet werden. Es ist jedoch effizienter, powermod zu benutzen:

```
>> 123^12345 mod 17 = powermod(123, 12345, 17)
      4 = 4
```

Beispiel 4. Die Wurzelfunktion sqrt liefert einfachere Ergebnisse als _power:

```
>> sqrt(4*x*y), (4*x*y)^(1/2)
      2 (x y)^(1/2), (4 x y)^(1/2)
```

Beispiel 5. Für endliche Mengen X, Y ist X^Y die Menge $\{x^y; x \in X, y \in Y\}$:

```
>> {a, b, c}^2, {a, b, c}^{q, r, s}
      {a^2, b^2, c^2}, {a^q, a^r, a^s, b^q, b^r, b^s, c^q, c^r, c^s}
```

Beispiel 6. Viele Bibliotheksdatentypen wie z.B. Matrizen oder Restklassen überladen _power:

```
>> x := Dom::Matrix(Dom::IntegerMod(7))([[2, 3], [3, 4]]):
      x^2, x^(-1), x^3 * x^(-3)

      +-+ +-+ +-+
      | 6 mod 7, 4 mod 7 | | 3 mod 7, 3 mod 7 |
      | 4 mod 7, 4 mod 7 | | 3 mod 7, 5 mod 7 |
      +-+ +-+ +-+

      +-+ +-+
      | 1 mod 7, 0 mod 7 |
      | 0 mod 7, 1 mod 7 |
      +-+ +-+
```

```
>> delete x:
```

Beispiel 7. Dieses Beispiel demonstriert das Verhalten von `_power` für benutzerdefinierte Domains. Ohne einen `"_power"`-Slot werden Potenzen von Domain-Elementen wie jede andere symbolische Potenz behandelt:

```
>> myDomain := newDomain("myDomain"): x := new(myDomain, 1): x^2
                                2
                                (new(myDomain, 1))
>> type(x^2), op(x^2, 0), op(x^2, 1), op(x^2, 2)
                                "_power", _power, new(myDomain, 1), 2
```

Nach der Definition des `"_power"`-Slots wird diese Methode zur Potenzierung von `myDomain`-Objekten verwendet:

```
>> myDomain::_power := proc() begin "The result" end: x^2
                                "The result"
>> delete myDomain, x:
```

Änderungen:

☞ Keine Änderungen.

@ – Komposition von Funktionen

$f@g$ stellt die Verknüpfung $x \mapsto f(g(x))$ der Funktionen f und g dar.

Aufruf(e):

☞ `f @ g @ ...`
 ☞ `_fconcat(f, g, ...)`

Parameter:

`f, g, ...` — Funktionen

Rückgabewert: ein Ausdruck vom Typ `"_fconcat"`.

Überladbar durch: `f, g, ...`

Verwandte Funktionen: @@

Details:

- ☞ In MuPAD werden Funktionen oft durch Prozeduren vom Typ `DOM_PROC`, durch Funktionsumgebungen oder auch durch funktionale Ausdrücke wie z. B. `f@g@exp + id^2` dargestellt. In der Tat kann praktisch jedes MuPAD-Objekt als Funktion benutzt werden.
 - ☞ `f @ g` ist äquivalent zum Funktionsaufruf `_fconcat(f, g)`.
 - ☞ `_fconcat()` liefert die identische Abbildung `id`; `_fconcat(f)` liefert `f`.
 - ☞ `_fconcat` ist eine Funktion des Systemkerns.
-

Beispiel 1. Die folgende Funktion `h` ist die Komposition der Systemfunktionen `abs` und `sin`:

```
>> h := abs@sin

                                abs@sin

>> h(x), h(y + 2), h(0.5)

                                abs(sin(x)), abs(sin(y + 2)), 0.4794255386
```

Die folgenden funktionalen Ausdrücke stellen Polynome dar:

```
>> f := id^3 + 3*id - 1: f(x), (f@f)(x)

                                3          3          3          3
                                3 x + x  - 1, 9 x + 3 x  + (3 x + x  - 1)  - 4
```

Der Zufallszahlengenerator `random` erzeugt nichtnegative ganze Zahlen mit maximal 12 Ziffern. Die folgende Komposition von `float` und `random` erzeugt zufällige Gleitpunktzahlen zwischen 0.0 und 1.0:

```
>> rand := float@random/10^12: rand() $ k = 1..12

0.427419669, 0.3211106933, 0.3436330737, 0.4742561436,

0.5584587189, 0.7467538305, 0.03206222209, 0.7229741218,

0.6043056139, 0.7455800374, 0.2598119527, 0.3100754872
```

Zusammen mit der Funktion `map` ist der Kompositionsoperator `@` ein hilfreiches Werkzeug, um Funktionen auf die Operanden einer Datenstruktur anzuwenden:


```
>> map([1, 2, 3, 4], (PI + id^2)@sin),
      map({1, 2, 3, 4}, cos@float)

      2          2          2          2
[PI + sin(1) , PI + sin(2) , PI + sin(3) , PI + sin(4) ],

      {-0.9899924966, -0.6536436209, -0.4161468366, 0.5403023059}

>> delete h, f, rand:
```

Beispiel 2. Einige Vereinfachungen von funktionalen Ausdrücken sind mit `simplify` möglich:

```
>> cos@arccos + exp@ln = simplify(cos@arccos + exp@ln)

      cos@arccos + exp@ln = 2 id
```

Änderungen:

☞ Keine Änderungen.

@@ – Iterierte einer Funktion

`f@@n` stellt die n -fache Komposition $x \rightarrow f(f(\dots(f(x))\dots))$ der Funktion f dar.

Aufruf(e):

☞ `f @@ n`
 ☞ `_fnest(f, n)`

Parameter:

f — eine Funktion
 n — eine ganze Zahl

Rückgabewert: eine Funktion

Verwandte Funktionen: `@`, `fp::fixargs`, `fp::nest`, `fp::nestvals`, `fp::fold`

Details:

- ⌘ Die Anweisung `f@@n` ist äquivalent zum Aufruf `_fnest(f, n)`.
- ⌘ Für positives `n` ist `f@@n` auch äquivalent zu `_fconcat(f $ n)`.
- ⌘ `f@@0` gibt die identische Abbildung `id` zurück.
- ⌘ Wenn `f` eine Funktionsumgebung ist, die den Slot "inverse" hat, darf `n` auch eine negative ganze Zahl sein. Siehe Beispiel Beispiel ??.
- ⌘ Iteration ist nur für Funktionen sinnvoll, die ihre Rückgabewerte wieder als Argumente akzeptieren. Die Systemfunktion `fp::fixargs` ist oft nützlich, Funktionen mit Parametern in univariate Funktionen zu konvertieren, die gegebenenfalls iteriert werden können. Siehe Beispiel ??.

Beispiel 1. Für nicht-negatives ganzzahliges `n` ist `f@@n` äquivalent zum Aufruf `_fconcat(f $ n)`:

```
>> f@@4, (f@@4)(x)

f@f@f@f, f(f(f(f(x))))
```

`@@` vereinfacht die Komposition symbolischer Iterierter:

```
>> (f@@n)@@m

f@@(m n)
```

Die Iterierte kann wie jede MuPAD-Funktion aufgerufen werden. Wenn `f` zu einer Prozedur und `n` zu einer ganzen Zahl evaluiert, wird ein entsprechender Wert berechnet:

```
>> f := x -> x^2: (f@@n)(x) $ n = 0..10

      2    4    8    16    32    64    128    256    512    1024
    x, x , x , x , x , x , x , x , x , x , x
>> delete f:
```

Beispiel 2. Hat `f` eine bekannte Umkehrfunktion, so kann `n` auch negativ sein. Die Funktion muß dabei als Funktionsumgebung mit einem "inverse"-Slot deklariert sein. Beispielsweise sind die trigonometrischen Funktionen in MuPAD als solche Funktionsumgebungen implementiert:

```
>> sin::"inverse", sin@@-3, (sin@@(-3))(x)

"arcsin", arcsin@arcsin@arcsin, arcsin(arcsin(arcsin(x)))
```

Beispiel 3. @@ kann nur auf Funktionen angewendet werden, deren Bildbereich im eigenen Definitionsbereich enthalten ist, also $f : M \mapsto M$ für eine passende Menge M . Wenn @@ mit einer Funktion verwendet werden soll, die weitere Parameter benötigt, kann oft `fp::fixargs` benutzt werden, um eine entsprechende univariate Funktion zu erzeugen. Der folgende Aufruf iteriert die Funktion $f: x \rightarrow g(x, p)$:

```
>> g := (x, y) -> x^2 + y: f := fp::fixargs(g, 1, p): (f@@4)(x)

                2 2 2 2
            p + (p + (p + (p + x ) ) )

>> delete g, f:
```

Änderungen:

⌘ @@ hieß früher `repcom`.

⌘ Der Slot "inverse" von Funktionsumgebungen wird nun verwendet.

\$ – Erzeugen von Ausdrucksfolgen

`$ a..b` erzeugt die Folge ganzer Zahlen von a bis b .

`f $ n` erzeugt die Folge f, \dots, f von n Kopien von f .

`f(i) $ i = a..b` erzeugt die Folge $f(a), f(a+1), \dots, f(b)$.

`f(i) $ i in object` erzeugt die Folge $f(i_1), f(i_2), \dots$, wobei i_1, i_2 etc. die Operanden des Objekts `object` sind.

Aufruf(e):

```
⌘ $ a..b
⌘ _seqgen(a..b)
⌘ f $ n
⌘ _seqgen(f, n)
⌘ f $ i = a..b
⌘ _seqgen(f, i, a..b)
⌘ f $ i in object
⌘ _seqin(f, i, object)
```

Parameter:

f , $object$ — beliebige MuPAD-Objekte
 n , a , b — ganze Zahlen
 i — ein Bezeichner oder eine lokale Variable (DOM_VAR) einer Prozedur

Rückgabewert: eine Ausdrucksfolge vom Typ `"_exprseq"` oder das leere Objekt vom Typ `DOM_NULL`.

Überladbar durch: `a..b`, `f`, `n`, `i`, `object`

Verwandte Funktionen: `_exprseq`, `null`

Details:

☞ Der `$`-Operator ist ein äußerst nützliches Hilfsmittel zur Erzeugung von Folgen von Objekten. Mit Folgen können Mengen oder Listen erzeugt werden, sie können auch als Argumentfolgen an Systemfunktionen übergeben werden. Siehe Beispiel ??.

☞ `$ a..b` und der äquivalente Funktionsaufruf `_seqgen(a..b)` erzeugen die Folge ganzer Zahlen a , $a + 1$, \dots , b . Bei $a > b$ wird das leere Objekt vom Typ `DOM_NULL` erzeugt.

☞ `f $ n` und der äquivalente Funktionsaufruf `_seqgen(f, n)` erzeugen eine Folge von n Kopien des Objekts f . Dabei wird f nur einmal vor dem Aufbau der Folge evaluiert. Ist n nicht positiv, so wird die leere Folge vom Typ `DOM_NULL` erzeugt.

☞ `f $ i = a..b` und der äquivalente Funktionsaufruf `_seqgen(f, i, a..b)` substituieren $i = a$ bis $i = b$ in f und werten die Ergebnisse aus. Die folgende Sequenz wird zurückgeliefert:

$$\text{eval}(\text{subs}(f, i=a)), \text{eval}(\text{subs}(f, i=a+1)), \dots, \text{eval}(\text{subs}(f, i=b)).$$

Hierbei wird f vor der Substitution nicht evaluiert. Bei $a > b$ wird das leere Objekt vom Typ `DOM_NULL` erzeugt.

☞ `f $ i in object` und der dazu äquivalente Funktionsaufruf `_seqin(f, i, object)` substituieren $i = \text{op}(\text{object}, 1)$ bis $i = \text{op}(\text{object}, n)$ in f und werten die Ergebnisse aus ($n = \text{nops}(\text{object})$ ist die Anzahl der Operanden). Die folgende Sequenz wird zurückgeliefert:

$$\text{eval}(\text{subs}(f, i=\text{op}(\text{object}, 1))), \dots, \text{eval}(\text{subs}(f, i=\text{op}(\text{object}, n))).$$

Hierbei wird f vor der Substitution nicht evaluiert. Die leere Folge vom Typ `DOM_NULL` wird erzeugt, wenn das Objekt `object` keine Operanden hat.

⌘ Die „Schleifenvariable“ i in $f \ \$ \ i = a..b$ und $f \ \$ \ i \ \text{in} \ \text{object}$ darf einen Wert haben. Dieser Wert wird durch die Verwendung von i in einer $\$$ -Anweisung nicht verändert.

⌘ `_seqgen` ist eine Funktion des Systemkerns.

Beispiel 1. Die folgende Zahlensequenz kann als Argumentfolge an die Systemfunktion `_plus` übergeben werden, welche die Argumente addiert:

```
>> i^2 $ i = 1..5
```

1, 4, 9, 16, 25

```
>> _plus(i^2 $ i = 1..5)
```

55

Die 5-te Ableitung des Ausdrucks $\exp(x^2)$ ist:

```
>> diff(exp(x^2), x $ 5)
```

$$120 x^2 \exp(x^2) + 160 x^3 \exp(x^2) + 32 x^5 \exp(x^2)$$

Die ersten Ableitungen von $\sin(x)$ werden berechnet:

```
>> diff(sin(x), x $ i) $ i = 0..5
```

$\sin(x), \cos(x), -\sin(x), -\cos(x), \sin(x), \cos(x)$

Die ersten 10 Primzahlen werden mittels `ithprime` berechnet:

```
>> ithprime(i) $ i = 1..10
```

2, 3, 5, 7, 11, 13, 17, 19, 23, 29

Mit `isprime` werden die Primzahlen aus der Menge der ganzen Zahlen zwischen 1990 und 2010 ausgewählt:

```
>> select({$ 1990..2010}, isprime)
```

{1993, 1997, 1999, 2003}

Die 3×3 -Matrix $A_{ij} = i \cdot j$ wird erzeugt:

```
>> n := 3: matrix([[i*j $ j = 1..n] $ i = 1..n])
```

```

+-      +-
|  1, 2, 3  |
|  2, 4, 6  |
|  3, 6, 9  |
+-      +-

```

```
>> delete n:
```

Beispiel 2. In `f $ n` wird das Objekt `f` nur einmal evaluiert und dann `n`-fach kopiert. Dementsprechend erzeugt der folgende Aufruf Kopien einer einzigen Zufallszahl:

```
>> random() $ 3
```

```
427419669081, 427419669081, 427419669081
```

Der folgende Aufruf evaluiert `random` für jeden Wert von `i`:

```
>> random() $ i = 1..3
```

```
321110693270, 343633073697, 474256143563
```

Beispiel 3. Im folgenden Aufruf durchläuft `i` die Liste:

```
>> i^2 $ i in [3, 2, 1]
```

```
9, 4, 1
```

Die Bildschirmausgabe von Mengen stimmt nicht notwendigerweise mit der internen Reihenfolge der Operanden überein:

```
>> Set := {1, 2, 3, 4}: Set, [op(Set)]
```

```
{1, 2, 3, 4}, [4, 3, 2, 1]
```

Der `$`-Operator bezieht sich auf die interne Ordnung:

```
>> i^2 $ i in Set
```

```
16, 9, 4, 1
```

```
>> delete Set:
```

Beispiel 4. In `f $ i = a..b` kann `f` ein beliebiges Objekt sein. Im folgenden Aufruf ist `f` eine Zuweisung, die mit zusätzliche Klammern anzugeben ist. Die Folge berechnet eine Tabelle `f[i] = i!`:

```
>> f[0] := 1: (f[i] := i*f[i - 1]) $ i = 1..4: f

      table(
        4 = 24,
        3 = 6,
        2 = 2,
        1 = 1,
        0 = 1
      )

>> delete f:
```

Änderungen:

- ⌘ In früheren MuPAD-Versionen durfte die „Schleifenvariable“ `i` in Ausdrücken wie `f $ i = a..b` keinen Wert haben. Nun darf `i` einen Wert haben.
- ⌘ Ausdrücke wie `f $ hold(i) = a..b` mussten in früheren MuPAD-Versionen benutzt werden, falls die Schleifenvariable `i` einen Wert trug. Diese `hold`-Konstruktion ist nun weder nötig noch zulässig.

`_exprseq` – Ausdrucksfolgen

Der Funktionsaufruf `_exprseq(object1, object2, ...)` ist die interne Darstellung der Ausdrucksfolge `object1, object2, ...`.

Aufruf(e):

- ⌘ `object1, object2, ...`
- ⌘ `_exprseq(object1, object2, ...)`

Parameter:

`object1, object2, ...` — beliebig MuPAD-Objekte

Rückgabewert: ein Ausdruck vom Typ `"_exprseq"` oder das leere Objekt vom Typ `DOM_NULL`.

Verwandte Funktionen: `_stmtseq, null`

Details:

- ⌘ „Folgen“ in MuPAD sind Aneinanderreihungen beliebiger durch Kommata getrennter Objekte. Man kann sich das Komma als einen Operator vorstellen, der Folgen verkettet. Intern werden Folgen als Funktionsaufrufe `_exprseq(object1, object2, ...)` dargestellt, auf dem Bildschirm erscheinen sie als `object1, object2,`
- ⌘ `_exprseq()` und der äquivalente Aufruf `null()` erzeugen das leere Objekt vom Typ `DOM_NULL`.
- ⌘ Bei der Evaluierung einer Ausdrucksfolge werden leere Objekte vom Typ `DOM_NULL` automatisch aus der Folge entfernt.
- ⌘ Der `$`-Operator ist ein nützliches Hilfsmittel zu Erzeugung von Folgen.
- ⌘ Wird eine MuPAD-Funktion oder -Prozedur mit mehr als einem Argument aufgerufen, so werden die Argumente als Ausdrucksfolge übergeben.
- ⌘ `_exprseq` ist eine Funktion des Systemkerns.

Beispiel 1. Eine Folge wird durch Verknüpfung von Objekten mittels Kommata eingegeben. Das so entstehende Objekt ist vom Typ `"_exprseq"`:

```
>> a, b, sin(x)
```

```
a, b, sin(x)
```

```
>> op(%, 0), type(%)
```

```
_exprseq, "_exprseq"
```

In der Bildschirmausgabe liefert `_exprseq` lediglich seine Argumentfolge zurück:

```
>> _exprseq(1, 2, x^2 + 5) = (1, 2, x^2 + 5)
```

```
          2                2
(1, 2, x  + 5) = (1, 2, x  + 5)
```

Beispiel 2. Das Objekt vom Domain `DOM_NULL` (die „leere Folge“) wird automatisch aus Ausdrucksfolgen entfernt:

```
>> 1, 2, null(), 3
```

```
1, 2, 3
```


Ausdrucksfolgen werden ausgeglichen. Die folgende Sequenz hat nicht 2 Operanden, von denen der zweite selbst wiederum eine Folge ist. Die Folge wird zu einer „flachen“ Folge mit 3 Operanden ausgeglichen:

```
>> x := 1: y := 2, 3: x, y
                                     1, 2, 3

>> delete x, y:
```

Beispiel 3. Folgen dienen zum Aufbau von Mengen und Listen. Folgen können auch als Argumentfolgen an Funktionen übergeben werden:

```
>> s := 1, 2, 3: {s}, [s], f(s)
                                     {1, 2, 3}, [1, 2, 3], f(1, 2, 3)

>> delete s:
```

Änderungen:

⌘ Keine Änderungen.

`_index` – indizierter Zugriff

`x[i]` bzw. `x[i1, i2, ...]` liefert den dem Index `i` bzw. `i1, i2, ...` entsprechenden Eintrag von `x`.

Aufruf(e):

```
⌘ x[i]
⌘ _index(x, i)
⌘ x[i1, i2, ...]
⌘ _index(x, i1, i2, ...)
```

Parameter:


<code>x</code>	— ein beliebiges MuPAD-Objekt. Insbesondere ein „Behälterobjekt“: eine Liste, eine endliche Menge, ein Array, eine Matrix, eine Tabelle, eine Folge oder eine Zeichenkette.
<code>i, i1, i2, ...</code>	— Indizes. Für die meisten „Behälter“ <code>x</code> sind nur ganze Zahlen als Indizes zugelassen. Ist <code>x</code> eine Tabelle, so können beliebige MuPAD-Objekte als Indizes verwendet werden.

Rückgabewert: der dem Index entsprechende Eintrag von x . Ist x weder eine Liste, eine Menge, ein Array etc., so wird ein indiziertes Objekt vom Typ `"_index"` zurückgeliefert.

Überladbar durch: x

Verwandte Funktionen: `:=, _assign, array, contains, DOM_ARRAY, DOM_LIST, DOM_SET, DOM_STRING, DOM_TABLE, indexval, op, slot, table`

Details:

- ☞ $x[i]$ bzw. $x[i1, i2, \dots]$ ist äquivalent zum Funktionsaufruf `_index(x, i)` bzw. `_index(x, i1, i2, \dots)`.
 - ☞ Jedes MuPAD-Objekt x läßt einen indizierten Aufruf der Form $x[i]$ oder $x[i1, i2, \dots]$ zu. Ist x kein „Behälterobjekt“ wie z. B. eine Liste, eine Menge, ein Array etc., so wird ein symbolisches indiziertes Objekt zurückgeliefert. Insbesondere erhält man „indizierte Bezeichner“, wenn x ein Bezeichner ist. In diesem Fall können beliebige MuPAD-Objekte als Indizes verwendet werden. Siehe Beispiel ??.
 - ☞ Für Listen, endliche Mengen und Ausdrucksfolgen ist der Index i auf die ganzen Zahlen von 1 bis `nops(x)` beschränkt. Für Listen und Ausdrucksfolgen gilt: $x[i] = op(x, i)$.
 - ☞ Für endliche Mengen liefert $x[i]$ das i -te Element wie auf dem Bildschirm dargestellt. Beachten Sie, daß die Funktion `op` bezüglich der *internen* Ordnung auf diese Elemente zugreift. Im allgemeinen gilt für Mengen: $x[i] <> op(x, i)$. Vor der Bildschirmausgabe und dem indizierten Zugriff werden die Elemente einer Menge mittels `DOM_SET::sort` sortiert.
- 

NOTE

NOTE
- ☞ Für Arrays sind Indizes i oder Mehrfachindizes $i1, i2, \dots$ aus dem Indexbereich von `array` zulässig. Ist einer der spezifizierten Indizes eine ganze Zahl außerhalb des zulässigen Bereichs, so führt dies zu einem Fehler. Ist einer der spezifizierten Indizes keine ganze Zahl (z. B. ein Symbol i), dann wird der symbolische Ausdruck $x[i]$ bzw. $x[i1, i2, \dots]$ zurückgegeben. Bei eindimensionalen Arrays $x := array(1..n, [\dots])$ entsprechen die Einträge den Operanden: $x[i] = op(x, i)$.
 - ☞ Für Matrizen müssen Indizes i oder Mehrfachindizes $i1, i2, \dots$ aus dem von `matrix` definierten Indexbereich verwendet werden. Indizes außerhalb dieses Bereichs sowie symbolische Indizes führen zu einem Fehler. Für eindimensionale Matrizen, die einen Spaltenvektor repräsentieren, gilt: $x[i] = x[i, 1] = op(x, i)$. Für eindimensionale Matrizen, die einen Zeilenvektor repräsentieren, gilt: $x[i] = x[1, i] = op(x, i)$.

☞ Für Tabellen dürfen beliebige Indizes verwendet werden. Ist kein entsprechender Eintrag in der Tabelle, so wird der symbolische Ausdruck $x[i]$ oder $x[i1, i2, \dots]$ zurückgegeben.

☞ Für Zeichenketten ist der Index i auf die ganzen Zahlen von 0 bis $\text{length}(x) - 1$ beschränkt. Beachten Sie, daß das erste Zeichen einer Zeichenkette den Index 0 trägt!



☞ In der Regel wird der von einem indizierten Aufruf gelieferte Eintrag vollständig evaluiert.

Lediglich Matrizen verhalten sich anders: $x[i]$ und $x[i1, i2, \dots]$ liefern den Wert des Eintrags, aber nicht dessen vollständige Auswertung. Siehe Beispiel ??.



Für Arrays und Tabellen kann die Evaluierung innerhalb von indizierten Aufrufen mittels `indexval` vermieden werden.

☞ Man beachte, daß eine indizierte Zuweisung wie $x[i] := \text{value}$ automatisch x in eine Tabelle mit einem einzelnen Eintrag konvertiert, falls x nicht von einem der oben aufgeführten „Behälter“-Typen ist.



☞ `_index` ist eine Funktion des Systemkerns.

Beispiel 1. Indizierte Bezeichner sind nützlich, um Gleichungen in vielen Unbekannten zu lösen:

```
>> n := 4:
    equations := {x[i-1] - 2*x[i] + x[i+1] = 1 $ i = 1..n}:
    unknowns := {x[i] $ i = 1..n}:
    linsolve(equations, unknowns)

--          4 x[0]    x[5]                3 x[0]    2 x[5]
|  x[1] = ----- + ----- - 2, x[2] = ----- + ----- - 3,
--          5          5                  5          5

          2 x[0]    3 x[5]                x[0]    4 x[5]      -
-
x[3] = ----- + ----- - 3, x[4] = ----- + ----- - 2 |
          5          5                  5          5      -
-
```

Symbolische indizierte Objekte sind vom Typ `"_index"`:

```
>> type(x[i])

           "_index"

>> delete n, equations, unknowns:
```

Beispiel 2. Listen, Arrays und Tabellen sind typische „Behälter“, die indizierten Zugriff auf ihre Einträge erlauben:

```
>> L := [1, 2, [3, 4]]:
      A := array(1..2, 2..3, [[a12, a13], [a22, a23]]):
      T := table( 1 = T1, x = Tx, (1, 2) = T12):

>> L[1], L[3][2], A[2, 3], T[1], T[x], T[1, 2]

      1, 4, a23, T1, Tx, T12
```

Die Einträge können durch indizierte Zuweisungen geändert werden:

```
>> L[2] := 22: L[3][2] := 32: A[2, 3] := 23: T[x] := T12: L, A, T

      +-          +- table(
      |  a12, a13 |      (1, 2) = T12,
      |  [1, 22, [3, 32]], |      x = T12,
      |  a22, 23 |      1 = T1
      +-          +- )

>> delete L, A, T:
```

Beispiel 3. Für endliche Mengen liefert $x[i]$ das i -te der auf dem Bildschirm ausgegebenen Elemente. Dieses Element stimmt nicht unbedingt mit dem i -ten internen Operanden überein, der von op geliefert wird:

```
>> S := {1, 2, 3, x}

      {x, 1, 2, 3}

>> S[i] $ i = 1..4

      x, 1, 2, 3

>> op(S, i) $ i = 1..4

      x, 3, 2, 1

>> delete S:
```

Beispiel 4. Der Index-Operator operiert auch auf Zeichenketten. Man beachte, daß die Zeichen mit 0 beginnend gezählt werden:

```
>> "ABCDEF"[0], "ABCDEF"[5]

      "A", "F"
```

Beispiel 5. Üblicherweise werten indizierte Aufrufe den zurückgelieferten Eintrag vollständig aus:

```
>> delete a: x := [a, b]: a := c: x[1] = eval(x[1])  
  
c = c  
  
>> delete a: x := table(1 = a, 2 = b): a := c: x[1] = eval(x[1])  
  
c = c  
  
>> delete a: x := array(1..2, [a, b]): a := c: x[1] = eval(x[1])  
  
c = c
```

Matrizen verhalten sich anders:

```
>> delete a: x := matrix([a, b]): a := c: x[1], eval(x[1])  
  
a, c  
  
>> delete x, a:
```

Änderungen:

⌘ Keine Änderungen.

intersect, minus, union – Mengen- und Intervalloperatoren

intersect berechnet die Schnittmenge von Mengen und Intervallen.

minus berechnet die Differenzmenge von Mengen und Intervallen.

union berechnet die Vereinigung von Mengen und Intervallen.

Aufruf(e):

```
⌘ set1 intersect set2  
⌘ _intersect(set1, set2, ...)  
⌘ set1 minus set2  
⌘ _minus(set1, set2)  
⌘ set1 union set2  
⌘ _union(set1, set2, ...)
```

Parameter:

set1, set2, ... — endliche Mengen vom Typ DOM_SET,
Intervalle vom Typ Dom::Interval, oder
arithmetische Ausdrücke

Rückgabewert: eine Menge, ein Intervall oder ein symbolischer Ausdruck vom Typ "_intersect", "_minus", "_union", oder universe.

Überladbar durch: set1, set2, ...

Verwandte Funktionen: universe

Details:

⌘ set1 intersect set2 ist äquivalent zu _intersect(set1, set2).

⌘ set1 minus set2 ist äquivalent zu _minus(set1, set2).

⌘ set1 union set2 ist äquivalent zu _union(set1, set2).

⌘ Die Prioritäten von intersect, minus und union sind wie folgt: Der Operator intersect bindet stärker als minus, also

$$\text{set1 intersect set2 minus set3} = (\text{set 1 intersect set2}) \text{ minus set3.}$$

Der Operator minus bindet stärker als union, also

$$\text{set1 minus set2 union set3} = (\text{set1 minus set2}) \text{ union set3.}$$

Weiterhin gilt

$$\text{set1 minus set2 minus set3} = (\text{set 1 minus set2}) \text{ minus set3.}$$

Im Zweifelsfall verwende man Klammern um sicherzustellen, dass der eingegebene Ausdruck in der gewünschten Form interpretiert wird.

⌘ Werden Mengen oder Intervalle durch symbolische Ausdrücke mit Bezeichnern oder indizierten Bezeichnern angegeben, so werden symbolische Aufrufe von _intersect, _minus, _union zurückgeliefert. Auf dem Bildschirm erscheinen sie in der Operatornotation set1 intersect set2 etc.

⌘ Auf endlichen Mengen vom Typ DOM_SET agieren diese Operatoren auf rein *syntaktische* Weise. Z.B. wird {1} minus {x} zu {1} vereinfacht. Mathematisch ist dieses Ergebnis i.A. nicht immer korrekt, da x den Wert 1 repräsentieren könnte.



- ☞ Auf Intervallen vom Typ `Dom :: Interval` agieren diese Operatoren auf *semantische* Weise. Insbesondere werden Eigenschaften von Bezeichnern berücksichtigt.
- ☞ `_intersect()` liefert `universe` (vom Typ `stdlib :: Universe`), welches die Menge aller mathematischen Objekte darstellt.
- ☞ `_union()` liefert die leere Menge `{}`.
- ☞ `_intersect` ist eine Funktion des Systemkerns.
- ☞ `_minus` ist eine Funktion des Systemkerns.
- ☞ `_union` ist eine Funktion des Systemkerns.

Beispiel 1. `intersect`, `minus` und `union` operieren auf endlichen Mengen:

```
>> {x, 1, 5} intersect {x, 1, 3, 4},
    {x, 1, 5} union {x, 1, 3, 4},
    {x, 1, 5} minus {x, 1, 3, 4}

    {x, 1}, {x, 1, 3, 4, 5}, {5}
```

Für durch Bezeichner oder indizierte Bezeichner spezifizierte symbolische Mengen werden symbolische Aufrufe zurückgeliefert:

```
>> {1, 2} union A union {2, 3}

    {1, 2, 3} union A
```

Man beachte, dass die Mengenoperatoren auf endlichen Mengen in rein syntaktischer Weise arbeiten. Im folgenden Aufruf entspricht `x` syntaktisch keiner der Zahlen 1, 2, 3:

```
>> {1, 2, 3} minus {1, x}

    {2, 3}
```

Beispiel 2. `intersect`, `minus` und `union` sind durch den Datentyp `Dom :: Interval` überladen:

```
>> Dom :: Interval([0, 1]) union Dom :: Interval(1, 4)

    [0, 4[

>> Dom :: Interval([0, 1]) union Dom :: Interval(4, infinity)

    [0, 1] union ]4, infinity[
```

```
>> Dom::Interval(2, infinity) intersect Dom::Interval([1, 3])
      ]2, 3]
```

```
>> {PI/2, 2, 2.5, 3} intersect Dom::Interval(1,3)
```

$$\begin{array}{c} \{ \quad \quad \quad \text{PI} \} \\ \{ 2, 2.5, \text{--} \} \\ \{ \quad \quad \quad 2 \} \end{array}$$

```
>> Dom::Interval(1, PI) minus {2, 3}
```

$$]3, \text{PI}[\text{ union }]1, 2[\text{ union }]2, 3[$$

Im Gegensatz zu endlichen Mengen vom Typ `DOM_SET` werden Intervalle mit mathematischer Semantik verarbeitet. Insbesondere werden Eigenschaften von Bezeichnern berücksichtigt:

```
>> Dom::Interval(-1, 1) minus {x}
```

$$]x, 1[\text{ union }]-1, x[$$

```
>> assume(x > 2): Dom::Interval(-1, 1) minus {x}
```

$$]-1, 1[$$

```
>> unassume(x):
```

Beispiel 3. Die folgende Liste enthält eine Ansammlung von Mengen:

```
>> L := [{a, b}, {1, 2, a, c}, {3, a, b}, {a, c}]:
```

Das funktionale Äquivalent `_intersect` des `intersect`-Operators akzeptiert beliebig viele Argumente. Damit kann der Schnitt aller Mengen in `L` folgendermaßen berechnet werden:

```
>> _intersect(op(L))
```

$$\{a\}$$

Die Vereinigung aller Mengen in `L` ist:

```
>> _union(op(L))
```

$$\{a, b, c, 1, 2, 3\}$$

```
>> delete L:
```


Beispiel 4. `universe` repräsentiert die Menge aller mathematischer Objekte:

```
>> _intersect()
```

```
universe
```

Änderungen:

⌘ Keine Änderungen.

`_invert` – der Kehrwert eines Ausdrucks

`_invert(x)` berechnet den Kehrwert $1/x$ von x .

Aufruf(e):

⌘ $1/x$

⌘ `_invert(x)`

Parameter:

x — ein arithmetischer Ausdruck oder eine Menge

Rückgabewert: ein arithmetischer Ausdruck oder eine Menge.

Überladbar durch: x

Verwandte Funktionen: `_divide`, `_negate`, `^`, `/`, `*`, `+`, `-`

Details:

⌘ $1/x$ ist äquivalent zum Funktionsaufruf `_invert(x)`. Hierdurch wird das Inverse des Elements x bezüglich der Multiplikation repräsentiert, d.h., $x * (1/x) = 1$.

⌘ Der Kehrwert von Zahlen vom Typ `Type::Numeric` wird als Zahl zurückgegeben.

⌘ $1/x$ ist für Matrixdatentypen (`matrix`) überladen und liefert das Inverse der Matrix x .

⌘ Ist x nicht von einem Bibliotheksdatentyp mit einer `"_invert"`-Methode, so wird $1/x$ intern als $x^{(-1)} = \text{power}(x, -1)$ dargestellt.

- ☞ Ist x ein Element eines Datentyps, der einen Slot `"_invert"` besitzt, dann wird diese Methode zur Berechnung von $1/x$ aufgerufen. Zahlreiche Bibliotheksdatentypen überladen den `/`-Operator durch einen entsprechenden `"_invert"`-Slot. Man beachte, daß a/x nur für $a = 1$ den überladenden Slot `x::dom::_invert(x)` aufruft.
- ☞ Wenn weder x noch y den binären Operator `/` durch eine `"_divide"`-Methode überladen, ist der Quotient x/y äquivalent zu $x * y^{(-1)} = \text{_mult}(x, \text{_power}(y, -1))$.
- ☞ Für endliche Mengen X ist $1/X$ die Menge $\{1/x; x \in X\}$.
- ☞ `_invert` ist eine Funktion des Systemkerns.

Beispiel 1. Der Kehrwert eines Ausdrucks ist das Inverse bezüglich `*`:

```
>> _invert(x), x * (1/x) = x * _invert(x)
```

$$\frac{1}{-}, 1 = 1$$

```
>> 3 * y * x^2 / 27 / x
```

$$\frac{x \ y}{9}$$

Intern wird ein symbolischer Ausdruck $1/x$ als $x^{(-1)} = \text{_power}(x, -1)$ dargestellt:

```
>> type(1/x), op(1/x, 0), op(1/x, 1), op(1/x, 2)
```

```
"_power", _power, x, -1
```

Beispiel 2. Für endliche Mengen X ist $1/X$ die Menge $\{1/x; x \in X\}$:

```
>> 1/{a, b, c}
```

$$\begin{Bmatrix} 1 & 1 & 1 \\ -, & -, & - \\ a & b & c \end{Bmatrix}$$

Beispiel 3. Viele Bibliotheksdatentypen wie z. B. Matrizen oder Restklassen überladen `_invert`:

```
>> x := Dom::Matrix(Dom::IntegerMod(7))([[2, 3], [3, 4]]):
      x, 1/x, x * (1/x)
```

```
+--      +-+ +-+      +-+
|  2 mod 7, 3 mod 7 |  |  3 mod 7, 3 mod 7 |
|                    |  |                    |
|  3 mod 7, 4 mod 7 |  |  3 mod 7, 5 mod 7 |
+-+      +-+ +-+      +-+
```

```
+--      +-+
|  1 mod 7, 0 mod 7 |
|                    |
|  0 mod 7, 1 mod 7 |
+-+      +-+
```

```
>> delete x:
```

Änderungen:

⌘ Keine Änderungen.

`_lazy_and`, `_lazy_or` – „lazy evaluation“ boolscher Ausdrücke

`_lazy_and(b1, b2, ...)` wertet den boolschen Ausdruck `b1` and `b2` and `...` mittels „lazy evaluation“ aus.

`_lazy_or(b1, b2, ...)` wertet den boolschen Ausdruck `b1` or `b2` or `...` mittels „lazy evaluation“ aus.

Aufruf(e):

⌘ `_lazy_and(b1, b2, ...)`

⌘ `_lazy_or(b1, b2, ...)`

Parameter:

`b1, b2, ...` — boolsche Ausdrücke

Rückgabewert: `TRUE`, `FALSE` oder `UNKNOWN`.

Überladbar durch: `b1, b2, ...`

Verwandte Funktionen: `and`, `bool`, `if`, `is`, `or`, `repeat`, `while`, `FALSE`, `TRUE`, `UNKNOWN`

Details:

☞ `_lazy_and(b1, b2, ...)` liefert dasselbe Ergebnis wie `bool(b1 and b2 and ...)`, falls der letztere Aufruf keinen Fehler liefert. Der Unterschied zwischen diesen Aufrufen ist wie folgt:

`bool(b1 and b2 and ...)` evaluiert stets *alle* boolschen Ausdrücke und kombiniert sie dann mit dem logischen ‚und‘.

Man beachte, dass das Endergebnis `FALSE` ist, wenn nur einer der Ausdrücke `b1`, `b2` etc. `FALSE` ergibt. „Lazy evaluation“ basiert auf dieser Tatsache: `_lazy_and(b1, b2, ...)` evaluiert seine Argumente von links nach rechts. Die Evaluierung wird gestoppt, sobald ein Argument sich zu `FALSE` evaluiert. Dann wird der Wert `FALSE` als Ergebnis geliefert, *ohne dass die verbleibenden Ausdrücke evaluiert werden*. Liefert keines der Argumente `b1`, `b2` etc. den Wert `FALSE`, so werden alle Argumente evaluiert und der entsprechende Wert `TRUE` oder `UNKNOWN` wird zurückgeliefert.

`_lazy_and` wird auch als „konditionales und“ bezeichnet.

☞ `_lazy_or(b1, b2, ...)` liefert dasselbe Ergebnis wie `bool(b1 or b2 or ...)`, falls der letztere Aufruf keinen Fehler liefert. Der Unterschied zwischen diesen Aufrufen ist wie folgt:

`bool(b1 or b2 or ...)` evaluiert stets *alle* boolschen Ausdrücke und kombiniert sie dann mit dem logischen ‚oder‘.

Man beachte, dass das Endergebnis `TRUE` ist, wenn nur einer der Ausdrücke `b1`, `b2` etc. `TRUE` ergibt. „Lazy evaluation“ basiert auf dieser Tatsache: `_lazy_or(b1, b2, ...)` evaluiert seine Argumente von links nach rechts. Die Evaluierung wird gestoppt, sobald ein Argument sich zu `TRUE` evaluiert. Dann wird der Wert `TRUE` als Ergebnis geliefert, *ohne dass die verbleibenden Ausdrücke evaluiert werden*. Liefert keines der Argumente `b1`, `b2` etc. den Wert `TRUE`, so werden alle Argumente evaluiert und der entsprechende Wert `FALSE` oder `UNKNOWN` wird zurückgeliefert.

`_lazy_or` wird auch als „konditionales oder“ bezeichnet.

☞ Kann einer der betrachteten Ausdrücke `b1`, `b2` etc. nicht zu einem der Werte `TRUE`, `FALSE` oder `UNKNOWN` ausgewertet werden, so liefern `_lazy_and`, `_lazy_or` Fehler.

☞ `_lazy_and` und `_lazy_or` werden intern von den Anweisungen `if`, `repeat` und `while` verwendet. Beispielsweise ist `if b1 and b2 then ...` äquivalent zu `if _lazy_and(b1, b2) then ...`.

☞ `_lazy_and()` liefert den Wert `TRUE`.

☞ `_lazy_or()` liefert den Wert `FALSE`.

⌘ `_lazy_and` ist eine Funktion des Systemkerns.

⌘ `_lazy_or` ist eine Funktion des Systemkerns.

Beispiel 1. Dieses Beispiel demonstriert den Unterschied zwischen „lazy evaluation“ und vollständiger Evaluierung boolscher Ausdrücke. Für $x = 0$ liefert die Auswertung von $\sin(1/x)$ einen Fehler:

```
>> x := 0: bool(x <> 0 and sin(1/x) = 0)
```

```
Error: Division by zero
```

Mit „lazy evaluation“ wird der Ausdruck $\sin(1/x) = 0$ gar nicht ausgewertet, wodurch der obige Fehler vermieden wird:

```
>> _lazy_and(x <> 0, sin(1/x) = 0)
```

```
FALSE
```

```
>> bool(x = 0 or sin(1/x) = 0)
```

```
Error: Division by zero
```

```
>> _lazy_or(x = 0, sin(1/x) = 0)
```

```
TRUE
```

```
>> delete x:
```

Beispiel 2. Die folgenden Anweisungen erzeugen keinen Fehler, da `if` intern „lazy evaluation“ verwendet:

```
>> for x in [0, PI, 1/PI] do
    if x = 0 or sin(1/x) = 0 then
        print(x)
    end_if;
end_for:
```

```
0
```

```
1
```

```
--
```

```
PI
```

```
>> delete x:
```

Beispiel 3. Beide Funktionen können ohne Argumente aufgerufen werden:

```
>> _lazy_and(), _lazy_or()  
  
TRUE, FALSE
```

Änderungen:

☞ Keine Änderungen.

`_negate` – das Negative eines Ausdrucks

`_negate(x)` berechnet das Negative von x .

Aufruf(e):

☞ $-x$
☞ `_negate(x)`

Parameter:

x — ein arithmetischer Ausdruck, ein Polynom vom Typ `DOM_POLY` oder eine Menge

Rückgabewert: ein arithmetischer Ausdruck, ein Polynom oder eine Menge.

Überladbar durch: x

Verwandte Funktionen: `_invert`, `_subtract`, `^`, `/`, `*`, `+`, `-`, `poly`

Details:

- ☞ $-x$ ist äquivalent zum Funktionsaufruf `_negate(x)`. Hierdurch wird das Inverse des Elements x einer additiven Gruppe repräsentiert. Auf Ausdrücken ist $-x$ das Inverse bezüglich der $+$ Operation.
- ☞ Das Negative von Zahlen vom Typ `Type::Numeric` wird als Zahl zurückgegeben.
- ☞ Ist x nicht von einem Bibliotheksdatentyp (Domain) mit einer `"_negate"`-Methode, so wird $-x$ intern als `x*(-1) = _mult(x, -1)` dargestellt.
- ☞ Ist x ein Element eines Domains, das einen Slot `"_negate"` besitzt, dann wird diese Methode zur Berechnung von $-x$ aufgerufen. Zahlreiche Bibliotheksdatentypen überladen den unären $-$ Operator durch einen entsprechenden `"_negate"`-Slot.

- ⌘ Wenn weder x noch y den *binären* Operator $-$ durch eine `"_subtract"`-Methode überladen, ist die Differenz $x - y$ äquivalent zu $x + y*(-1) = \text{_plus}(x, \text{_mult}(y, -1))$.
 - ⌘ Das Negative eines Polynoms vom Typ `DOM_POLY` ergibt ein entsprechendes Polynom mit negierten Koeffizienten.
 - ⌘ Für endliche Mengen X ist $-X$ die Menge $\{-x; x \in X\}$.
 - ⌘ `_negate` ist eine Funktion des Systemkerns.
-

Beispiel 1. Das Negative eines Ausdrucks ist das Inverse bezüglich $+$:

```
>> x - x = x + \_negate(x)
                                0 = 0
>> -1 + x - 2*x + 23
                                22 - x
```

Intern wird ein symbolisches $-x$ als $x*(-1) = \text{_mult}(x, -1)$ dargestellt:

```
>> type(-x), op(-x, 0), op(-x, 1), op(-x, 2)
                                "\_mult", \_mult, x, -1
```

Beispiel 2. Das Negative eines Polynoms ergibt ein Polynom:

```
>> -poly(x^2 + x - 1, [x])
                                2
                                poly(- x  - x + 1, [x])
>> -poly(x, [x], Dom::Integer)
                                poly((-1) x, [x], Dom::Integer)
```

Beispiel 3. Für endliche Mengen X ist $-X$ die Menge $\{-x; x \in X\}$:

```
>> -{a, b, c}
                                {-a, -b, -c}
```

Beispiel 4. Viele Bibliotheksdatentypen wie z. B. Matrizen oder Restklassen überladen `_negate`:

```
>> x := Dom::Matrix(Dom::IntegerMod(7))([2, 10]): x, -x, x + (-x)
```

$$\begin{array}{ccccc} \begin{array}{c} + - \\ | \quad 2 \text{ mod } 7 \\ | \\ | \quad 3 \text{ mod } 7 \\ + - \end{array} & \begin{array}{c} - + \\ | \\ | \\ + - \end{array} & \begin{array}{c} + - \\ | \quad 5 \text{ mod } 7 \\ | \\ | \quad 4 \text{ mod } 7 \\ + - \end{array} & \begin{array}{c} - + \\ | \\ | \\ + - \end{array} & \begin{array}{c} + - \\ | \quad 0 \text{ mod } 7 \\ | \\ | \quad 0 \text{ mod } 7 \\ + - \end{array} & \begin{array}{c} - + \\ | \\ | \\ + - \end{array} \end{array},$$

```
>> delete x:
```

Beispiel 5. Dies Beispiel demonstriert die Implementation eines Slots "`_negate`" für ein Domain. Das folgende Domain `myString` dient der Darstellung von Zeichenketten. Für eine solche Zeichenkette `x` wird `-x` durch das Umdrehen der Reihenfolge der enthaltenen Zeichen definiert.

Die Methode "`new`" verwendet `expr2text` zur Konvertierung eines beliebigen MuPAD Objektes in eine Zeichenkette. Diese Zeichenkette wird als interne Repräsentation von Elementen des Domains `myString` verwendet. Die Methode "`print`" benutzt diese Zeichenkette als Bildschirmausgabe von `myString`-Objekten:

```
>> myString := newDomain("myString"):
myString::new := proc(x)
begin
  if args(0) = 0 then x := "" end_if;
  case domtype(x)
  of myString do return(x);
  of DOM_STRING do return(new(dom, x));
  otherwise return(new(dom, expr2text(x)));
  end_case
end_proc:
myString::print := x -> extop(x, 1):
```

Ohne eine "`_negate`"-Methode behandelt das System Elemente dieses Domains wie jedes andere symbolische Objekt:

```
>> x := myString(x): -x, type(-x), op(-x, 0), op(-x, 1), op(-x, 2)
```

```
-x, "_mult", _mult, x, -1
```

Nun implementieren wir die Methode "`_negate`". Es besteht keine Notwendigkeit, das Argument zu überprüfen, denn `_negate(x)` ruft diesen Slot genau dann auf, wenn `x` vom Typ `myString` ist. Der Slot verwendet `revert`, um die Reihenfolge der Zeichen in der internen Zeichenkette umzudrehen:


```
>> myString::_negate := x -> myString::new(revert(extop(x, 1))):
```

Nun können myString-Objekte durch den - Operator „invertiert“ werden:

```
>> -myString("This is a string")

gnirts a si sihT
```

Im folgenden Beispiel wird myString::_negate nicht aufgerufen, weil für Objekte vom Typ myString kein Slot "_subtract" definiert wurde:

```
>> myString("This is a string") - myString("a string")

This is a string - a string
```

Die Slots "_plus" und "_subtract"-Slot werden implementiert:

```
>> myString::_plus := proc()
  begin
    myString::new(_concat(map(args(), extop, 1))):
  end_proc:
  myString::_subtract := (x, y) -> x + myString::_negate(y):
```

Nun wird der "_negate"-Slot aufgerufen:

```
>> myString("This is a string") - myString("This is a string")

This is a stringgnirts a si sihT

>> delete myString, x:
```

Änderungen:

☞ Keine Änderungen.

_stmtseq – Anweisungsfolgen

Der Funktionsaufruf `_stmtseq(object1, object2, ...)` ist äquivalent zur Anweisungsfolge `(object1; object2; ...)`.

Aufruf(e):

☞ `(object1; object2; ...)`
☞ `(object1: object2: ...)`
☞ `_stmtseq(object1, object2, ...)`

Parameter:

object1, object2, ... — beliebige MuPAD-Objekte und -Anweisungen

Rückgabewert: der Rückgabewert des letzten Anweisung in der Folge.

Verwandte Funktionen: `_exprseq`

Details:

- ⌘ Der Funktionsaufruf `_stmtseq(object1, object2, ...)` evaluiert die Anweisungen `(object1; object2; ...)` von links nach rechts.
 - ⌘ `_stmtseq()` liefert das leere Objekt vom Typ `DOM_NULL`.
 - ⌘ `_stmtseq` ist eine Funktion des Systemkerns.
-

Beispiel 1. Anweisungen werden meist imperativ eingegeben:

```
>> x := 2; x := x^2 + 17; sin(x + 1)

2

21

sin(22)
```

Diese Folge von Anweisungen kann in eine einzelne Anweisung (eine „Anweisungsfolge“) verwandelt werden, indem man sie mit Klammern einschließt. Nun wird nur das Ergebnis der Anweisungsfolge ausgegeben, dies ist das Ergebnis der letzten Anweisung innerhalb der Folge:

```
>> (x := 2; x := x^2 + 17; sin(x + 1))

sin(22)
```

Alternativ kann die Anweisungsfolge mittels `_stmtseq` eingegeben werden. Aus syntaktischen Gründen müssen die Zuweisungen geklammert werden, um sie als Argumente an `_stmtseq` übergeben zu können. Nur der Rückgabewert der Anweisungsfolge (der Rückgabewert der letzten Anweisung) wird ausgegeben:

```
>> _stmtseq((x := 2), (x := x^2 + 17), sin(x + 1))

sin(22)
```

Anweisungsfolgen können iteriert werden:

```
>> x := 1: (x := x + 1; x := x^2; print(i, x)) $ i = 1..4

1, 4

2, 25

3, 676

4, 458329

>> delete x:
```

Änderungen:

⌘ Keine Änderungen.

%if – Bedingte Erzeugung von Code durch den Übersetzer

%if kontrolliert die Erzeugung von Code durch den Übersetzer abhängig von einer Bedingung.

Aufruf(e):

```
⌘ %if condition
    then casetrue
    <elif condition then casetrue, ...>
    <else casefalse>
end_if
```

Parameter:


condition — ein boolscher Ausdruck
 casetrue — eine Anweisungsfolge
 casefalse — eine Anweisungsfolge

Verwandte Funktionen: if

Details:

⌘ Diese Anweisung ist eine der mehr esoterischen Eigenschaften von MuPAD. Sie wird *nicht* zur Laufzeit vom Interpreter ausgeführt, sondern steuert die Erzeugung von Interpreter-Code durch den Übersetzer.

- ☞ `%if` kann verwendet werden, um unterschiedliche Versionen einer Bibliothek zu erzeugen, die auf gemeinsamen Quelltext basieren, oder um Debugging-Anweisungen zu schreiben, die nicht in der Endfassung einer Bibliothek enthalten sein sollen.
- ☞ Die erste Bedingung wird vom Übersetzer in einem boolschen Kontext ausgeführt und muss den Wert `TRUE` oder `FALSE` liefern:
 - Liefert die Bedingung `TRUE`, so wird vom Übersetzer die Anweisungsfolge `casetrue` als Code für die `%if`-Anweisung erzeugt. Der Rest der Anweisung wird vom Übersetzer ignoriert, für ihn wird kein Code erzeugt.
 - Liefert die Bedingung `FALSE`, so wird die Bedingung des nächsten `elif`-Teils ausgewertet, und der Übersetzer verfährt wie zuvor.
 - Liefern alle Bedingungen `FALSE` und existiert kein weiterer `elif`-Teil, so wird vom Übersetzer die Anweisungsfolge `casefalse` als Code für die `%if`-Anweisung erzeugt. Falls `casefalse` nicht existiert, wird `NIL` erzeugt.
- ☞ Die gesamte Anweisung wird vom Übersetzer gelesen und muss syntaktisch korrekt sein. Dies gilt auch für diejenigen Teile, für die kein Code erzeugt wird.
- ☞ Anstelle des Schlüsselwortes `end_if` kann auch das einfachere Schlüsselwort `end` verwendet werden.
- ☞ Für eine leere Anweisungsfolge erzeugt der Übersetzer `NIL` als Code.
- ☞ Die Bedingungen werden im umgebenden lexikalischen Kontext übersetzt, werden vom Übersetzer aber *in demjenigen Kontext ausgeführt, in dem auch der Übersetzer ausgeführt wird*. Der Grund hierfür ist, dass die Umgebung, in der die Bedingungen lexikalisch gebunden sind, während der Übersetzung noch nicht existiert. Man muss sicherstellen, dass die Namen in den Bedingungen nicht mit Namen von lokalen Variablen oder Prozedur-Argumenten aus dem umgebenden lexikalischen Kontext übereinstimmen. Das wird vom Übersetzer nicht geprüft.


- ☞ Es gibt keine Funktion im Interpreter, die die `%if`-Anweisung ausführen kann. Der Grund hierfür ist, dass die Anweisung vom Übersetzer und nicht vom Interpreter implementiert wird.

Beispiel 1. Im folgenden Beispiel wird Debugging-Code in einer Prozedur erzeugt, abhängig vom Wert des globalen Bezeichners `DEBUG`.

Man beachte, dass das Beispiel etwas akademisch ist. Die Bibliotheksfunktion `prog::trace` bietet wesentlich elegantere Möglichkeiten, die Ausführung einer Prozedur beim Debuggen zu beobachten.

```
>> DEBUG := TRUE:
  p := proc(x) begin
    %if DEBUG = TRUE then
      print("entering p")
    end;
    x^2
  end_proc:
  p(2)

      "entering p"
```

4

Beim Blick auf p sieht man, dass vom Übersetzer nur das print-Kommando in die Prozedur eingefügt wurde:

```
>> expose(p)

      proc(x)
        name p;
      begin
        print("entering p");
        x^2
      end_proc
```

Nun setze man DEBUG auf FALSE und lese die Prozedur erneut ein, um die Endversion zu erzeugen. Keine Debug-Ausgaben werden erzeugt:

```
>> DEBUG := FALSE:
  p := proc(x) begin
    %if DEBUG = TRUE then
      print("entering p")
    end;
    x^2
  end_proc:
  p(2)
```

4

Beim Blick auf die Prozedur sieht man, dass vom Übersetzer nun NIL anstelle der %if-Anweisung eingesetzt wurde:

```
>> expose(p)

      proc(x)
        name p;
      begin
        NIL;
        x^2
      end_proc
```

Hintergründe:

- ⌘ Diese Anweisung mag C-Programmierer an bedingte Übersetzung erinnern. In C wird dies durch einen Präprozessor implementiert, der vor dem eigentlichen Übersetzer läuft. In MuPAD gibt es keinen solchen Präprozessor, die `%if`-Anweisung ist Teil des Übersetzungsvorgangs.

Änderungen:

- ⌘ `%if` ist eine neue Funktion.
-

Ci – die Integral-Cosinusfunktion

$\text{Ci}(x)$ stellt die Integral-Cosinusfunktion $\text{EULER} + \ln(x) + \int_0^x (\cos(t) - 1)/t \, dt$ dar.

Aufruf(e):

- ⌘ $\text{Ci}(x)$

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `Ei`, `int`, `Si`, `cos`

Details:

- ⌘ Wenn x eine Gleitkommazahl ist, dann liefert $\text{Ci}(x)$ den numerischen Wert der Integral-Cosinusfunktion. Die speziellen Werte $\text{Ci}(\infty) = 0$ und $\text{Ci}(-\infty) = i\pi$ sind implementiert. Für alle anderen Argumente wird ein symbolischer Funktionsaufruf zurückgeliefert.
 - ⌘ Das `float`-Attribut von `Ci` ist eine Kernfunktion, d. h., die numerische Auswertung ist schnell.
-

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> Ci(1), Ci(sqrt(2)), Ci(x + 1), Ci(infinity), Ci(-infinity)

1/2
Ci(1), Ci(2 ), Ci(x + 1), 0, I PI
```

Für Gleitkommazahlen wird der numerische Wert von Ci berechnet:

```
>> Ci(1.0), Ci(2.0 + 10.0*I)

0.3374039229, - 242.5252694 - 1185.8387 I
```

Beispiel 2. Ci ist singularär am Nullpunkt:

```
>> Ci(0)

Error: singularity [Ci]
```

Die negative reelle Halbachse ist ein Verzweigungsschnitt von Ci. Beim Überschreiten des Schnitts springen die Funktionswerte um $2\pi i$:

```
>> Ci(-1.0), Ci(-1.0 + 10^(-10)*I), Ci(-1.0 - 10^(-10)*I)

0.3374039229 + 3.141592654 I, 0.3374039229 + 3.141592654 I,
0.3374039229 - 3.141592654 I
```

Beispiel 3. Die Funktionen diff und float verarbeiten Ci:

```
>> diff(Ci(x), x, x, x), float(ln(3 + Ci(sqrt(PI))))

2 cos(x) cos(x) 2 sin(x)
----- - ----- + -----, 1.241299561
3 x 2
x x
```

Hintergründe:

- ⌘ Die Funktion $Ci(x) - \ln(x)$ ist analytisch. Ci hat eine logarithmische Singularität im Ursprung und einen Verzweigungsschnitt längs der negativen reellen Halbachse. Auf dieser Halbachse stimmen die Werte mit dem Grenzwert „von oben“ überein:

$$Ci(x) = \lim_{\epsilon \rightarrow 0_+} Ci(x + \epsilon i), \quad x \text{ real}, \quad x < 0.$$

- ⌘ Literatur: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

Änderungen:

☞ Ci ist eine neue Funktion.

D – Differentialoperator für Funktionen

$D(f)$, oder gleichbedeutend f' , berechnet die Ableitung der univariaten Funktion f .

$D([n_1, n_2, \dots], f)$ berechnet die partielle Ableitung $\frac{\partial}{\partial x_{n_1}} \frac{\partial}{\partial x_{n_2}} \dots f$ der multivariaten Funktion $f(x_1, x_2, \dots)$.

Aufruf(e):

☞ f'

☞ $D(f)$

☞ $D([n_1, n_2, \dots], f)$

Parameter:

f — eine Funktion oder ein funktionaler Ausdruck, ein Feld, eine Liste, ein Polynom, eine Menge oder eine Tabelle

n_1, n_2, \dots — Indizes: positive ganze Zahlen

Rückgabewert: die Ableitung oder partielle Ableitung: eine Funktion oder ein funktionaler Ausdruck, oder ein Objekt vom selben Typ wie f

Überladbar durch: f

Weitere Dokumentation: Abschnitt 7.1 des Tutoriums.

Verwandte Funktionen: `diff`, `int`, `poly`

Details:

☞ $D(f)$ liefert die Ableitung f' der univariaten Funktion f zurück. Alternativ kann statt $D(f)$ auch f' geschrieben werden.

☞ Sei f eine multivariate Funktion. Es bezeichne $D_n f$ die partielle Ableitung von f nach ihrem n -ten Argument. Dann liefert $D([n_1, \dots, n_k], f)$ die partielle Ableitung $D_{n_1} D_{n_2} \dots f$ zurück. Siehe Beispiel ??
 $D([], f)$ liefert f .

- ☞ Als f ist jegliches Objekt erlaubt, das eine Funktion repräsentieren kann. Insbesondere ist ein Ausdruck erlaubt, der aus solchen Objekten mittels arithmetischer Operatoren aufgebaut ist (funktionaler Ausdruck). Ein Bezeichner, der nicht zu den konstanten Bezeichnern CATALAN, EULER und π gehört, wird als „unbekannte“ Funktion aufgefasst; dasselbe gilt für Objekte von einem Datentyp, der auf dieser Seite nicht erwähnt wird. Siehe Beispiel ?? . Eine Zahl sowie jeder der o. g. konstanten Bezeichner wird als konstante Funktion aufgefasst. Siehe Beispiel ?? .
 - ☞ Ist f eine Liste, eine Menge, eine Tabelle oder ein Feld, so wird D auf jeden Eintrag von f angewandt. Siehe Beispiel ?? .
 - ☞ Ist f ein Polynom, so wird es als Polynomfunktion aufgefasst; seine Unbestimmten werden als Argumente der Polynomfunktion aufgefasst. Siehe Beispiel ?? .
 - ☞ Ist f eine Funktionsumgebung, eine Prozedur oder eine Kernfunktion, so kann D die Ableitung nur in den unter „Hintergrund“ beschriebenen Fällen bestimmen. Falls die Ableitung nicht bestimmt werden kann, so wird der unevaluierte Aufruf von D zurückgeliefert.
 - ☞ Verschachtelte *partielle* Ableitungen, d. h., partielle Ableitungen von partiellen Ableitungen, werden zu partiellen Ableitungen höherer Ordnung zusammengefasst. Siehe Beispiel ?? .
 - ☞ Die Ableitung einer Funktion f – geschrieben als $D(f)$ – und die partielle Ableitung von f nach ihrem einzigen Argument – geschrieben als $D([1], f)$ – werden voneinander unterschieden.
 - ☞ Die üblichen Ableitungsregeln sind implementiert:
 - $D(f + g) = D(f) + D(g)$
 - $D(f * g) = f * D(g) + g * D(f)$
 - $D(1/f) = -D(f) / f^2$
 - $D(f @ g) = D(g) * D(f) @ g$
- Beachten Sie bitte, das die Hintereinanderausführung von Funktionen als $f@g$ und *nicht* als $f(g)$ geschrieben wird.
- ☞ In MuPAD existieren zwei Ableitungsfunktionen: `diff` und `D`. `D` darf nur auf Funktionen angewandt werden; `diff` dient dagegen zum Differenzieren von Ausdrücken. `D`-Ausdrücke können mit Hilfe von `rewrite` in `diff`-Ausdrücke umgeschrieben werden. Siehe Beispiel ?? .
 - ☞ Um für symbolisches n die n -te Ableitung einer Funktion auszudrücken, kann der Operator `@@` verwendet werden. Siehe Beispiel ?? .
-

Beispiel 1. $D(f)$ berechnet die Ableitung der Funktion f :

```
>> D(sin), D(x -> x^2), D(id)
cos, 2 id, 1
```

Hierbei bezeichnet id die identische Abbildung. D funktioniert auch für komplexere funktionale Ausdrücke:

```
>> D(sin @ exp + 2*(x -> x*ln(x)) + id^2)
2 id + 2 ln + exp cos@exp + 2
```

Ist f ein Bezeichner ohne Wert, so wird ein symbolischer Aufruf von D zurückgeliefert:

```
>> delete f: D(f + sin)
D(f) + cos
```

Dasselbe gilt für Objekte eines Kern-Datentyps, der keine Funktionen repräsentiert:

```
>> D(NIL), D(point(3,2))
D(NIL), D(point(3, 2))
```

f' ist eine Kurzform für $D(f)$:

```
>> (f + sin)', (x -> x^2)', id'
D(f) + cos, 2 id, 1
```

Beispiel 2. Konstanten werden als konstante Funktionen aufgefasst:

```
>> PI', 3', (1/2)'
0, 0, 0
```

Beispiel 3. Die üblichen Ableitungsregeln sind implementiert. Listen und Mengen sind ebenfalls als Eingabe zulässig; in diesem Fall wird D auf jedes einzelne Element der Liste oder Menge angewendet:

```
>> delete f, g: D([f+g, f*g]); D({f@g, 1/f})
[D(f) + D(g), f D(g) + g D(f)]
{ D(f) }
{ D(g) D(f)@g, - ---- }
{ 2 }
{ f }
```

Beispiel 4. Die Ableitungen der meisten speziellen Funktionen der Library können bestimmt werden. Die Funktion `id` dient zur Darstellung der identischen Funktion:

```
>> D(tan); D(sin*cos); D(1/sin); D(sin@cos); D(2*sin + ln)
```

$$\begin{aligned} & \tan^2 + 1 \\ & \cos \cos^2 - \sin^2 \\ & -\frac{\cos^2}{\sin^2} \\ & -\sin \cos @ \cos \\ & \frac{1}{--} + 2 \cos \\ & id \end{aligned}$$

Beispiel 5. Die Ableitung von Prozeduren kann ebenfalls bestimmt werden:

```
>> f := x -> x^2:
    g := proc(x) begin tan(ln(x)) end:
    D(f), D(g)
```

$$\begin{aligned} & \tan @ \ln^2 + 1 \\ 2 \text{ id}, & \frac{\text{-----}}{\text{id}} \end{aligned}$$

Wir differenzieren eine Funktion, die von zwei Argumenten abhängt, durch Übergabe einer Liste von Indizes als erstes Argument an `D`. Im folgenden Beispiel differenzieren wir zunächst bezüglich des zweiten Arguments und dann das Ergebnis bezüglich des ersten Arguments:

```
>> D([1, 2], (x, y) -> sin(x*y))
(x, y) -> cos(x*y) - x*y*sin(x*y)
```

Die Reihenfolge der partiellen Ableitungen ist aber im angegebenen Beispiel gleichgültig:

```
>> D([2, 1], (x, y) -> sin(x*y))
(x, y) -> cos(x*y) - x*y*sin(x*y)
```

Beispiel 6. Ein Polynom wird als Polynomfunktion aufgefasst:

```
>> D(poly(x^2 + 3*x + 2, [x]))
      poly(2 x + 3, [x])
```

In diesem Beispiel wird ein Polynom f in zwei Variablen einmal nach seiner ersten Variable x und zweimal nach seiner zweiten Variable y differenziert:

```
>> f := poly(x^3*y^3, [x, y]):
      D([1, 2, 2], f) = diff(f, y, y, x)
      poly(18 x^2 y, [x, y]) = poly(18 x^2 y, [x, y])
>> delete f:
```

Beispiel 7. Verschachtelte Aufrufe von D werden als einzelner Aufruf umgeschrieben:

```
>> D([1], D([2], f))
      D([1, 2], f)
```

Dies gilt jedoch nicht für Aufrufe mit nur einem Argument, da $D(f)$ und $D([1], f)$ als verschieden angesehen werden:

```
>> D(D(f))
      D(D(f))
```

Beispiel 8. D darf nur auf Funktionen angewandt werden; diff ist dagegen nur für Ausdrücke sinnvoll:

```
>> D(sin), diff(sin(x), x)
      cos, cos(x)
```

Das Anwenden von D auf Ausdrücke oder von diff auf Funktionen ist sinnlos:

```
>> D(sin(x)), diff(sin, x)
      D(sin(x)), 0
```

rewrite bietet die Möglichkeit, Ausdrücke mit D in Ausdrücke mit diff umzuschreiben:

```
>> rewrite(D(f)(y), diff), rewrite(D(D(f))(y), diff)
      diff(f(y), y), diff(f(y), y, y)
```

Beispiel 9. Manchmal muss man die n -te Ableitung einer Funktion mit unbekanntem n verwenden. Dies kann mit dem Operator @@ erreicht werden. Als Beispiel geben wir eine Prozedur, die das k -te Taylorpolynom einer Funktion f an einem Punkt x_0 berechnet und x als Variable des Polynoms verwendet:

```
>> nthtaylorpoly:=
  (f, k, x, x0) -> _plus(((D@@n)(f)(x0) * (x-x0)^n / n!) $n=0..k):
  nthtaylorpoly(sin, 7, x, 0)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$$

Beispiel 10. Fortgeschrittene Benutzer können D auf eigene Funktionen ausdehnen (siehe „Hintergrund“). Hierzu ist die Funktion in eine Funktionsumgebung f einzubinden und das Verhalten von D im Eintrag "D" der Funktionsumgebung festzulegen. Dieser Eintrag muss zwei Fälle behandeln: er kann entweder mit nur einem Argument aufgerufen werden, das dann gleich f ist, oder mit zwei Argumenten, wobei dann das zweite gleich f ist. Im letzteren Fall ist das erste Argument eine Liste mit beliebig vielen Indizes; d. h., der Eintrag muss auch den Fall höherer partieller Ableitungen behandeln.

Angenommen, wir wissen über eine Funktion $f(t, x, y)$ nur, dass sie unendlich oft differenzierbar ist und die partielle Differenzialgleichung $\frac{\partial^2 f}{(\partial x)^2} + \frac{\partial^2 f}{(\partial y)^2} = \frac{\partial f}{\partial t}$ erfüllt. Um MuPAD zu veranlassen, partielle Ableitungen nach t zu ersetzen, kann man wie folgt vorgehen:

```
>> f:= funcenv((t, x, y) -> procname(args())):
  f::D :=
  proc(indexlist, ff)
    local
      n          : DOM_INT,    // number of d/dt to replace
      list_2_3   : DOM_LIST;  // list of indices of 2's and 3's
                                // these remain unchanged
  begin
    if args(0)<>2 then
      error("Wrong number of arguments")
    end_if;
    n          := nops(select(indexlist, _equal, 1));
    list_2_3   := select(indexlist, _unequal, 1);
    _plus(binomial(n, k) *
          hold(D)([2 $ 2*(n-k), 3 $ 2*k].list_2_3, hold(f))
    $k=0..n)
  end_proc:
  D([1, 2, 1], f)
```

$$D([2, 2, 2, 2, 2], f) + 2 D([2, 2, 3, 3, 2], f) + \\ D([3, 3, 3, 3, 2], f)$$

Hintergründe:

☞ Ist f ein Domain oder eine Funktionsumgebung mit Slot "D", dann wird dieser Slot zur Bestimmung der Ableitung verwendet. Im Slot muss eine Prozedur gespeichert sein, die die gleiche Syntax wie D hat. Insbesondere – und im Gegensatz zum Slot "diff" – muss der Slot auch höhere partielle Ableitungen berechnen können, da die Liste von Indizes beliebige Länge haben kann. Siehe Beispiel ??.

Ist kein Slot "D" vorhanden, so wird der erste Operand der Funktionsumgebung verwendet, um die Ableitung zu bestimmen.

☞ Ist f eine Prozedur oder eine Kern-Funktion (ein „ausführbares Objekt“), so wird f mit Hilfsbezeichnern als Argument aufgerufen. Das Ergebnis des Aufrufs wird mit der Funktion `diff` nach den Hilfsbezeichnern abgeleitet. Falls das Ergebnis von `diff` dann einen Ausdruck liefert, der als Funktion in den Hilfsbezeichnern aufgefasst werden kann, so wird diese Funktion als Ableitung von f zurückgeliefert. Kann die Ableitung auf diese Weise nicht bestimmt werden, so wird der unevaluierte Aufruf von D zurückgeliefert.

☞ Sei die Funktionsumgebung `sin` als Beispiel gegeben. Sie hat keinen Slot "D", also wird die für die Auswertung der Sinusfunktion verantwortliche Prozedur `op(sin, 1)` wie folgt zur Bestimmung der Ableitung verwendet. Diese Prozedur wird auf einen Hilfsbezeichner, etwa x , angewandt, und das Ergebnis wird mit `diff` nach x abgeleitet. Das Resultat ist `diff(sin(x), x) = cos(x)`. Also wird `cos` als Ableitung von `sin` zurückgeliefert.

Änderungen:

☞ Keine Änderungen.

DIGITS – die Anzahl signifikanter Stellen von Gleitpunktzahlen

Die Umgebungsvariable `DIGITS` legt die Anzahl der signifikanten Dezimalstellen in Gleitpunktzahlen fest. Die Voreinstellung ist `DIGITS = 10`.

Aufruf(e):

☞ `DIGITS`

☞ `DIGITS := n`

Parameter:

n — eine positive ganze Zahl kleiner 2^{31} .

Verwandte Funktionen: `float`, `Pref::floatFormat`,
`Pref::trailingZeroes`

Details:

- ☞ Gleitpunktzahlen werden durch Anwendung der Funktion `float` auf Zahlen oder numerische Ausdrücke erzeugt. Elementare Objekte werden durch die Gleitpunktzahlen auf eine relative Genauigkeit von $10^{-(\text{DIGITS})}$ approximiert, d. h., die ersten `DIGITS` Dezimalstellen sind korrekt. Siehe Beispiel ??.
- ☞ Bei einer arithmetischen Operation mit Gleitpunktzahlen werden nur die ersten `DIGITS` Dezimalstellen verwendet. Der numerische Fehler vererbt sich dabei und kann sich im Laufe von Rechnungen verstärken. Siehe Beispiel ??.
- ☞ Bei direkter Eingabe von Gleitpunktzahlen wie z. B. durch `x := 1.234` wird eine Zahl mit mindestens `DIGITS` internen Dezimalstellen erzeugt. Man beachte jedoch, daß ein Konvertierungsfehler nach der letzten Stelle auftreten kann, da die interne Darstellung binär ist.
Wird eine reelle Gleitpunktzahl mit mehr als `DIGITS` Stellen eingegeben, so werden auch alle überzähligen Stellen intern gespeichert. Diese werden jedoch in arithmetischen Operation ignoriert, wenn nicht `DIGITS` entsprechend hochgesetzt wird. Siehe Beispiel ??.
Insbesondere entstehen komplexe Gleitpunktzahlen durch Addition des Real- und Imaginärteils. Durch diese Addition gehen eventuell vorhandenen überzählige Stellen im Real- und Imaginärteil verloren.
- ☞ Der Wert von `DIGITS` kann jederzeit im Laufe einer Berechnung verändert werden. Wird `DIGITS` verringert, so werden in folgenden arithmetischen Operation nur die führenden `DIGITS` Stellen existierender Gleitpunktzahlen berücksichtigt. Wird `DIGITS` vergrößert, so werden existierende Gleitpunktzahlen intern mit binären Nullen aufgefüllt. Siehe Beispiel ??.
- ☞ In Abhängigkeit von `DIGITS` weisen einige Funktionen wie z. B. die trigonometrischen Funktionen Gleitpunktzahlen als Argument zurück, wenn diese zu ungenau sind. Siehe Beispiel ??.
- ☞ Nur die durch `DIGITS` als signifikant deklarierten Stellen einer Gleitpunktzahl werden auf dem Bildschirm angezeigt. Durch Aufruf der Präferenzen `Pref::floatFormat` und `Pref::trailingZeroes` kann die Ausgabe nach eigenen Wünschen eingestellt werden. Siehe Beispiel ??.
Mindestens eine Stelle nach dem Dezimalpunkt wird immer angezeigt. Ist sie nicht signifikant, so wird sie durch Null ersetzt. Siehe Beispiel ??.

☞ Intern werden Gleitpunktzahlen mit zusätzlichen „Schutzziffern“ erzeugt und abgespeichert. Diese werden auch von den arithmetischen Grundoperationen berücksichtigt, sind aber in der Bildschirmausgabe nicht sichtbar.

Beispielsweise konvertiert die Funktion `float` für `DIGITS = 10` exakte Zahlen in Gleitpunktapproximation mit einer internen Genauigkeit von etwa 19 Dezimalstellen. Die Anzahl der Schutzziffern ist jedoch nicht konstant, sondern hängt vom Wert von `DIGITS` ab. Beispielsweise wird für alle `DIGITS`-Werte von 10 bis 19 dieselbe interne Genauigkeit von 19 Dezimalstellen verwendet. Insbesondere gibt es damit für `DIGITS = 19` keine Schutzziffer. Siehe die Beispiele ?? und ??.

☞ Umgebungsvariable wie `DIGITS` sind globale Variablen. Nach der Rückkehr aus einer Prozedur, die den Wert von `DIGITS` verändert, ist der neue Wert auch außerhalb des Kontextes der Prozedur gültig! Mit `save DIGITS` kann die Gültigkeit des modifizierten Wertes von `DIGITS` auf die Prozedur eingeschränkt werden. Siehe Beispiel ??.

☞ Der voreingestellte Wert für `DIGITS` ist 10. Dieser Wert gilt nach dem Systemstart sowie nach einer Neuinitialisierung mittels `reset`. Auch der Befehl `delete DIGITS` stellt den Standardwert wieder her.

☞ Weitere Informationen sind auf der Hilfeseite von `float` zu finden.

Beispiel 1. Einige exakte Zahlen und numerischen Ausdrücke werden in Gleitpunktapproximationen konvertiert:

```
>> DIGITS := 10:
    float(PI), float(1/7), float(sqrt(2) + exp(3)), float(exp(-
20))

3.141592654, 0.1428571429, 21.49975049, 0.000000002061153622

>> DIGITS := 20:
    float(PI), float(1/7), float(sqrt(2) + exp(3)), float(exp(-
20))

3.1415926535897932385, 0.14285714285714285714,

21.49975048556076279, 0.000000002061153622438557828

>> delete DIGITS:
```

Beispiel 2. Die Fehlerfortpflanzung in numerischen Rechnungen wird demonstriert. Die folgende rationale Zahl approximiert `exp(2)` auf 17 Dezimalstellen:

wieder in die Dezimaldarstellung konvertiert. Hierbei werden die angehängten Binär-Nullen in 11 Dezimalziffern verwandelt, von denen einige ungleich 0 sein können. Der Aufruf `Pref::trailingZeroes(TRUE)` verhindert, daß eventuell vorhandene abschließende Nullen der Dezimalausgabe nicht abgeschnitten werden:

```
>> DIGITS := 10: a := float(1/9)

0.1111111111

>> Pref::trailingZeroes(TRUE): DIGITS := 30: a

0.1111111111111111111109605274000

>> Pref::trailingZeroes(FALSE): delete a, DIGITS:
```

Beispiel 5. Für die Gleitpunktauswertung der Sinus-Funktion wird das Argument zunächst auf das Standardintervall $[0, 2\pi]$ reduziert. Hierzu muss das Argument auf einige Stellen hinter dem Dezimalpunkt bekannt sein. Ist das Argument eine große Gleitpunktzahl, so sind die Nachkommastellen durch Rundungseffekte bestimmt:

```
>> DIGITS := 10: sin(float(2*10^20))

0.9576594803
```

Erhöht man DIGITS auf 50, so ist das Argument der Sinus-Funktion auf etwa 30 Nachkommastellen genau. Die ersten 30 Stellen des folgenden Resultats sind verlässlich:

```
>> DIGITS := 50: sin(float(2*10^20))

-0.9859057707420871849896773829691365946134713391129
```

Für sehr große Gleitpunktzahlen liefern MuPADs trigonometrische Funktionen Fehler, falls DIGITS nicht groß genug ist:

```
>> DIGITS := 10: sin(float(2*10^30))

Error: Loss of precision;
during evaluation of 'sin'

>> DIGITS := 50: sin(float(2*10^30))

0.17950046751493908795061771243112520647287791588203

>> delete DIGITS:
```

Beispiel 6. Es wird immer mindestens eine Nachkommastelle angezeigt. Im folgenden Beispiel wird die Zahl 3.9 als 3.0 ausgegeben, um anzuzeigen, dass die Ziffer 9 nicht signifikant ist:

```
>> DIGITS := 1: float(PI), 3.9, -3.2
      3.0, 3.0, -3.0

>> delete DIGITS:
```

Beispiel 7. Die Gleitpunktzahl $\text{float}(10^{40} \cdot 8/9)$ wird für mehrere DIGITS-Werte berechnet. Die Rundung auf eine ganze Zahl berücksichtigt die internen Schutzziffern, die hierdurch sichtbar werden:

```
>> for DIGITS in [9, 10, 11, 19, 20, 21, 28, 29, 30] do
      print("DIGITS" = DIGITS, round(float(10^40*8/9)))
end_for:

"DIGITS" = 9, 8888888887243627086483687557525021917184

"DIGITS" = 10, 8888888888888888888888888303079319556646240256

"DIGITS" = 11, 8888888888888888888888888303079319556646240256

"DIGITS" = 19, 8888888888888888888888888303079319556646240256

"DIGITS" = 20, 88888888888888888888888888888888827804909568

"DIGITS" = 21, 88888888888888888888888888888888827804909568

"DIGITS" = 28, 88888888888888888888888888888888827804909568

"DIGITS" = 29, 888888888888888888888888888888888888888864

"DIGITS" = 30, 888888888888888888888888888888888888888864
```

Die angezeigten Werte zeigen an, daß die interne Darstellung für DIGITS-Werte zwischen 10 und 19 übereinstimmt. Ab DIGITS = 20 wird die interne Darstellung erweitert und bis DIGITS = 28 beibehalten. Von DIGITS = 29 an wird die interne Darstellung wiederum erweitert etc.

Beispiel 8. Die folgende Prozedur erlaubt die Berechnung numerischer Approximationen auf eine vorgegebene Genauigkeit, ohne DIGITS als globale Variable zu ändern. Intern wird DIGITS auf die gewünschte Genauigkeit gesetzt, mit der die Gleitpunktapproximation berechnet werden soll. Wegen `save DIGITS` wird der Wert von DIGITS außerhalb der Prozedur nicht verändert:

```
>> Float := proc(x, digits)
    save DIGITS;
    begin
        DIGITS := digits;
        float(x);
    end_proc;
```

Die numerische Berechnung des folgenden Wertes x leidet an numerischer Auslöschung. Speziell für $\text{DIGITS} = 9$, wo keine internen Schutzstellen zur Verfügung stehen, ist der mit `float` berechnete Wert auf nur 3 führende Stellen genau. Mittels `Float` wird intern mit 30 Stellen ausgewertet. Das Ergebnis wird zwar wegen des außerhalb von `Float` gültigen DIGITS -Wertes auf nur 9 Stellen ausgegeben, die angezeigten Stellen sind aber alle korrekt:

```
>> x := PI^7 - exp(80131/10000): DIGITS := 9:
    float(x), Float(x, 30)

0.0277910233, 0.0277894265
```

```
>> delete Float, x, DIGITS:
```

Hintergründe:

- ☞ Wurde eine Gleitpunktzahl x mit hoher Genauigkeit erzeugt, und soll sie in der weiteren Rechnung nur mit geringerer Genauigkeit verwendet werden, so ist $x := x + 0.0$ die einfachste Möglichkeit, alle nicht signifikanten Stellen zu löschen und Speicher zu sparen.

Änderungen:

- ☞ Änderungen von DIGITS innerhalb einer Prozedur sind nun auch außerhalb der Prozedur gültig, wenn sie nicht mit `save DIGITS` implementiert ist.

Ei – die Integral-Exponentialfunktion

$\text{Ei}(x)$ stellt die Integral-Exponentialfunktion $\int_1^\infty e^{-xt}/t \, dt$ dar.

Aufruf(e):

☞ $\text{Ei}(x)$

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `Ci`, `exp`, `igamma`, `int`, `Si`

Details:

- ☞ Wenn x eine Gleitkommazahl ist, dann liefert $Ei(x)$ den numerischen Wert der Integral-Cosinusfunktion. Die speziellen Werte $Ei(\infty) = 0$ und $Ei(-\infty) = -\infty$ sind implementiert. Für alle anderen Argumente wird ein symbolischer Funktionsaufruf zurückgeliefert.
 - ☞ $Ei(x)$ ist äquivalent zu `igamma(0, x)` für reelle Argumente $x > 0$.
 - ☞ Das `float`-Attribut von `Ei` ist eine Kernfunktion, d. h., die numerische Auswertung ist schnell.
-

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> Ei(1), Ei(sqrt(2)), Ei(x + 1), Ei(infinity), Ei(-infinity)
1/2
Ei(1), Ei(2 ), Ei(x + 1), 0, -infinity
```

Für Gleitkommazahlen wird der numerische Wert von `Ei` berechnet:

```
>> Ei(-1000.0), Ei(1.0), Ei(12.3), Ei(2.0 + 10.0*I)
- 1.972045137e431 - 3.141592654 I, 0.2193839344,
0.0000003439533949, 0.003675663008 + 0.01234609005 I
```

Beispiel 2. `Ei` ist singulär am Nullpunkt:

```
>> Ei(0)
Error: singularity [Ei]
```

Die negative reelle Halbachse ist ein Verzweigungsschnitt von `Ei`. Beim Überschreiten des Schnitts springen die Funktionswerte um $2\pi i$:

```
>> Ei(-1.0), Ei(-1.0 + 10^(-10)*I), Ei(-1.0 - 10^(-10)*I)
```

$$\begin{aligned}
& - 1.895117816 - 3.141592654 \, i, \quad - 1.895117816 - 3.141592653 \, i, \\
& - 1.895117816 + 3.141592653 \, i
\end{aligned}$$

Beispiel 3. Die Funktionen diff, float, limit und series verarbeiten Ei:

```
>> diff(Ei(x), x, x, x), float(ln(3 + Ei(sqrt(PI))))
```

$$\begin{aligned}
& \frac{\exp(-x)}{x} - \frac{2 \exp(-x)}{x^2} - \frac{2 \exp(-x)}{x^3}, \quad 1.120796995
\end{aligned}$$

```
>> limit(Ei(2*x^2/(1+x)), x = infinity)
```

0

```
>> series(Ei(x), x = 0, 3),
series(Ei(x), x = infinity, 3),
series(Ei(x), x = -infinity, 3)
```

$$\begin{aligned}
& - (\ln(x) + \text{EULER}) + x - \frac{x^2}{4} + O(x^3),
\end{aligned}$$

$$\begin{aligned}
& \frac{\exp(-x)}{x} - \frac{\exp(-x)}{x^2} + O\left(\frac{\exp(-x)}{x^3}\right),
\end{aligned}$$

$$\begin{aligned}
& \frac{\exp(-x)}{x} - \frac{\exp(-x)}{x^2} + O\left(\frac{\exp(-x)}{x^3}\right)
\end{aligned}$$

Hintergründe:

☞ Die Funktion $Ei(x) + \ln(x)$ ist analytisch. Ei hat eine logarithmische Singularität im Ursprung und einen Verzweigungsschnitt längs der negativen reellen Halbachse. Auf dieser Halbachse stimmen die Werte mit dem Grenzwert „von oben“ überein:

$$Ei(x) = \lim_{\epsilon \rightarrow 0_+} Ei(x - \epsilon i), \quad x \text{ real}, \quad x < 0.$$

☞ $Ei(x)$ stimmt mit $Ei(1, x)$ aus der Funktionenfamilie

$$Ei(n, x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt$$

überein. Diese sind mit der unvollständigen Gamma-Funktion `igamma` über $Ei(n, x) = x^{n-1} \text{igamma}(1 - n, x)$ verknüpft. Man beachte, daß gegenwärtig die Gleitpunktevaluation von `igamma` nur für reelle Argumente $x > 0$ implementiert ist, während Ei für jedes komplexe $x \neq 0$ ausgewertet werden kann.

☞ Die spezielle Funktion $ei(x) = \int_{-\infty}^x e^t/t dt$ mit *reellem* x (für $x > 0$ aufzufassen als Cauchy-Hauptwertintegral) ist mit der implementierten Integral-Exponentialfunktion Ei über $ei(x) = -\text{Re}(Ei(-x))$ verknüpft, d. h.:

$$ei(x) = \begin{cases} -Ei(-x), & x < 0, \\ -Ei(-x) + i\pi, & x > 0. \end{cases}$$

☞ Literatur: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

Änderungen:

☞ Ei hieß früher `eint`.

FAIL – Anzeigen einer fehlgeschlagenen Berechnung

FAIL ist ein Schlüsselwort der MuPAD-Sprache. Viele Bibliotheksfunktionen benutzen FAIL als Rückgabewert für fehlgeschlagene Rechnungen oder nicht existierende Elemente.

Aufruf(e):

☞ FAIL

Verwandte Funktionen: `error`, `NIL`, `null`

Details:

☞ FAIL ist das einzige Element des Domains `DOM_FAIL`.

☞ FAIL wird als Rückgabewert für fehlgeschlagene Berechnungen verwendet. Auch das Anfordern nicht existierender Slots von Domains oder Funktionsumgebungen liefert FAIL. Mit diesem Verhalten können Bibliotheksfunktion Rechnungen austesten, ohne Fehler auszulösen.

☞ Jede Funktion sollte FAIL oder einen Fehler liefern, wenn mindestens eines ihrer Argumente FAIL ist.

Beispiel 1. Der folgende Versuch, `sqrt(3)` in eine ganze Zahl eines Restklassenrings zu verwandeln, muss scheitern:

```
>> poly(sqrt(3)*x, [x], Dom::IntegerMod(3))
```

FAIL

Die folgende Matrix ist nicht invertierbar. Man kann versuchen, sie zu invertieren, ohne daß ein Fehler ausgelöst wird:

```
>> A := matrix([[1, 1], [1, 1]]): 1/A
```

FAIL

Der "inverse"-Slot einer Funktionsumgebung liefert die Umkehrfunktion. Für die Sinus-Funktion ist die Inverse in MuPAD implementiert, aber nicht für die Dilogarithmus-Funktion:

```
>> sin::inverse, dilog::inverse
```

"arcsin", FAIL

```
>> delete A:
```

Beispiel 2. Die meisten MuPAD-Funktionen liefern bei Eingabe von FAIL einen Fehler oder FAIL:

```
>> poly(FAIL)
```

FAIL

```
>> sin(FAIL)
```

Error: argument must be of 'Type::Arithmetical' [sin]

Beispiel 3. FAIL evaluiert zu sich selbst:

```
>> FAIL, eval(FAIL), level(FAIL, 5)
```

FAIL, FAIL, FAIL

Änderungen:

- ☞ Keine Änderungen.
-

HISTORY – die maximale Anzahl Elemente in der History-Tabelle

Die Umgebungsvariable `HISTORY` bestimmt die maximale Anzahl von Einträgen in der History-Tabelle auf interaktiver Ebene.

Aufruf(e):

- ☞ `HISTORY`
- ☞ `HISTORY := n`

Parameter:

- `n` — eine nichtnegative ganze Zahl kleiner als 2^{31} .

Verwandte Funktionen: `history`, `last`

Details:

- ☞ Die Befehle, die interaktiv in einer MuPAD-Sitzung eingegeben, innerhalb von Prozeduren ausgeführt oder aus Dateien eingelesen werden, speichert MuPAD zusammen mit den dazugehörigen Ergebnissen in einer internen Datenstruktur: der History-Tabelle. `HISTORY` bestimmt die maximale Anzahl der Einträge in dieser Tabelle für die interaktiv eingegebenen Befehle. Nur die neuesten Einträge werden aufbewahrt.
 - ☞ Auf die Einträge der History-Tabelle kann mit `history` oder `last` zugegriffen werden.
 - ☞ Der Standard-Wert für `HISTORY` ist 20. `HISTORY` hat diesen Wert nach dem Starten von MuPAD und nach jedem Neustart mit `reset`. Der Befehl `delete HISTORY` setzt `HISTORY` auf den Standard-Wert zurück.
 - ☞ Innerhalb einer Prozedur ist die Anzahl der Einträge in der lokalen History-Tabelle immer 3, unabhängig vom Wert von `HISTORY`.
-

Beispiel 1. Im folgenden Beispiel wird für `HISTORY` der Wert 2 gesetzt. Danach werden nur noch die beiden neuesten Einträge in der History-Tabelle für die interaktive Eingabe gespeichert:

```
>> HISTORY := 2:
a := 1: b := 2: max(a, b):
history(history() - 1), history(history())
```

```
[ (b := 2), 2], [max(a, b), 2]
```

Der Versuch, auf den drittletzten Eintrag in der History-Tabelle zuzugreifen, führt zu einem Fehler:

```
>> history(history() - 2)

Error: Illegal argument [history]
```

Mit `delete` wird `HISTORY` auf seinen Standard-Wert zurückgesetzt:

```
>> delete HISTORY: HISTORY
```

20

Änderungen:

- ⌘ Mögliche Werte sind nur noch nichtnegative ganze Zahlen.
 - ⌘ `HISTORY` hat keine Auswirkungen mehr auf Prozeduren.
-

LEVEL – Auswertungstiefe von Bezeichnern

Die Umgebungsvariable `LEVEL` bestimmt die maximale Auswertungstiefe von Bezeichnern.

Aufruf(e):

- ⌘ `LEVEL`
- ⌘ `LEVEL := n`

Parameter:

`n` — eine positive ganze Zahl kleiner als 2^{31} .

Weitere Dokumentation: Kapitel 5 des Tutoriums.

Verwandte Funktionen: `context`, `eval`, `hold`, `level`, `MAXLEVEL`, `MAXDEPTH`, `val`

Details:

- ☞ Wenn ein MuPAD-Objekt evaluiert wird, werden die Bezeichner, die in dem Objekt vorkommen, durch ihre Werte ersetzt. Das geschieht rekursiv, d. h. wenn die Werte wieder Bezeichner enthalten, werden diese ebenfalls ersetzt. LEVEL bestimmt die maximale Rekursionstiefe dieses Prozesses.

- ☞ Die Evaluierung (Auswertung) von MuPAD-Objekten erfolgt aus technischer Sicht wie folgt.

Bei zusammengesetzten Objekten werden zuerst die Operanden rekursiv evaluiert, danach wird das Objekt selbst evaluiert. Wenn das Objekt z. B. eine Funktion mit Argumenten ist, werden zuerst die Argumente evaluiert, dann erfolgt der Funktionsaufruf mit den evaluierten Argumenten.

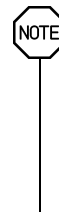
Für die Evaluierung von Bezeichnern bedeutet das: Die *aktuelle Auswertungstiefe* wird intern aufgezeichnet. Anfangs ist sie Null. Wenn ein Bezeichner in dem eben beschriebenen Evaluierungs-Prozess auftritt und die aktuelle Auswertungstiefe kleiner als LEVEL ist, wird der Bezeichner durch seinen Wert ersetzt, die Auswertungstiefe um Eins erhöht und die Evaluierung des Wertes des Bezeichners fortgesetzt. Nachdem die Evaluierung eines Bezeichners beendet wurde, wird die Auswertungstiefe wieder auf ihren ursprünglichen Wert gesetzt und mit der Evaluierung fortgefahren. Wenn bei der Evaluierung die aktuelle Auswertungstiefe gleich dem Wert von LEVEL ist, wird der Bezeichner nicht durch seinen Wert ersetzt.

- ☞ Der Standard-Wert von LEVEL bei der interaktiven Eingabe ist 100. Der Standard-Wert von LEVEL innerhalb von Prozeduren ist jedoch 1. Dadurch werden Bezeichner innerhalb von Prozeduren nur durch ihren Wert ersetzt, der aber nicht weiter evaluiert wird.



Der Wert von LEVEL kann innerhalb von Prozeduren geändert werden. In jeder neuen Prozedur ist er aber immer 1. Nach dem Verlassen einer Prozedur wird LEVEL auf seinen Wert vor dem Aufruf der Prozedur zurückgesetzt. Siehe Beispiel ??.

- ☞ Die Evaluierung von lokalen Variablen und formalen Parametern von Prozeduren (vom Typ DOM_VAR) wird von LEVEL nicht beeinflusst: sie werden immer mit der Auswertungstiefe 1 evaluiert. D.h. eine lokale Variable oder ein formaler Parameter einer Prozedur wird durch den entsprechenden Wert ersetzt, der aber nicht weiter evaluiert wird. Siehe Beispiel ??.



- ☞ LEVEL beeinflusst nicht die Evaluierung von Feldern und Polynomen. Siehe Beispiel ??.



☞ Die Funktion `eval` evaluiert ihr Argument mit der durch `LEVEL` gegebenen Auswertungstiefe, und das Ergebnis noch einmal mit derselben Auswertungstiefe.

Der Aufruf `level(object, n)` evaluiert `object` mit der Auswertungstiefe `n`, unabhängig von `LEVEL`.

☞ Wenn während der Evaluierung die Auswertungstiefe den Wert von `MAXLEVEL` erreicht, wird die Evaluierung mit einer Fehlermeldung abgebrochen. Mit diesem Vorgehen sollen Endlos-Rekursionen verhindert werden, die in Beispielen wie `delete a: a := a + 1; a` vorkommen. Hier wird `a` unendlich oft durch `a + 1` ersetzt. Wenn `MAXLEVEL` größer als `LEVEL` ist, ergibt das angegebene Beispiel keine Fehlermeldung!

Der Standard-Wert von `MAXLEVEL` ist 100, d.h. gleich dem Wert von `LEVEL` während der interaktiven Eingabe. Anders als bei `LEVEL` wird der Wert von `MAXLEVEL` innerhalb von Prozeduren *nicht* geändert, und dadurch rekursive Definitionen nicht erkannt. Die Hilfeseite von `MAXLEVEL` enthält dazu Beispiele.

☞ Der voreingestellte Wert für `LEVEL` ist 100 auf interaktiver Ebene. Dieser Wert gilt nach dem Systemstart sowie nach einer Neuinitialisierung mittels `reset`. Innerhalb einer Prozedur ist der voreingestellte Wert 1. Der Befehl `delete LEVEL` stellt den Standardwert wieder her.

Beispiel 1. Das folgende Beispiel zeigt die Auswirkungen verschiedener Werte von `LEVEL` bei interaktiver Eingabe:

```
>> delete a0, a1, a2, a3, a4, b: b := b + 1:
    a0 := a1: a1 := a2 + 2: a2 := a3 + a4: a3 := a4^2: a4 := 5:

>> LEVEL := 1: a0, a0 + a2, b;
    LEVEL := 2: a0, a0 + a2, b;
    LEVEL := 3: a0, a0 + a2, b;
    LEVEL := 4: a0, a0 + a2, b;
    LEVEL := 5: a0, a0 + a2, b;
    LEVEL := 6: a0, a0 + a2, b;
delete LEVEL:

    a1, a1 + a3 + a4, b + 1

                2
    a2 + 2, a2 + a4  + 7, b + 2

    a3 + a4 + 2, a3 + a4 + 32, b + 3

                2          2
    a4  + 7, a4  + 37, b + 4
```

32, 62, b + 5

32, 62, b + 6

Beispiel 2. Bei folgendem Aufruf wird der Bezeichner a voll evaluiert:

```
>> delete a, b, c:  
      a := b: b := c: c := 7: a
```

7

Nach der Zuweisung von 2 an LEVEL wird a nur noch mit der Auswertungstiefe 2 evaluiert:

```
>> LEVEL := 2: a;  
      delete LEVEL:
```

c

Wenn MAXLEVEL auch auf den Wert 2 gesetzt wird, ergibt die Evaluierung von a eine Fehlermeldung, auch wenn keine rekursive Definition vorliegt:

```
>> LEVEL := 2: MAXLEVEL := 2: a  
  
      Error: Recursive definition [See ?MAXLEVEL]  
  
>> delete LEVEL, MAXLEVEL:
```

Beispiel 3. Dieses Beispiel zeigt den Unterschied bei der Evaluierung von Bezeichnern und lokalen Variablen. Der Standard-Wert von LEVEL innerhalb von Prozeduren ist 1, d.h. ein globaler Bezeichner wird durch seinen Wert ersetzt, der Wert wird aber nicht weiter evaluiert. Dies kann durch einen größeren Wert von LEVEL innerhalb einer Prozedur geändert werden:

```
>> delete a0, a1, a2, a3:  
      a0 := a1 + a2: a1 := a2 + a3: a2 := a3^2 - 1: a3 := 5:  
      p := proc()  
            save LEVEL;  
            begin  
              print(a0, eval(a0)):  
              LEVEL := 2:  
              print(a0, eval(a0)):  
            end_proc:  
  
>> p()
```

$$a_1 + a_2, a_2 + a_3 + a_3^2 - 1$$

$$a_2 + a_3 + a_3^2 - 1, 53$$

Im Gegensatz dazu werden lokale Variablen bei der Evaluierung innerhalb von Prozeduren immer nur durch ihren Wert ersetzt, der nicht weiter evaluiert wird. Wenn `eval` auf ein Objekt angewendet wird, das eine lokale Variable enthält, wird der Wert der Variable mit der Auswertungstiefe `LEVEL` evaluiert:

```
>> q := proc()
      save LEVEL;
      local x;
      begin
        x := a0:
        print(x, eval(x)):
        LEVEL := 2:
        print(x, eval(x)):
      end_proc:
    q()
```

$$a_1 + a_2, a_2 + a_3 + a_3^2 - 1$$

$$a_1 + a_2, a_3^2 + 28$$

Der Befehl `x:=a0` weist den Wert des Bezeichners `a0`, und zwar den unevaluierten Ausdruck `a1+a2`, der lokalen Variable `x` zu. `x` wird bei jeder Evaluierung durch diesen Wert ohne weitere Evaluierung ersetzt, unabhängig von `LEVEL`.

Beispiel 4. `LEVEL` hat keine Auswirkungen auf die Evaluierung von Polynomen:

```
>> delete a, x: p := poly(a*x, [x]): a := 2: x := 3:
      p, eval(p);
      LEVEL := 1: p, eval(p);
      delete LEVEL:
```

`poly(a x, [x]), poly(a x, [x])`

`poly(a x, [x]), poly(a x, [x])`

Dasselbe gilt für Felder:

```
>> delete a, b:
  A := array(1..2, [a, b]):  T := table(a = b):
  a := 1:  b := 2:
  A, eval(A), T, eval(T);
  LEVEL := 1: A, eval(A), T, eval(T);
  delete LEVEL:

      +-      -+  +-      -+  table(    table(
      | a, b |, | a, b |,   a = b ,   a = b
      +-      -+  +-      -+  )          )

      +-      -+  +-      -+  table(    table(
      | a, b |, | a, b |,   a = b ,   a = b
      +-      -+  +-      -+  )          )
```

Änderungen:

☞ Keine Änderungen.

Line-Editor – Editieren von Zeilen in der MuPAD Terminalversion

Diese Seite beschreibt den Zeileneditor der Terminalversion MuPADs.

Details:

- ☞ Der Zeileneditor steht nur in der Terminalversion von MuPAD zur Verfügung, nicht unter dem X-Window-System, auf dem Macintosh oder unter Windows.
- ☞ Bei der interaktiven Eingabe kann mit dem Text-Editor die aktuelle Zeile bearbeitet werden. Die meisten Befehle werden durch das gleichzeitige Drücken der Taste <Ctrl> (<Steuerung>) in Verbindung mit der angegebenen Taste aktiviert. Die möglichen Kommandos sind:

<Ctrl-A>	– Cursor an den Zeilenanfang
<Ctrl-Y>	– Cursor an den Anfang des vorherigen Wortes. Dies funktioniert nicht unter Solaris, wo <Ctrl-Y> ein nicht-POSIX Signal zur Unterbrechung der Sitzung auslöst.
<Ctrl-B>, <Cursor-Links>	– Cursor ein Zeichen nach links
<Ctrl-F>, <Cursor-Rechts>	– Cursor ein Zeichen nach rechts
<Ctrl-E>	– Cursor an das Zeilenende
<Ctrl-U>	– Löschen der gesamten Eingabezeile
<Ctrl-W>	– Löschen von der Cursor-Position bis zum Anfang des vorherigen Wortes
<Ctrl-H>	– Löschen des Zeichens links vom Cursor
<Ctrl-D>	– Löschen des Zeichens an der Cursorposition
<Ctrl-T>	– Löschen des nächsten Wortes
<Ctrl-K>	– Löschen bis zum Ende der Zeile
<Ctrl-L>	– Einfügen der letzten Eingabezeile vor der Cursorposition
<Ctrl-P>, <Cursor-Oben>	– Wiederherstellung der letzten Eingabezeile. Wiederholtes Drücken von <Ctrl-P> durchläuft die letzten Eingabezeilen. Steht der Cursor nicht am Anfang der Zeile, werden die vorherigen Zeilen nach Einträgen durchsucht, die mit den Zeichen der aktuellen Zeile übereinstimmen.
<Ctrl-N>, <Cursor-Unten>	– Analog zu <Ctrl-P>, aber die vorherigen Zeilen werden in umgekehrter Reihenfolge durchlaufen.
<Ctrl-C>	– Führt während des Editierens zum Abbruch der Eingabe und Rückkehr zum MuPAD-Prompt. Folgt <Ctrl-C> direkt nach dem MuPAD-Prompt, so wird der MuPAD-Prozess beendet. Wird während einer MuPAD-Berechnung <Ctrl-C> ausgeführt, so wird die laufende Rechnung abgebrochen.
<TAB>	– Vervollständigung der bisherige Eingabe zum Namen eines Objektes, d.h., zu einem Funktionsnamen, einem Bibliotheksnamen oder dem Namen einer Variablen. Existieren mehrere Objekte, deren Namen mit der momentanen Eingabe beginnt, so werden alle vervollständigten Namen auf dem Bildschirm ausgegeben.

Beispiel 1. Die <TAB>-Vervollständigung wird demonstriert: nach Eingabe von `lin` wird die <TAB>-Taste gedrückt. Das System reagiert durch Rückgabe der drei Systemobjekte, die mit `lin` beginnen. Dies sind die Bibliotheken `linalg`, `linopt` sowie die Systemfunktion `linsolve`:

```
>> lin<TAB>
```



```
linalg, linopt, linsolve
```

Die folgende Eingabe listet alle Funktionen der linalg Bibliothek auf, die mit ‚a‘ beginnen:

```
>> linalg::a<TAB>
```

```
linalg::addCol, linalg::addRow, linalg::adjoint, linalg::angle
```

Die folgende Eingabe listet alle Funktionen der groebner Bibliothek auf:

```
>> groebner::<TAB>
```

```
groebner::dimension, groebner::gbasis, groebner::normalf,  
groebner::spoly
```

Änderungen:

⌘ Die <TAB>-Vervollständigung wurde eingeführt.

MAXDEPTH – Verhinderung von Endlosrekursionen bei Prozeduraufrufen

Die Umgebungsvariable MAXDEPTH bestimmt die maximale Rekursionstiefe von geschachtelten Prozeduraufrufen. Wenn diese Rekursionstiefe erreicht wird, dann wird eine Fehlermeldung ausgegeben.

Aufruf(e):

⌘ MAXDEPTH

⌘ MAXDEPTH := n

Parameter:

n — eine positive ganze Zahl kleiner als 2^{31} .

Verwandte Funktionen: eval, freeze, LEVEL, level, MAXLEVEL, proc

Details:

⌘ Mit Hilfe von MAXDEPTH wird versucht, Endlosrekursionen von Prozeduraufrufen wie in `p := x -> p(x): p(0)` zu erkennen und zu vermeiden. Würde in diesem Beispiel die Rekursionstiefe nicht beschränkt, dann würde sich die Prozedur p unendlich oft selbst aufrufen, und das System würde sich „aufhängen“.

- ☞ Wenn während der Evaluierung eines Objektes die Rekursionstiefe MAXDEPTH erreicht wird, bricht die Evaluierung mit einer Fehlermeldung ab.
- ☞ Gleichermäßen dient die Umgebungsvariable MAXLEVEL dazu, um Endlos-Rekursionen bei der Ersetzung von Bezeichnern durch ihre Werte zu verhindern. Die entsprechende Hilfeseite enthält weitere Informationen und Beispiele.
- ☞ Der voreingestellte Wert für MAXDEPTH ist 500. Dieser Wert gilt nach dem Systemstart sowie nach einer Neuinitialisierung mittels `reset`. Auch der Befehl `delete MAXDEPTH` stellt den Standardwert wieder her.
- ☞ MAXDEPTH ist eine globale Variable. Mit der Anweisung `save MAXDEPTH` bei der Definition einer Prozedur bleiben Änderungen von MAXDEPTH auf diese Prozedur beschränkt.

Beispiel 1. Die Evaluierung von Objekten, die durch eine Endlos-Rekursion definiert werden, ergibt eine Fehlermeldung:

```
>> p := proc() begin p() end_proc: p()

Error: Recursive definition [See ?MAXDEPTH];
during evaluation of 'p'
```

Dies gilt auch für verschachtelte rekursive Definitionen:

```
>> p := proc(x) begin q(x + 1)^2 end_proc:
  q := proc(y) begin p(x) + 2 end_proc:
  p(0)

Error: Recursive definition [See ?MAXDEPTH];
during evaluation of 'p'
```

Beispiel 2. Wird die maximale Rekursionstiefe erreicht, dann bedeutet das nicht unbedingt, daß eine Endlosrekursion vorliegt. Die folgende rekursive Prozedur berechnet die Fakultät einer nicht-negativen ganzen Zahl. Setzt man die maximale Rekursionstiefe auf einen kleineren Wert als zur Berechnung von 4! nötig, dann wird eine Fehlermeldung ausgegeben:

```
>> factorial := proc(n) begin
  if n = 0 then 1
  else n*factorial(n - 1)
  end_if
end_proc:
MAXDEPTH := 4: factorial(5)
```

```
Error: Recursive definition [See ?MAXDEPTH];  
during evaluation of 'factorial'
```

Die Rekursionstiefe 5 ist zur Berechnung von 4! jedoch ausreichend. Die Anweisung `delete MAXDEPTH` setzt `MAXDEPTH` auf den Standard-Wert 500:

```
>> MAXDEPTH := 5: factorial(5); delete MAXDEPTH:
```

120

Änderungen:

☞ Keine Änderungen.

MAXLEVEL – Verhinderung von Endlosrekursionen bei der Evaluierung

Die Umgebungsvariable `MAXLEVEL` bestimmt die maximale Auswertungstiefe von Bezeichnern. Wenn diese Auswertungstiefe erreicht wird, dann wird eine Fehlermeldung ausgegeben.

Aufruf(e):

☞ `MAXLEVEL`
☞ `MAXLEVEL := n`

Parameter:

`n` — eine ganze Zahl zwischen 2 und 2^{31} .

Verwandte Funktionen: `context`, `eval`, `hold`, `LEVEL`, `level`, `MAXDEPTH`, `val`

Details:

☞ Wenn ein MuPAD-Objekt evaluiert wird, werden die Bezeichner, die in dem Objekt vorkommen, durch ihre Werte ersetzt. Das geschieht rekursiv, d. h. wenn die Werte wieder Bezeichner enthalten, werden diese ebenfalls ersetzt. `MAXLEVEL` bestimmt die maximale Rekursionstiefe dieses Prozesses. Wird diese Auswertungstiefe erreicht, dann wird eine Fehlermeldung ausgegeben.

- ☞ Mit Hilfe von `MAXLEVEL` wird versucht, Endlosrekursionen bei der Ersetzung von Bezeichnern durch ihre Werte wie in `delete a: a := a + 1; a` zu erkennen und zu vermeiden. Würde in diesem Beispiel die Auswertungstiefe nicht beschränkt, dann würde der Bezeichner `a` unendlich oft durch `a + 1` ersetzt, und das System würde sich „aufhängen“.
- ☞ Gleichermäßen dient die Umgebungsvariable `MAXDEPTH` dazu, um Endlos-Rekursionen bei Funktionsaufrufen zu verhindern. Die entsprechende Hilfeseite enthält weitere Informationen und Beispiele.
- ☞ Es besteht ein enger Zusammenhang zwischen `LEVEL` und `MAXLEVEL`. Wenn die aktuelle Auswertungstiefe während des Ersetzungsvorgangs den Wert von `LEVEL` erreicht, wird der Vorgang abgebrochen und eventuell noch vorhandene Bezeichner bleiben unevaluiert, aber es wird keine Fehlermeldung ausgegeben.
 Wenn also `MAXLEVEL > LEVEL` gilt, hat `MAXLEVEL` keine Auswirkung. `MAXLEVEL` hat standardmäßig den selben Wert 100 wie `LEVEL`. Der Wert von `LEVEL` innerhalb von Prozeduren ist jedoch 1, daher hat `MAXLEVEL` üblicherweise keine Auswirkungen in Prozeduren.
- ☞ Es gibt einige bemerkenswerte Unterschiede zwischen `LEVEL` und `MAXLEVEL`. Der Wert von `LEVEL` hängt vom Kontext ab, d. h. ob die Evaluierung auf interaktiver Ebene oder in Prozeduren stattfindet. Außerdem reagieren einige Systemfunktionen wie `context` und `level` *nicht* auf den aktuellen Wert von `LEVEL`. Im Gegensatz dazu ist der Wert von `MAXLEVEL` global gültig. `MAXLEVEL` dient als letzte Begrenzung, wenn die Steuerung der Evaluierung mittels `LEVEL` fehlschlägt.
- ☞ Der voreingestellte Wert für `MAXLEVEL` ist 100. Dieser Wert gilt nach dem Systemstart sowie nach einer Neuinitialisierung mittels `reset`. Auch der Befehl `delete MAXLEVEL` stellt den Standardwert wieder her.
- ☞ `MAXLEVEL` ist eine globale Variable. Mit der Option `save MAXLEVEL` bei der Definition einer Prozedur bleiben Änderungen von `MAXLEVEL` auf diese Prozedur beschränkt.

Beispiel 1. Die Evaluierung von Objekten, die durch eine Endlos-Rekursion definiert werden, ergibt eine Fehlermeldung:

```
>> delete a: a := a + 1: a
Error: Recursive definition [See ?MAXLEVEL]
```

Dies gilt auch für verschachtelte rekursive Definitionen:

```
>> delete a, b: a := b^2: b := a + 1: b
Error: Recursive definition [See ?MAXLEVEL]
```

Beispiel 2. Wenn MAXLEVEL kleiner oder gleich LEVEL ist, wie die Voreinstellung auf interaktiver Ebene, werden Objekte bis zur Tiefe MAXLEVEL - 1 vollständig evaluiert. Die Evaluierung wird mit einer Fehlermeldung abgebrochen, wenn die rekursive Auswertung die Tiefe MAXLEVEL erreicht, auch wenn gar keine rekursive Definition vorliegt:

```
>> delete a, b, c, d:
      a := b: b := c: c := 7: d := d + 1:
      MAXLEVEL := 2: LEVEL := 2: c
```

7

```
>> a
```

```
Error: Recursive definition [See ?MAXLEVEL]
```

```
>> d
```

```
Error: Recursive definition [See ?MAXLEVEL]
```

Andererseits hat MAXLEVEL keine Auswirkungen, wenn sein Wert größer als LEVEL ist. Dann wird jedes Objekt maximal bis zur Tiefe LEVEL ausgewertet, und es tritt kein Fehler auf:

```
>> MAXLEVEL := 3: a, b, c, d
      c, 7, 7, d + 2
```

Insbesondere hat MAXLEVEL üblicherweise innerhalb von Prozeduren keine Auswirkungen, wo LEVEL standardmäßig den Wert 1 hat:

```
>> MAXLEVEL := 2:
      p := proc() begin a, d end_proc:
      p();
      delete MAXLEVEL, LEVEL:
      b, d + 1
```

Änderungen:

☞ Keine Änderungen.

NIL – das einzige Element des Domains DOM_NIL

NIL ist ein Schlüsselwort der MuPAD-Sprache, welches das einzige Element des Domains DOM_NIL repräsentiert.

Aufruf(e):

⌘ NIL

Verwandte Funktionen: `delete`, `FAIL`, `null`

Details:

- ⌘ Das Kern-Domain `DOM_NIL` hat nur ein einziges Element. `NIL` ist ein Schlüsselwort der MuPAD-Sprache, welches dieses Element repräsentiert. `NIL` wird durch die Evaluierung nicht verändert, siehe `DOM_NIL`.
 - ⌘ Meistens wird `NIL` verwendet, um „fehlende“ oder „leere“ Einträge in einer Datenstruktur zu repräsentieren. Das „leere Objekt“, das von der Funktion `null` zurückgeliefert wird, ist für diesen Zweck nicht geeignet, da es während der Evaluation aus dem meisten Behältern (wie Listen, Mengen oder Ausdrücken) entfernt wird.
 - ⌘ Wenn ein neues Array vom Kern-Domain `DOM_ARRAY` erzeugt wird, werden dessen Elemente mit `NIL` vorbelegt. Die Funktion `op` liefert `NIL` für solche nicht initialisierten Elemente. Man beachte aber, das im Gegensatz dazu beim indizierten Zugriff auf nicht initialisierte Elemente der indizierte Ausdruck selbst zurückgeliefert wird.
 - ⌘ Lokale Variablen von Prozeduren, die mit `proc` definiert werden, werden mit `NIL` vorbelegt. Trotzdem wird eine Warnung ausgegeben, falls solch eine nicht explizit initialisierte Variable verwendet wird.
 - ⌘ In früheren Versionen von MuPAD wurde `NIL` verwendet, um den Wert von Bezeichnern, Array- oder Tabelleneinträgen zu löschen, indem `NIL` an den Bezeichner oder Eintrag zugewiesen wurde. Dies funktioniert nicht mehr: man muss nun `delete` verwenden, um Werte zu löschen. `NIL` ist nun ein gültiger Wert von Bezeichnern, Array- oder Tabelleneinträgen.
-

Beispiel 1. Im Gegensatz zum „leeren Objekt“, das von `null` geliefert wird, wird `NIL` nicht aus Listen oder Mengen entfernt:

```
>> [1, NIL, 2, NIL], [1, null(), 2, null()],
    {1, NIL, 2, NIL}, {1, null(), 2, null()}

      [1, NIL, 2, NIL], [1, 2], {NIL, 1, 2}, {1, 2}
```

Beispiel 2. NIL wird verwendet, um „fehlende“ Einträge in Prozeduren zu repräsentieren. Die einfachste denkbare Prozedur hat z. B. folgende Operanden:

```
>> op(proc() begin end)

NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL
```

Das erste NIL z. B. steht für die leere Argument-Liste, das zweite für die fehlenden lokalen Variablen und das dritte für die fehlenden Prozedur-Optionen.

Beispiel 3. Array-Elemente werden mit NIL vorbelegt, soweit sie nicht anderweitig definiert werden. Man beachte aber, dass beim indizierten Zugriff auf solche Elemente der indizierte Ausdruck zurückgeliefert wird:

```
>> A := array(1..2): A[1], op(A,1)

A[1], NIL

>> delete A:
```

Beispiel 4. Lokale Variablen in Prozeduren werden implizit mit NIL vorbelegt. Trotzdem wird eine Warnung ausgegeben, falls eine nicht explizit initialisierte Variable verwendet wird:

```
>> p := proc() local l; begin print(l) end: p():

Warning: Uninitialized variable 'l' used;
during evaluation of 'p'

NIL

>> delete p:
```

Beispiel 5. NIL kann wie jedes andere Datum einem Bezeichner oder indiziertem Bezeichner zugewiesen werden. Solch eine Zuweisung löscht nicht länger den Wert des Bezeichners:

```
>> a := NIL: b[1] := NIL: a, b[1]

NIL, NIL

>> delete a, b:
```

Änderungen:

- ☞ Durch die Zuweisung von NIL an Bezeichner, Array- oder Tabelleneinträge werden deren Werte nicht länger gelöscht.
-

NOTEBOOKFILE, NOTEBOOKPATH – Name von Notebook-Datei und -Verzeichnis

Die Umgebungsvariablen NOTEBOOKFILE bzw. NOTEBOOKPATH enthalten den absoluten Dateinamen bzw. das Verzeichnis des aktuellen Notebooks in MuPAD Pro für Windows in Form einer Zeichenkette.

Aufruf(e):

- ☞ NOTEBOOKFILE
- ☞ NOTEBOOKPATH

Verwandte Funktionen: LIBPATH, READPATH, TESTPATH, UNIX, WRITEPATH

Details:

- ☞ Die Umgebungsvariable NOTEBOOKFILE enthält den Namen des aktuellen Notebooks, das mit dem MuPAD-Kern verbunden ist.
 - ☞ Die Umgebungsvariable NOTEBOOKPATH enthält den Namen des Verzeichnisses, in dem das Notebook enthalten ist.
 - ☞ Diese Variablen sind z. B. nützlich, um Dateien einzulesen, die relativ zum Notebook-Verzeichnis gespeichert wurden.
Sie haben nur dann einen Wert, wenn das Notebook einen Namen besitzt. Das ist generell der Fall, falls ein vorhandenes Notebook geöffnet wurde oder ein neues Notebook gesichert wurde.
 - ☞ Der durch NOTEBOOKFILE gegebene Name ist ein absoluter Dateiname.
 - ☞ Beide Variablen dürfen nur gelesen und nicht verändert werden. Man kann NOTEBOOKFILE insbesondere keinen neuen Wert zuweisen, um den Namen des Notebooks zu ändern.
 - ☞ NOTEBOOKFILE und NOTEBOOKPATH sind nur in MuPAD Pro für Windows definiert. Auf anderen Plattformen sind beide Variablen ganz normale Bezeichner.
-

Beispiel 1. In MuPAD Pro für Windows kann man für ein Notebook Start-Kommandos angeben. Diese werden ausgeführt, sobald das Notebook mit einem Kern verbunden wird. (Siehe das Menü Datei/Eigenschaften in der On-Line Hilfe.)

In den Start-Kommandos kann man NOTEBOOKPATH verwenden, um eine Datei „my_init.mu“ einzulesen, die im Notebook-Verzeichnis gespeichert ist:

```
>> fread(NOTEBOOKPATH."my_init.mu")
```

Änderungen:

☞ NOTEBOOKFILE und NOTEBOOKPATH sind neue Umgebungsvariablen.

O – das Domain der Ordnungsterme (Landau Symbole)

$O(f, x = x_0)$ repräsentiert das Landau-Symbol $O(f, x \rightarrow x_0)$.

Aufruf(e):

☞ $O(f <, x = x_0, y = y_0, \dots >)$

Parameter:

f — ein arithmetischer Ausdruck, als Funktion in x, y etc. zu interpretieren
 x, y, \dots — die Variablen: Bezeichner
 x_0, y_0, \dots — die Grenzpunkte: arithmetische Ausdrücke

Rückgabewert: ein Element des Domains O .

Verwandte Funktionen: `asympt, limit, series, taylor`

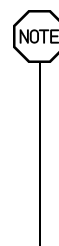
Details:

☞ Für eine Funktion f in den Variablen (x, y, \dots) stellt das Landau Symbol

$$g := O(f, x \rightarrow x_0, y \rightarrow y_0, \dots)$$

eine Funktion in diesen Variablen mit der folgenden Eigenschaft dar: es existiert eine Konstante c und eine Umgebung des Grenzpunktes (x_0, y_0, \dots) , sodass $|g| \leq c|f|$ für alle Punkte (x, y, \dots) dieser Umgebung gilt.

Typischerweise wird diese Symbolik benutzt, um die Ordnungsterme („Fehlerterme“) von Reihenentwicklungen zu kennzeichnen. Man beachte jedoch, dass die von den Funktionen `asympt`, `series` und `taylor` gelieferten Reihenentwicklungen Ordnungsterme als Teil der Datenstrukturen `Series::Puisseux` und `Series::gseries` repräsentieren und *nicht* das Domain `O` verwenden.



- ⌘ Bei Angabe von Gleichungen $x = x_0$, $y = y_0$ etc. wird f als Funktion in den angegebenen Variablen aufgefasst. Alle weiteren in f enthaltenen Bezeichner werden als konstante Parameter angesehen.

Werden keine Variablen und Grenzpunkte angegeben, so werden alle in f vorhandenen Bezeichner als Variablen benutzt, die jeweils gegen den Grenzpunkt `0` streben.

- ⌘ Zur Zeit können nur endliche Grenzpunkte angegeben werden.
- ⌘ Variablen mit dem Grenzpunkt `0` werden auf dem Bildschirm nicht ausgegeben.
- ⌘ Die Variablen des Ordnungsterms können mit der Funktion `indets` erfragt werden. Die Grenzpunkte werden durch die Funktion `O::points` geliefert.
- ⌘ Die arithmetischen Operationen $+$, $-$, $*$, $/$ und $^$ sind für Ordnungsterme überladen.
- ⌘ Automatische Vereinfachungen sind gegenwärtig auf polynomiale Ausdrücke f beschränkt. Univariate polynomiale Ausdrücke werden auf das führende Monom der Entwicklung um den Grenzpunkt reduziert. In multivariaten polynomialen Ausdrücken werden alle Summanden entfernt, die durch Summanden niedriger Ordnung exakt teilbar sind. Für nicht-polynomiale Ausdrücke werden lediglich ganzzahlige Faktoren entfernt.

Beispiel 1. Für polynomiale Ausdrücke werden einige Vereinfachungen durchgeführt:

```
>> O(x^4 + 2*x^2), O(7*x^3), O(x, x = 1)
```

```
      2      3
O(x ), O(x ), O(1, x = 1)
```

Ein Grenzpunkt `0` wird in der Ausgabe unterdrückt:

```
>> O(1), O(1, x = 1), O(x^2/(y + 1), x = 0, y = -1, z = PI)
```

```
      /      2      \
      |      x      |
O(1), O(1, x = 1), O| ----, z = PI, y = -1 |
      \ y + 1      /
```

Die arithmetischen Operationen sind für Ordnungsterme überladen:

```
>> 7*O(x), O(x^2) + O(x^13), O(x^3) - O(x^3), O(x^2)^2 + O(x^4)
      2      3      4
O(x), O(x ), O(x ), O(x )
```

Beispiel 2. Bei multivariaten polynomialen Ausdrücken werden diejenigen Summanden entfernt, die exakt durch Summanden niedriger Ordnung teilbar sind:

```
>> O(15*x*y^2 + 3*x^2*y + x^2*y^2)
      2      2
O(5 x y + x y)

>> O(x + x^2*y) = O(x)*O(1 + x*y)
      O(x) = O(x)
```

Beispiel 3. Es wird gezeigt, wie man auf die Variablen und Grenzpunkte eines Ordnungsterms zugreifen kann:

```
>> a := O(x^2*y^2)
      2      2
O(x y )

>> indets(a) = O::indets(a), O::points(a)
      {x, y} = {x, y}, {x = 0, y = 0}

>> delete a:
```

Änderungen:

☞ Keine Änderungen.

ORDER – die Standardanzahl von Termen in Reihenentwicklungen

Die Umgebungsvariable ORDER legt die Anzahl der Terme fest, die das System standardmäßig zurückgibt, wenn es eine Reihenentwicklung berechnet.

Aufruf(e):

⌘ ORDER
⌘ ORDER := n

Parameter:

n — eine positive ganze Zahl kleiner als 2^{31} . Der Standardwert ist 6.

Verwandte Funktionen: `asympt`, `limit`, `O`, `series`, `taylor`

Details:

- ⌘ Die Funktionen `taylor`, `series` und `asympt` haben ein optionales drittes Argument, welches die gewünschte Anzahl der Terme der berechneten Reihenentwicklung angibt, vom Term mit der kleinsten Ordnung an gezählt (relative Ordnung). Wenn dieses optionale Argument fehlt, dann wird dafür der Wert von `ORDER` eingesetzt.
 - ⌘ `ORDER` kann auch die von der Funktion `limit` gelieferten Ergebnisse beeinflussen.
 - ⌘ Löschen von `ORDER` mittels „delete ORDER“ setzt `ORDER` auf den Standardwert 6 zurück. Ein Aufruf der Funktion `reset` stellt ebenfalls den Standardwert wieder her.
 - ⌘ In manchen Fällen kann es vorkommen, dass die von `taylor`, `series` oder `asympt` zurückgegebene Anzahl von Termen nicht mit dem Wert von `ORDER` übereinstimmt. Siehe Beispiel ??.
-

Beispiel 1. In diesem Beispiel werden die ersten 6 Terme der Reihenentwicklung der Exponentialfunktion am Ursprung berechnet:

```
>> series(exp(x), x = 0)
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O(x^6)$$

Um die ersten 10 Terme zu bekommen, wird der Funktion `series` ein drittes Argument mitgegeben:

```
>> series(exp(x), x = 0, 10)
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + \frac{x^9}{362880} + O(x^{10})$$

Alternativ kann man den Wert von ORDER erhöhen. Dies beeinflusst alle zukünftigen Aufrufe von `series` und allen anderen Funktionen, die Reihenentwicklungen berechnen:

```
>> ORDER := 10: series(exp(x), x = 0)
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + \frac{x^9}{362880} + O(x^{10})$$

```
>> taylor(x^2/(1 - x), x = 0)
```

$$x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12} + O(x^{13})$$

Mit dem folgenden Befehl wird ORDER auf seinen Standardwert 6 gesetzt:

```
>> delete ORDER: taylor(x^2/(1 - x), x = 0)
```

$$x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + O(x^8)$$

Beispiel 2. Hier sind einige Beispiele, in denen die Anzahl der Terme in der von `series` berechneten Reihenentwicklung vom Wert von ORDER abweicht:

```
>> ORDER := 3:
```

```
>> series((x^2 + x)^2, x = 0)
```

$$x^2 + O(x^3)$$

```
>> series(exp(x) - 1 - x, x = 0)
```

$$\frac{x^2}{2} + O(x^3)$$

```
>> series(1/(1 - sqrt(x)), x = 0)
```

$$1 + \frac{1}{2}x + \frac{3}{2}x^2 + x^3 + O(x^{5/2})$$

```
>> delete ORDER:
```

Änderungen:

☞ Keine Änderungen.

Pfadvariablen – Suchpfade für Dateien

LIBPATH bestimmt die Verzeichnisse, in denen die Funktionen `loadlib` und `loadproc` nach Bibliotheksdateien suchen.

READPATH bestimmt die Verzeichnisse, in denen die Funktion `read` nach Dateien sucht.

WRITEPATH bestimmt das Verzeichnis, in dem die Funktionen `fopen`, `fprint`, `write` und `protocol` Dateien ablegen.

Aufruf(e):

```
☞ LIBPATH := path
☞ READPATH := path
☞ WRITEPATH := path
```


Parameter:

`path` — der Verzeichnispfad: eine Zeichenkette oder eine Folge von Zeichenketten.

Verwandte Funktionen: `fclose`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `loadlib`, `loadproc`, `NOTEBOOKFILE`, `NOTEBOOKPATH`, `package`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `TESTPATH`, `UNIX`, `write`, `WRITEPATH`

Details:

- ☞ LIBPATH legt die Verzeichnisse fest, in denen Dateien von den Funktionen `loadproc` und `loadlib` gesucht werden. LIBPATH ist in der UNIX/Linux-Version von MuPAD auf `$MuPAD_ROOT_PATH/share/lib` voreingestellt und kann beim Aufruf von MuPAD mittels der Option `-l` undefiniert werden.
- ☞ READPATH bezeichnet den Suchpfad der Funktion `read`. `read` sucht zuerst in den durch READPATH gegebenen Verzeichnissen, dann in dem „aktuellen Arbeitsverzeichnis“ und zuletzt in dem durch LIBPATH angegebenen Verzeichnis.
- ☞ Den Variablen LIBPATH und READPATH können mehrere Verzeichnisse als eine durch Kommata getrennte Folge von Zeichenketten zugewiesen werden. Jedes Element der Folge stellt ein Verzeichnis dar, in dem Dateien gesucht werden.

- ☞ `WRITEPATH` bezeichnet das Verzeichnis, in das von den Funktionen `fopen`, `fprint`, `write` und `protocol` Dateien geschrieben werden, die nicht über einen vollständigen (absoluten) Pfadnamen spezifiziert wurden. Ist `WRITEPATH` nicht definiert, so wird in das „aktuelle Arbeitsverzeichnis“ geschrieben.
- ☞ Man beachte, daß das „aktuelle Arbeitsverzeichnis“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.
- ☞ Die durch die Pfadvariablen angegebenen Verzeichnisse müssen in Verbindung mit einem Dateinamen ein auf dem Betriebssystem gültigen Pfadnamen ergeben. 
- ☞ Pfadnamen sind systemabhängig. In UNIX/Linux beginnt ein Unterverzeichnis mit einem `/`, auf einem Macintosh mit einem `:` und unter Windows mit einem einfachen „Backslash“ `\`.
- ☞ Man beachte, daß ein einfacher Backslash in einer MuPAD-Zeichenkette durch zwei aufeinanderfolgende Backslashes eingegeben wird. Beispielsweise ist die zum Pfad „C:\Programs\MuPAD“ zugehörige MuPAD Zeichenkette als `"C:\\Programs\\MuPAD"` einzugeben.
- ☞ Die Funktion `pathname` ermöglicht die systemunabhängige Eingabe von Pfadnamen.
- ☞ Für Entwickler von MuPAD-Bibliotheken ist es nützlich, sich ein lokales Verzeichnis mit derselben Struktur wie die Bibliotheksdateien im Installationsverzeichnis anzulegen, um geänderte Bibliotheksdateien dort zu speichern. Wenn man in der Benutzer-Konfigurationsdatei `user-init.mu` den Namen des lokalen Verzeichnisses vorne an `LIBPATH` anhängt, dann sucht MuPAD zuerst in diesem Verzeichnis nach Bibliotheksdateien, und dann erst im Systemverzeichnis. Siehe Beispiel ??.

Beispiel 1. Dieses Beispiel zeigt, wie man einen `READPATH` definiert. Es kann mehr als ein Pfad angegeben werden. `read` sucht in den durch `READPATH` gegebenen Verzeichnissen nach zu öffnenden Dateien. Dieses Beispiel erzeugt nur auf UNIX/Linux-Systemen einen gültigen Pfadnamen, da die Pfadtrennsymbole direkt in den Zeichenketten angegeben sind:

```
>> READPATH := "math/lib/", "math/local/"
               "math/lib/", "math/local/"
```

Es ist guter Programmierstil, plattformunabhängige Pfade zu verwenden. Dazu dient die Funktion `pathname`:

```
>> READPATH := pathname("math", "lib"),
               pathname("math", "local")
```

```
"math/lib/", "math/local/"
```

Alle Pfadvariablen können auf ihre Standardwerte zurückgesetzt werden, indem man sie löscht:

```
>> delete READPATH:
```

Beispiel 2. Die Pfadvariable WRITEPATH akzeptiert nur einen Pfad:

```
>> WRITEPATH := "math/lib/", "math/local/"
```

```
Error: Illegal argument [WRITEPATH]
```

Beispiel 3. Vorsicht beim Ändern von LIBPATH. Sie können ihre MuPAD-Sitzung unbenutzbar machen:

```
>> LIBPATH := "does/not/exist":
```

```
linalg::det
```

```
Error: can't read file 'LIBFILES/linalg.mu' [loadproc]
```

Die Standardeinstellung läßt sich durch Löschen von LIBPATH wiederherstellen:

```
>> delete LIBPATH:
```

```
linalg::det
```

```
proc linalg::det(A) ... end
```

Das Ändern des LIBPATH ist nützlich bei der Bibliotheksentwicklung. Sie können ein Verzeichnis "mylib" mit derselben Verzeichnisstruktur wie in der MuPAD Bibliothek aufbauen. Angenommen, sie haben eine geänderte Version der Systemfunktion `linalg::det` in der Datei "mylib/LINALG/det.mu" installiert. Wenn die Funktion `linalg::det` das erste Mal aufgerufen wird, so versucht MuPAD die Datei "LINALG/det.mu" zu lesen. Da das Verzeichnis "mylib" diese Datei enthält, wird sie anstelle der entsprechenden Datei in der Standard-Bibliothek gelesen:

```
>> reset(): Pref::verboseRead(2):
```

```
LIBPATH := pathname("mylib"), LIBPATH:
```

```
linalg::det
```

```
loading package 'linalg' [<YourMuPADpath>/share/lib/]
```

```
reading file mylib/LINALG/det.mu
```

```
proc linalg::det(A) ... end
```

Bitte restaurieren Sie Ihre Sitzung:

```
>> delete LIBPATH: Pref::verboseRead(0):
```


Änderungen:

- ☞ Die Suchpfade hießen früher `LIB_PATH`, `READ_PATH`, `WRITE_PATH`.
-

PRETTYPRINT – Steuerung der formatierten Ausgabe

Die Umgebungsvariable `PRETTYPRINT` bestimmt, ob MuPAD-Ergebnisse im eindimensionalen oder zweidimensionalen Format ausgegeben werden.

Aufruf(e):

- ☞ `PRETTYPRINT`
- ☞ `PRETTYPRINT := value`

Parameter:

`value` — `TRUE` oder `FALSE`

Verwandte Funktionen: `print`, `TEXTWIDTH`

Details:

- ☞ `PRETTYPRINT` steuert den Pretty-Printer, der für die formatierte zweidimensionale Ausgabe zuständig ist. Hat `PRETTYPRINT` den Wert `TRUE`, so wird der Pretty-Printer für die Ausgabe benutzt.
- ☞ Der Standardwert für `PRETTYPRINT` ist `TRUE`. Dieser Wert gilt nach dem Systemstart sowie nach einer Neuinitialisierung mittels `reset`. Auch der Befehl `delete PRETTYPRINT` stellt den Standardwert wieder her.
- ☞ Auf Windows-Plattformen hat `PRETTYPRINT` normalerweise keinen Einfluss auf das Ausgabeformat, wenn der Formelsatz aktiviert ist. Eine Ausnahme bilden sehr breite MuPAD-Ausgaben, für welche `PRETTYPRINT` das Ausgabeformat bestimmt, auch wenn der Formelsatz aktiviert ist.

Standardmäßig ist der Formelsatz eingeschaltet. Man kann ihn ein- bzw. ausschalten, indem man „Optionen“ aus dem Menü „Ansicht“ des MuPAD-Fensters wählt und dann auf „Formelsatz für die Ausgaben“ klickt.

Beispiel 1. Die folgende Anweisung schaltet den Pretty-Printer ab:

```
>> PRETTYPRINT := FALSE  
  
FALSE
```

Nun gibt MuPAD alle Ergebnisse in eindimensionaler Form aus:

```
>> series(sin(x), x = 0, 15)
```

```
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/3991680\
00*x^11 + 1/6227020800*x^13 + O(x^15)
```

Durch Rücksetzen von PRETTYPRINT auf TRUE erhält man wieder das zwei-dimensionale Ausgabeformat:

```
>> PRETTYPRINT := TRUE: series(sin(x), x = 0, 15)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} - \frac{x^{11}}{39916800} + \frac{x^{13}}{6227020800} + O(x^{15})$$

Änderungen:

☞ PRETTYPRINT hieß früher PRETTY_PRINT.

Re, Im – Real- und Imaginärteil eines arithmetischen Ausdrucks

$\text{Re}(z)$ liefert den Realteil von z .

$\text{Im}(z)$ liefert den Imaginärteil von z .

Aufruf(e):

☞ $\text{Re}(z)$

☞ $\text{Im}(z)$

Parameter:

z — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: z

Seiteneffekte: Die Ergebnisse dieser Funktionen hängen von Eigenschaften von Bezeichnern ab, die mittels `assume` gesetzt worden sind. Siehe Beispiel ??.

Verwandte Funktionen: `abs`, `assume`, `conjugate`, `rectform`, `sign`

Details:

- ☞ `Re` und `Im` dienen zum Extrahieren des Real- bzw. Imaginärteils konstanter arithmetischer Ausdrücke. Besonders für Zahlen vom Typ `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` oder `DOM_COMPLEX` wird der Real- und Imaginärteil direkt und damit sehr schnell bestimmt.
 - ☞ `Re` und `Im` behandeln auch symbolische Ausdrücke, wobei Eigenschaften von Bezeichnern berücksichtigt werden (siehe `assume`). Bezeichner ohne Eigenschaften werden als komplexwertig angenommen. Siehe Beispiel ??.
Für beliebige symbolische Ausdrücke jedoch sind `Re` und `Im` nicht in der Lage, den Real- bzw. Imaginärteil von z zu bestimmen. In diesen Fällen kann die Funktion `rectform` nützlich sein (siehe Beispiel ??). Es ist zu beachten, daß Berechnungen mit `rectform` zeitintensiv sind.
 - ☞ Wenn `Re` nicht den vollständigen Realteil von z extrahieren kann, so enthält das Ergebnis symbolische `Re`- oder `Im`-Aufrufe. Das Gleiche gilt für die Funktion `Im`. Siehe Beispiel ??.
-

Beispiel 1. Der Real- und Imaginärteil von $2e^{1+i}$ lauten:

```
>> Re(2*exp(1 + I)), Im(2*exp(1 + I))  
2 cos(1) exp(1), 2 sin(1) exp(1)
```

Beispiel 2. `Re` und `Im` sind nicht in der Lage, den vollständigen Real- bzw. Imaginärteil von symbolischen Ausdrücken, in denen Bezeichner enthalten sind, zu bestimmen. Jedoch können sie die Eingabe in vereinfachter Form zurückliefern, was in den folgenden beiden Beispielen auftritt:

```
>> delete u, v: Re(u + v*I), Im(u + v*I)  
Re(u) - Im(v), Im(u) + Re(v)  
  
>> delete z: Re(z + 2), Im(z + 2)  
Re(z) + 2, Im(z)
```

Bezeichner ohne Werte repräsentieren standardmäßig beliebige komplexe Zahlen. Diese Voreinstellung läßt sich mit der Funktion `assume` ändern. Die folgende Anweisung teilt dem System mit, daß der Bezeichner z nur reelle Zahlen repräsentiert:

```
>> assume(z, Type::Real): Re(z + 2), Im(z + 2)  
z + 2, 0
```

Beispiel 3. Hier ist ein weiteres Beispiel, für das `Re` und `Im` nicht in der Lage sind, den Real- bzw. Imaginärteil eines symbolischen Ausdrucks zu bestimmen:

```
>> delete z: Re(exp(I*sin(z))), Im(exp(I*sin(z)))

Re(exp(I sin(z))), Im(exp(I sin(z)))
```

Das kann aber mit der weitaus mächtigeren Funktion `rectform` erreicht werden, die einen komplexwertigen Ausdruck `z` in dessen Real- und Imaginärteil zerlegt:

```
>> r := rectform(exp(I*sin(z)))

cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z))) +

(sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))) I
```

Mit `Re(r)` und `Im(r)` erhält man den Real- bzw. Imaginärteil von `r`:

```
>> Re(r)

cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))

>> Im(r)

sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))
```

Beispiel 4. Symbolische Ausdrücke vom Typ `"Re"` und `"Im"` haben die Eigenschaft `Type::Real`, auch wenn keinem Bezeichner, der im symbolischen Ausdruck enthalten ist, eine Eigenschaft zugewiesen wurde:

```
>> is(Re(sin(2*x)), Type::Real)

TRUE
```

Beispiel 5. Fortgeschrittene Benutzer können die Funktionen `Re` und `Im` für ihre eigenen speziellen mathematischen Funktionen erweitern (siehe „Hintergründe“ unten). Dazu wird eine solche Funktion in eine Funktionsumgebung eingebettet und das Verhalten von `Re` und `Im` für diese Funktion in den Slots `"Re"` und `"Im"` der Funktionsumgebung implementiert.

Enthält nun ein Ausdruck `z` einen Teilausdruck der Form `f(u, ...)`, so bewirken die Funktionen `Re` und `Im` die Aufrufe `f::Re(u, ...)` bzw. `f::Im(u, ...)` der betreffenden Slot-Routinen, um den Real- und Imaginärteil zu bestimmen.

Wir zeigen diese Funktionsweise am Beispiel der Sinus-Funktion und dem Slot `"Re"`. Die Funktionsumgebung `sin` besitzt natürlich schon den Slot `"Re"`. Wir nennen unsere Funktionsumgebung daher `Sin`, um nicht die Systemfunktion `sin` zu überschreiben:

```

>> Sin := funcenv(Sin):
    Sin::Re := proc(u) // compute Re(Sin(u))
        local r, s;
    begin
        r := Re(u);
        if r = u then
            return(Sin(u))
        elif not has(r, {hold(Im), hold(Re)}) then
            s := Im(u);
            if not has(s, {hold(Im), hold(Re)}) then
                return(Sin(r)*cosh(s))
            end_if
        end_if;
        return(FAIL)
    end:
>> Re(Sin(2)), Re(Sin(2 + 3*I))

```

Sin(2), Sin(2) cosh(3)

Anhand des Rückgabewertes FAIL erkennt die Funktion Re, daß "Sin::Re" nicht in der Lage ist, den Realteil des Eingabeausdrucks zu bestimmen. Das Ergebnis ist dann ein symbolischer Re-Aufruf:

```

>> delete f, z: Re(2 + Sin(f(z)))

    Re(Sin(f(z))) + 2

```

Hintergründe:

☞ Wenn in z ein Teilausdruck der Form $f(u, \dots)$ vorkommt, wobei f eine Funktionsumgebung ist, so versucht Re den Slot "Re" von f aufzurufen, um den Realteil von $f(u, \dots)$ zu bestimmen. Damit läßt sich die Funktionalität von Re für benutzereigene mathematische Funktionen erweitern.

Der Slot "Re" wird mit den Argumenten u, \dots von f aufgerufen. Falls die Slot-Routine $f::\text{Re}$ nicht in der Lage ist, den Realteil von $f(u, \dots)$ zu bestimmen, so muss sie den Wert FAIL zurückliefern.

Falls f keinen Slot "Re" besitzt oder falls die Routine $f::\text{Re}$ den Wert FAIL zurückliefert, so wird $f(u, \dots)$ in dem zurückgelieferten Ausdruck durch den symbolischen Aufruf $\text{Re}(f(u, \dots))$ ersetzt.

Analog dazu ruft die Funktion Im den Slot "Im" von f auf.

Siehe Beispiel ??.

☞ Entsprechendes gilt für Domainelemente: Tritt ein Domainelement d eines Bibliotheksdomains T als Teilausdruck in z auf, so versucht Re den

Slot "Re" dieses Domains mit dem Element d als Argument aufzurufen, um den Realteil von d zu bestimmen.

Falls die Routine $T::\text{Re}$ nicht in der Lage ist, den Realteil von d zu bestimmen, so muss sie den Wert `FAIL` zurückliefern.

Falls T keinen Slot "Re" besitzt oder falls die Routine $T::\text{Re}$ den Wert `FAIL` zurückliefert, so wird d in dem zurückgelieferten Ausdruck durch den symbolischen Aufruf $\text{Re}(d)$ ersetzt.

Analog dazu ruft die Funktion Im den Slot "Im" des entsprechenden Domains auf.

Änderungen:

- Re und Im berücksichtigen Eigenschaften von Bezeichnern, die mit `assume` gesetzt worden sind.
-

RootOf – die Nullstellenmenge eines Polynoms

$\text{RootOf}(f, x)$ definiert die Menge der Lösungen der Polynomgleichung $f = 0$ nach x , ohne dabei diese Lösungen in Form von Wurzelausdrücken auszuzeichnen.

Aufruf(e):

- $\text{RootOf}(f, x)$
- $\text{RootOf}(f)$

Parameter:

- f — Gleichung oder Ausdruck
- x — Unbekannte

Rückgabewert: RootOf liefert den unevaluierten Aufruf zurück, wobei das zweite Argument x ergänzt wird, falls es vorher fehlte.

Verwandte Funktionen: `numeric::polyroots`, `poly`, `solve`

Details:

- RootOf dient zur symbolischen Darstellung der Nullstellenmenge eines Polynoms; dies ist die einzige Möglichkeit der Darstellung, wenn ein Polynom nicht durch Radikale auflösbar ist. RootOf erscheint hauptsächlich als Ergebnis von `solve` und damit verwandter Funktionen.

- ☞ Der Parameter f muss entweder ein arithmetischer Ausdruck, der ein Polynom in x repräsentiert, oder eine Gleichung $p=q$ sein, wobei p und q arithmetische Ausdrücke sind, die Polynome in x repräsentieren. Im letzteren Fall repräsentiert das Ergebnis die Nullstellen von $p-q$ bezüglich x .
- ☞ Das Polynom f braucht nicht irreduzibel, ja nicht einmal quadratfrei zu sein. Auch wenn f mehrfache Nullstellen besitzt, repräsentiert `RootOf` die Nullstellen nur mit Vielfachheit 1.
- ☞ Fehlt x , so wird hierfür die einzige in f vorkommende Variable verwendet; f muss in diesem Fall genau eine Variable enthalten.
- ☞ x braucht kein Bezeichner oder indizierter Bezeichner sein; zulässig ist jeder Ausdruck, der weder rational noch konstant ist.
- ☞ Ein unevaluierter Aufruf von `RootOf` mittels `float` in eine Menge von Gleitkommazahlen konvertiert werden, falls f außer x keine weiteren Unbestimmten enthält.

Beispiel 1. Jeder der folgenden Aufrufe repräsentiert die Nullstellen des Polynoms $x^3 - x^2$, d. h. die Menge $\{0, 1\}$.

```
>> RootOf(x^3 - x^2, x), RootOf(x^3 = x^2, x)
      3      2      3      2
RootOf(x  - x , x), RootOf(x  = x , x)

>> RootOf(x^3 - x^2), RootOf(x^3 = x^2)
      3      2      3      2
RootOf(x  - x , x), RootOf(x  = x , x)

>> RootOf(poly(x^3 - x^2, [x]), x)
      3      2
RootOf(x  - x , x)
```

Im allgemeinen wird `RootOf` jedoch nur verwendet, wenn sich die Nullstellen nicht explizit symbolisch darstellen lassen.

Beispiel 2. Das erste Argument an `RootOf` darf Parameter enthalten:

```
>> RootOf(y*x^2 - x + y^2, x)
      2      2
RootOf(y  - x + x  y, x)
```

Die Menge aller Nullstellen eines Polynoms wird wie ein Ausdruck behandelt; sie kann beispielsweise nach einem freien Parameter differenziert werden. Das Ergebnis ist die Menge der Ableitungen der Nullstellen; es wird mittels `RootOf`, d. h. durch Angabe eines Minimalpolynoms, ausgedrückt:

```
>> diff(%, y)
```

$$\text{RootOf}(2 y^4 - x^4 + y^3 + 4 y^2 x - y^2 x^2 + 4 y x^2, x)$$

Für reduzible Polynome kann das Ergebnis ein Vielfaches des korrekten Minimalpolynoms sein.

Beispiel 3. Objekte vom Typ `RootOf` werden von `solve` zurückgeliefert, wenn die Nullstellen eines Polynoms nicht durch Wurzelausdrücke dargestellt werden können:

```
>> solve(x^5 + x + 7, x)
```

$$\text{RootOf}(X^5 + X + 7, X)$$

Um Gleitkommadarstellungen der Nullstellen zu erhalten, braucht nicht erneut `solve` aufgerufen zu werden, denn ein `RootOf` kann unmittelbar in eine endliche Menge von `DOM_FLOATs` konvertiert werden:

```
>> float(%)
```

$$\{-1.410813851, -0.5084694089 + 1.368616488 \text{ I}, \\ -0.5084694089 - 1.368616488 \text{ I}, \\ 1.213876335 + 0.9241881109 \text{ I}, 1.213876335 - 0.9241881108 \text{ I}\}$$

Beispiel 4. Die Funktion `sum` kann Summen über alle Nullstellen eines Polynoms berechnen:

```
>> sum(i^2, i = RootOf(x^3 + a*x^2 + b*x + c, x))
```

$$a^2 - 2 b$$

```
>> sum(1/(z + i), i = RootOf(x^4 - y*x + 1, x))
```

$$\frac{y^3 + 4 y^2 z}{y^4 z + z^4 + 1}$$

Änderungen:

- ⌘ `RootOf` war bislang ein Domain; ein Aufruf lieferte ein Element dieses Domains. Nunmehr ist `RootOf` eine Funktionsumgebung, die den Aufruf unevaluiert zurückliefert.
-

`Si` – die Integral-Sinusfunktion

`Si(x)` stellt die Integral-Sinusfunktion $\int_0^x \sin(t)/t \, dt$ dar.

Aufruf(e):

- ⌘ `Si(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `Ci`, `Ei`, `int`, `sin`

Details:

- ⌘ Wenn `x` eine Gleitkommazahl ist, dann liefert `Si(x)` den numerischen Wert der Integral-Sinusfunktion. Die speziellen Werte $Si(0) = 0$ und $Si(\pm\infty) = \pm\pi/2$ sind implementiert. Für alle anderen Argumente wird ein symbolischer Funktionsaufruf zurückgeliefert.
 - ⌘ Die Spiegelung $Si(x) = -Si(-x)$ wird bei negativen ganzzahligen oder rationalen Argumenten benutzt. Sie wird auch bei symbolischen Produkten angewendet, die einen solchen Faktor enthalten. Siehe Beispiel ??.
 - ⌘ Das `float`-Attribut von `Si` ist eine Kernfunktion, d. h., die numerische Auswertung ist schnell.
-

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> Si(0), Si(1), Si(sqrt(2)), Si(x + 1), Si(infinity)
```

$$0, \operatorname{Si}(1), \operatorname{Si}\left(2^{\frac{1}{2}}\right), \operatorname{Si}(x+1), \frac{\pi}{2}$$

Für Gleitkommazahlen wird der numerische Wert von Si berechnet:

```
>> Si(-5.0), Si(1.0), Si(2.0 + 10.0*I)
-1.549931245, 0.9460830704, 1187.409493 - 242.5252717 I
```

Beispiel 2. Die Spiegelung $\operatorname{Si}(-x) = -\operatorname{Si}(x)$ wird bei negativen reellen Zahlen und Produkten mit solchen Faktoren angewendet:

```
>> Si(-3), Si(-3/7), Si(-sqrt(2)), Si(-x/7), Si(-0.3*x)
-Si(3), -Si(3/7), -Si(2^{1/2}), -Si(x/7), -Si(0.3 x)
```

Diese „Normalisierung“ wird jedoch weder bei komplexen Zahlen noch bei Ausdrücken angewendet, die keine Produkte sind:

```
>> Si(-3 - I), Si(3 + I), Si(x - 1), Si(1 - x)
Si(-3 - I), Si(3 + I), Si(x - 1), Si(1 - x)
```

Beispiel 3. Die Funktionen diff, float, limit und series verarbeiten die Integral-Sinusfunktion:

```
>> diff(Si(x), x, x, x), float(ln(3 + Si(sqrt(PI))))
```

$$\frac{2 \sin(x)}{x^3} - \frac{\sin(x)}{x^2} - \frac{2 \cos(x)}{x^2}, 1.502020149$$

```
>> limit(Si(2*x^2/(1+x)), x = infinity)
```

$$\frac{\pi}{2}$$

```
>> series(Si(x), x = 0), series(Si(x), x = infinity, 3)
```

$$x - \frac{x^3}{18} + \frac{x^5}{600} + O(x^6), \quad \frac{\pi}{2} - \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} + O\left(\frac{1}{x^3}\right)$$

Hintergründe:

- ☞ \sin ist eine analytische Funktion.
- ☞ Literatur: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

Änderungen:

- ☞ Keine Änderungen.

TESTPATH – Schreibpfad für `prog::test`

TESTPATH bestimmt das Verzeichnis, in das die Funktion `prog::test` ihre Dateien schreibt.

Aufruf(e):

- ☞ `TESTPATH := path`

Parameter:

`path` — ein zulässiger Verzeichnispfad: eine Zeichenkette.

Verwandte Funktionen: `LIBPATH`, `NOTEBOOKFILE`, `NOTEBOOKPATH`, `prog::test`, `READPATH`, `UNIX`, `WRITEPATH`

Details:

- ☞ TESTPATH ist ein spezieller Pfad für Ergebnisdateien, die von `prog::test` erzeugt werden.
- ☞ Die Hilfeseite zu "``Pfadvariablen``" erklärt, wie man Pfadvariablen korrekt und systemunabhängig definiert.

Beispiel 1. Eine Pfadangabe muss mit einem Verzeichnistrennzeichen enden. Hier ist ein Beispiel unter UNIX:

```
>> TESTPATH := "testresults/"  
  
"testresults/"
```

Änderungen:

⌘ TESTPATH hieß früher TEST_PATH.

TEXTWIDTH – die maximale Anzahl von Zeichen einer Ausgabezeile

Der Wert der Umgebungsvariablen TEXTWIDTH legt die maximale Anzahl der Zeichen fest, mit der eine Zeile auf dem Bildschirm ausgegeben wird.

Aufruf(e):

⌘ TEXTWIDTH
⌘ TEXTWIDTH := n

Parameter:

n — eine positive ganze Zahl kleiner als 2^{31} . Der Standardwert ist 75.

Verwandte Funktionen: fprintf, PRETTYPRINT, print

Details:

- ⌘ Bildschirmausgaben werden umbrochen, falls sie mehr als TEXTWIDTH Zeichen pro Zeile benötigen.
- ⌘ Löschen von TEXTWIDTH mittels „delete TEXTWIDTH“ setzt TEXTWIDTH auf den Standardwert zurück. Ein Aufruf der Funktion reset stellt ebenfalls den Standardwert wieder her.
- ⌘ Der Mindestwert von TEXTWIDTH hängt von der Länge des Prompt-Zeichens ab, welches durch Pref::promptString gegeben ist: Der Mindestwert ist 7 plus der Länge des Prompt-Zeichens. Das Prompt-Zeichen ist als ">> " voreingestellt, also ist in diesem Fall 10 der Mindestwert von TEXTWIDTH.
- ⌘ TEXTWIDTH wird bei der Ausgabe in eine Textdatei mittels fprintf auf den Maximalwert $2^{31} - 1$ gesetzt. Dadurch findet kein zusätzlicher Zeilenumbruch statt.

- ☞ TEXTWIDTH hat keinen Einfluss auf den Formelsatz, der in einigen Benutzungsoberflächen von MuPAD verfügbar ist.
 - ☞ Für die Ausgabe der meisten Beispiele dieses Handbuchs wurde TEXTWIDTH auf 63 gesetzt.
-

Beispiel 1. Die Zeilenlänge wird zuerst auf 20 Zeichen gesetzt:

```
>> oldTEXTWIDTH := TEXTWIDTH:
      TEXTWIDTH := 20: 30!
```

```
2652528598121910586\
36308480000000
```

Danach wird wieder der vorherige Wert verwendet:

```
>> TEXTWIDTH := oldTEXTWIDTH: 30!

26525285981219105863630848000000
```

Beispiel 2. Die folgende Prozedur erzeugt durch Hinzufügen einer passenden Anzahl von Leerzeichen eine rechtsbündige Ausgabe:

```
>> myprint := proc(x) local l; begin
      if domtype(x) <> DOM_STRING then
        x := expr2text(x);
      end_if;
      l := length(x);
      print(Unquoted, _concat(" " $ TEXTWIDTH - l, x))
    end_proc:

>> myprint("hello world"):  myprint(30!):  myprint("bye bye"):

                                hello world

26525285981219105863630848000000

                                bye bye

>> delete myprint:
```

Änderungen:

☞ Keine Änderungen.

TRUE, FALSE, UNKNOWN – boolsche Konstanten (Wahrheitswerte)

MuPAD verwendet eine dreiwertige Logik mit den boolschen Werten TRUE (‚wahr‘), FALSE (‚falsch‘) und UNKNOWN (‚unbekannt‘).

Verwandte Funktionen: `_lazy_and`, `_lazy_or`, `and`, `bool`, `DOM_BOOL`, `if`, `is`, `not`, `or`, `repeat`, `while`

Details:

- ☞ Die boolschen Konstanten TRUE, FALSE und UNKNOWN sind vom Datentyp DOM_BOOL.
 - ☞ Auf der Hilfeseite für `and`, `or`, `not` finden sich die Verknüpfungsregeln der dreiwertigen Logik MuPADs.
 - ☞ Boolsche Konstanten werden von Systemfunktionen wie `bool` und `is` als Ergebnis zurückgegeben. Diese Funktionen werten boolsche Ausdrücke wie Gleichungen, Ungleichungen oder Vergleiche aus.
-

Beispiel 1. Die boolschen Konstanten können mit `and`, `or` und `not` verknüpft werden:

```
>> (TRUE and (not FALSE)) or UNKNOWN
      TRUE
```

Beispiel 2. Die Funktion `bool` dient zur Auswertung von boolschen Ausdrücken wie z. B. Gleichungen oder Ungleichungen:

```
>> bool(x = x and 2 < 3 and 3 <> 4 or UNKNOWN)
      TRUE
```

Die Funktion `is` evaluiert symbolische boolsche Ausdrücke mit Eigenschaften:

```
>> assume(x > 2): is(x^2 > 4), is(x^3 < 0), is(x^4 > 17)
      TRUE, FALSE, UNKNOWN

>> unassume(x):
```

Beispiel 3. Boolesche Konstanten treten im Konditionalteil von Kontrollstrukturen wie `if`, `repeat` und `while` auf. Die folgende Schleife sucht nach der kleinsten Mersenne-Primzahl größer als 500 (`numlib::mersenne` liefert weitere Informationen). Die Funktion `isprime` liefert `TRUE`, wenn ihr Argument eine Primzahl ist, anderenfalls `FALSE`. Sobald eine Mersenne-Primzahl p gefunden wurde, wird die `while`-Schleife mittels `break` abgebrochen:

```
>> p := 500:
    while TRUE do
      p := nextprime(p + 1):
      if isprime(2^p - 1) then
        print(p);
        break;
      end_if;
    end_while;
```

521

Beachten Sie, dass der Konditionalteil von `if`-, `repeat`- und `while`-Anweisungen sich entweder zu `TRUE` oder zu `FALSE` evaluieren muss. Anderenfalls wird ein Fehler ausgelöst:

```
>> if UNKNOWN then "true" else "false" end_if

Error: Can't evaluate to boolean [if]

>> delete p;
```

Änderungen:

⌘ Keine Änderungen.

UNIX – MuPAD-Kommandozeilenoptionen und Initialisierungsdateien unter UNIX

Diese Seite beschreibt alle Kommandozeilen-Optionen und Initialisierungsdateien für MuPAD auf UNIX Betriebssystemen.

Aufruf(e):

```
⌘ mupad [-g] [-n] [-S] [-v] [-h helppath] [-l libpath]
    [-L primelimit]
        [-m modpath] [-p stacksize] [-P [pPeEsSwW]] [-u
    userpath]
        [-U opts] [-w sec] [file...]
```

```
# xmpad [-n] [-S] [-v] [-h helppath] [-l libpath] [-L  
primelimit]  
        [-m modpath] [-p stacksize] [-P [pPeEsSwW]]  
        [-u userpath]  
        [-U opts] [-w sec] [file...]
```

Verwandte Funktionen: LIBPATH, NOTEBOOKFILE, NOTEBOOKPATH,
READPATH, TESTPATH, UNIX, WRITEPATH

Details:

Die folgende Tabelle enthält alle Kommandozeilen-Optionen. der Terminalversion mupad und der graphischen Benutzungsschnittstelle xmu-
pad.

Die Option -g kann für die X11 Oberfläche nicht benutzt werden, da sie dort zur Spezifizierung der Fenster Geometrie reserviert ist. Der Debug-Modus kann jedoch im „Options“-Menu in der Oberfläche aktiviert werden.

OptionBeschreibung

-g	Debug-Modus	
-h	Pfadangabe für den Hilfe-Index	(Standard: \$R/share/doc/ascii)
-l	Standard Bibliotheks-Pfad; kann interaktiv als LIBPATH geändert werden	(Standard: \$R/share/lib)
-L	vorberechnen einer Liste aller Primzahlen kleiner primelimit	(Standard: 1000000)
-m	Pfadangabe für dynamische Module	(Standard: \$R/\$A/modules)
-n	Unterdrückung der Benutzer-Initialisierungsdatei	
-p	Größe des PARI-Stacks in Bytes	(Standard: 250000)
-P	Unterdrücken (p) oder Drucken (P) des Promptes; zur Laufzeit änderbar mit Pref::prompt	(Standard: p)
	Unterdrücken (e) oder Drucken (E) des Eingabe-Echos; zur Laufzeit änderbar mit Pref::echo	(Standard: e)
	Unterdrücken (w) oder Drucken (W) von Warnungen zu Änderungen in der neuen Version von MuPAD; zur Laufzeit änderbar mit Pref::warnChanges	(Standard: w)
	Kernel im sicheren Modus starten (S) oder nicht (s); der sichere Modus verhindert den Zugriff auf Dateien und verbietet den Aufruf der MuPAD Funktion system	(default: s)
-S	Start ohne Ausgabe des MuPAD-Logos	
-u	Pfadangabe für die Benutzer-Initialisierungsdatei	(Standard: ~/.mu-pad/)
-U	übergibt beliebige Optionen an die MuPAD Sitzung, die interaktiv über Pref::userOptions erfragt werden können	(Standard: "")
-v	Verbose-Debugger-Modus	
-w	beendet den MuPAD-Prozess nach sec Sekunden	

Mit \$R ist das MuPAD Installationsverzeichnis gemeint. Mit \$A ist Name

der Architektur gemeint, wie er von dem Shell-Script `$R/share/bin/sysinfo` geliefert wird.

Die Namen der MuPAD-Initialisierungsdateien sind:

<code>~/.mupad/userinit.mu</code>	Benutzer-Initialisierungsdatei
<code>~/.mupad/mxdviRecentFiles</code>	Liste der letzten Dokumente (Hilfe Tool)
<code>~/.mupad/mxmupadrc</code>	Einstellungen der X11 Oberfläche
<code>~/.mupad/vcam_defaults</code>	Voreinstellungen der Graphik
<code>\$R/share/lib/sysinit.mu</code>	System-Initialisierungsdatei
<code>\$R/share/lib/.MMMinit</code>	MAMMUT-Initialisierungsdatei (Speicherverwaltung)

Zusätzlich zu den Optionen können auf der Kommandozeile ein oder mehrere MuPAD Quelltext Dateien `file...` angegeben werden. Sie werden in der angegebenen Reihenfolge eingelesen und ausgeführt.

Beispiel 1. Das folgende Beispiel startet die Terminal-Version von MuPAD, wobei die Benutzer-Initialisierungsdatei nicht gelesen wird (`-n`), kein MuPAD-Logo ausgegeben wird (`-S`), alle Primzahlen bis 2 000 000 vorab berechnet werden (`-L`) und dynamische Module im Verzeichnis `myModules` gesucht werden (`-m`):

```
# mupad -n -S -L 2000000 -m myModules
>>
```

Änderungen:

- ☞ Der Pfad zur MAMMUT-Initialisierungsdatei, der früher mit `-m` angegeben wurde, ist nun immer gleich dem Bibliotheks-Pfad.
- ☞ Der Pfad zur System-Initialisierungsdatei, der früher mit `-s` angegeben wurde, ist nun immer gleich dem Bibliotheks-Pfad.
- ☞ Der Modul-Pfad wird nun mit der Option `-m` angegeben, anstatt mit `-d`.
- ☞ Der Kernel Trace Modus und die entsprechende Option `-t` existieren nicht mehr.
- ☞ Der neue Warnungsschalter `-P W` wurde eingeführt.
- ☞ Die Benutzer-Initialisierungsdatei `~/.mupadinit` wurde nach `~/.mupad/userinit.mu` verschoben.
- ☞ Die System-Initialisierungsdatei `$R/share/lib/.mupadsysinit` wurde umbenannt in `$R/share/lib/sysinit.mu`.

abs – der Absolutbetrag einer reellen oder komplexen Zahl

`abs(z)` liefert den Absolutbetrag der Zahl `z`.

Aufruf(e):

☞ `abs(z)`

Parameter:

`z` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `z`

Seiteneffekte: `abs` berücksichtigt Eigenschaften von Bezeichnern.

Verwandte Funktionen: `conjugate`, `Im`, `norm`, `Re`, `sign`

Details:

- ☞ Für zahlreiche konstante Ausdrücke liefert `abs` den Absolutbetrag als expliziten Ausdruck. Siehe Beispiel ??.
 - ☞ Ein symbolischer Aufruf von `abs` wird zurückgeliefert, wenn der Absolutbetrag nicht explizit bestimmt werden kann (z. B., wenn das Argument symbolische Bezeichner enthält). Das Resultat kann dabei vereinfacht werden: insbesondere extrahiert `abs` konstante Faktoren. Eigenschaften von Bezeichnern werden berücksichtigt. Siehe Beispiele ?? und ??.
 - ☞ Die `expand`-Funktion schreibt den Betrag eines Produkts in ein Produkt von Beträgen um: `expand(abs(x*y))` liefert `abs(x)*abs(y)`. Siehe Beispiel ??.
 - ☞ Die symbolischen Konstanten `CATALAN`, `E`, `EULER` und `PI` werden von `abs` verarbeitet. Siehe Beispiel ??.
 - ☞ Der Absolutbetrag von symbolischen Funktionsausdrücken kann über den "abs"-Slot von Funktionsumgebungen definiert werden. Siehe Beispiel ??.
 - ☞ Genauso kann der Absolutbetrag von Domain-Elementen mittels Überladung definiert werden. Siehe Beispiel ??.
-

Beispiel 1. Der Absolutbetrag vieler konstanter Ausdrücke kann explizit bestimmt werden:

```
>> abs(1.2), abs(-8/3), abs(3 + I), abs(sqrt(-3))
```

$$1.2, 8/3, 10^{1/2}, 3^{1/2}$$

```
>> abs(sin(42)), abs(PI^2 - 10), abs(exp(3) - tan(157/100))
```

$$-\sin(42), 10 - \pi^2, \tan(157/100) - \exp(3)$$

```
>> abs(exp(3 + I) - sqrt(2))
```

$$(\sin(1) \exp(3)^2 + (\cos(1) \exp(3) - 2^{1/2})^2)^{1/2}$$

Beispiel 2. Symbolische Ausdrücke werden zurückgeliefert, falls das Argument Bezeichner ohne Eigenschaften enthält:

```
>> abs(x), abs(x + 1), abs(sin(x + y))
```

$$\text{abs}(x), \text{abs}(x + 1), \text{abs}(\sin(x + y))$$

Dabei werden eventuell Vereinfachungen durchgeführt. Insbesondere spaltet abs in Produkten konstante Faktoren ab:

```
>> abs(PI*x*y), abs((1 + I)*x), abs(sin(4)*(x + sqrt(3)))
```

$$\pi \text{abs}(x y), \text{abs}(x) 2^{1/2}, -\sin(4) \text{abs}(x + 3^{1/2})$$

Beispiel 3. abs regiert auf Eigenschaften von Bezeichnern:

```
>> assume(x < 0): abs(3*x), abs(PI - x), abs(I*x), abs(x + I)
```

$$-3 x, \pi - x, -x, (x^2 + 1)^{1/2}$$

```
>> unassume(x):
```

Beispiel 4. Mittels `expand` werden Beträge von Produkten in Produkte von Beträgen umgeschrieben:

```
>> abs(x*(y + 1)), expand(abs(x*(y + 1)))
      abs(x (y + 1)), abs(x) abs(y + 1)
```

Beispiel 5. Der Absolutbetrag der symbolischen Konstanten `PI`, `EULER` etc. ist bekannt:

```
>> abs(PI), abs(EULER + CATALAN^2)
      PI, EULER + CATALAN2
```

Beispiel 6. Ausdrücke, die `abs` enthalten, können differenziert werden:

```
>> diff(abs(x), x), diff(abs(x), x, x)
      sign(x), 2 dirac(x)
```

Beispiel 7. Der Slot "`abs`" einer Funktionsumgebung `f` bestimmt den Absolutbetrag symbolischer Funktionsaufrufe:

```
>> abs(f(x))
      abs(f(x))

>> f := funcenv(f): f::abs := x -> f(x)/sign(f(x)): abs(f(x))
      f(x)
      -----
      sign(f(x))

>> delete f:
```

Beispiel 8. Der Slot "`abs`" eines Domains `d` bestimmt den Absolutbetrag seiner Elemente:

```
>> d := newDomain("d"): e1 := new(d, 2): e2 := new(d, x):
      abs(e1), abs(e2)
      abs(new(d, 2)), abs(new(d, x))
```

```
>> d::abs := x -> abs(extop(x, 1)): abs(e1), abs(e2)

                                2, abs(x)

>> delete d, e1, e2:
```

Änderungen:

☞ Keine Änderungen.

alias, unalias – Definieren bzw. Löschen einer Abkürzung oder eines Makros

alias(x = object) definiert x als Abkürzung für object.

alias(f(y1, y2, ...) = object) definiert f als Makro. Für beliebige Objekte a1, a2, ... ist danach f(a1, a2, ...) äquivalent zu object, mit y1 ersetzt durch a1, y2 ersetzt durch a2, usw.

alias() zeigt eine Liste aller im Moment gültigen Aliasse und Makros an.

unalias(x) löscht die Abkürzung bzw. das Makro x.

unalias() löscht alle Abkürzungen und Makros.

Aufruf(e):

```
☞ alias(x1 = object1, x2 = object2, ...)
☞ alias()
☞ unalias(z1, z2, ...)
☞ unalias()
```

Parameter:

x1, x2, ...	— Bezeichner oder symbolische Ausdrücke der Form f(y1, y2, ...), wo f, y1, y2, ... Bezeichner sind.
object1, object2, ...	— beliebige MuPAD-Objekte
z1, z2, ...	— Bezeichner

Rückgabewert: Sowohl alias als auch unalias liefern das leere Objekt vom Typ DOM_NULL zurück.

Falls alias ohne Argumente aufgerufen wird, so werden alle derzeit definierten Aliasse in Form einer Folge von Gleichungen angezeigt. Eine genauere Beschreibung ist weiter unten zu finden.

Seiteneffekte: `alias` mit mindestens einem Argument und `unalias` verändern die Konfiguration des Parsers in der unter „Details“ beschriebenen Weise.

Verwandte Funktionen: `:=`, `finput`, `fprint`, `fread`, `input`,
`Pref::alias`, `print`, `proc`, `read`, `subs`, `text2expr`, `write`

Details:

☞ Der Aufruf `alias(x = object)` definiert eine Abkürzung. Er ändert die Konfiguration des Parsers, so dass in jeder Eingabe der Bezeichner `x` durch `object` ersetzt wird, und umgekehrt wird auch `object` in der Ausgabe durch `x` ersetzt.

☞ `alias(f(y1, y2, ...) = object)` definiert ein Makro. Der Parser wird so konfiguriert, dass er jeden Funktionsaufruf der Form `f(a1, a2, ...)` mit einer Folge beliebiger Objekte `a1, a2, ...` durch dasjenige Objekt ersetzt, das aus `object` durch Substitution von `y1` durch `a1`, `y2` durch `a2`, etc. entsteht.

Dies gilt jedoch nicht, falls die Folgen `y1, y2, ...` und `a1, a2, ...` aus verschieden vielen Elementen bestehen; in diesem Fall findet keine Ersetzung der Eingabe statt.

In der Ausgabe findet keine Ersetzung statt.

Ein Makrodefinition ohne Argumente der Form `alias(f()=object)` ist zulässig.

☞ In einem Aufruf können mehrere Alias-Definitionen angegeben werden. Abkürzungen und Makros dürfen dabei gemischt werden.

☞ `alias()` zeigt alle im Moment definierten Aliasse als Folge von Gleichungen an. Für eine Abkürzung, die mittels `alias(x = object)` definiert wurde, wird `x = object` ausgegeben. Für ein Makro, das mittels `alias(f(y1, y2, ...) = object)` definiert wurde, wird die Gleichung `f = [object, [y1, y2, ...]]` ausgegeben. Sind keine Aliasse definiert, so erscheint die Meldung „No alias defined“. Siehe Beispiel ??.

☞ `unalias(x)` löscht die Abkürzung oder das Makro `x`. Um ein Makro zu Löschen, das durch `alias(f(y1, y2, ...) = object)` definiert wurde, verwendet man `unalias(f)`. Wenn kein Alias für `x` bzw. `f` definiert wurde, wird der Aufruf ignoriert.

Mehrere Alias Definitionen können durch einen einzigen Aufruf von `unalias` gelöscht werden. Der Aufruf `unalias()` löscht alle momentan definierten Aliasse.

☞ Weder `alias` noch `unalias` werten ihre Argumente aus. Daher ist es unerheblich, ob der als Alias verwendete Bezeichner einen Wert hat, und

`alias` erzeugt immer einen Alias für die *unausgewertete* rechte Seite. Siehe Beispiel ??.

- ☞ `alias` gleicht seine Argumente nicht aus; eine Folge ist daher auf der rechten Seite einer Aliasdefinition zulässig. Siehe Beispiel ??.
- ☞ Die Definition eines Alias bewirkt eine Substitution vergleichbar der durch `subs`, nicht nur eine rein textuelle Ersetzung. Siehe Beispiel ??.
- ☞ Jeder Bezeichner kann als Alias für nur ein Objekt dienen. Umgekehrt kann jedes Objekt nur auf eine Weise abgekürzt werden; andernfalls bricht `alias` mit einer Fehlermeldung ab.
- ☞ Eine Aliasdefinition ist gültig für alle Eingaben, die *geparst* werden, nachdem der Aufruf von `alias` ausgewertet wurde. Siehe Beispiel ??.
- ☞ Die Rückübersetzung von Aliassen in der Ausgabe wird nur für Abkürzungen durchgeführt und nicht für Makros. Nach einem Kommando der Form `alias(x = object)` werden sowohl das unevaluierte Objekt `object` als auch das evaluierte Objekt in der Ausgabe durch den unevaluierten Bezeichner `x` ersetzt. Siehe Beispiel ??.
- ☞ Der Anwender kann dieses Verhalten der Rückübersetzung in der Ausgabe über `Pref::alias` steuern; Details dazu findet man auf der entsprechenden Hilfeseite.
- ☞ Die Rücksubstitution in der Ausgabe wird nur bei den Ergebnissen von Berechnungen auf interaktiver Ebene, nicht jedoch in der Ausgabe der Funktionen `fprint`, `print` und `write` durchgeführt.
- ☞ Alle Aliasse werden gleichzeitig ersetzt (gilt für Eingabe und Ausgabe); die Definition verschachtelter Aliasse ist daher nicht möglich. Siehe Beispiel ??.
- ☞ Wird ein Bezeichner als Abkürzung benutzt, so kann dieser Bezeichner selbst nicht mehr eingegeben werden.

Insbesondere muss zunächst `unalias` aufgerufen werden, damit der Bezeichner als Abkürzung oder Makro für etwas anderes verwendet werden kann. Siehe Beispiel ??.



- ☞ Nach Definition eines Makros `f(y1, y2, . . . , yn)` mit n Argumenten ist ein Aufruf von `f` selbst mit n Argumenten nicht mehr möglich. Jedoch behält ein Aufruf von `f` mit einer anderen Zahl von Argumenten seine ursprüngliche Bedeutung. Siehe Beispiel ??.

Es ist nicht notwendig, `unalias` aufzurufen, bevor eine neue Abkürzung oder ein Makro mit einer anderen Zahl von Argumenten als zuvor für den Bezeichner `f` definiert werden kann. Jede nachfolgenden Definition eines Alias, d. h., einer Abkürzung oder eines Makros, für diesen Bezeichner überschreibt die vorherige. Siehe Beispiel ??.

- ⌘ Ein Alias wirkt auf alle Arten der Eingabe: Kommandozeileneingabe, Eingabe mittels `input`, Einlesen einer Datei mittels `read`, `fread` oder `finput`, sowie Umwandeln einer Zeichenkette mittels `text2expr`. Für `read` und `fread` kann dies durch Angabe der Option `Plain` verhindert werden. Siehe Beispiel ??.
- ⌘ Eine Aliasdefinition hat auf den als Alias verwendeten Bezeichner keine Auswirkungen; dieser behält seinen Wert und seine Eigenschaften. Der Bezeichner und das Objekt, das für ihn eingesetzt wird, werden beim Auswerten voneinander unterschieden. Siehe Beispiel ??.
- ⌘ Das Zuweisen oder Löschen von Werten von Bezeichnern auf der linken Seite einer Aliasdefinition hat keine Auswirkungen, weder auf die Eingabe noch auf die Ausgabe. Siehe Beispiel ??.

Beispiel 1. Wir definieren `d` als Abkürzung für `diff`:

```
>> delete f, g, x, y: alias(d = diff):
      d(sin(x), x) = diff(sin(x), x);
      d(f(x, y), x) = diff(f(x, y), x)

      cos(x) = cos(x)

      d(f(x, y), x) = d(f(x, y), x)
```

Wir definieren ein Makro `Dx(f)` für `diff(f(x), x)`. Zu beachten ist, dass `hold` die Aliassubstitution nicht verhindert.

```
>> alias(Dx(f) = diff(f(x), x)):
      Dx(sin); Dx(f + g); hold(Dx(f + g))

      cos(x)

      d(f(x), x) + d(g(x), x)

      d((f + g)(x), x)
```

Nach Aufruf von `unalias(d, Dx)` finden keine Alias-Ersetzungen mehr statt:

```
>> unalias(d, Dx):
      d(sin(x), x), diff(sin(x), x), d(f(x, y), x), diff(f(x, y), x);
      Dx(sin), Dx(f + g)

      d(sin(x), x), cos(x), d(f(x, y), x), diff(f(x, y), x)

      Dx(sin), Dx(f + g)
```

Beispiel 2. Um das Tippen von `longhardtotypeident` zu vermeiden, definieren wir dafür eine Abkürzung `a`:

```
>> longhardtotypeident := 10; alias(a = longhardtotypeident):
10
```

Da `alias` seine Argumente nicht evaluiert, ist `a` jetzt eine Abkürzung für `longhardtotypeident` und nicht etwa für die Zahl 10:

```
>> type(a), type(hold(a))
DOM_INT, DOM_IDENT

>> a + 1, hold(a) + 1, eval(hold(a) + 1)
11, a + 1, 11
```

```
>> longhardtotypeident := 2:
a + 1, hold(a) + 1, eval(hold(a) + 1)
3, a + 1, 3
```

Standardmäßig geschieht die Rücksubstitution der Aliasse in der Ausgabe für den Bezeichner und seinen Wert:

```
>> 2, 10, longhardtotypeident, hold(longhardtotypeident)
a, 10, a, a
```

Das Kommando `Pref::alias(FALSE)` schaltet die Rücksubstitution der Aliasse ab:

```
>> p := Pref::alias(FALSE):
a, hold(a), 2, longhardtotypeident, hold(longhardtotypeident);
Pref::alias(p): unalias(a):
2, longhardtotypeident, 2, 2, longhardtotypeident
```

Beispiel 3. Aliasse werden substituiert und nicht nur textuell ersetzt. Daher wird `3*succ(u)` zu `3*(u+1)` und nicht zu `3*u+1` wie beim Suchen und Ersetzen in einem Textverarbeitungsprogramm:

```
>> alias(succ(x) = x + 1): 3*succ(u);
unalias(succ):
3 u + 3
```

Beispiel 4. Definiert man a als Abkürzung für b, so ist die folgende Aliasdefinition für a in Wirklichkeit eine Aliasdefinition für b:

```
>> delete a, b:
    alias(a = b): alias(a = 2): type(a), type(b); unalias(b):

DOM_IDENT, DOM_INT
```

Bevor man einen anderen Alias für a definiert, muss man unalias verwenden:

```
>> unalias(a): alias(a = 2): type(a), type(b); unalias(a):

DOM_INT, DOM_IDENT
```

Eine Makrodefinition kann hingegen sofort überschrieben werden; dies aber nur, falls das neu definierte Makro eine andere Argumentanzahl hat:

```
>> alias(a(x)=sin(x^2)): a(y); alias(a(x)=cos(x^2)):

      2
      sin(y )
Error: Illegal operand [_power];
during evaluation of 'alias'

>> alias(a(x, y) = sin(x + y)): a(u, v); unalias(a)

sin(u + v)
```

Beispiel 5. Ein Makro wird nur wirksam, wenn es mit der richtigen Anzahl von Argumenten aufgerufen wird, und die Folge der Argumente wird nicht ausgeglichen:

```
>> alias(plus(x, y) = x + y):
    plus(1), plus(3, 2), plus((3, 2));
    unalias(plus):

plus(1), 5, plus(3, 2)
```

Auch Folgen sind als rechte Seite einer Aliasdefinition zulässig; sie müssen eingeklammert werden:

```
>> alias(x = (1, 2)): f := 0, 1, 2, x;
    nops(f); unalias(x):

0, 1, 2, 1, 2

5
```

Beispiel 6. Ein als Abkürzung verwendeter Bezeichner kann in Ausdrücken, die vor der Aliasdefinition eingegeben wurden, weiterhin für sich selbst stehen:

```
>> delete x: f := [x, 1]: alias(x = 1): f;
    map(f, type); unalias(x):

[x, x]

[DOM_IDENT, DOM_INT]
```

Beispiel 7. Es hat keine Auswirkungen, falls der als Alias verwendete Bezeichner einen Wert hat:

```
>> a := 5: alias(a = 7): 7, 5; print(a); unalias(a):

a, 5

7
```

Beispiel 8. Aliasdefinitionen wirken sich auch auf die Eingabe aus Dateien oder Zeichenketten aus:

```
>> alias(a = 3): type(text2expr("a")); unalias(a)

DOM_INT
```

Beispiel 9. Ein Alias wirkt auf alle Eingaben, die nach der Ausführung von *alias* eingelesen werden. Eine Anweisung in einer Kommandozeile wird erst nach Ausführung aller vorherigen Anweisungen in derselben Kommandozeile eingelesen. Daher ist im folgenden Beispiel die Abkürzung bereits für die zweite Anweisung gültig:

```
>> alias(a = 3): type(a); unalias(a)

DOM_INT
```

Dies kann man durch Eingabe zusätzlicher Klammern umgehen:

```
>> (alias(a = 3): type(a)); unalias(a)

DOM_INT
```

Beispiel 10. Wir definieren `b` als Alias für `c` und dies wiederum als Alias für `2`. Solche Ketten sollten vermieden werden, da leicht unerwünschte Effekte auftreten können.

```
>> alias(b=c): alias(c=2):
```

Danach wird jedes `b` in jeder Eingabe durch `c` ersetzt, das neu entstandene `c` aber nicht durch `2`:

```
>> print(b)
```

`c`

Andererseits wird in jeder Ausgabe `2` durch `c`, dieses aber nicht weiter durch `b` ersetzt:

```
>> 2
```

`c`

```
>> unalias(c): unalias(b):
```

Beispiel 11. Wird `alias` ohne Argumente aufgerufen, so werden einfach alle im Moment wirksamen Aliasse angezeigt:

```
>> alias(a = 5, F(x) = sin(x^2)):
    alias(); unalias(F, a):
```

```
a = 5,
F = [sin(x^2), [x]]
```

Hintergründe:

- ☞ Alle Aliasse werden in der Parserkonfigurationstabelle gespeichert, die man sich mit `_parser_config()` ansehen kann. Bei der Ausgabe dieser Tabelle wird die übliche Rücksubstitution der Aliasse vorgenommen; um dies zu verhindern, verwendet man `print(_parser_config())`.
- ☞ Aliasse wirken nicht auf das Einlesen einer Datei mittels `read` oder `fread` mit Option `Plain` und mittels `loadproc`. Umgekehrt gelten die in einer mit einem dieser Befehle eingelesenen Datei definierten Aliasse nur, bis die Datei vollständig eingelesen ist.

Änderungen:

- ⌘ Abkürzungen werden jetzt auch für die Ausgabe von Objekten zurücks-
ubstituiert.
-

`anames` – Bezeichner, die einen Wert oder Eigenschaften haben

`anames(All)` liefert alle Bezeichner, die einen Wert haben.

`anames(Properties)` liefert alle Bezeichner, die Eigenschaften haben.

`anames(d)` liefert alle Bezeichner, die einen Wert aus dem gegebenen Domain `d` haben.

Aufruf(e):

- ⌘ `anames(All <, User>)`
- ⌘ `anames(Properties <, User>)`
- ⌘ `anames(d <, User>)`

Parameter:

`d` — ein Domain

Optionen:

- `All` — alle Bezeichner, die einen Wert haben
- `Properties` — alle Bezeichner, die Eigenschaften haben
- `User` — schließt alle System-Variablen aus

Rückgabewert: eine Menge von Bezeichnern.

Verwandte Funktionen: `:=`, `_assign`, `assume`, `DOM_IDENT`

Details:

- ⌘ Das von `anames` zurückgelieferte Ergebnis ist eine Menge von *unevaluierten* Bezeichnern. Siehe Beispiel ??.
- ⌘ `anames` berücksichtigt keine Slots von Funktionsumgebungen oder Domains. Funktionen aus einer MuPAD-Bibliothek werden nur berücksichtigt, wenn sie exportiert sind.

Option *<User>*:

- ☞ Ist die Option *User* angegeben, so werden nur solche Bezeichner zurückgeliefert, denen vom Benutzer ein Wert bzw. eine Eigenschaft zugewiesen wurde.

Beispiel 1. `anames(DOM_IDENT)` liefert alle Bezeichner, deren Werte selbst wieder Bezeichner sind:

```
>> anames(DOM_IDENT)

      {'*', '+', '-', '/', '**', '^'}
```

Die Elemente der zurückgegebenen Menge sind nicht evaluiert. Man kann `eval` verwenden, um sie auszuwerten:

```
>> map(%, x -> x = eval(x))

      {'*' = _mult, '+' = _plus, '**' = _power, '^' = _power,
       '-' = _negate, '/' = _divide}
```

Beispiel 2. `anames(All, User)` liefert alle benutzerdefinierten Bezeichner:

```
>> a := b: b := 2: c := {2, 3}:
    anames(All, User)

      {a, b, c}
```

Ist das erste Argument ein Domain, so werden nur Bezeichner berücksichtigt, deren *Wert* zu dem Domain gehört. Das sind im allgemeinen andere als die Bezeichner, deren *Auswertung* ein Element des Domains liefert:

```
>> a, b;
    anames(DOM_IDENT, User);
    anames(DOM_INT, User)

      2, 2

      {a}

      {b}
```

Beispiel 3. `anames(Properties)` liefert alle Bezeichner, denen mittels `assume` eine Eigenschaft zugewiesen wurde:

```
>> assume(x > y): anames(Properties)
                        {x, y}
```

Änderungen:

- ⌘ Ein Domain ist nun ein zulässiges Argument.
 - ⌘ Die neue Option *User* wurde eingeführt.
 - ⌘ `anames(name)` wird nicht mehr unterstützt; benutzen Sie statt dessen `bool(name = hold(name))`.
 - ⌘ `anames(0)` wird nicht mehr unterstützt; benutzen Sie statt dessen `anames(DOM_FUNC_ENV)`.
 - ⌘ `anames(1)` wird nicht mehr unterstützt; benutzen Sie statt dessen `anames(DOM_PROC)`.
 - ⌘ `anames(2)` wird nicht mehr unterstützt; benutzen Sie statt dessen `anames(All) minus (anames(DOM_FUNC_ENV) union anames(DOM_PROC))`.
 - ⌘ `anames(3)` wird nicht mehr unterstützt; benutzen Sie statt dessen `anames(All)`.
 - ⌘ `anames` ist nun eine Bibliotheks-Funktion.
-

`and`, `or`, `not` – boolsche Operatoren

`b1 and b2` stellt die logische ‚und‘-Verknüpfung der boolschen Ausdrücke `b1, b2` dar.

`b1 or b2` stellt die logische ‚oder‘-Verknüpfung der boolschen Ausdrücke `b1, b2` dar.

`not b` stellt die logische Verneinung des boolschen Ausdrucks `b` dar.

Aufruf(e):

- ⌘ `b1 and b2`
- ⌘ `_and(b1, b2, ...)`
- ⌘ `b1 or b2`
- ⌘ `_or(b1, b2, ...)`
- ⌘ `not b`
- ⌘ `_not(b)`

Parameter:

$b, b1, b2, \dots$ — boolsche Ausdrücke

Rückgabewert: ein boolscher Ausdruck.

Überladbar durch: $b, b1, b2, \dots$

Verwandte Funktionen: `_lazy_and`, `_lazy_or`, `bool`, `is`, `FALSE`, `TRUE`, `UNKNOWN`

Details:

- ☞ MuPAD verwendet eine dreiwertige Logik mit den boolschen Konstanten `TRUE`, `FALSE` und `UNKNOWN`. Sie werden wie folgt verknüpft:

and	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN
or	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

$\text{not } \text{TRUE} = \text{FALSE}, \text{not } \text{FALSE} = \text{TRUE}, \text{not } \text{UNKNOWN} = \text{UNKNOWN}.$

- ☞ Boolesche Ausdrücke können aus diesen Konstanten sowie beliebigen arithmetischen Ausdrücken aufgebaut werden. Typischerweise werden Gleichungen wie etwa $x = y$ oder Ungleichungen wie $x <> y$, $x < y$, $x \leq y$ etc. zu komplexeren boolschen Ausdrücken verknüpft.

- ☞ `_and(b1, b2, ...)` ist äquivalent zu $b1 \text{ and } b2 \text{ and } \dots$. Dieser Ausdruck repräsentiert `TRUE`, wenn jeder einzelne Ausdruck sich zu `TRUE` evaluiert. Er repräsentiert `FALSE`, wenn mindestens ein Ausdruck sich zu `FALSE` evaluiert. Er repräsentiert `UNKNOWN`, wenn mindestens ein Ausdruck `UNKNOWN` liefert und sich alle anderen zu `TRUE` evaluieren.

`_and()` liefert den Wert `TRUE`.

- ☞ `_or(b1, b2, ...)` ist äquivalent zu $b1 \text{ or } b2 \text{ or } \dots$. Dieser Ausdruck repräsentiert `FALSE`, wenn jeder einzelne Ausdruck sich zu `FALSE` evaluiert. Er repräsentiert `TRUE`, wenn mindestens ein Ausdruck sich zu `TRUE` evaluiert. Er repräsentiert `UNKNOWN`, wenn mindestens ein Ausdruck `UNKNOWN` liefert und sich alle anderen zu `FALSE` evaluieren.

`_or()` liefert den Wert `FALSE`.

- ☞ `_not(b)` ist äquivalent zu $\text{not } b$.

- ☞ Kombinationen der Konstanten `TRUE`, `FALSE`, `UNKNOWN` innerhalb eines boolschen Ausdrucks werden automatisch vereinfacht. Im Gegensatz dazu werden symbolische Objekte, Gleichungen und Ungleichungen von `and`, `or`, `not` weder evaluiert noch vereinfacht. Man verwende `bool`, um solche Ausdrücke zu einer der boolschen Konstanten zu evaluieren. Man beachte dabei, dass `bool` Ungleichungen $x < y$, $x \leq y$ etc. nur dann auswerten kann, wenn sie Vergleiche von reellen Zahlen vom Typ `Type::Real` darstellen. Siehe Beispiel ??.

Über die Funktion `simplify` mit der Option `logic` können boolsche Ausdrücke mit symbolischen Objekten vereinfacht werden. Siehe Beispiel ??.

- ☞ Die Prioritäten von `and`, `or` und `not` sind wie folgt: Der Operator `not` bindet stärker als `and`, also

$$\text{not } b1 \text{ and } b2 = (\text{not } b1) \text{ and } b2.$$

Der Operator `and` bindet stärker als `or`, also

$$b1 \text{ and } b2 \text{ or } b3 = (b1 \text{ and } b2) \text{ or } b3.$$

Im Zweifelsfall verwende man Klammern um sicherzustellen, dass der eingegebene Ausdruck in der gewünschten Form interpretiert wird.

- ☞ Weitere logische Operatoren können leicht durch den Benutzer implementiert werden. Siehe Beispiel ??.
- ☞ Im Konditionalteil von `if`-, `repeat`- und `while`-Anweisungen werden diese boolschen Operatoren via „lazy evaluation“ ausgewertet (siehe `_lazy_and` und `_lazy_or`). In jedem anderen Kontext werden alle Operanden evaluiert.
- ☞ `_and` ist eine Funktion des Systemkerns.
- ☞ `_or` ist eine Funktion des Systemkerns.
- ☞ `_not` ist eine Funktion des Systemkerns.

Beispiel 1. Kombinationen der boolschen Konstanten `TRUE`, `FALSE` und `UNKNOWN` werden automatisch zu einer dieser Konstanten vereinfacht:

```
>> TRUE and not (FALSE or TRUE)
```

FALSE

```
>> FALSE and UNKNOWN, TRUE and UNKNOWN
```

FALSE, UNKNOWN

```
>> FALSE or UNKNOWN, TRUE or UNKNOWN
UNKNOWN, TRUE

>> not UNKNOWN
UNKNOWN
```

Beispiel 2. Die Operatoren and, or, not vereinfachen Teilausdrücke, die sich zu TRUE, FALSE, UNKNOWN evaluieren:

```
>> b1 or b2 and TRUE
b1 or b2

>> FALSE or ((not b1) and TRUE)
not b1

>> b1 and (b2 or FALSE) and UNKNOWN
UNKNOWN and b1 and b2

>> FALSE or (b1 and UNKNOWN) or x < 1
UNKNOWN and b1 or x < 1

>> TRUE and ((b1 and FALSE) or (b1 and TRUE))
b1
```

Gleichungen und Ungleichungen werden jedoch nicht evaluiert:

```
>> (x = x) and (1 < 2) and (2 < 3) and (3 < 4)
x = x and 1 < 2 and 2 < 3 and 3 < 4
```

Die boolsche Auswertung wird mittels bool erzwungen:

```
>> bool(%)
TRUE
```

Man beachte, dass bool nur reelle Zahlen vom syntaktischen Typ `Type::Real` vergleichen kann:

```
>> bool(1 < 2 and PI < sqrt(10))
Error: Can't evaluate to boolean [_less]
```

Beispiel 3. Ausdrücke mit symbolischen boolschen Teilausdrücken werden nicht automatisch von `and`, `or`, `not` vereinfacht. Die Vereinfachung muss explizit mittels `simplify` angefordert werden:

```
>> (b1 and b2) or (b1 and (not b2)) and (1 < 2)

      b1 and b2 or b1 and not b2 and 1 < 2

>> simplify(%, logic)

      b1
```

Beispiel 4. Die boolschen Funktionen `_and` und `_or` akzeptieren beliebige Folgen boolscher Ausdrücke. Der folgende Aufruf testet mittels `isprime`, ob *alle* Zahlen der folgenden Menge Primzahlen sind:

```
>> Set := {1987, 1993, 1997, 1999, 2001}:
      _and(isprime(i) $ i in Set)

      FALSE
```

Der folgende Aufruf testet, ob *mindestens eine* der Zahlen eine Primzahl ist:

```
>> _or(isprime(i) $ i in Set)

      TRUE

>> delete Set:
```

Beispiel 5. Die folgende Funktion implementiert das logische ‚exklusive oder‘:

```
>> exor := (b1, b2) -> (b1 or b2) and not (b1 and b2):
```

Es ergeben sich folgende Verknüpfungsregeln:

```
>> for b1 in [TRUE, FALSE, UNKNOWN] do
      for b2 in [TRUE, FALSE, UNKNOWN] do
        print(hold(exor)(b1, b2) = exor(b1, b2))
      end_for
    end_for:
```

```

exor(TRUE, TRUE) = FALSE

exor(TRUE, FALSE) = TRUE

exor(TRUE, UNKNOWN) = UNKNOWN

exor(FALSE, TRUE) = TRUE

exor(FALSE, FALSE) = FALSE

exor(FALSE, UNKNOWN) = UNKNOWN

exor(UNKNOWN, TRUE) = UNKNOWN

exor(UNKNOWN, FALSE) = UNKNOWN

exor(UNKNOWN, UNKNOWN) = UNKNOWN

```

Der folgende Aufruf erzeugt einen Operator `exor`, sodass `b1 exor b2` zum Funktionsaufruf `exor(b1, b2)` führt:

```

>> operator("exor", exor, Binary, 50): TRUE exor FALSE

TRUE

>> operator("exor", Delete): delete exor, b1, b2:

```

Änderungen:

☞ Keine Änderungen.

append – Hinzufügen von Elementen zu einer Liste

`append(l, object)` fügt `object` an die Liste `l` an.

Aufruf(e):

☞ `append(l, object1, object2, ...)`

Parameter:

`l` — eine Liste
`object1, object2, ...` — beliebige MuPAD Objekte

Rückgabewert: die verlängerte Liste.

Überladbar durch: `l`

Verwandte Funktionen: `_concat`, `_index`, `DOM_LIST`, `op`

Details:

- ⌘ `append(l, object1, object2, ...)` fügt `object1`, `object2` usw. an das Ende der Liste `l` an und liefert die neue Liste als Ergebnis zurück.
 - ⌘ Der Aufruf `append(l)` ist zulässig und liefert `l` zurück.
 - ⌘ `append(l, object1, object2, ...)` ist äquivalent zu jedem der beiden Aufrufe `[op(l), object1, object2, ...]` und `l.[object1, object2, ...]`. Allerdings ist `append` effizienter.
 - ⌘ Die Funktion `append` liefert immer ein neues Objekt. Das erste Argument bleibt unverändert. Siehe Beispiel ??.
 - ⌘ `append` ist eine Funktion des Systemkerns.
-

Beispiel 1. Die Funktion `append` fügt neue Elemente an das Ende einer Liste an:

```
>> append([a, b], c, d)

[a, b, c, d]
```

Werden keine neuen Elemente angegeben, so wird das erste Argument unverändert zurückgegeben:

```
>> l := [a, b]: append(l)

[a, b]
```

Das erste Argument kann auch eine leere Liste sein:

```
>> append([], c)

[c]
```

Beispiel 2. Die Funktion `append` liefert immer ein neues Objekt. Das erste Argument bleibt unverändert:

```
>> l := [a, b]: append(l, c, d), l

[a, b, c, d], [a, b]
```

Beispiel 3. Anwender können `append` für ihre eigenen Domains überladen. Zur Illustration erzeugen wir ein Domain `T` und definieren einen "append"-Slot, der einfach die verbleibenden Argumente zu den internen Operanden seines ersten Arguments hinzufügt:

```
>> T := newDomain("T"):
      T::append := x -> new(T, extop(x), args(2..args(0))):
```

Wenn man nun `append` auf ein Objekt vom Domain-Typ `T` anwendet, dann wird die Methode `T::append` aufgerufen:

```
>> e := new(T, 1, 2): append(e, 3)

      new(T, 1, 2, 3)
```

Änderungen:

☞ Keine Änderungen.

`arcsin`, `arccos`, `arctan`, `arccsc`, `arcsec`, `arccot` – die inversen trigonometrischen Funktionen

`arcsin(x)` stellt die Inverse der Sinus-Funktion dar.

`arccos(x)` stellt die Inverse der Kosinus-Funktion dar.

`arctan(x)` stellt die Inverse der Tangens-Funktion dar.

`arccsc(x)` stellt die Inverse der Kosekans-Funktion dar.

`arcsec(x)` stellt die Inverse der Sekans-Funktion dar.

`arccot(x)` stellt die Inverse der Kotangens-Funktion dar.

Aufruf(e):

☞ `arcsin(x)`

☞ `arccos(x)`

☞ `arctan(x)`

☞ `arccsc(x)`

☞ `arcsec(x)`

☞ `arccot(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: \times

Seiteneffekte: Für Gleitpunktargumente reagieren die Funktionen auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `sin`, `cos`, `tan`, `csc`, `sec`, `cot`

Details:

- ☞ Der zurückgelieferte Winkel ist im Bogenmaß angegeben, nicht in Grad. Beispielsweise stellt das Ergebnis π einen Winkel von 180° dar.
- ☞ Alle inversen trigonometrischen Funktionen sind für komplexe Argumente definiert.
- ☞ Für Gleitpunktargumente werden Gleitpunktwerte berechnet. Für die meisten exakten Argumente werden unevaluierte Funktionsaufrufe zurückgeliefert.
- ☞ Die trigonometrischen Funktionen liefern für gewisse rationale Vielfache von π explizite Ausdrücke. Für diese Ausdrücke liefern die inversen Funktion ein entsprechendes rationales Vielfaches von π auf dem unten definierten Hauptzweig. Siehe Beispiel ??.
- ☞ Das Ergebnis wird durch Hyperbel-Funktionen ausgedrückt, wenn das Argument ein rationales Vielfaches von \mathbb{I} ist. Siehe Beispiel ??.
- ☞ Die inversen trigonometrischen Funktionen sind mehrdeutig. Die MuPAD-Funktionen liefern Werte auf dem im Folgenden definierten Hauptzweig. Für jedes endliche komplexe x gilt:
 - $y := \arcsin(x)$ erfüllt $-\pi/2 \leq \Re(y) \leq \pi/2$,
 - $y := \arccos(x)$ erfüllt $0 \leq \Re(y) \leq \pi$,
 - $y := \arctan(x)$ erfüllt $-\pi/2 < \Re(y) < \pi/2$,
 - $y := \operatorname{arccot}(x)$ erfüllt $-\pi/2 < \Re(y) \leq \pi/2$.
- ☞ Für `arcsin` und `arccos` sind die Verzweigungsschnitte die reellen Intervalle $(-\infty, -1)$ und $(1, \infty)$.
Für `arctan` sind die Verzweigungsschnitte die Intervalle $(-\infty \cdot i, -i]$ und $[i, \infty \cdot i)$ auf der imaginären Achse.
Für `arccsc` und `arcsec` ist der Verzweigungsschnitt das reelle Intervall $(-1, 1)$.
Für `arccot` ist der Verzweigungsschnitt das Intervall $[-i, i]$ auf der imaginären Achse.

Die Funktionswerte springen, wenn man einen Verzweigungsschnitt überschreitet. Siehe Beispiel ??.

- ☞ Die Ausdrücke $\text{arccsc}(x)$ und $\text{arcsec}(x)$ werden unmittelbar in der Form $\text{arccsc}(x) = \arcsin(1/x)$ und $\text{arcsec}(x) = \arccos(1/x)$ umgeschrieben.

Der Arcus-Kotangens ist in MuPAD gemäß $\text{arccot}(x) = \arctan(1/x)$ implementiert. Allerdings kann arccot sich selbst unevaluiert zurückliefern, ohne mittels arctan umgeschrieben zu werden. Aufgrund dieser Definition schneidet die reelle Achse den Verzweigungsschnitt und arccot hat am Ursprung eine Sprungstelle!



- ☞ Die float-Attribute sind Kernfunktionen, d. h., die Gleitpunktauswertung ist schnell.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
    arcsec(I), arccot(1)
```

```

      PI  PI                                PI                                PI
      --, --, arctan(5 + I), arcsin(3), -- + I arcsinh(1), -
      2   4                                2                                4
```

```
>> arcsin(-x), arccos(x + 1), arctan(1/x)
```

```

                                     / 1 \
      -arcsin(x), arccos(x + 1), arctan| - |
                                     \ x /
```

Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)

0.1237153458, 0.9506879769 - 2.956002937 I, 1.570796327
```

Beispiel 2. Einige spezielle Werte sind implementiert:

```
>> arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)
```

```

      PI  2 PI      PI
      --, ----, - --
      4     5      12
```

Solche Vereinfachungen geschehen für Argumente, die Bildwerte eines rationalen Vielfachen von π unter einer trigonometrischen Funktion sind:

```
>> sin(9/10*PI), arcsin(sin(9/10*PI))
```

$$\frac{1}{2} \frac{5}{4} - \frac{1}{4}, \frac{\pi}{10}$$

```
>> cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))
```

$$\frac{\frac{1}{2} + \frac{1}{2}}{(2 - 2)^{\frac{1}{2}}}, \frac{3\pi}{8}$$

Beispiel 3. Ist das Argument ein rationales Vielfaches von I , so wird das Ergebnis durch Hyperbel-Funktionen ausgedrückt:

```
>> arcsin(5*I), arccos(5/4*I), arctan(-3*I)
```

$$\frac{\pi}{2} I \operatorname{arcsinh}(5), -\frac{\pi}{2} I \operatorname{arcsinh}(5/4), -I \operatorname{arctanh}(3)$$

Für andere komplexe Argumente werden unevaluierte Funktionsaufrufe ohne Vereinfachungen zurückgeliefert:

```
>> arcsin(1/2^(1/2) + I), arccos(1 - 3*I)
```

$$\operatorname{arcsin}\left(\frac{1}{2} + I\right), \operatorname{arccos}(1 - 3I)$$

Beispiel 4. Die Funktionswerte springen, wenn ein Verzweigungsschnitt überschritten wird:

```
>> arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)
```

$$1.570796327 + 1.316957897 I, 1.570796327 - 1.316957897 I$$

Auf dem Verzweigungsschnitt gilt: für reelles $x > 1$ sind die Werte von arcsin durch den „Grenzwert von unten“, für reelles $x < -1$ durch den „Grenzwert von oben“ gegeben:

```
>> arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)
1.570796327 - 0.6223625037 I, 1.570796327 - 0.6223625037 I,
1.570796327 + 0.6223625037 I
>> arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)
- 1.570796327 + 0.6223625037 I,
- 1.570796327 + 0.6223625037 I,
- 1.570796327 - 0.6223625037 I
```

Beispiel 5. Die inversen trigonometrischen Funktionen können über die Logarithmus-Funktion mit komplexen Argumenten ausgedrückt werden:

```
>> rewrite(arcsin(x), ln), rewrite(arctan(x), ln)
2 1/2
- I ln(I x + (1 - x ) ), 1/2 I ln(1 - I x) -
1/2 I ln(I x + 1)
```

Beispiel 6. Systemfunktionen wie z.B. diff, float, limit oder series verarbeiten die inversen trigonometrischen Funktionen:

```
>> diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))
2 x
-----, - 0.06540673615 + 2.433548516 I
4 1/2
(1 - x )
>> limit(arcsin(x^2)/arctan(x^2), x = 0)
1
>> series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)
```

$$- x^3 - \frac{83 x^7}{120} - \frac{4 x^9}{189} + O(x^{10})$$

Änderungen:

- ⌘ Weitere spezielle Werte und Vereinfachungen wurden implementiert. Die `series`-Attribute wurden überarbeitet. Die Gleitpunktauswertung ist nun konsistent mit den durch die logarithmische Darstellung festgelegten Verzweigungsschnitten. Die neue Funktion `arg` ersetzt die zweiarumentige Version der `arctan`-Funktion aus früheren MuPAD-Versionen.
 - ⌘ Die Namen `asin`, ..., `acot` in früheren MuPAD-Versionen wurden in die gebräuchlicheren Namen `arcsin`, ..., `arccot` abgeändert.
-

`arcsinh`, `arccosh`, `arctanh`, `arccsch`, `arcsech`, `arccoth` – die inversen trigonometrischen Funktionen

`arcsinh(x)` stellt die Inverse der Sinus-Funktion dar.

`arccosh(x)` stellt die Inverse der Kosinus-Funktion dar.

`arctanh(x)` stellt die Inverse der Tangens-Funktion dar.

`arccsch(x)` stellt die Inverse der Kosekans-Funktion dar.

`arcsech(x)` stellt die Inverse der Sekans-Funktion dar.

`arccoth(x)` stellt die Inverse der Kotangens-Funktion dar.

Aufruf(e):

⌘ `arcsinh(x)`

⌘ `arccosh(x)`

⌘ `arctanh(x)`

⌘ `arccsch(x)`

⌘ `arcsech(x)`

⌘ `arccoth(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Seiteneffekte: Für Gleitpunktargumente reagieren die Funktionen auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `sinh`, `cosh`, `tanh`, `csch`, `sech`, `coth`

Details:

- ⌘ Diese Funktionen sind für komplexe Argumente definiert.
- ⌘ Für Gleitpunktargumente werden Gleitpunktwerte berechnet. Für die meisten exakten Argumente werden unevaluierte Funktionsaufrufe zurückgeliefert.
- ⌘ Die folgenden speziellen Werte sind implementiert:
 $\operatorname{arcsinh}(0) = 0$, $\operatorname{arccosh}(0) = i\pi/2$, $\operatorname{arccosh}(1) = 0$,
 $\operatorname{arctanh}(0) = 0$, $\operatorname{arccosh}(0) = i\pi/2$.
- ⌘ Die inversen trigonometrischen Funktionen sind mehrdeutig. Die MuPAD-Funktionen liefern Werte auf dem im Folgenden definierten Hauptzweig. Für jedes endliche komplexe x gilt:
 $y := \operatorname{arcsinh}(x)$ erfüllt $-\pi/2 \leq \Im(y) \leq \pi/2$,
 $y := \operatorname{arccosh}(x)$ erfüllt $-\pi < \Im(y) \leq \pi$,
 $y := \operatorname{arctanh}(x)$ erfüllt $-\pi/2 < \Im(y) < \pi/2$,
 $y := \operatorname{arccoth}(x)$ erfüllt $-\pi/2 < \Im(y) \leq \pi/2$.
- ⌘ Die inversen Hyperbel-Funktionen sind gemäß der folgenden Beziehung zur Logarithmus-Funktion implementiert:
 $\operatorname{arcsinh}(x) = \ln(x + \sqrt{x^2 + 1})$,
 $\operatorname{arccosh}(x) = \ln(x + \sqrt{x^2 - 1})$,
 $\operatorname{arctanh}(x) = (\ln(1 + x) - \ln(1 - x))/2$,
 $\operatorname{arccsch}(x) = \operatorname{arcsinh}(1/x)$,
 $\operatorname{arcsech}(x) = \operatorname{arccosh}(1/x)$,
 $\operatorname{arccoth}(x) = \operatorname{arctanh}(1/x)$.
Siehe Beispiel ??.
- ⌘ Dementsprechend haben diese Funktionen die folgenden Verzweigungsschnitte:
Für $\operatorname{arcsinh}$ sind die Verzweigungsschnitte die Intervalle $(-i \cdot \infty, -i)$ und $(i, i \cdot \infty)$ auf der imaginären Achse.
Für $\operatorname{arccosh}$ sind die Verzweigungsschnitte das reelle Intervall $(-\infty, 1)$ und die imaginäre Achse.
Für $\operatorname{arctanh}$ sind die Verzweigungsschnitte die reellen Intervalle $(-\infty, -1]$ und $[1, \infty)$.
Für $\operatorname{arccsch}$ ist der Verzweigungsschnitt das Intervall $(-i, i)$ auf der imaginären Achse.

Für $\operatorname{arcsech}$ sind die Verzweigungsschnitte die reellen Intervalle $(-\infty, 0)$ und $(1, \infty)$ sowie die imaginäre Achse.

Für $\operatorname{arccoth}$ ist der Verzweigungsschnitt das reelle Intervall $[-1, 1]$.

Die Funktionswerte springen, wenn man einen Verzweigungsschnitt überschreitet. Siehe Beispiel ??.

☞ Die Ausdrücke $\operatorname{arccsch}(x)$ und $\operatorname{arcsech}(x)$ werden unmittelbar in der Form $\operatorname{arccsch}(x) = \operatorname{arcsinh}(1/x)$ und $\operatorname{arcsech}(x) = \operatorname{arccosh}(1/x)$ umgeschrieben. MuPADs $\operatorname{arccoth}$ erfüllt $\operatorname{arccoth}(x) = \operatorname{arctanh}(1/x)$. Allerdings schreibt sich $\operatorname{arccoth}$ nicht selbstständig mittels $\operatorname{arctanh}$ um.

☞ Die float-Attribute sind Kernfunktionen, d. h., die Gleitpunktauswertung ist schnell.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> arcsinh(1), arccosh(1/sqrt(2)), arctanh(5 + I), arccsch(1/3),
    arcsech(I), arccoth(2)
```

$$\operatorname{arcsinh}(1), \operatorname{arccosh}\left(\frac{1}{\sqrt{2}}\right), \operatorname{arctanh}(5 + I), \operatorname{arcsinh}(3),$$

$$\operatorname{arccosh}(-I), \operatorname{arccoth}(2)$$

```
>> arcsinh(-x), arccosh(x + 1), arctanh(1/x)
```

$$-\operatorname{arcsinh}(x), \operatorname{arccosh}(x + 1), \operatorname{arctanh}\left(\frac{1}{x}\right)$$

Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)
0.1230889466, 2.956002937 + 0.9506879769 I, - 1.570796327 I
```

Beispiel 2. Die inversen trigonometrischen Funktionen können über die Logarithmus-Funktion ausgedrückt werden:

```
>> rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)
```

$$\ln(x + \sqrt{x^2 + 1}), \frac{\ln(x + 1)}{2} - \frac{\ln(1 - x)}{2}$$

Beispiel 3. Die Funktionswerte springen, wenn ein Verzweigungsschnitt überschritten wird:

```
>> arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)
0.5493061443 + 1.570796327 I, 0.5493061443 - 1.570796327 I
```

Auf dem Verzweigungsschnitt gilt: für reelles $x > 1$ sind die Werte von $\operatorname{arctanh}$ durch den „Grenzwert von unten“, für reelles $x < -1$ durch den „Grenzwert von oben“ gegeben:

```
>> arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)
1.198947636 - 1.570796327 I, 1.198947636 - 1.570796327 I,
1.198947636 + 1.570796327 I
>> arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)
- 1.198947636 + 1.570796327 I, - 1.198947636 + 1.570796327 I,
- 1.198947636 - 1.570796327 I
```

Beispiel 4. Systemfunktionen wie z.B. `diff`, `float`, `limit` oder `series` verarbeiten die inversen trigonometrischen Funktionen:

```
>> diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))
      2 x
-----, 0.3427241326 + 2.698556745 I
      4      1/2
(x  + 1)

>> limit(arcsinh(x)/arctanh(x), x = 0)
1

>> series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)
      3      83 x      4 x      10
x  + ---- - ---- + O(x )
      120      189
```

Änderungen:

- ⌘ Die `series`-Attribute wurden überarbeitet. Die Gleitpunktauswertung ist nun konsistent mit den durch die logarithmische Darstellung festgelegten Verzweigungsschnitten.
 - ⌘ Die Namen `asinh`, ..., `acoth` in früheren MuPAD-Versionen wurden in die gebräuchlicheren Namen `arcsinh`, ..., `arccoth` abgeändert.
-

`arg` – das Argument (Polarwinkel) einer komplexen Zahl

`arg(x, y)` liefert das Argument der komplexen Zahl mit Realteil `x` und Imaginärteil `y`.

Aufruf(e):

- ⌘ `arg(x, y)`

Parameter:

`x, y` — arithmetische Ausdrücke, die reelle Zahlen repräsentieren

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x, y`

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `arctan`, `Im`, `Re`, `rectform`

Details:

- ⌘ Das Argument einer nicht-verschwindenden komplexen Zahl $z = x + iy = |z| e^{i\phi}$ ist der (reelle) Polarwinkel ϕ . `arg(x, y)` stellt den Hauptwert $\phi \in (-\pi, \pi]$ dar. Für $x \neq 0$, $y \neq 0$ ist dieser Winkel durch

$$\arg(x, y) = \arctan\left(\frac{y}{x}\right) + \frac{\pi}{2} \operatorname{sign}(y) (1 - \operatorname{sign}(x))$$

gegeben.

- ⌘ Ist eines der Argumente `x, y` ein nicht-reeller numerischer Wert, so wird ein Fehler ausgelöst. Symbolische Argumente werden als reell angenommen.

- ☞ Ein Gleitpunktwert wird berechnet, wenn beide Argumente numerisch sind und mindestens eines der Argumente eine Gleitpunktzahl ist.
- ☞ Kann das Vorzeichen der Argumente ermittelt werden, so wird das Ergebnis mittels `arctan` ausgedrückt. Siehe Beispiel ?? . Anderenfalls wird ein unevaluierter Aufruf von `arg` zurückgeliefert, wobei numerische Faktoren im ersten Argument eliminiert werden. Siehe Beispiel ?? .
- ☞ Der Aufruf `arg(0,0)` liefert 0.
- ☞ Eine alternative Darstellung des Argumentes ist $\arg(x,y) = -i \ln(z/|z|) = -i \ln(\text{sign}(z))$. Siehe Beispiel ?? .

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> arg(2, 3), arg(x, 4), arg(4, y), arg(x, y), arg(10, y + PI)
```

```
arctan(3/2), arg(x, 4), arctan| / y \
                             - | , arg(x, y),
                             \ 4 /
```

```
arctan| / y    PI \
        -- + -- |
        \ 10   10 /
```

```
>> arg(x, infinity), arg(-infinity, 3), arg(-infinity, -3)
```

```
PI
-- , PI, -PI
2
```

Für Gleitkommazahlen wird der numerische Wert von `arg` berechnet:

```
>> arg(2.0, 3), arg(2, 3.0), arg(10.0^100, 10.0^(-100))

0.9827937233, 0.9827937233, 1.0e-200
```

Beispiel 2. `arg` reagiert auf Eigenschaften:

```
>> assume(x > 0): assume(y < 0): arg(x, y)
```

```
arctan| / y \
        - |
        \ x /
```

```
>> assume(x < 0): assume(y > 0): arg(x, y)
```

$$\text{PI} + \arctan\left|\frac{y}{x}\right|$$

```
>> assume(x <> 0): arg(x, 3)
```

$$\frac{\text{PI} (1 - \text{sign}(x))}{2} + \arctan\left|\frac{y}{x}\right|$$

```
>> unassume(x), unassume(y):
```

Beispiel 3. Gewisse Vereinfachungen werden auch bei unevaluierte Rückgabe durchgeführt. Insbesondere werden numerische Faktoren im ersten Argument eliminiert:

```
>> arg(3*x, 9*y), arg(-12*sqrt(2)*x, 12*y)
```

$$\arg(x, 3 y), \arg(-x^2, y)$$

Beispiel 4. Mittels `rewrite` können unevaluierte Aufrufe von `arg` in die logarithmische Darstellung umgeformt werden:

```
>> rewrite(arg(x, y), ln)
```

$$-I \ln\left|\frac{x + I y}{\text{abs}(x + I y)}\right|$$

Beispiel 5. Systemfunktionen wie z.B. `diff`, `float`, `limit` oder `series` verarbeiten `arg`:

```
>> diff(arg(x, y), x), float(arg(PI, ln(2)))
```

$$-\frac{y}{x^2 + y^2}, 0.2171564814$$

```
>> limit(arg(x, x^2/(1+x)), x = infinity)
```

$$\frac{\text{PI}}{4}$$

```
>> series(arg(x, x^2), x = 1, 4, Real)
```

$$\frac{\pi}{4} + \frac{x}{2} - \frac{1}{2} \sqrt{x-1} - \frac{(x-1)^2}{4} + \frac{(x-1)^3}{12} + O((x-1)^4)$$

Änderungen:

⌘ arg hieß früher atan.

args – Zugriff auf Prozedurparameter

args(0) liefert die Anzahl Parameter zurück, mit denen die aktuelle Prozedur aufgerufen wurde.

args(i) liefert den Wert des i-ten Parameters der aktuellen Prozedur.

Aufruf(e):

⌘ args()
 ⌘ args(0)
 ⌘ args(i)
 ⌘ args(i..j)

Parameter:

i, j — positive ganze Zahlen

Rückgabewert: args(0) liefert eine natürliche Zahl zurück. Alle anderen Formen des Aufrufs liefern ein beliebiges MuPAD Objekt oder eine Folge solcher Objekte.

Verwandte Funktionen: context, DOM_PROC, DOM_VAR,
 Pref::typeCheck, proc, procname, testargs

Details:

⌘ args ermöglicht den Zugriff auf die aktuellen Parameter einer Prozedur und kann nur in Prozeduren verwendet werden. args ist hauptsächlich für Prozeduren mit variabler Anzahl von Parametern gedacht, denn sonst kann man die Parameter natürlich einfach über ihre Namen ansprechen.

- ⌘ `args()` gibt eine Ausdrucksfolge aller aktuellen Parameter zurück.
 - ⌘ `args(0)` gibt die Anzahl der aktuellen Parameter an.
 - ⌘ `args(i)` gibt den *i*-ten Parameter zurück.
 - ⌘ `args(i..j)` liefert eine Ausdrucksfolge des *i*-ten bis *j*-ten Parameters.
 - ⌘ Durch `args` erfolgt keine Auswertung der Parameter in Prozeduren mit der Option `hold`. Dies kann durch die Funktionen `context` oder `eval` nachträglich erzwungen werden. Siehe Beispiel ??.
 - ⌘ `procname(args())` gibt einen symbolischen Aufruf der aktuellen Prozedur mit evaluierten Argumenten zurück.
 - ⌘ Zuweisung von Werten an formale Parameter eines Prozeduraufrufs verändert das Ergebnis von `args` (siehe Beispiel ??). `args(0)` bleibt unverändert.
 - ⌘ `args` ist eine Funktion des Systemkerns.
-

Beispiel 1. Dieses Beispiel demonstriert die verschiedenen Aufrufformen von `args`:

```
>> f := proc() begin
    print(Unquoted, "number of arguments" = args(0)):
    print(Unquoted, "sequence of all arguments" = args()):
    if args(0) > 0 then
        print(Unquoted, "first argument" = args(1)):
    end_if:
    if args(0) >= 3 then
        print(Unquoted, "second, third argument" = args(2..3)):
    end_if:
end_proc:

>> f():

        number of arguments = 0

        sequence of all arguments = null()

>> f(42):

        number of arguments = 1

        sequence of all arguments = 42

        first argument = 42

>> f(a, b, c, d):
```

```

number of arguments = 4

sequence of all arguments = (a, b, c, d)

first argument = a

second, third argument = (b, c)

```

Beispiel 2. `args` wertet die zurückgelieferten Parameter in Prozeduren mit der Option `hold` nicht aus. Man kann `context` benutzen, um eine nachträgliche Auswertung zu erreichen:

```

>> f := proc()
    option hold;
    begin
        args(1), context(args(1))
    end_proc;

>> delete x, y: x := y: y := 2: f(x)

x, 2

```

Beispiel 3. Hier wird `args` für den Zugriff auf die Parameter einer Prozedur mit beliebiger Anzahl von Argumenten verwendet:

```

>> f := proc() begin
    args(1) * _plus(args(2..args(0)))
end_proc:

f(2, 3), f(2, 3, 4)

6, 14

```

Beispiel 4. Die Zuweisung von Werten an formale Parameter beeinflusst `args`. Im folgenden Beispiel liefert `args` den Wert 4, der innerhalb der Prozedur zugewiesen wird, und nicht den Wert 1, der das Argument des Prozedur-Aufrufs ist:

```

>> f := proc(a) begin a := 4; args() end_proc:

f(1)

4

```

Änderungen:

- ⌘ Zuweisungen an formale Parameter beeinflussen jetzt args.
-

array – Erzeugung eines Feldes

`array(m1..n1, m2..n2, ...)` erzeugt ein Feld mit nicht initialisierten Einträgen, wobei der erste Index von m_1 bis n_1 läuft, der zweite Index von m_2 bis n_2 , usw.

`array(m1..n1, m2..n2, ..., list)` erzeugt ein Feld mit Einträgen, die aus `list` initialisiert werden.

Aufruf(e):

- ⌘ `array(m1..n1 <, m2..n2, ...>)`
- ⌘ `array(m1..n1, <m2..n2, ...,> index1 = entry1, index2 = entry2, ...)`
- ⌘ `array(m1..n1, <m2..n2, ...,> list)`

Parameter:


- `m1, n1, m2, n2, ...` — die Grenzen: ganze Zahlen
- `index1, index2, ...` — eine Folge ganzer Zahlen, die einen gültigen Feld-Eintrag definieren
- `entry1, entry2, ...` — beliebige Objekte
- `list` — eine Liste, möglicherweise geschachtelt

Rückgabewert: ein Objekt vom Typ `DOM_ARRAY`.

Verwandte Funktionen: `_assign, _index, assignElements, delete, DOM_ARRAY, DOM_LIST, DOM_TABLE, indexval, matrix, table`

Details:

- ⌘ Felder sind Container-Objekte zur Speicherung von Daten. Im Gegensatz zu Tabellen müssen die Indizes Folgen ganzer Zahlen sein. Während Tabellen dynamisch in der Größe wachsen können, ist die Anzahl der Einträge eines Feldes fest.
- ⌘ Für ein Feld `A` und eine Folge von ganzen Zahlen `index`, die einen gültigen Feld-Index bildet, liefert der indizierte Aufruf `A[index]` den zugehörigen Eintrag. Siehe Beispiele ?? und ??.
- ⌘ Eine indizierte Zuweisung der Form `A[index] := entry` initialisiert oder überschreibt den zu `index` zugehörigen Eintrag. Siehe Beispiele ?? und ??.

- ☞ `array` erzeugt ein Feld. Die Grenzen müssen die Bedingungen $m_1 \leq n_1$, $m_2 \leq n_2$, usw. erfüllen. Die Dimension des resultierenden Feldes ist die Anzahl der gegebenen Bereiche; wenigstens ein Bereich muss angegeben werden. Die gesamte Anzahl von Einträgen des sich ergebenden Feldes ist $(m_1 - n_1 + 1)(m_2 - n_2 + 1) \cdots$.
 - ☞ Wenn nur Index-Bereiche als Argumente gegeben sind, dann wird ein Feld mit nicht initialisierten Einträgen erzeugt. Siehe Beispiel ??.
 - ☞ Wenn Gleichungen der Form `index=entry` gegeben sind, dann wird der zu `index` gehörige Feld-Eintrag mit `entry` initialisiert. Dies ist nützlich, um einige bestimmte Feld-Einträge selektiv zu initialisieren.
 Jeder Index muss ein gültiger Feld-Index der Form `i1` für ein-dimensionale Felder und `(i1,i2,...)` für höher-dimensionale Felder sein, wobei `i1,i2,...` ganze Zahlen innerhalb der gültigen Grenzen sind, also $m_1 \leq i_1 \leq n_1$, $m_2 \leq i_2 \leq n_2$, usw. erfüllen und die Anzahl der ganzen Zahlen in `index` mit der Dimension des Feldes übereinstimmt.
 - ☞ Wenn das Argument `list` gegeben ist, dann wird das Feld mit den Operanden von `list` initialisiert. Dies ist nützlich, um alle Einträge des Feldes auf einmal zu initialisieren. Die Struktur der Liste muss mit der Struktur des Feldes genau übereinstimmen, so dass die Schachtelungstiefe der Liste größer oder gleich der Dimension des Feldes ist und die Anzahl der Listen-Einträge in der k -ten Schachtelungstiefe gleich der Größe des k -ten Index-Bereichs ist. Siehe Beispiel ??.
 - ☞ Ein Aufruf der Form `delete A[index]` entfernt den `index` zugehörigen Eintrag, so dass er wieder uninitialisiert ist. Siehe Beispiel ??.
 - ☞ Intern haben uninitialisierte Feld-Einträge den Wert `NIL`. Somit hat die Zuweisung von `NIL` an einen Feld-Eintrag die gleiche Wirkung wie die Entfernung mit `delete` und anschließend liefert ein Aufruf der Form `A[index]` den symbolischen Ausdruck `A[index]` und nicht `NIL` zurück, wie man es erwarten würde. Siehe Beispiel ??.
- 
- ☞ Ein ein-dimensionales Feld wird als Zeilenvektor ausgegeben. Der Index gibt die Spalte in der Zeile an.
 - ☞ Ein zwei-dimensionales Feld wird als Matrix ausgegeben. Dabei gibt der erste Index die Matrix-Zeile und der zweite Index die Matrix-Spalte an.
 - ☞ Ein ein- oder zwei-dimensionales Feld, das so groß ist, dass es die maximale Ausgabebreite `TEXTWIDTH` überschreitet, wird in der Form `array(m1..n1, m2..n2, ..., index1 = entry1, index2 = entry2, ...)` ausgegeben. Siehe Beispiel ??.
- Dasselbe gilt für Felder mit einer Dimension größer als zwei. Siehe Beispiele ?? und ??.

- ☞ Auf Feldern sind keine arithmetischen Operationen definiert. Mit `matrix` lassen sich ein-dimensionale Vektoren und zwei-dimensionale Matrizen im mathematischen Sinn erzeugen.
- ☞ Wenn ein Feld evaluiert wird, so wird es nur zurückgeliefert. Die Evaluierung wird nicht rekursiv auf alle Feld-Einträge angewendet. Dies macht den Datentyp `Feld` effizienter. Um die Einträge eines Feldes zu evaluieren, muss man die Funktion `eval` mittels `map` explizit auf die Einträge anwenden. Siehe Beispiel ??.
- ☞ `array` ist eine Funktion des Systemkerns.

Beispiel 1. Wir erzeugen ein uninitialisiertes ein-dimensionales Feld mit Indizes von 2 bis 4:

```
>> A := array(2..4)
```

```

+-              +-
| ?[2], ?[3], ?[4] |
+-              +-

```

Die Fragezeichen in der Ausgabe zeigen an, dass die Feld-Einträge nicht initialisiert sind. Wir setzen den mittleren Eintrag auf 5 und den letzten Eintrag auf "MuPAD":

```
>> A[3] := 5: A[4] := "MuPAD": A
```

```

+-              +-
| ?[2], 5, "MuPAD" |
+-              +-

```

Man kann über indizierte Aufrufe auf Feld-Einträge zugreifen. Weil der Eintrag `A[2]` nicht initialisiert ist, wird `A[2]` zurückgeliefert:

```
>> A[2], A[3], A[4]
```

```
A[2], 5, "MuPAD"
```

Man kann Felder bei der Erzeugung initialisieren, indem man Initialisierungsgleichungen an `array` übergibt:

```
>> A := array(2..4, 3 = 5, 4 = "MuPAD")
```

```

+-              +-
| ?[2], 5, "MuPAD" |
+-              +-

```

Man kann alle Einträge eines Feldes bei der Erzeugung initialisieren, indem man eine Liste mit Anfangswerten an `array` übergibt:


```
>> array(2..4, [PI, 5, "MuPAD"])
```

```

+-              +-
|  PI, 5, "MuPAD" |
+-              +-

```

Beispiel 2. Feldgrenzen dürfen auch negative ganze Zahlen sein:

```
>> A := array(-1..1, [2, sin(x), FAIL])
```

```

+-              +-
|  2, sin(x), FAIL |
+-              +-

```

```
>> A[-1], A[0], A[1]
```

```
2, sin(x), FAIL
```

Beispiel 3. Der \$-Operator kann zur Erzeugung einer Folge von Initialisierungs-Gleichungen verwendet werden:

```
>> array(1..8, i = i^2 $ i = 1..8)
```

```

+-              +-
|  1, 4, 9, 16, 25, 36, 49, 64 |
+-              +-

```

Ebenso kann der \$-Operator zur Erzeugung einer Initialisierungs-Liste verwendet werden:

```
>> array(1..8, [i^2 $ i = 1..8])
```

```

+-              +-
|  1, 4, 9, 16, 25, 36, 49, 64 |
+-              +-

```

Beispiel 4. Wir erzeugen eine 2×2 Matrix als ein zwei-dimensionales Feld:

```
>> A := array(1..2, 1..2, (1, 2) = 42, (2, 1) = 1 + I)
```

```

+-              +-
|  ?[1, 1],      42      |
|                  |
|  1 + I,    ?[2, 2]    |
+-              +-

```

```
>> op(A, 1), op(A, 2), op(A, 3), op(A, 4)
                                NIL, 42, 1 + I, NIL
```

```
>> A[1, 1], A[1, 2], A[2, 1], A[2, 2]
      A[1, 1], 42, 1 + I, A[2, 2]
```

```
>> A[1, 1] := 0: A[1, 2] := 5: A
```

	+-		-+	
		0,	5	
		1 + I,	?[2, 2]	
	+-		-	

```
>> delete A[2, 1]: A[2, 1], op(A, 3)
                                A[2, 1], NIL
```

```
>> A[1, 2] := NIL: A[1, 2], op(A, 2)
                                A[1, 2], NIL
```

```
>> A := array(1..8, 1..8, 1..8,
              (1, 1, 1) = 111,
              (8, 8, 8) = 888)

              array(1..8, 1..8, 1..8,
                    (1, 1, 1) = 111,
                    (8, 8, 8) = 888
              )

>> A[1, 1, 1], A[1, 1, 2]

              111, A[1, 1, 2]
```

Beispiel 6. Eine geschachtelte Liste kann zur Initialisierung eines zwei-dimensionalen Feldes verwendet werden. Die inneren Listen sind die Zeilen der erzeugten Matrix:

```
>> array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

```

+-      +-
|  1, 2, 3  |
|           |
|  4, 5, 6  |
+-      +-

```

Hier wird ein drei-dimensionales Feld erzeugt und durch geschachtelte Listen der Tiefe drei initialisiert. Die äußere Liste hat zwei Einträge für die erste Dimension. Jeder dieser Einträge ist eine Liste mit drei Einträgen für die zweite Dimension. Schließlich haben die innersten Listen jeweils einen Eintrag für die dritte Dimension:

```
>> array(2..3, 1..3, 1..1,
[
[ [1], [2], [3] ],
[ [4], [5], [6] ]
])
```

```

array(2..3, 1..3, 1..1,
(2, 1, 1) = 1,
(2, 2, 1) = 2,
(2, 3, 1) = 3,
(3, 1, 1) = 4,
(3, 2, 1) = 5,
(3, 3, 1) = 6
)
```

Beispiel 7. Wenn ein Feld evaluiert wird, dann wird es nur zurückgeliefert. Die Evaluierung wird nicht rekursiv auf alle Feld-Einträge angewendet. Hier werden die Einträge a und b nicht evaluiert:

```
>> A := array(1..2, [a, b]):
a := 1: b := 2:
A, eval(A)
```

```

+-      +-      +-      +-
|  a, b  |, |  a, b  |
+-      +-      +-      +-

```

Aufgrund der speziellen Evaluierung von Feldern, evaluiert der Index-Operator die Feld-Einträge, nachdem er sie aus dem Feld extrahiert hat:

```
>> A[1], A[2]
```

```
1, 2
```

Man muss `eval` mittels `map` explizit auf das Feld anwenden, um seine Einträge zu evaluieren:

```
>> map(A, eval)
```

```
+-      +-
| 1, 2 |
+-      +-
```

Beispiel 8. Ein zwei-dimensionales Feld wird üblicherweise in Matrix-Form ausgegeben:

```
>> A := array(1..4, 1..4,
              (1, 1) = 11,
              (4, 4) = 44)
```

```
+-                                     +-
|      11,    ?[1, 2], ?[1, 3], ?[1, 4] |
|      ?[2, 1], ?[2, 2], ?[2, 3], ?[2, 4] |
|      ?[3, 1], ?[3, 2], ?[3, 3], ?[3, 4] |
|      ?[4, 1], ?[4, 2], ?[4, 3],    44  |
+-                                     +-
```

Wenn die Ausgabe nicht in die durch `TEXTWIDTH` gegebene Textbreite passt, so wird eine kompaktere Ausgabe verwendet:

```
>> TEXTWIDTH := 20:
    A;
    delete TEXTWIDTH:

array(1..4, 1..4,
      (1, 1) = 11,
      (4, 4) = 44
)
```

Änderungen:

- ☞ Unterteilte Felder werden nicht mehr unterstützt.
- ☞ Negative Zahlen sind nun als Grenzen erlaubt.

assign – als Gleichungen gegebene Zuweisungen ausführen

Für jede Gleichung in einer angegebenen Liste, Menge oder Tabelle L von Gleichungen wertet `assign(L)` beide Seiten der Gleichung aus und weist das Ergebnis der Auswertung der rechten Seite dem Ergebnis der Auswertung der linken Seite zu.

`assign(L, S)` führt nur für diejenigen Gleichungen, deren linke Seite in der Menge S liegt, eine Zuweisung durch.

Aufruf(e):

⇒ `assign(L)`
⇒ `assign(L, S)`

Parameter:

L — eine Liste, Menge oder Tabelle bestehend aus Gleichungen
 S — eine Menge

Rückgabewert: L .

Verwandte Funktionen: `:=`, `_assign`, `assignElements`, `delete`, `evalassign`

Details:

- ⇒ Da die Argumente an `assign` ausgewertet werden, muss das *Ergebnis der Auswertung* der linken Seite jeder Gleichung eine zulässige linke Seite für eine Zuweisung sein. Nähere Informationen darüber finden sich auf der Hilfeseite des Zuweisungsoperators `:=`.
 - ⇒ Die Zuweisungen werden in der Reihenfolge von links nach rechts durchgeführt. Siehe Beispiel ??.
 - ⇒ `assign` bietet eine bequeme Möglichkeit, eine von `solve` zurückgelieferten Lösung eines Gleichungssystems den Unbestimmten zuzuweisen. Siehe Beispiel ??.
-

Beispiel 1. Wir weisen den drei Bezeichnern $B1, B2, B3$ Werte zu:

```
>> delete B1, B2, B3:  
    assign([B1 = 42, B2 = 13, B3 = 666]): B1, B2, B3  
  
          42, 13, 666
```

Wir geben ein zweites Argument an, um nur die Zuweisungen mit linker Seite B1 durchzuführen:

```
>> delete B1, B2, B3:
      assign([B1 = 42, B2 = 13, B3 = 666], {B1}): B1, B2, B3
              42, B2, B3
```

Das erste Argument darf auch eine Tabelle von Gleichungen sein:

```
>> delete B1, B2, B3:
      assign(table(B1 = 42, B2 = 13, B3 = 666)): B1, B2, B3
              42, 13, 666
```

Beispiel 2. Im Gegensatz zu `_assign` werden die linken Seiten der Gleichungen evaluiert:

```
>> delete a, b: a := b: assign({a = 3}): a, b
              3, 3

>> delete a, b: a := b: a := 3: a, b
              3, b
```

Beispiel 3. Das zugewiesene Objekt darf auch eine Folge sein:

```
>> assign([X=(2,7)])
              [X = (2, 7)]

>> X
              2, 7
```

Beispiel 4. Die Zuweisungen werden nacheinander von links nach rechts ausgeführt. Da die rechte Seite ausgewertet wird, erhält im folgenden Beispiel der Bezeichner C den Wert 3:

```
>> assign([B=3, C=B])
              [B = 3, C = B]

>> level(C,1)
              3
```

Beispiel 5. Ruft man `solve` mit einem System algebraischer Gleichungen auf, so ist das Ergebnis oft eine Menge von Listen von Zuweisungen. `assign` kann dafür verwendet werden, diese Zuweisungen durchzuführen:

```
>> sys:={x^2+y^2=2, x+y=5}:
      S:= solve(sys)
```

```
{[x = 5/2 - 1/2 I 211/2, y = 1/2 I 211/2 + 5/2],
 [x = 1/2 I 211/2 + 5/2, y = 5/2 - 1/2 I 211/2]}
```

Wir überprüfen, ob die erste Lösung wirklich eine Lösung ist:

```
>> assign(S[1]): sys
```

```
{5 = 5, (5/2 - 1/2 I 211/2)2 + (1/2 I 211/2 + 5/2)2 = 2}
```

Besseren Aufschluss ergibt eine Gleitkommaauswertung:

```
>> float(sys)
```

```
{5.0 = 5.0, 2.0 - 8.67361738e-19 I = 2.0}
```

Änderungen:

☞ Keine Änderungen.

assignElements – Zuweisung an Einträge eines Felds, einer Liste oder einer Tabelle

`assignElements(L, [index1] = value1, [index2] = value2, ...)` liefert eine abgeänderte Kopie von `L`, in der `value1` unter `index1`, `value2` unter `index2` etc. eingetragen sind.

Aufruf(e):

```
☞ assignElements(L, [index1] = value1, [index2] = value2, ...)
☞ assignElements(L, [[index1], value1], [[index2], value2], ...)
```

Parameter:

`L` — ein Feld, eine Liste, oder eine Tabelle
`index1, index2, ...` — gültige Indizes für `L`
`value1, value2, ...` — beliebige MuPAD-Objekte

Rückgabewert: ein Objekt vom selben Typ wie `L`.

Verwandte Funktionen: `:=`, `_assign`, `_index`, `array`, `assign`, `delete`, `DOM_ARRAY`, `DOM_LIST`, `DOM_TABLE`, `evalassign`, `table`

Details:

- ☞ `R:=assignElements(L,[index1]=value1,[index2]=value2,...)` bewirkt dasselbe wie die Zuweisungen `R:=L: R[index1]:=value1: R[index2]:=value2: ...`. `R` ist jedoch aus Effizienzgründen zu bevorzugen.
- ☞ `assignElements` liefert lediglich eine abgeänderte Kopie des ersten Arguments zurück; das erste Argument selbst bleibt unverändert. Siehe Beispiel ??.
- ☞ Die zweite Art des Aufrufs von `assignElements` (mit Listen anstelle von Gleichungen) ist zur ersten Art äquivalent; es dürfen auch Listen und Gleichungen gemischt verwendet werden. Siehe Beispiel ??.
- ☞ Alle Zuweisungen werden gleichzeitig durchgeführt; die Reihenfolge der Argumente ist daher ohne Belang. Siehe Beispiel ??.
- ☞ Alle Regeln für indizierte Zuweisungen sind auch hier unverändert gültig, insbesondere was die Zulässigkeit von Indizes betrifft. Ist `L` eine Liste, so sind positive ganze Zahlen kleiner oder gleich der Länge von `L` zulässig; ist `L` ein Feld, so müssen die Indizes ganze Zahlen (bzw. Folgen von so vielen ganzen Zahlen, wie es der Dimension des Feldes entspricht) innerhalb der Feldgrenzen sein. Für die Indizes von Tabellen gibt es keine Beschränkungen.
- ☞ `assignElements` ist eine Funktion des Systemkerns.

Beispiel 1. Zuweisungen können als Gleichungen oder Listen gegeben werden, und beide Formen dürfen auch gemischt vorkommen:

```
>> L := array(1..3, [3, 4, 5]);
      assignElements(L, [1] = one, [2] = two, [3] = three);
      assignElements(L, [[1], one], [[2], two], [[3], three]);
      assignElements(L, [1] = one, [[2], two], [3] = three);
```



```

+-          +-
| 3, 4, 5 |
+-          +-

```

```

+-          +-
| one, two, three |
+-          +-

```

```

+-          +-
| one, two, three |
+-          +-

```

```

+-          +-
| one, two, three |
+-          +-

```

Das Feld `L` selbst wird durch `assignElements` nicht verändert:

```
>> L
```

```

+-          +-
| 3, 4, 5 |
+-          +-

```

Beispiel 2. Auch Folgen können einem Feld als Einträge zugewiesen werden. Dazu sind sie zu klammern:

```
>> R := assignElements(array(1..2), [1] = (1, 7), [2] = PI)
```

```

+-          +-
| 1, 7, PI |
+-          +-

```

```
>> [R[1]], [R[2]]
```

```
[1, 7], [PI]
```

Beispiel 3. Der Folgenerator `$` ist nützlich, um Folgen von Zuweisungen zu erzeugen:

```
>> L := [i $ i = 1..10];
      assignElements(L, [i] = L[i] + L[i + 1] $ i = 1..9)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 10]
```

Die Reihenfolge der Argumente ist gleichgültig:

```
>> assignElements(L, [10 - i] = L[10 - i] + L[11 - i] $ i = 1..9)
      [3, 5, 7, 9, 11, 13, 15, 17, 19, 10]
```

Beispiel 4. In Tabellen dürfen beliebige Indizes, z. B. Zeichenketten, verwendet werden:

```
>> assignElements(table(), [expr2text(i)] = i^2 $ i = 1..4)

      table(
        "4" = 16,
        "3" = 9,
        "2" = 4,
        "1" = 1
      )
```

Beispiel 5. Für zwei- oder mehrdimensionale Felder sind die Indizes Folgen von so vielen ganzen Zahlen, wie die Dimension des Feldes angibt:

```
>> assignElements(array(1..3, 1..3),
      ([i, j] = i + j $ i = 1..3) $ j = 1..3)
```

```
  +-      +-
  |  2, 3, 4  |
  |  3, 4, 5  |
  |  4, 5, 6  |
  +-      +-
```

Änderungen:

⚡ `assignElements` hieß früher `assign_elems`.

assume – definiert die Eigenschaften von Bezeichnern

`assume(x, prop)` heftet die Eigenschaft `prop` an den Bezeichner `x`.

`assume(prop)` setzt eine „globale“ Eigenschaft, die für alle Bezeichner gültig ist.

Aufruf(e):

```

# assume(x, prop <, _and_or>)
# assume(prop <, _and_or>)
# assume(y rel z <, _and_or>)

```

Parameter:

<code>x</code>	— ein Bezeichner oder einer der Ausdrücke <code>Re(u)</code> oder <code>Im(u)</code> mit einem Bezeichner <code>u</code>
<code>prop</code>	— eine Eigenschaft
<code>_and_or</code>	— entweder <code>_and</code> oder <code>_or</code> . Ohne dieses optionale Argument wird jede bestehende Eigenschaft durch die neue Eigenschaft überschrieben. Mit <code>_and</code> oder <code>_or</code> wird die bestehende Eigenschaft nicht gelöscht, sondern durch ein logisches „und“ oder „oder“ mit der neuen Eigenschaft verknüpft.
<code>y, z</code>	— arithmetische Ausdrücke
<code>rel</code>	— einer der Vergleichsoperatoren <code><</code> , <code><=</code> , <code>=</code> , <code><></code> , <code>>=</code> oder <code>></code>

Rückgabewert: eine Eigenschaft vom Typ `Type::Property`.

Verwandte Funktionen: `_assign`, `anames`, `getprop`, `is`, `property::hasprop`, `property::implies`, `Type`, `Type::Property`, `unassume`

Details:

Eigenschaften repräsentieren Teilmengen der komplexen Ebene. Das Definieren einer Eigenschaft `prop` eines Bezeichners `x` entspricht der Aussage: „`x` repräsentiert eine Zahl der Menge `prop`“. Die Bibliothek `Type` enthält verschiedene vordefinierte Eigenschaften. `?properties` verweist auf eine Liste aller möglichen Eigenschaften.

Im Normalfall werden Bezeichner von allen Bibliotheksfunktionen als Symbole interpretiert, die komplexe Zahlen repräsentieren. Aus diesem Grund können viele Ausdrücke wie z. B. `sign(1 + x^2)` nicht vereinfacht werden, wenn nicht `x` mittels `assume` auf Teilmengen der komplexen Ebene eingeschränkt wird. Wird `x` z. B. als reell angenommen, kann der Ausdruck `sign(1 + x^2)` zu 1 vereinfacht werden.

Daher helfen Eigenschaften, Ausdrücke zu vereinfachen. Etliche Systemfunktionen berücksichtigen ebenfalls Eigenschaften von Bezeichnern und geben vereinfachte Ergebnisse zurück (siehe Beispiel ??).

Eigenschaften von Bezeichnern werden mittels `assume` gesetzt. Eigenschaften von Ausdrücken können mit den Funktionen `getprop` und `is` erfragt werden.

- ⇒ `assume(x, prop)` heftet die Eigenschaft `prop` an den Bezeichner `x`. Dabei werden existierende Eigenschaften von `x` überschrieben, falls das optionale Argument `_and_or` nicht angegeben wird.
- ⇒ Wenn das optionale Argument `_and_or` angegeben wird, werden bestehende Eigenschaften der Bezeichner `x`, `y` oder `z` nicht gelöscht, sondern durch ein logisches ‚und‘ oder ‚oder‘ mit der neuen Eigenschaft verknüpft. Die resultierende Eigenschaft wird dann an die Bezeichner `x`, `y` oder `z` geheftet. Siehe Beispiel ??.
- ⇒ Es kann dabei vorkommen, dass eine resultierende Eigenschaft nicht direkt mit den in MuPAD zur Verfügung stehenden Eigenschaften ausgedrückt werden kann. Dann wird eine allgemeinere (schwächere) Eigenschaft benutzt. Siehe Beispiel ??.
- ⇒ `assume(Re(u), prop)` und `assume(Im(u), prop)` heftet die Eigenschaft `prop` an den Real- bzw. Imaginärteil des Bezeichners `u`.
- ⇒ In `y rel z` muss mindestens eine der Größen `y` oder `z` ein Bezeichner oder von der Form `Re(u)` oder `Im(u)` sein. Die andere Seite der Relation kann ein beliebiger arithmetischer Ausdruck sein.
Die Eigenschaft, die durch die Relation gegeben ist, wird an `y` und `z` geheftet, wenn *beides* Bezeichner oder von der Form `Re(u)` oder `Im(u)` sind. Dabei werden die bestehenden Eigenschaften *beider* Seiten der Relation gelöscht, wenn nicht das optionale Argument `_and_or` angegeben wird.
Wenn `rel` entweder `<`, `>`, `<=` oder `>=` ist und `y` oder `z` Bezeichner oder von der Form `Re(u)` oder `Im(u)` sind, wird zusätzlich die Eigenschaft `Type::Real` an `y` und `z` geheftet.
Siehe Beispiel ??.
- ⇒ Der Aufruf `assume(prop <, _and_or>)` definiert eine „globale Eigenschaft“ `prop`, die als für alle Bezeichner gültig angenommen wird. Beim Erfragen von Eigenschaften von Ausdrücken (z. B. mittels `is`) wird die globale Eigenschaft logisch mittels „und“ mit den individuellen gesetzten Eigenschaften einzelner Bezeichner verknüpft.
Das Argument `_and_or` bewirkt, dass eine schon bestehende globale Eigenschaft mit der neuen globalen Eigenschaft logisch verknüpft wird.
Die globale Eigenschaft wird dem geschützten Bezeichner `Global` angeheftet, der beim Abfragen von Eigenschaften stets mitberücksichtigt wird.
Die Aufrufe `assume(prop <, _and_or>)` und `assume(Global, prop <, _and_or>)` sind äquivalent.
Siehe Beispiel ??.
- ⇒ Eigenschaften des Bezeichners `x` werden mit `unassume(x)` oder `delete x` gelöscht. Die globale Eigenschaft wird mit `unassume()` oder `unassume(Global)`

gelöscht (individuell gesetzte Eigenschaften von Bezeichnern werden dabei nicht gelöscht).

Das Zuweisen eines Wertes an einen Bezeichner überschreibt die Eigenschaften des Bezeichners. Eine Übereinstimmung der Eigenschaften des Bezeichners mit dem zugewiesenen Wert wird nicht überprüft. Siehe Beispiel ??.

Beispiel 1. Der folgende Befehl deklariert den Bezeichner n als ganze Zahl:

```
>> assume(n, Type::Integer)
```

```
      Type::Integer
```

MuPAD kann nun ableiten, dass n^2 eine nichtnegative ganze Zahl ist:

```
>> getprop(n^2), is(n^2, Type::NonNegInt)
```

```
      Type::NonNegInt, TRUE
```

Auch andere Systemfunktionen reagieren auf diese Eigenschaft:

```
>> abs(n^2 + 1), simplify(sin(2*n*PI))
```

```
      2
      n  + 1, 0
```

```
>> delete n:
```

Beispiel 2. Mit dem zusätzlichen Argument `_and` oder `_or` werden bestehende Eigenschaften nicht überschrieben und können mit neuen Eigenschaften kombiniert werden:

```
>> assume(n, Type::NonNegInt)
```

```
      Type::NonNegInt
```

```
>> assume(n, Type::NegInt, _or)
```

```
      Type::Integer
```

```
>> assume(n, Type::Positive, _and)
```

```
      Type::PosInt
```

```
>> delete n:
```

Beispiel 3. Eigenschaften des Real- und Imaginärteils eines Bezeichners können getrennt gesetzt werden:

```
>> assume(Re(z) > 0), assume(Im(z) < 0, _and)
      Re(.) > 0, Re(.) > 0 and Im(.) < 0

>> abs(Re(z)), sign(Im(z))
      Re(z), -1

>> is(z, Type::Real), is(z > 0)
      FALSE, FALSE

>> delete z:
```

Beispiel 4. Eine Relation zwischen Bezeichnern wie z.B. $x > y$ beeinflusst die Eigenschaften beider Bezeichner:

```
>> assume(x > y)
      < x
```

Eigenschaften können mittels `getprop` erfragt werden. Sowohl x als auch y haben Eigenschaften:

```
>> getprop(x), getprop(y)
      > y, < x
```

Im nächsten Aufruf wird `_and` verwendet, damit die vorher gesetzte Eigenschaft von y nicht verloren geht: y wird als größer als 0 *und* kleiner als x vorausgesetzt:

```
>> assume(y > 0, _and)
      ]0, x[ of Type::Real

>> is(x^2 >= y^2)
      TRUE
```

Der zweite `assume`-Befehl im nächsten Beispiel würde *ohne* den Operator `_and` die Eigenschaften von x überschreiben. Mit `_and` bleibt die Eigenschaft $x \geq 0$ bestehen:

```
>> unassume(y):
      assume(x >= 0): assume(y >= x, _and): is(y >= 0)
```

TRUE

Relationen zwischen Bezeichnern wie $x > y$ setzen implizit voraus, dass die beteiligten Bezeichner reell sind:

```
>> is(x, Type::Real), is(y, Type::Real)
```

TRUE, TRUE

```
>> delete x, y:
```

Beispiel 5. Im nächsten Beispiel wird eine globale Eigenschaft gesetzt:

```
>> assume(Type::NonNegative)
```

Type::NonNegative

Nun wird jeder Bezeichner automatisch als reell und nichtnegativ angenommen:

```
>> Re(x), Im(y), sign(1 + z^2)
```

x, 0, 1

Individuelle Eigenschaften einzelner Bezeichner können unabhängig von der globalen Eigenschaft gesetzt werden:

```
>> assume(x, Type::Integer)
```

Type::Integer

Beim Ableiten von Eigenschaften mittels `getprop` oder `is` werden individuelle Eigenschaften immer über das logische „und“ mit der globalen Eigenschaft verknüpft:

```
>> getprop(x), is(x < 0)
```

Type::NonNegInt, FALSE

Auch die globale Eigenschaft kann mittels `_and` bzw. `_or` mit neuen Eigenschaften kombiniert werden:

```
>> assume(Type::Negative, _or)
```

Type::Real

Um Relationen als globale Eigenschaft zu definieren, muss der Bezeichner `Global` benutzt werden:

```
>> assume(Global > 0): is(x + y + z > 0)
```

TRUE

Die globale Eigenschaft kann nur durch den Aufruf `unassume()` gelöscht werden:

```
>> delete x: unassume():
```

Beispiel 6. `_assign` und `:=` ignorieren die Eigenschaften des mit einem Wert zu belegenden Bezeichners. Alle Eigenschaften des Bezeichners werden überschrieben:

```
>> assume(a > 0): a := -2: a, getprop(a)

-2, -2

>> delete a:
```

Beispiel 7. Einige Systemfunktionen berücksichtigen die Eigenschaften von Bezeichnern:

```
>> assume(x > 0): abs(x), sign(x), Re(x), Im(x)

x, 1, x, 0
```

Die Gleichung $\ln(z_1 \cdot z_2) = \ln(z_1) + \ln(z_2)$ ist nicht für allgemeine z_1, z_2 in der komplexen Ebene gültig:

```
>> expand(ln(z1*z2))

ln(z1 z2)
```

Diese Identität gilt jedoch, wenn eine der beiden Zahlen reell und positiv ist:

```
>> assume(z1 > 0): expand(ln(z1*z2))

ln(z1) + ln(z2)

>> unassume(x): unassume(z1):
```

Beispiel 8. Wenn eine Kombination von Eigenschaften nicht direkt mit vorhandenen Eigenschaften ausgedrückt werden kann, wird dem Bezeichner eventuell eine allgemeinere Eigenschaft zugeordnet. In diesem Beispiel wird die Eigenschaft „eine Primzahl oder das Negative einer Primzahl“ zu „eine ganze Zahl ungleich Null“ verallgemeinert:

```
>> assume(x, Type::Prime):
    assume(x, -Type::Prime, _or)

Type::Integer and not Type::Zero

>> unassume(x):
```


Hintergründe:

⇒ `assume` ist eine exportierte Funktion der Bibliothek `property`.

Änderungen:

⇒ Die Fähigkeiten des „`property`“-Mechanismus wurden verbessert.

`asympt` – Berechnung einer asymptotischen Reihenentwicklung

`asympt(f, x)` berechnet die ersten Terme einer asymptotischen Entwicklung von `f` bezüglich der Variablen `x` um den Punkt `infinity`.

Aufruf(e):

⇒ `asympt(f, x)`

⇒ `asympt(f, x <= x0> <, order> <, dir>)`

Parameter:

- `f` — ein arithmetischer Ausdruck, als Funktion in `x` zu interpretieren
- `x` — ein Bezeichner
- `x0` — der Entwicklungspunkt: ein arithmetischer Ausdruck; ohne Angabe dieses Punktes wird der Entwicklungspunkt `infinity` benutzt
- `order` — die Anzahl der zu berechnenden Terme: eine nicht-negative ganze Zahl; die Standardordnung ist durch die Umgebungsvariable `ORDER` mit dem voreingestellten Wert 6 gegeben

Optionen:

- `dir` — *Left* oder *Right*. Mit *Left* ist die Entwicklung für $x < x_0$ gültig, mit *Right* für $x > x_0$. Für endliche Entwicklungspunkte x_0 ist der Standardwert *Right*.

Rückgabewert: ein Objekt vom Domain-Typ `Series::gseries` oder ein Ausdruck vom Typ `"asympt"`.

Seiteneffekte: Die Ergebnisse der Funktion hängen vom Wert der Umgebungsvariablen `ORDER` ab, die standardmäßig die Anzahl der Terme in Reihenentwicklungen bestimmt.

Überladbar durch: `f`

Verwandte Funktionen: `limit`, `O`, `ORDER`, `series`, `Series::gseries`, `taylor`, `Type::Series`

Details:

- ☞ `asympt` dient zur Berechnung asymptotischer Entwicklungen von f für $x \rightarrow \infty$. Kann eine solche Entwicklung berechnet werden, wird ein Objekt vom Domain-Typ `Series::gseries` zurückgeliefert.

Es können auch gerichtete Entwicklungen um einen endlichen reellen Punkt x_0 berechnet werden. Aus Effizienzgründen wird jedoch empfohlen, hierfür die Funktion `series` zu verwenden.

Im Gegensatz zum Standardverhalten von `series` liefert `asympt` nur gerichtete Entwicklungen, die nur längs der reellen Achse gültig zu sein brauchen.

- ☞ Kann `asympt` keine asymptotische Reihenentwicklung bestimmen, so wird ein symbolischer Ausdruck vom Typ `"asympt"` zurückgeliefert. Siehe Beispiel ??.

- ☞ Das optionale Argument `order` bestimmt die Anzahl der Terme der Entwicklung. Ohne Angabe von `order` wird der Wert der Umgebungsvariablen `ORDER` benutzt, deren Standardwert 6 durch Zuweisung an `ORDER` verändert werden kann.

Die Anzahl der Terme der Entwicklung wird vom Term mit dem kleinsten Grad an gezählt, d. h. „`order`“ ist als „relative Abbruchordnung“ anzusehen.

Es kann vorkommen, dass die Anzahl der Terme in der berechneten Reihenentwicklung von der angeforderten Anzahl abweicht. Siehe `series` für Details.



- ☞ Die Funktion `asympt` gibt ein Objekt vom Domain-Typ `Series::gseries` zurück, das mit den arithmetischen Standardoperationen sowie einigen Systemfunktionen behandelt werden kann. Beispielsweise liefert `coeff` die Koeffizienten, `expr` konvertiert die Reihe in einen Ausdruck ohne Fehlerterm, `lmonomial` liefert das führende Monom, `lterm` den führenden Term, `lcoeff` den führenden Koeffizienten, `map` wendet eine Funktion auf die Koeffizienten der Reihe an, `nthterm` liefert den n -ten Term und `nthmonomial` das n -te Monom.

Beispiel 1. Wir berechnen eine asymptotische Entwicklung für $x \rightarrow \infty$:

```
>> s := asympt(sin(1/x + exp(-x)) - sin(1/x), x)
```

$$\exp(-x) - \frac{\exp(-x)^2}{2x} + \frac{\exp(-x)^4}{24x} + O\left(\frac{\exp(-x)^6}{x}\right)$$

Auf den führenden Term bzw. den dritten Term der Entwicklung kann wie folgt zugegriffen werden:

```
>> lmonomial(s), nthterm(s, 3)
```

$$\exp(-x), \frac{\exp(-x)}{4x}$$

In der folgenden asymptotischen Entwicklung werden nur 2 Terme angefordert:

```
>> asympt(
    exp(sin(1/x + exp(-exp(x)))) - exp(sin(1/x)), x, 2
)
```

$$\exp(-\exp(x)) + \frac{\exp(-\exp(x))}{x} + O\left(\frac{\exp(-\exp(x))}{x^2}\right)$$

```
>> delete s:
```

Beispiel 2. Eine Entwicklung um einen endlichen reellen Punkt wird betrachtet. Die Reihe ist „rechts vom Entwicklungspunkt“ gültig:

```
>> asympt(abs(x/(1+x)), x = 0)
```

$$x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + O(x^8)$$

„Links vom Entwicklungspunkt“ ist eine andere Entwicklung gültig:

```
>> asympt(abs(x)/(1 + x), x = 0, Left)
```

$$-x^2 + x^3 - x^4 + x^5 - x^6 + x^7 + O(-x^8)$$

Beispiel 3. Die folgende Entwicklung ist exakt und besitzt daher keinen „Fehlerterm“:

```
>> asympt(x/exp(x), x = -infinity)
```

$$x \exp(-x)$$

Beispiel 4. Hier ist ein Beispiel, bei dem `asympt` keine asymptotische Reihenentwicklung bestimmen kann:

```
>> asympt(cos(x*s)/s, x = infinity)
```

```

      / cos(s x)
asympt| -----, x = infinity |
      \      s      \

```

Änderungen:

- ⌘ Von `asympt` gelieferte Entwicklungen sind nun vom Domain-Typ `Series::gseries`.
 - ⌘ `asympt` liefert nun einen symbolischen Ausdruck vom Typ `"asympt"`, falls keine verallgemeinerte Reihenentwicklung berechnet werden kann.
 - ⌘ The new options *Left* and *Right* were introduced.
-

bernoulli – die Bernoulli-Zahlen und -Polynome

`bernoulli(n)` liefert die n -te Bernoulli-Zahl.

`bernoulli(n, x)` liefert das n -te Bernoulli-Polynom in x .

Aufruf(e):

- ⌘ `bernoulli(n)`
- ⌘ `bernoulli(n, x)`

Parameter:

- n — ein arithmetischer Ausdruck, der eine nicht-negative ganze Zahl repräsentiert
- x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Seiteneffekte: Ist x eine Gleitpunktzahl, so reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Details:

☞ Die Bernoulli-Polynome sind durch die erzeugende Funktion

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} \frac{\text{bernoulli}(n, x)}{n!} t^n.$$

definiert.

☞ Die Bernoulli-Zahlen sind durch $\text{bernoulli}(n) = \text{bernoulli}(n, 0)$ definiert.

☞ Ein Fehler wird ausgelöst, falls n numerisch, aber keine nicht-negative ganze Zahl ist. Ein symbolischer Aufruf $\text{bernoulli}(n)$ wird zurückgeliefert, falls n nicht-numerische symbolische Bezeichner enthält. Einige Vereinfachungen von $\text{bernoulli}(n, x)$ werden für symbolisches n und spezielle numerische Werte x durchgeführt. Siehe Beispiel ??.

☞ Man beachte, dass Gleitpunktauswertung für hochgradige Polynome numerisch instabil sein kann. Siehe Beispiel ??.



Beispiel 1. Die ersten Bernoulli-Zahlen:

```
>> bernoulli(n) $ n = 0..11
1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66, 0
```

Die ersten Bernoulli-Polynome:

```
>> bernoulli(n, x) $ n = 0..4
1, x - 1/2, x^2 - x + 1/6, -x^3 + x^2/2, x^3 - 2x^2 + x - 1/30
```

Für symbolisches n wird ein symbolischer Aufruf zurückgeliefert:

```
>> bernoulli(n, x), bernoulli(n + 3/2, x), bernoulli(n + 5*I, x)
bernoulli(n, x), bernoulli(n + 3/2, x), bernoulli(n + 5 I, x)
```

Ein Fehler wird ausgelöst, falls n einen numerischen Wert darstellt, der keine nicht-negative ganze Zahl ist:

```
>> bernoulli(sin(3), x)
Error: first argument must be symbolic or a nonnegative \
integer [bernoulli]
```

Beispiel 2. Wenn x keine Unbestimmte ist, dann wird der Wert des Bernoulli-Polynoms an der Stelle x zurückgeliefert:

```
>> bernoulli(50, 1 + I)
```

$$132549963452557267373179389125/66 + 25 I$$

```
>> bernoulli(3, 1 - y), expand(bernoulli(3, 1 - y))
```

$$\frac{(1-y)^2 (3-3y)}{3} - \frac{(1-y)(3-3y)}{2} - \frac{y}{2} + \frac{1}{2}, \frac{3y^2}{2} - \frac{y^3}{2}$$

Beispiel 3. Für gewisse numerische Werte x können auch für symbolisches n Vereinfachungen durchgeführt werden:

```
>> bernoulli(n, -2), bernoulli(n, -1/2), bernoulli(n, -1/6)
```

$$(-1)^n \text{bernoulli}(n, 2) + n (-1)^{n-1} 2,$$

$$\text{bernoulli}(n) (-1)^{n-1} (2^{1-n} - 1) + n (-1)^{n-1} (1/2),$$

$$(-1)^n \text{bernoulli}(n, 1/6) + n (-1)^{n-1} (1/6)$$

```
>> bernoulli(n, 1/2), bernoulli(n, 2/3), bernoulli(n, 0.7)
```

$$\text{bernoulli}(n) (2^{1-n} - 1), (-1)^n \text{bernoulli}(n, 1/3),$$

$$(-1)^n \text{bernoulli}(n, 0.3)$$

Beispiel 4. Die Gleitpunktauswertung hochgradiger Polynome kann numerisch instabil sein:

```
>> exact := bernoulli(50, 1 + I): float(exact);
```

$$2.00833278e27 + 25.0 I$$

```
>> bernoulli(50, float(1 + I))
```

```

2.00833278e27 + 437450444.9 I
>> DIGITS := 40: bernoulli(50, float(1 + I))

2008332779584201020805748320.075757575758 + 25.0000000000000000\
00000435528380270361207 I

>> delete exact, DIGITS:

```

Hintergründe:

⌘ Literatur: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

Änderungen:

⌘ Weitere Vereinfachungen für symbolisches n und spezielle numerische Werte von x wurden implementiert. Die Effizienz wurde verbessert.

besselI, besselJ, besselK, bessely – die Bessel-Funktionen

`besselJ(v, z)` und `besselI(v, z)` repräsentieren die Bessel- und modifizierten Bessel-Funktionen

$$J_v(z) = \frac{(z/2)^v}{\sqrt{\pi} \Gamma(v + 1/2)} \int_0^\pi \cos(z \cos(t)) \sin(t)^{2v} dt ,$$

$$I_v(z) = \frac{(z/2)^v}{\sqrt{\pi} \Gamma(v + 1/2)} \int_0^\pi \exp(z \cos(t)) \sin(t)^{2v} dt ,$$

der ersten Art.

`bessely(v, z)` and `besselK(v, z)` repräsentieren die Bessel- und modifizierten Bessel-Funktionen

$$Y_v(z) = \frac{J_v(z) \cos(v\pi) - J_{-v}(z)}{\sin(v\pi)} , \quad K_v(z) = \frac{\pi}{2} \frac{I_{-v}(z) - I_v(z)}{\sin(v\pi)}$$

der zweiten Art.

Aufruf(e):

⌘ `besselI(v, z)`
⌘ `besselJ(v, z)`
⌘ `besselK(v, z)`
⌘ `bessely(v, z)`

Parameter:

v, z — arithmetische Ausdrücke

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: z

Seiteneffekte: Für Gleitpunktargumente reagieren diese Funktionen auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Details:

- ⌘ Die Bessel-Funktionen sind für komplexe Argumente v und z definiert.
- ⌘ Ein Gleitpunktwert wird berechnet, wenn beide Argumente numerisch sind und mindestens ein Argument eine Gleitpunktzahl ist. Für die meisten exakten Argumente liefern die Bessel-Funktionen unevaluierte symbolische Aufrufe zurück. Spezielle Werte sind für $v = 0$ bzw. $z = 0$ implementiert. Für halbzahlige Indizes werden explizite symbolische Darstellungen der Bessel-Funktionen zurückgeliefert. Siehe Beispiel ??.
- ⌘ Für nicht-negative Indizes v haben einige der Bessel-Funktionen einen Verzweigungsschnitt längs der negativen reellen Halbachse. Beim Überschreiten des Schnitts springen die Funktionswerte. Siehe Beispiel ??.
- ⌘ Soll eine Gleitpunktnäherung berechnet werden, so ist ein Aufruf der Form `besselJ(v, float(x))` dem Aufruf `float(besselJ(v, x))` vorzuziehen. Man beachte, das speziell für halbzahlige Indizes v das symbolische Zwischenergebnis `besselJ(v, x)` kostspielig zu berechnen ist und nicht immer numerisch stabil ausgewertet werden kann. Siehe Beispiel ??.

Beispiel 1. Für exakte und symbolische Argumente werden unevaluierte Aufrufe zurückgeliefert:

```
>> besselJ(2, 1 + I), besselK(0, x), besselY(v, x)
      besselJ(2, 1 + I), besselK(0, x), besselY(v, x)
```

Für Gleitpunktargumente werden Gleitpunktwerte berechnet:

```
>> besselI(2, 5.0), besselK(3.2 + I, 10000.0)
      17.50561497, 1.423757712e-4345 + 4.555796986e-4349 I
```


Beispiel 2. Besselfunktionen, deren Index ein ungerades ganzzahliges Vielfaches von $1/2$ ist, können elementar dargestellt werden:

```
>> besselJ(1/2, x), besselY(3/2, x)
```

$$\frac{\sin(x)}{x}, \frac{\cos(x)}{x^2} - \frac{\sin(x)}{x^3}$$

```
>> besselI(7/2, x), besselK(-7/2, x)
```

$$\frac{(15x^2 + 6x^3 + x^4 + 15) \cosh(x) - \sinh(x) \exp(-x)}{x^3}$$

Beispiel 3. Die negative reelle Halbachse ist ein Verzweigungsschnitt einiger Bessel-Funktionen. Beim Überschreiten des Schnitts springen die Funktionswerte:

```
>> besselI(-3/4, -1.2), besselI(-3/4, -1.2 + I/10^10),
    besselI(-3/4, -1.2 - I/10^10)
- 0.7606149199 - 0.7606149199 I,
- 0.76061492 - 0.7606149199 I, - 0.76061492 + 0.7606149199 I
```

Beispiel 4. Die von Bessel-Funktionen mit halbzahligem Index gelieferten exakten Ausdrücke können für Gleitpunktapproximationen ungeeignet sein:

```
>> y := besselJ(51/2, PI)
```

$$\frac{1}{2} - \frac{450675225}{4\pi^4} + \frac{52650}{2\pi^2} - \frac{1466947857375}{6\pi^6} + \dots + 1$$

Hintergründe:

☞ Die Bessel-Funktionen sind holomorphe Funktionen von z in der längs der negativen reellen Achse aufgeschnittenen z -Ebene. Für $z \neq 0$ sind sie ganze Funktionen in v .

☞ $J_v(z)$ und $Y_v(z)$ erfüllen die Besselsche Differentialgleichung in $w(v, z)$:

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - v^2) w = 0 .$$

$I_v(z)$ und $K_v(z)$ erfüllen die modifizierte Besselsche Differentialgleichung

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} - (z^2 + v^2) w = 0 .$$

☞ Für ganzzahlige Indizes v gelten die Reflektionsregeln:

$$I_{-v}(z) = I_v(z) , \quad J_{-v}(z) = (-1)^v J_v(z) ,$$

$$K_{-v}(z) = K_v(z) , \quad Y_{-v}(z) = (-1)^v Y_v(z) .$$

Änderungen:

☞ `besselI` und `besselJ` sind neue Funktion, `besselJ` und `besselY` wurden neu implementiert und verbessert.

☞ Literatur: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

beta – die Betafunktion

`beta(x, y)` stellt die Betafunktion $\Gamma(x)\Gamma(y)/\Gamma(x+y)$ dar.

Aufruf(e):

☞ `beta(x, y)`

Parameter:

`x`, `y` — arithmetische Ausdrücke

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `gamma`, `psi`

Details:

- ⌘ Die Betafunktion ist für komplexe Argumente x und y definiert.
 - ⌘ Das Ergebnis wird durch Aufrufe der Gammafunktion ausgedrückt, wenn beide Argumente vom Typ `Type::Numeric` sind. Man beachte, dass die Betafunktion reguläre Werte annehmen kann, auch wenn $\Gamma(x)$ oder $\Gamma(y)$ und $\Gamma(x+y)$ singulär sind. In solchen Fällen liefert `beta` den Grenzwert der Quotienten der singulären `gamma`-Ausdrücke zurück.
 - ⌘ Ein Gleitpunktwert wird berechnet, wenn beide Argumente numerisch sind und mindestens ein Argument eine Gleitpunktzahl ist.
 - ⌘ Verschwindet keines der Argumente und ist mindestens eines nicht vom Typ `Type::Numeric`, so wird ein unevaluierter Aufruf von `beta` zurückgeliefert.
-

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> beta(1, 5), beta(I, 3/2), beta(1, y + 1), beta(x, y)
```

$$\frac{1}{5}, \frac{\pi \Gamma(I)}{2 \Gamma(3/2 + I)}, \frac{1}{y + 1}, \text{beta}(x, y)$$

Für Gleitkommazahlen wird der numerische Wert von `beta` berechnet:

```
>> beta(3.5, sqrt(2)), beta(sqrt(2), 2.0 + 10.0*I)
```

$$0.1395855454, -0.01112350756 - 0.03108193098 I$$

Beispiel 2. Die Gammafunktion ist für nicht-positive ganzzahlige Argumente singulär. Trotzdem hat `beta` einen regulären Wert für die folgenden Argumente:

```
>> beta(-3, 2)
```

$1/6$

Beispiel 3. Die Funktionen `diff`, `expand` und `float` verarbeiten `beta`:

```
>> diff(beta(x^2, x), x)

          2          2          2
    beta(x, x ) (psi(x) + 2 x psi(x ) - psi(x + x ) (2 x + 1))

>> expand(beta(x - 1, y + 1))

          y gamma(x) gamma(y)
          -----
        gamma(x + y) (x - 1)

>> float(beta(100, 1000))

          7.730325902e-147
```

Änderungen:

☞ Das `expand`-Attribut drückt `beta` durch `gamma` aus und expandiert das Resultat.

`binomial` – Binomialkoeffizienten

`binomial(n, k)` stellt den Binomialkoeffizienten $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ dar.

Aufruf(e):

☞ `binomial(n, k)`

Parameter:

`n`, `k` — arithmetische Ausdrücke

Rückgabewert: ein arithmetischer Ausdruck.

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `fact`, `gamma`

Details:

- ⌘ Binomialkoeffizienten sind für komplexe Argumente mittels der gamma-Funktion definiert:

$$\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}.$$

Mit $\Gamma(n+1) = n!$ stimmt dies für ganzzahlige Argumente $0 \leq k \leq n$ mit der üblichen Definition der Binomialkoeffizienten überein.

- ⌘ Ein symbolischer Funktionsaufruf wird zurückgeliefert, falls eines der beiden Argumente nicht zu einer Zahl vom Typ `Type::Numeric` evaluiert werden kann. Für $k = 0$, $k = 1$, $k = n - 1$ und $k = n$ wird jedoch für beliebiges n ein vereinfachtes Resultat geliefert.
- ⌘ Sei n ein numerischer Wert vom Typ `Type::Numerical`. Für nicht-negatives ganzzahliges k wird $n \times (n-1) \times \cdots \times (n-k+1)/k!$ zurückgeliefert. Für negatives ganzzahliges k wird 0 zurückgeliefert. Für Gleitpunktzahlen k wird ein Gleitpunktwert berechnet. In allen anderen Fällen wird ein symbolischer Funktionsaufruf von `binomial` zurückgeliefert.
- ⌘ Ein Gleitpunktwert wird berechnet, wenn beide Argumente numerisch sind und mindestens ein Argument eine Gleitpunktzahl ist.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> binomial(10, k) $ k=-2..12
      0, 0, 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1, 0, 0
>> binomial(-23/12, 3), binomial(1 + I, 3)
      -37835/10368, - 1/3 I
>> binomial(n, k), binomial(n, 1), binomial(n, 4)
      binomial(n, k), n, binomial(n, 4)
```

Für Gleitkommazahlen wird eine Gleitpunktnäherung von `binomial` berechnet:

```
>> binomial(-235/123, 3.0), binomial(3.0, 1 + I)
      -3.624343742, 4.411293493 + 2.205646746 I
```

Beispiel 2. Die Funktion `expand` verarbeitet `binomial`:

```
>> binomial(n, 3) = expand(binomial(n, 3))
```

$$\text{binomial}(n, 3) = \frac{n^2}{3} - \frac{n}{2} + \frac{n}{6}$$

```
>> binomial(2, k) = expand(binomial(2, k))
```

$$\text{binomial}(2, k) = - \frac{2}{k \, \Gamma(k) \, \Gamma(-k) \, (1 - k) \, (2 - k)}$$

Das `float`-Attribut verarbeitet `binomial`, wenn alle Argumente zu Gleitpunktzahlen konvertiert werden können:

```
>> binomial(sin(3), 5/4), float(binomial(sin(3), 5/4))
      binomial(sin(3), 5/4), -0.08360571366
```

Änderungen:

- ⌘ Das `expand`-Attribut drückt das Ergebnis nun durch `gamma` und nicht mehr durch `fact` aus.

`bool` – boolsche Auswertung

`bool(b)` evaluiert den boolschen Ausdruck `b`.

Aufruf(e):

- ⌘ `bool(b)`

Parameter:

`b` — ein boolscher Ausdruck

Rückgabewert: `TRUE`, `FALSE` oder `UNKNOWN`.

Überladbar durch: `b`

Verwandte Funktionen: `_lazy_and`, `_lazy_or`, `FALSE`, `if`, `is`, `repeat`, `TRUE`, `UNKNOWN`, `while`

Details:

- Die Funktion `bool` dient dazu, boolesche Ausdrücke zu einer der logischen Konstanten `TRUE`, `FALSE` oder `UNKNOWN` zu vereinfachen.

Boolesche Ausdrücke sind Ausdrücke, die aus Gleichungen, Ungleichungen sowie den obigen Konstanten mittels der logischen Verknüpfungsooperatoren `and`, `or`, `not` aufgebaut sind.

Die Funktion `bool` evaluiert alle Gleichungen und Ungleichungen innerhalb eines booleschen Ausdrucks zu `TRUE` oder `FALSE`. Die resultierende logische Kombination dieser Konstanten wird gemäß der Regeln der dreiwertigen Logik MuPADs vereinfacht (siehe `and`, `or`, `not`).

- Gleichungen $x = y$ und Ungleichungen $x <> y$ werden von `bool` rein *syntaktisch* ausgewertet. Es wird nicht überprüft, ob die zu vergleichenden Objekte mathematisch äquivalent sind.



- Ungleichungen $x < y$, $x \leq y$ etc. können von `bool` nur dann evaluiert werden, wenn x und y reelle Zahlen vom Typ `Type::Real` sind. Ist dies nicht der Fall, wird ein Fehler ausgelöst.



- `bool` evaluiert *alle* Teilausdrücke eines booleschen Ausdrucks, bevor das Resultat vereinfacht wird. Die Funktionen `_lazy_and`, `_lazy_or` bieten eine alternative Auswertungsstrategie: „lazy evaluation“.

- Im Bedingungsanteil von `if`-, `repeat`- und `while`-Anweisungen braucht `bool` nicht explizit benutzt werden. Diese Anweisungen erzwingen intern automatisch boolesche Auswertung von Gleichungen und Ungleichungen mittels `_lazy_and` und `_lazy_or`. Siehe Beispiel ??.

- Über die Funktion `simplify` mit der Option `logic` können boolesche Ausdrücke mit symbolischen Objekten vereinfacht werden. Siehe Beispiel ??.

- `bool` ist eine Funktion des Systemkerns.

Beispiel 1. Auch in MuPAD ist 1 kleiner als 2:

```
>> 1 < 2 = bool(1 < 2)
```

```
(1 < 2) = TRUE
```

Man beachte in der booleschen Auswertung von Ungleichungen, dass `bool` nur reelle Zahlen vom syntaktischen Typ `Type::Real` vergleichen kann:

```
>> bool(PI < 2 + sqrt(2))
```



```
Error: Can't evaluate to boolean [_less]
```

Man kann Gleitpunktapproximationen vergleichen oder alternativ `is` verwenden:

```
>> bool(float(PI) < float(2 + sqrt(2))), is(PI < 2 + sqrt(2))

TRUE, TRUE
```

Beispiel 2. Die boolschen Verknüpfungsoperatoren `and`, `or`, `not` werten Gleichungen und Ungleichungen nicht logisch aus, sondern liefern einen symbolischen boolschen Ausdruck. Die boolsche Auswertung und Vereinfachung wird durch `bool` erzwungen:

```
>> a = a and 3 < 4

a = a and 3 < 4

>> bool(a = a and 3 < 4)

TRUE
```

Beispiel 3. `bool` verarbeitet die spezielle boolsche Konstante `UNKNOWN`:

```
>> bool(UNKNOWN and 1 < 2), bool(UNKNOWN or 1 < 2),
    bool(UNKNOWN and 1 > 2), bool(UNKNOWN or 1 > 2)

UNKNOWN, TRUE, FALSE, UNKNOWN
```

Beispiel 4. `bool` muss alle Teile eines zusammengesetzten boolschen Ausdrucks zu einer der boolschen Konstanten reduzieren können. Dementsprechend dürfen keine symbolischen boolschen Teilausdrücke vorkommen:

```
>> b := b1 and b2 or b3: bool(b)

Error: Can't evaluate to boolean [bool]

>> b1 := 1 < 2: b2 := x = x: b3 := FALSE: bool(b)

TRUE

>> delete b, b1, b2, b3:
```

Beispiel 5. Im Bedingungsteil von `if`-, `repeat`- und `while`-Anweisungen braucht `bool` nicht explizit benutzt zu werden. Diese Anweisungen erzwingen intern automatisch boolsche Auswertung von Gleichungen und Ungleichungen mittels `_lazy_and` und `_lazy_or`:

```
>> x := 0: if x <> 0 and sin(1/x) = 0 then 1 else 2 end
2
```

Im Gegensatz dazu evaluiert `bool` *alle* Bedingungen. Dementsprechend wird bei der Auswertung von `sin(1/x) = 0` durch Null geteilt:

```
>> bool(x <> 0 and sin(1/x) = 0)
Error: Division by zero
>> delete x:
```

Beispiel 6. Ausdrücke mit symbolischen boolschen Teilausdrücken können nicht von `bool` verarbeitet werden. Man kann jedoch `simplify` mit der Option `logic` zur Vereinfachung verwenden:

```
>> (b1 and b2) or (b1 and (not b2)) and (1 < 2)
b1 and b2 or b1 and not b2 and 1 < 2
>> simplify(%, logic)
b1
```

Änderungen:

⌘ Keine Änderungen.

`break` – vorzeitiges Beenden von Schleifen oder `case`-Anweisungen

`break` beendet vorzeitig `for`-, `repeat`-, und `while`-Schleifen sowie `case`-Anweisungen.

Aufruf(e):

⌘ `break`
⌘ `_break()`

Verwandte Funktionen: case, for, next, quit, repeat, return, while

Details:

- ⌘ Die break-Anweisung ist äquivalent zum Funktionsaufruf `_break()`. Der Rückgabewert ist das leere Objekt vom Typ `DOM_NULL`.
 - ⌘ Innerhalb von `for`-, `repeat`-, `while`-Schleifen und `case`-Verzweigungen führt die `break`-Anweisung zum sofortigen Abbruch. Die dem entsprechenden `end` folgende Anweisung wird als nächstes ausgeführt.
 - ⌘ In verschachtelten Schleifen wird durch `break` nur die innerste Schleife abgebrochen.
 - ⌘ `break` bricht auch eine Anweisungsfolge `_stmtseq(..., break, ...)` ab.
 - ⌘ Außerhalb von `for`-, `repeat`-, `while`-, `case`- und `_stmtseq`-Anweisungen hat `break` keinerlei Effekt.
 - ⌘ `_break` ist eine Funktion des Systemkerns.
-

Beispiel 1. Schleifen werden durch `break` vorzeitig abgebrochen:

```
>> for i from 1 to 10 do
    print(i);
    if i = 2 then break end_if
end_for
```

1

2

```
>> delete i:
```

Beispiel 2. In einer `case`-Anweisung werden alle Anweisungen startend mit dem ersten passenden Zweig durchgeführt:

```
>> x := 2:
case x
  of 1 do print(1); x^2;
  of 2 do print(2); x^2;
  of 3 do print(3); x^2;
  otherwise print(UNKNOWN)
end_case:
```

2

3

UNKNOWN

In der folgenden Version sorgt `break` dafür, daß nur die Anweisungen des zutreffenden Zweiges durchgeführt werden:

```
>> case x
  of 1 do print(1); x^2; break;
  of 2 do print(2); x^2; break;
  of 3 do print(3); x^2; break;
  otherwise print(UNKNOWN)
end_case:
```

2

```
>> delete x:
```

Änderungen:

⌘ Keine Änderungen.

builtin – Repräsentanten von C-Funktionen des Systemkerns

`builtin` repräsentiert eine C-Funktion des Systemkerns.

Aufruf(e):

⌘ `builtin(i, j, str, tbl)`
⌘ `builtin(i, j1, str1, str)`


Parameter:

`i` — die Nummer einer C-Funktion des Kerns: eine nichtnegative ganze Zahl
`j` — die Nummer einer C-Funktion des Kerns: eine nichtnegative ganze Zahl oder `NIL`
`str` — der Name des erzeugten `DOM_EXEC`-Objekts: eine Zeichenkette
`tbl` — die Remember-Tafel der Funktion: eine Tabelle oder `NIL`
`j1` — die Priorität eines Operators: eine nichtnegative ganze Zahl
`str1` — das Operator-Symbol: eine Zeichenkette oder `NIL`

Rückgabewert: ein Objekt vom Typ `DOM_EXEC`.

Verwandte Funktionen: `funcenv`

Details:

- ⌘ `builtin` ist nur für den internen Gebrauch gedacht! Ein Anwender sollte diese systemnahe Funktion nicht aufrufen. 
 - ⌘ Die Funktion `builtin` stellt die Schnittstelle zwischen der MuPAD-Sprache und den C-Funktionen des Systemkerns dar. Die von `builtin` gelieferten MuPAD-Funktionen sind Objekte vom Typ `DOM_EXEC`. Sie dürfen nur als erster oder zweiter Eintrag von mittels `funcenv` erzeugten Funktionsumgebungen verwendet werden.
 - ⌘ Funktionen, die als erstes Argument in `funcenv` verwendet werden, dienen zur Auswertung von Funktionsaufrufen der Funktionsumgebung. Eine Kernfunktion, die zu diesem Zweck eingesetzt werden soll, muss durch einen Aufruf der Form `builtin(i, j, str, tbl)` erzeugt werden. Die Zeichenkette `str` wird für die Ausgabe symbolischer Aufrufe der Funktion verwendet. Die Tabelle `tbl` definiert die Remember-Tafel der Funktion. Siehe Beispiel ?? . Bei Übergabe von `NIL` wird der Funktion keine Remember-Tafel zugeordnet.
 - ⌘ Funktionen, die als zweites Argument in `funcenv` verwendet werden, bestimmen die Ausgabe symbolischer Funktionsaufrufe. Eine Kernfunktion, die zu diesem Zweck eingesetzt werden soll, muss durch einen Aufruf der Form `builtin(i, j1, str1, str)` erzeugt werden. Soll der symbolische Funktionsaufruf in Operatornotation ausgegeben werden, wird die Zeichenkette `str1` als Operatorzeichen verwendet. Siehe Beispiel ?? . `NIL` muss übergeben werden, wenn die Funktion nicht als Operator dargestellt werden soll. Die Zeichenkette `str` wird für die Ausgabe des `DOM_EXEC`-Objekts selbst verwendet.
 - ⌘ `builtin` ist eine Funktion des Systemkerns.
-

Beispiel 1. Mit `expose` können die Operanden einer Funktionsumgebung wie z. B. `_mult` angezeigt werden. Die beiden folgenden Kernfunktionen sind zuständig für die Auswertung von Produkten bzw. für die Bildschirmdarstellung des Resultats:

```
>> expose(op(_mult, 1)), expose(op(_mult, 2))

builtin(815, NIL, "_mult", NIL),

    builtin(1100, 15, "*", "_mult")

>> _mult(a, b) = builtin(815, NIL, "_mult", NIL)(a, b)
```

```
a b = a b
```

Beispiel 2. Wir demonstrieren, wie die Remember-Tafel von Kernfunktionen manipuliert werden kann. Die Funktionsumgebung `isprime` benutzt eine C-Funktion des Kerns zur Auswertung ihres Arguments:

```
>> expose(isprime)

      builtin(1000, 1305, "isprime", NIL)
```

Sie sieht 1 (korrekterweise) nicht als Primzahl an:

```
>> isprime(1)

      FALSE
```

Der Schutz der Systemfunktion wird mittels `unprotect` aufgehoben, dann wird die Remember-Tafel aufgefüllt, indem `isprime(1)` der Wert `TRUE` zugewiesen wird:

```
>> unprotect(isprime): isprime(1) := TRUE:
```

Der Wert wird in der Remember-Tafel abgelegt. Dies ist der vierte Eintrag der Evaluierungsfunktion von `isprime`:

```
>> expose(isprime)

      /
      builtin| 1000, 1305, "isprime",  table(  \
      \      1 = TRUE  |
      \      )      /
```

Nach dieser Änderung sieht `isprime` die Zahl 1 als Primzahl an:

```
>> isprime(1)

      TRUE
```

Das ursprüngliche Verhalten von `isprime` wird wiederhergestellt, indem die Remember-Tafel durch den ursprünglichen Wert `NIL` ersetzt wird:

```
>> isprime := subsop(isprime, [1, 4] = NIL): protect(isprime):

>> isprime(1)

      FALSE
```

Beispiel 3. Es wird demonstriert, wie das Ausgabesymbol der Kern-Funktion `_power` geändert werden kann. Diese Funktion ist dafür zuständig, Potenzen darzustellen:

```
>> op(a^b, 0), _power(a, b)

                                     b
                                _power, a
```

Der zweite Operand der Funktionsumgebung `_power` ist die Kernfunktion, welche für die Bildschirmausgabe zuständig ist:

```
>> expose(op(_power, 2))

      builtin(1100, 16, "^", "_power")
```

Der dritte Operand ist das Symbol, das für die Ausgabe symbolischer Potenzen benutzt wird. Es soll durch `**` ersetzt werden. Vor der Änderung muss jedoch der Schutz der Systemfunktion `_power` aufgehoben werden:

```
>> unprotect(_power): _power := subsop(_power, [2, 3] = "***"):
>> expose(op(_power, 2)), a^b

      builtin(1100, 16, "***", "_power"), a**b
```

Das ursprüngliche Verhalten von `_power` wird wiederhergestellt:

```
>> _power := subsop(_power, [2, 3] = "^"): protect(_power):
```

Änderungen:

⌘ `builtin` hieß früher `built_in`.

bytes – der Speicherverbrauch der MuPAD-Sitzung

`bytes()` liefert den momentan belegten Speicherplatz.

Aufruf(e):

⌘ `bytes()`

Rückgabewert: eine Folge aus drei ganzen Zahlen.

Verwandte Funktionen: `rtime`, `share`, `time`

Details:

☞ `bytes` gibt die drei folgenden Zahlen zurück:

- Der logisch benutzte Speicherplatz in Bytes; dies ist der Speicher, der wirklich aktuell durch MuPAD-Daten belegt ist.
- Der durch die Speicherverwaltung physikalisch belegte Speicherplatz in Bytes; dies ist der Speicher, den MuPAD vom Betriebssystem angefordert hat. Die Differenz zwischen physikalischen und logischen Bytes ist der Speicher, der intern bereits für zukünftige Berechnungen reserviert ist.
- Bei Computern mit virtueller Speicherverwaltung ist die dritte Zahl der konstante Wert $2^{31} - 1$. Auf anderen Rechnern wie z. B. dem Apple Macintosh wird der verbleibende freie Speicher (auf dem *Programm-Heap*) angezeigt.

☞ `bytes` ist eine Funktion des Systemkerns.

Beispiel 1. In einer gerade gestarteten MuPAD-Sitzung könnte `bytes` folgende Werte für den Speicherverbrauch der Sitzung liefern:

```
>> bytes()
```

```
506584, 717312, 2147483647
```

Jede Rechnung erhöht den Speicherverbrauch:

```
>> int(x, x): bytes()
```

```
2040956, 2201624, 2147483647
```

Änderungen:

☞ Keine Änderungen.

case – Auswahlanweisung

`case-end_case` erlaubt innerhalb eines Programms die Auswahl verschiedener Äste einer Verzweigung.

Aufruf(e):

```

⇨ case x
    of match1 do statements1
    of match2 do statements2
    ...
    <otherwise otherstatements>
end_case
⇨ _case(x, match1, statements1, match2, statements2,
    ...
    <, otherstatements>)

```

Parameter:

x, match1, match2, ...	— beliebige MuPAD-Objekte
statements1, ..., otherstatements	— beliebige Anweisungsfolgen

Rückgabewert: das Ergebnis des letzten innerhalb der `case`-Anweisung ausgeführten Kommandos. Das Ergebnis ist das leere Objekt vom Typ `DOM_NULL`, falls kein passender Zweig gefunden wurde und kein `otherwise`-Zweig existiert. `NIL` wird zurückgeliefert, wenn zwar ein passender Zweig gefunden wurde, darin aber kein Kommando ausgeführt wurde.

Verwandte Funktionen: `break`, `if`, `return`

Details:

- ⇨ Die `case`-Anweisung ist eine Kontrollstruktur, welche die Funktionalität der `if`-Anweisung erweitert. In einer `case`-Anweisung wird ein Objekt mit mehreren möglichen Werten verglichen, und eine oder mehrere Anweisungsfolgen werden ausgeführt.
- ⇨ Wenn der Wert von `x` gleich einem der Werte `match1`, `match2` etc. ist, so wird der erste entsprechende Zweig *und alle darauf folgenden Zweige (inclusive otherwise)* ausgeführt, bis die Ausführung durch eine `break`- oder `return`-Anweisung bzw. das `end_case` beendet wird.
- ⇨ Wenn der Wert von `x` mit keinem der Werte `match1`, `match2`, ... übereinstimmt, so wird lediglich der `otherwise`-Zweig ausgeführt. Existiert kein `otherwise`-Zweig, so terminiert die `case`-Anweisung und liefert das leere Objekt vom Typ `DOM_NULL` zurück.
- ⇨ Statt des Schlüsselworts `end_case` kann auch das Schlüsselwort `end` benutzt werden.
- ⇨ `_case` ist eine Funktion des Systemkerns.

Beispiel 1. Alle Zweige nach dem ersten passenden Zweig werden ausgeführt:

```
>> x := 2:
case x
  of 1 do print(1)
  of 2 do print(4)
  of 3 do print(9)
  otherwise print("otherwise")
end_case:

4

9

"otherwise"
```

Mittels break kann dafür gesorgt werden, daß nur jeweils ein Zweig durchlaufen wird:

```
>> case x
  of 1 do print(1); 1; break
  of 2 do print(4); 4; break
  of 3 do print(9); 9; break
  otherwise print("otherwise")
end_case:

4
```

```
>> delete x:
```

Beispiel 2. Die Funktionalität der case-Anweisung erlaubt die gemeinsame Benutzung von Code-Stücken durch mehrere Zweige. Die folgende Funktion benutzt die Anweisung `print(x, "is a real number")` für alle Zweige, die den drei reellen Zahlentypen MuPADs entsprechen:

```
>> isReal := proc(x)
begin
  case domtype(x)
  of DOM_INT do
  of DOM_RAT do
  of DOM_FLOAT do print(x, "is a real number"); break
  of DOM_COMPLEX do print(x, "is not a real number"); break
  otherwise print(x, "cannot decide");
  end_case
end_proc:
isReal(3), isReal(3/7), isReal(1.23), isReal(3 + I), isReal(z)
```

```

3, "is a real number"

3/7, "is a real number"

1.23, "is a real number"

3 + I, "is not a real number"

z, "cannot decide"

>> delete isReal:

```

Beispiel 3. Der Zusammenhang zwischen der funktionalen und der imperativen Form der case-Anweisung wird demonstriert:

```

>> hold(_case(x, match1, (1; break), match2, (4; break),
              print("otherwise")))

case x
  of match1 do
    1;
    break
  of match2 do
    4;
    break
  otherwise
    print("otherwise")
end_case

>> hold(_case(x, match1, (1; break), match2, (4; break)))

case x
  of match1 do
    1;
    break
  of match2 do
    4;
    break
end_case

```

Hintergründe:

- ⌘ Die Funktionalität der case-Anweisung entspricht der switch-Anweisung der Programmiersprache C.

Änderungen:

☞ `end` kann als Alternative zu `end_case` benutzt werden.

`ceil`, `floor`, `round`, `trunc` – Runden auf eine ganze Zahl

`ceil` rundet auf.

`floor` rundet ab.

`round` rundet auf die nächstgelegene ganze Zahl.

`trunc` rundet auf die nächste ganze Zahl in Richtung 0.

Aufruf(e):

☞ `ceil(x)`

☞ `floor(x)`

☞ `round(x)`

☞ `trunc(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Seiteneffekte: Die Funktionen reagieren auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `frac`

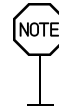
Details:

☞ Für komplexe Argumente wird die Rundung getrennt auf Real- und Imaginärteil angewendet.

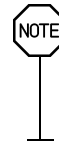
☞ Für reelle Zahlen und exakte Ausdrücke, die reelle Zahlen darstellen, werden ganze Zahlen zurückgeliefert. Für Argumente, die symbolische Bezeichner enthalten, werden unevaluierte Funktionsaufrufe zurückgeliefert.

☞ `trunc` schneidet die Stellen nach dem Dezimalpunkt ab. Dementsprechend stimmt `trunc` für positive reelle Argumente mit `floor`, für negative reelle Argumente mit `ceil` überein.

☞ Für Gleitpunktzahlen mit einem Betrag größer als 10^{DIGITS} wird der gerundete Wert von den internen nicht-signifikanten Stellen mitbestimmt! Siehe Beispiel ??.



☞ Intern werden exakte numerische Ausdrücke durch Gleitpunkt-
werte approximiert, bevor sie gerundet werden. Dementspre-
chend kann das Ergebnis vom momentanen Wert von DIGITS ab-
hängen! Siehe Beispiel ??.



Beispiel 1. Die Rundung einiger reeller und komplexer Zahlen:

```
>> ceil(3.5), floor(3.5), round(3.5), trunc(3.5)
      4, 3, 4, 3
>> ceil(-7/2), floor(-7/2), round(-7/2), trunc(-7/2)
      -3, -4, -3, -3
>> ceil(3 + 5/2*I), floor(4.3 + 7*I), round(I/2), trunc(I/2)
      3 + 3 I, 4 + 7 I, I, 0
```

Auch symbolische Ausdrücke können gerundet werden, wenn sie Zahlen dar-
stellen:

```
>> x := PI*I + 7*sin(exp(2)): ceil(x), floor(x), round(x), trunc(x)
      7 + 4 I, 6 + 3 I, 6 + 3 I, 6 + 3 I
```

Die Rundung von Ausdrücken mit symbolischen Bezeichnern führt zu uneva-
luierter Rückgabe:

```
>> delete x: ceil(x), floor(x - 1), round(x + 1), trunc(x^2 + 3)
      2
      ceil(x), floor(x - 1), round(x + 1), trunc(x + 3)
```

Beispiel 2. Die Rundung von betragsmäßig großen Gleitkommazahlen kann
problematisch sein:

```
>> x := 10^30/3.0
      3.333333333e29
```

Man beachte, dass nur die ersten 10 Dezimalstellen „signifikant“ sind, die wei-
teren Stellen werden durch die interne Binärdarstellung erzeugt. Diese „unsi-
gnifikanten“ Stellen fließen in die Rundung ein:

```
>> floor(x), ceil(x)
      333333333333333333307205615616, 3333333333333333333307205615616
>> delete x:
```

```
>> x := 10^30 - exp(30)^ln(10)

                                     ln(10)
      100000000000000000000000000000000 - exp(30)
```

```
>> DIGITS := 10: float(x), floor(x), ceil(x)
1.030792151e13, 10307921510400, 10307921510400
```

```
>> DIGITS := 20: float(x), floor(x), ceil(x)
                        2896.0, 2896, 2896
```

```
>> DIGITS := 30: float(x), floor(x), ceil(x)
```

```
                0.00000087916851043701171875, 0, 1
```

Keine Änderungen.

```

coef(p)
coef(p, <x,> n)
coef(f <, vars>)
coef(f, <vars,> <x,> n)

```

Parameter:

- p — ein Polynom vom Typ `DOM_POLY`
 x — eine Unbestimmte
 n — die Potenz: eine nichtnegative ganze Zahl
 f — ein polynomialer Ausdruck
 $vars$ — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: ein oder mehrere Koeffizienten des Koeffizientenrings des Polynoms, oder ein Polynom oder `FAIL`.

Überladbar durch: p , f

Verwandte Funktionen: `collect`, `content`, `degree`, `degreevec`, `ground`, `icontent`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- ⌘ Wenn das erste Argument f nicht Element eines Polynom-Domains ist, so konvertiert `coeff` diesen Ausdruck intern mittels `poly(f)` in ein Polynom vom Typ `DOM_POLY`. Ist eine Liste von Unbestimmten angegeben, so wird `poly(f, vars)` betrachtet.

Koeffizienten polynomialer Ausdrücke f werden als arithmetische Ausdrücke zurückgeliefert.

- ⌘ Es gibt verschiedene Formen, `coeff` mit einem Polynom p vom Typ `DOM_POLY` aufzurufen:

- `coeff(p)` gibt eine Folge aller nicht-trivialen Koeffizienten von p in der lexikographischen Ordnung der Terme aus.
Die zurückgelieferten Koeffizienten sind Elemente des Koeffizientenrings von p .
- `coeff(p, x, n)` fasst p als univariates Polynom in der Variablen x auf und liefert den Koeffizienten des Terms x^n .
Ist p ein univariates Polynom in x , so werden Koeffizienten als Elemente des Koeffizientenrings von x zurückgeliefert.
Ist p ein multivariates Polynom, so werden Koeffizienten als Polynome vom Typ `DOM_POLY` in den „verbleibenden“ Variablen zurückgeliefert.
- `coeff(p, n)` entspricht `coeff(p, x, n)`, wobei x die „Hauptvariable“ von p ist. Dies ist das erste Element der durch `op(p, 2)` gegebenen Variablenliste.

- ⌘ `coeff` liefert 0 oder ein Null-Polynom, falls das Eingabepolynom keinen der eingegebenen Potenz n entsprechenden Term enthält. Dies geschieht insbesondere, wenn n größer ist als der Polynomgrad.

- ☞ `coeff` liefert FAIL, wenn ein Ausdruck nicht als Polynom interpretiert werden kann.
 - ☞ Das Ergebnis von `coeff` wird nicht weiter evaluiert. Vollständige Evaluation kann mit der Funktion `eval` erzwungen werden. Siehe Beispiel ??.
 - ☞ `coeff` ist eine Funktion des Systemkerns.
-

Beispiel 1. `coeff(f)` liefert die Folge aller nicht-trivialen Koeffizienten:

```
>> f := 10*x^10 + 5*x^5 + 2*x^2: coeff(f)
```

10, 5, 2

`coeff(f, i)` liefert einen einzelnen Koeffizienten:

```
>> coeff(f, i) $ i = 0..15
```

0, 0, 2, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0

```
>> delete f:
```

Beispiel 2. Es wird gezeigt, wie die Unbestimmten das Ergebnis beeinflussen:

```
>> f := 3*x^3 + x^2*y^2 + 17*x + 23*y + 2
```

$$17x + 23y + 3x^3 + x^2y^2 + 2$$

```
>> coeff(f); coeff(f, [x, y]); coeff(f, [y, x])
```

1, 23, 3, 17, 2

3, 1, 17, 23, 2

1, 23, 3, 17, 2

```
>> delete f:
```

Beispiel 3. Die Koeffizienten von `f` werden bezüglich der Hauptvariablen `x` bestimmt, welche der erste Eintrag in der Liste der Unbestimmten ist:

```
>> f := 3*x^3 + x^2*y^2 + 2: coeff(f, [x, y], i) $ i = 0..3
```

$$2, 0, y^2, 3$$

Die Koeffizienten von f können bezüglich einer anderen Hauptvariablen bestimmt werden, in diesem Fall y :

```
>> coeff(f, [y, x], i) $ i = 0..2
```

$$3x^3 + 2, 0, x^2$$

Alternativ:

```
>> coeff(f, y, i) $ i = 0..2
```

$$3x^3 + 2, 0, x^2$$

```
>> delete f:
```

Beispiel 4. `coeff` kann auf gleiche Weise auf Polynome vom Typ `DOM_POLY` angewendet werden:

```
>> p := poly(3*x^3 + x, [x], Dom::IntegerMod(7)):
coeff(p)
```

$$3 \bmod 7, 1 \bmod 7$$

```
>> coeff(p, i) $ i = 0..3
```

$$0 \bmod 7, 1 \bmod 7, 0 \bmod 7, 3 \bmod 7$$

Für multivariate Polynome sind die Koeffizienten bezüglich einer Unbestimmten Polynome in den anderen Unbestimmten:

```
>> p := poly(3*x^3 + x^2*y^2 + 2, [x, y]):
>> coeff(p, y, 0), coeff(p, y, 1), coeff(p, y, 2);
```

$$\text{poly}(3x^3 + 2, [x]), \text{poly}(0, [x]), \text{poly}(x^2, [x])$$

```
>> coeff(p, x, 0), coeff(p, x, 1), coeff(p, x, 2)
```

$$\text{poly}(2, [y]), \text{poly}(0, [y]), \text{poly}(y^2, [y])$$

Beachten Sie, dass die an `coeff` übergebenen Unbestimmten verwendet werden, auch wenn das Polynom sie gar nicht enthält:

```
>> coeff(p, z, 0), coeff(p, z, 1), coeff(p, z, 2)
```

$$\text{poly}(3x^3 + x^2y^2 + 2, [x, y]), \text{poly}(0, [x, y]),$$

$$\text{poly}(0, [x, y])$$

```
>> delete p:
```

Beispiel 5. `coeff` evaluiert sein Ergebnis nicht vollständig:

```
>> p := poly(27*x^2 + a*x, [x]): a := 5:
      coeff(p, x, 1), eval(coeff(p, x, 1))
                        a, 5

>> delete p, a:
```

Änderungen:

☞ Keine Änderungen.

`coerce` – Typkonvertierung

`coerce(object, T)` versucht, `object` in ein Element des Domains `T` zu konvertieren.

Aufruf(e):

☞ `coerce(object, T)`

Parameter:

`object` — ein beliebiges Objekt
`T` — ein beliebiges Domain

Rückgabewert: ein Objekt des Domains `T` oder der Wert `FAIL`.

Überladbar durch: `T`

Verwandte Funktionen: `domtype`, `expr`, `testtype`, `type`

Details:

☞ `coerce(object, T)` versucht, das Objekt `object` in ein Element des Datentyps `T` zu konvertieren. Sollte das nicht möglich sein oder die Konvertierung nicht implementiert sein, so wird der Wert `FAIL` zurückgeliefert.

☞ Domains implementieren in der Regel die zwei Methoden "`convert`" und "`convert_to`" für Konvertierungen von Objekten.

`coerce` verwendet diese Methoden wie folgt: Zuerst wird die Methode `T::convert(object)` aufgerufen. Liefert diese Methode einen Wert ungleich `FAIL`, so wird das Ergebnis zurückgeliefert. Anderenfalls wird die Methode `object::dom::convert_to(object, T)` aufgerufen und ihr Ergebnis zurückgeliefert, was erneut der Wert `FAIL` sein kann.

- ☞ Die möglichen Konvertierungen für `object` bzw. die Konvertierungen, die das Domain `T` anbietet, sind unter der Methode `"coerce"` bzw. `"convert"` auf der Hilfeseite des Domains `T` und unter der Methode `"convert_to"` auf der Hilfeseite des Domains von `object` dokumentiert.
- ☞ Derzeit implementieren nur wenige Basisdomains die Methoden `"convert"` und `"convert_to"`. In zukünftigen Versionen von MuPAD wird es entsprechende Erweiterungen geben.
- ☞ Die Funktion `expr` konvertiert ein Objekt in ein Element eines Basisdomains.
- ☞ Oft kann die gewünschte Konvertierung auch durch den Aufruf des Konstruktors des Domains `T` erreicht werden. Siehe Beispiel ??.

Beispiel 1. Wir beginnen mit der Konvertierung eines Feldes in eine Liste vom Domain-Typ `DOM_LIST`:

```
>> a := array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

```

+-          +-
|  1, 2, 3  |
|           |
|  4, 5, 6  |
+-          +-

```

```
>> coerce(a, DOM_LIST)
```

```
[1, 2, 3, 4, 5, 6]
```

Die Konvertierung eines Feldes in ein Polynom ist nicht implementiert, daher liefert `coerce` im folgenden den Wert `FAIL`:

```
>> coerce(a, DOM_POLY)
```

```
FAIL
```

Man kann ein- oder zweidimensionale Felder in Matrizen konvertieren, und umgekehrt. Hierzu ein Beispiel:

```
>> A := coerce(a, matrix); domtype(A)
```

```

+-          +-
|  1, 2, 3  |
|           |
|  4, 5, 6  |
+-          +-

```

```
Dom::Matrix()
```

Die Konvertierung einer Matrix in eine Liste ist ebenso möglich. Das Ergebnis ist eine Liste von Listen, wobei die inneren Listen die Zeilen der Matrix repräsentieren:

```
>> coerce(A, DOM_LIST)

[[1, 2, 3], [4, 5, 6]]
```

Man kann Listen in Mengen konvertieren, und umgekehrt. Ein Beispiel:

```
>> coerce([1, 2, 3, 2], DOM_SET)

{1, 2, 3}
```

Jedes MuPAD Objekt kann in eine Zeichenkette umgewandelt werden, wie beispielsweise der arithmetische Ausdruck $2x + \sin(x^2)$:

```
>> coerce(2*x + sin(x^2), DOM_STRING)

"2*x + sin(x^2)"
```

Beispiel 2. Die Funktion `factor` liefert die Faktorisierung eines Polynoms als Objekt des Bibliotheks-Domains `Factored`:

```
>> f := factor(x^2 + 2*x + 1);
    domtype(f)

          2
      (x + 1)

Factored
```

Dieses Domain implementiert die oben erwähnte Konvertierungsroutine `"convert_to"`, die man direkt aufrufen kann, um die Faktorisierung in eine Liste zu konvertieren (siehe `factor` für Details):

```
>> Factored::convert_to(f, DOM_LIST)

[1, x + 1, 2]
```

Es ist jedoch bequemer, `coerce` zu verwenden, was implizit einen Aufruf von `Factored::convert_to` bewirkt:

```
>> coerce(f, DOM_LIST)

[1, x + 1, 2]
```

Beispiel 3. Oft kann die gewünschte Konvertierung auch durch den Aufruf des Konstruktors eines Domains T erreicht werden. Beispielsweise konvertiert die folgende Eingabe ein Feld in eine Matrix vom Domain-Typ $\text{Dom}::\text{Matrix}(\text{Dom}::\text{Rational})$:

```
>> a := array(1..2, 1..2, [[1, 2], [3, 4]]):
      MatQ := Dom::Matrix(Dom::Rational):

>> MatQ(a)
```

```
+-      +-
|  1, 2  |
|        |
|  3, 4  |
+-      +-

```

Der Aufruf $\text{MatQ}(a)$ bewirkt den Aufruf der Methode "new" des Domains MatQ , der wiederum den Aufruf der Methode "convert" des Domains MatQ zur Folge hat, um das Feld in eine Matrix zu konvertieren.

Das Gleiche kann hier mit Hilfe der Funktion `coerce` erreicht werden:

```
>> A := coerce(a, MatQ);
      domtype(A)
```

```
+-      +-
|  1, 2  |
|        |
|  3, 4  |
+-      +-

```

```
Dom::Matrix(Dom::Rational)
```

Man beachte, dass der Konstruktor eines Domains T zur *Erzeugung* von Objekten dient, nicht zur Konvertierung von Objekten anderer Domains in den Domain-Typ T . Der Konstruktor gestattet es häufig, mehrere Argumente anzugeben, was die Implementation zahlreicher benutzerfreundlicher Möglichkeiten zur Erzeugung von Objekten bietet (siehe beispielsweise die verschiedenen Aufrufe zur Erzeugung von Matrizen, die von `matrix` zur Verfügung gestellt werden).

Änderungen:

⚡ `coerce` ist eine neue Funktion.

collect – Zusammenfassen von Koeffizienten eines polynomialen Ausdrucks

`collect(p, x)` formt den polynomialen Ausdruck p in einen Ausdruck der Form $\sum_{i=0}^n a_i x^i$ um, wobei die Koeffizienten a_i Ausdrücke sind, die x nicht als polynomiale Unbestimmte enthalten.

`collect(p, [x1, x2, ...])` formt den polynomialen Ausdruck `p` in einen Ausdruck der Form

$$\sum_{i_1, i_2, \dots} a_{i_1, i_2, \dots} x_1^{i_1} x_2^{i_2} \dots$$

um, wobei die Koeffizienten $a_{i_1, i_2, \dots}$ keines der x_i als polynomiale Unbestimmte enthalten.

Wird ein drittes Argument `f` angegeben, so wird jeder Koeffizient in den oben genannten Rückgabewerten durch $f(a_i)$ bzw. $f(a_{i_1, i_2, \dots})$ ersetzt.

Aufruf(e):

```
# collect(p, x <, f>)
# collect(p, [x1, x2, ...] <, f>)
```

Parameter:

<code>p</code>	— ein polynomialer Ausdruck
<code>x, x1, x2, ...</code>	— die Unbestimmten des Polynoms: üblicherweise Bezeichner oder indizierte Bezeichner
<code>f</code>	— eine Funktion

Rückgabewert: ein polynomialer Ausdruck oder `FAIL`, falls `p` nicht in ein Polynom konvertiert werden kann.

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `coeff`, `combine`, `expand`, `factor`, `indets`, `normal`, `poly`, `rectform`, `rewrite`, `simplify`

Details:

- # `collect` fasst die Terme von `p` nach Potenzen der angegebenen Unbestimmten zusammen. `collect` liefert eine abgeänderte Kopie von `p` zurück; das Argument selbst bleibt unverändert. Siehe Beispiel ??.
- # `collect` dient im Wesentlichen nur als Abkürzung für die Hintereinanderausführung von `expr` nach `poly`. Es wandelt zunächst `p` mittels `poly` in ein Polynom in den angegebenen Unbestimmten um; dadurch wird die gewünschte Zusammenfassung der Terme erzielt. Danach wird `expr` benutzt, um das Ergebnis in einen Ausdruck zu konvertieren. Siehe auch die Hilfe zu `poly` für mehr Informationen und Beispiele.
- # Die Unbestimmten brauchen keine Bezeichner oder indizierten Bezeichner zu sein; jeder Ausdruck, der weder rational noch konstant ist, ist zulässig. Zum Beispiel sind `sin(x)`, `f(x)` und `y^(1/3)` erlaubt, `sin(1)`

und $f(1)$ dagegen nicht. Genauer: x ist genau dann eine polynomiale Unbestimmte, wenn der Aufruf `indets(x, PolyExpr)` das Ergebnis $\{x\}$ liefert. Siehe die Hilfe zu `indets` sowie Beispiel ??.

⌘ `collect` wird nicht rekursiv auf die Operanden nicht-polynomialer Teilausdrücke angewandt. Siehe Beispiel ??.

⌘ Die Terme im Ergebnis von `collect` werden in der Regel nicht geordnet; dies kann man durch `poly` erreichen.

Die zu a_0 bzw. $a_{0,0,\dots}$ gehörigen „konstanten“ Terme brauchen im Ergebnis nicht nebeneinander zu stehen.



Siehe Beispiel ??.

⌘ `collect` liefert `FAIL`, falls p nicht in ein Polynom konvertiert werden kann; in der Hilfe zu `poly` findet man mehr darüber, wann dies der Fall ist. Siehe Beispiel ??.

Beispiel 1. Wir definieren einen polynomialen Ausdruck p und fassen alle Terme mit der gleichen Potenz von x und y zusammen:

```
>> p := x*y + z*x*y + y*x^2 - z*y*x^2 + x + z*x;
      collect(p, [x, y])
```

$$x^2 + x^2 y + x^2 z + x^2 y z + x^2 y - x^2 y z$$

$$x^2 (z + 1) + x^2 y (z + 1) + x^2 y (1 - z)$$

Der Ausdruck p selbst wird nicht verändert:

```
>> p
```

$$x^2 + x^2 y + x^2 z + x^2 y z + x^2 y - x^2 y z$$

Nun fassen wir Terme mit derselben x -Potenz zusammen:

```
>> collect(p, [x])
```

$$x^2 (y + z + y z + 1) + x^2 (y - y z)$$

Bei nur einer Unbestimmten dürfen die eckigen Klammern weggelassen werden:

```
>> collect(p, x)
```

$$x^2 (y + z + yz + 1) + x^2 (y - yz)$$

Durch Angabe von `factor` als drittem Argument sorgen wir dafür, dass jeder Koeffizient faktorisiert wird:

```
>> collect(p, x, factor)
```

$$x^2 (y + 1) (z + 1) - x^2 y (z - 1)$$

Beispiel 2. `collect` verhält sich gegenüber nicht-polynomialen Teilausdrücken wie `poly`. Ein solcher Teilausdruck bleibt unverändert, selbst wenn er eine der angegebenen Unbestimmten enthält; insbesondere wird `collect` nicht rekursiv angewandt:

```
>> collect(sin((x + 1)^2)*(x + 1) + 5*sin((x + 1)^2) + x, x)
```

$$6 \sin^2((x + 1)) + x (\sin^2((x + 1)) + 1)$$

Ein nicht-polynomialer Teilausdruck darf an `collect` als Unbestimmte übergeben werden, falls `poly` ihn als solche akzeptiert:

```
>> collect(sin((x + 1)^2)*(x + 1) + 5*sin((x + 1)^2) + x,
          sin((x + 1)^2))
```

$$x + (x + 6) \sin^2((x + 1))$$

Andernfalls hält `collect` mit einem Fehler:

```
>> collect(1 + I*(x + I), I)

Error: Illegal indeterminate [poly];
during evaluation of 'collect'
```

In diesem Beispiel liefert statt dessen `rectform` das gewünschte Ergebnis:

```
>> rectform(1 + I*(x + I))

- Im(x) + I Re(x)
```

Beispiel 3. `collect` liefert `FAIL` zurück, falls die Eingabe nicht in ein Polynom konvertiert werden kann:

```
>> collect(1/x, x)
```

FAIL

Beispiel 4. Die Terme des Ergebnisses werden in der Regel nicht nach ihrem Grad geordnet:

```
>> collect(1 + x^2 + x, [x])
```

$$x + x^2 + 1$$

Dies kann man jedoch mit `poly` erreichen:

```
>> poly(1 + x^2 + x, [x])
```

$$\text{poly}(x^2 + x + 1, [x])$$

Konstante Terme stehen nicht immer nebeneinander:

```
>> collect(sin(1) + (x + 1)^2, [x])
```

$$2x + \sin(1) + x^2 + 1$$

```
>> poly(sin(y) + (x + 1)^2, [x])
```

$$\text{poly}(x^2 + 2x + (\sin(y) + 1), [x])$$

Änderungen:

⌘ Keine Änderungen.

`combine` – Zusammenfassen von Termen gleicher algebraischer Struktur

`combine(f)` ersetzt in `f`, soweit möglich, Produkte von Potenzen durch einzelne Potenzen.

`combine(f, target)` fasst mehrere Aufrufe der Zielfunktion(en) zu einem einzelnen Aufruf zusammen.

Aufruf(e):

⌘ `combine(f)`

⌘ `combine(f, target)`

⌘ `combine(f, [target1, target2, ...])`

Parameter:

- `f` — ein arithmetischer Ausdruck, ein Feld, eine Liste, ein Polynom, oder eine Menge.
- `target` — einer der Bezeichner `arctan`, `exp`, `ln`, `sincos`, oder `sinhcosh`

Rückgabewert: ein Objekt desselben Typs wie `f`.

Seiteneffekte: `combine` reagiert auf Eigenschaften von Bezeichnern, die in der Eingabe vorkommen.

Überladbar durch: `f`

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `denom`, `expand`, `factor`, `normal`, `numer`, `radsimp`, `rectform`, `rewrite`, `simplify`

Details:

☞ `combine(f)` wendet die folgenden Ersetzungsregeln auf alle Produkte von Potenzen an, die in dem arithmetischen Ausdruck `f` vorkommen:

- $x^a x^b = x^{a+b}$
- $x^b y^b = (xy)^b$
- $(x^a)^b = x^{ab}$

Die letzten beiden Regeln sind nur unter bestimmten Voraussetzungen anwendbar, z. B. dann, wenn `b` eine ganze Zahl ist. Mit Ausnahme der dritten Regel stellt dieses Verhalten von `combine` eine Umkehrung des Verhaltens von `expand` dar. Siehe Beispiel ??.

Da MuPADs interner Vereinfacher manchmal die Umkehrung der obengenannten Regeln automatisch anwendet, hat `combine` in bestimmten Fällen keine Wirkung. Siehe Beispiel ??.



☞ `combine(f, target)` schreibt den arithmetischen Ausdruck `f` unter Anwendung der Ersetzungsregeln für die angegebenen Zielfunktionen um. Einige dieser Regeln sind nur unter bestimmten Voraussetzungen anwendbar. In den meisten Fällen handelt es sich bei diesen Regeln um die Umkehrung der von `expand` angewandten Regeln. Im einzelnen handelt es sich um folgende Regeln für die einzelnen Zielfunktionen:

`target = arctan`:

- $\arctan(x) + \arctan(y) = \arctan\left(\frac{x+y}{1-xy}\right)$

target = exp (siehe Beispiel ??:)

- $\exp(a)\exp(b) = \exp(a+b)$
- $\exp(a)^b = \exp(ab)$

target = ln (siehe Beispiel ??:)

- $\ln(a) + \ln(b) = \ln(ab)$
- $b \ln(a) = \ln(a^b)$

target = sincos (siehe Beispiel ??:)

- $\sin(x)\sin(y) = \frac{1}{2}\cos(x-y) - \frac{1}{2}\cos(x+y)$
- analoge Regeln für $\sin(x)\cos(y)$ und $\cos(x)\cos(y)$
- die obigen Regeln werden rekursiv angewandt auf Potenzen von sin und cos, wenn der Exponent eine natürliche Zahl ist

target = sinhcosh:

- $\sinh(x)\sinh(y) = \frac{1}{2}\cosh(x+y) - \frac{1}{2}\cosh(x-y)$
- analoge Regeln für $\sinh(x)\cosh(y)$ und $\cosh(x)\cosh(y)$
- die obigen Regeln werden rekursiv angewandt auf Potenzen von sinh und cosh, wenn der Exponent eine natürliche Zahl ist

⌘ combine arbeitet rekursiv auf Teilausdrücken von f.

⌘ Ist das zweite Argument eine Liste von Zielfunktionen, so wird combine nacheinander für jede Zielfunktion in der Liste auf f angewandt. Siehe Beispiel ??.

⌘ Ist f ein Feld, eine Liste oder eine Menge, so wird f nacheinander auf die Einträge von f angewandt; siehe Beispiel ??.

Ist f ein Polynom oder eine Reihe vom Typ `Series::Puisseux` oder `Series::gseries`, so wird combine auf jeden Koeffizienten einzeln angewandt; siehe Beispiel ??.

Beispiel 1. Wird kein zweites Argument angegeben, so werden Potenzen mit gleicher Basis zusammengefasst:

```
>> combine(sin(x) + x*y*x^(exp(1)))  
                                     exp(1) + 1  
sin(x) + y x
```

In manchen Fällen fasst combine auch Potenzen mit gleichem Exponenten zusammen:

```
>> combine(sqrt(2)*sqrt(3))  
1/2  
6
```

Beispiel 2. In dem meisten Fällen fasst `combine` gleiche Exponenten jedoch nicht zusammen:

```
>> combine(y^5*x^5)
```

$$x^5 y^5$$

Beispiel 3. Ist das zweite Argument `sincos`, so schreibt `combine` Produkte von Sinus- und Kosinusfunktionen als Summen mit solcher Funktionen mit komplizierterem Argument:

```
>> combine(sin(a)*cos(b) + sin(b)^2, sincos)
```

$$\frac{\sin(a+b)}{2} - \frac{\cos(2b)}{2} + \frac{\sin(a-b)}{2} + 1/2$$

Potenzen des Sinus und Kosinus mit negativem Exponenten werden nicht umgeschrieben:

```
>> combine(sin(b)^(-2), sincos)
```

$$\frac{1}{\sin(b)^2}$$

Beispiel 4. Ist das zweite Argument `exp`, so werden die bekannten Regeln für die Exponentialfunktion angewandt:

```
>> combine(exp(3)*exp(2), exp)
```

$$\exp(5)$$

```
>> combine(exp(a)^2, exp)
```

$$\exp(2a)$$

Beispiel 5. In diesem Beispiel wenden wir Ersetzungsregeln für den Logarithmus an; hierbei wird auch deutlich, dass es auf die Eigenschaften von Bezeichnern in der Eingabe ankommt. Der Logarithmus eines Produkts ist nicht immer gleich der Summe der Logarithmen der einzelnen Faktoren; aber für positive Zahlen gilt diese Regel:

```
>> combine(ln(a)+ln(b), ln)

ln(a) + ln(b)

>> assume(a>0): assume(b>0):
    combine(ln(a)+ln(b), ln)

ln(a b)

>> unassume(a): unassume(b):
```

Beispiel 6. Das zweite Argument darf auch eine Liste von Zielfunktionen sein. In diesem Fall werden die Ersetzungsregeln für jede Zielfunktion in der Liste angewandt:

```
>> combine(ln(2)+ln(3)+sin(a)*cos(a), [ln, sincos])

sin(2 a)
ln(6) + ----
      2
```

Beispiel 7. combine wird elementweise auf Mengen angewandt:

```
>> combine({sqrt(2)*sqrt(5), sqrt(2)*sqrt(11)})

1/2    1/2
{10    , 22    }
```

Beispiel 8. combine wird koeffizientenweise auf Polynome angewandt:

```
>> combine(poly(sin(x)*cos(x)*y, [y]), sincos)

/ / sin(2 x) \
poly| | ----- | y, [y] |
\ \      2    /
```

Die Unbestimmten des Polynoms bleiben unverändert:

```
>> combine(poly(sin(x)*cos(x)), sincos)

poly(sin(x) cos(x), [sin(x), cos(x)])
```

Hintergründe:

- ⌘ Fortgeschrittene Benutzer haben die Möglichkeit, `combine` durch eigene Ersetzungsregeln für weitere Zielfunktionen zu erweitern. Hierzu definiert man ein neues Attribut `"target"` von `combine`; dazu muss zunächst `unprotect` auf den Bezeichner `combine` angewandt werden. Danach führt `combine(f, target)` zum Aufruf `combine::target(f)` des entsprechenden Attributs.
- ⌘ Per Voreinstellung behandelt `combine` jeden Teilausdruck `g(x1, x2, ...)` von `f` dadurch, dass es sich selbst rekursiv für die Operanden `x1`, `x2`, etc. aufruft. Benutzer können dieses Verhalten für ihre selbstdefinierten Funktionsumgebungen `g` durch Definieren eines `"combine"` Attributs von `g` ändern. Trifft `combine` dann auf den o.g. Teilausdruck, so wird das Attribut `g::combine` mit der Argumentfolge `x1, x2, ...` aufgerufen.

Änderungen:

- ⌘ Die Zielfunktionen `_power` und `sqrt` sind obsolet; dieselbe Funktionalität wird jetzt durch den Aufruf von `combine` ohne zweites Argument erreicht.
 - ⌘ `combine` reagiert jetzt auf Eigenschaften von Bezeichnern in der Eingabe.
-

`complexInfinity` – unendlich ferner Punkt von \mathbb{C}

`complexInfinity` repräsentiert den bei der Einpunktkompaktifizierung zu den komplexen Zahlen hinzukommenden unendlichen Punkt.

Verwandte Funktionen: `infinity`

Details:

- ⌘ Mathematisch ist `complexInfinity` der Nordpol der Riemannschen Zahlenkugel, wobei der Einheitskreis den Äquator und der Nullpunkt den Südpol bildet.
 - ⌘ Bezüglich arithmetischer Operationen verhält sich `complexInfinity` wie „ $1/0$ “. Insbesondere können von 0 verschiedene komplexe Zahlen mit `complexInfinity` oder `1/complexInfinity` multipliziert werden. Die Addition einer von Null verschiedenen Zahl zu `complexInfinity` ergibt `undefined`.
 - ⌘ In Bezug auf arithmetische Operationen ist `complexInfinity` inkompatibel mit dem reellen Unendlich-Objekt `infinity`.
-

Beispiel 1. `complexInfinity` kann in arithmetischen Operationen mit komplexen Zahlen verwendet werden. Das Ergebnis bei einer Multiplikation oder einer Division ist entweder `complexInfinity`, 0, oder `undefined`:

```
>> 3*complexInfinity, I*complexInfinity, 0*complexInfinity;
    3/complexInfinity, I/complexInfinity, 0/complexInfinity;
    complexInfinity/3, complexInfinity/I;
    complexInfinity*complexInfinity, complexInfinity/complexInfinity;

    complexInfinity, complexInfinity, undefined

    0, 0, 0

    complexInfinity, complexInfinity

    complexInfinity, undefined
```

Von symbolische Ausdrücke in Multiplikationen und Divisionen wird implizit angenommen, dass sie sowohl von 0 als auch von `complexInfinity` verschieden sind:

```
>> delete x;
    x*complexInfinity, x/complexInfinity, complexInfinity/x

    complexInfinity, 0, complexInfinity
```

Das Ergebnis einer Addition ist stets `undefined`:

```
>> 3 + complexInfinity, I + complexInfinity, x + complexInfinity

    undefined, undefined, undefined
```

Hintergründe:

☞ `complexInfinity` ist einziges Element des Domains `stdlib::CInfinity`.

Änderungen:

☞ Keine Änderungen.

conjugate – Konjugierte einer komplexen Zahl

`conjugate(z)` berechnet die konjugiert komplexe Zahl $\Re(z) - i\Im(z)$ zu einer komplexen Zahl $z = \Re(z) + i\Im(z)$.

Aufruf(e):

☞ `conjugate(z)`

Parameter:

`z` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck

Überladbar durch: `z`

Seiteneffekte: `conjugate` reagiert auf Eigenschaften von Bezeichnern, die mit `assume` definiert worden sind.

Verwandte Funktionen: `abs`, `assume`, `Im`, `Re`, `rectform`, `sign`

Details:

☞ Für Zahlen der Typen `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, `DOM_COMPLEX` wird die Konjugierte unmittelbar und sehr effizient berechnet.

☞ `conjugate` kann auch symbolische Ausdrücke verarbeiten. Eigenschaften von Bezeichnern werden dabei berücksichtigt (siehe auch `assume`). Von einem Bezeichner `z` ohne Eigenschaften wird dabei angenommen, dass er für irgendeine komplexe Zahl steht, und es wird der symbolische Aufruf `conjugate(z)` zurückgeliefert. Siehe Beispiel ??.

☞ `conjugate` kann unter anderem Aufrufe der folgenden mathematischen Funktionen behandeln:

<code>_mult</code>	<code>_plus</code>	<code>_power</code>	<code>abs</code>	<code>cos</code>	<code>cosh</code>	<code>cot</code>
<code>coth</code>	<code>csc</code>	<code>csch</code>	<code>erf</code>	<code>erfc</code>	<code>exp</code>	<code>gamma</code>
<code>igamma</code>	<code>sec</code>	<code>sech</code>	<code>sin</code>	<code>sinh</code>	<code>tan</code>	<code>tanh</code>

Siehe Beispiel ??.

☞ Falls `conjugate` auf eine mathematische Funktion stößt, die es nicht behandeln kann, so wird ein symbolischer Aufruf von `conjugate` zurückgeliefert. Siehe Beispiel ??.

Beispiel 1. `conjugate` kann unter anderem Summen, Produkte, die Exponential- und die Sinusfunktion behandeln:

```
>> conjugate((1 + I)*exp(2 - 3*I))
(1 - I) exp(2 + 3 I)
>> delete z: conjugate(z + 2*sin(3 - 5*I))
conjugate(z) + 2 sin(3 + 5 I)
```


Beispiel 2. `conjugate` berücksichtigt Eigenschaften von Bezeichnern:

```
>> delete x, y: assume(x, Type::Real):
      conjugate(x), conjugate(y)

      x, conjugate(y)
```

Beispiel 3. Enthält die Eingabe eine Funktion, die das System nicht kennt, so wird ein symbolischer Aufruf von `conjugate` zurückgeliefert:

```
>> delete f, z: conjugate(f(z) + I)

      conjugate(f(z)) - I
```

Sei nun f eine benutzerdefinierte mathematische Funktion, von der bekannt ist, dass $\overline{f(z)} = f(\bar{z})$ für alle komplexen Zahlen z gilt. Um dies `conjugate` mitzuteilen, betten wir `conjugate` in eine Funktionsumgebung ein und definieren einen geeigneten Eintrag "conjugate":

```
>> f := funcenv(f):
      f::conjugate := u -> f(conjugate(u)):
```

Danach führt ein Aufruf von `conjugate` mit einem Argument der Form $f(u)$ zu einem Aufruf $f::conjugate(u)$, und dieser wiederum liefert $f(conjugate(u))$:

```
>> conjugate(f(z) + I), conjugate(f(I))

      f(conjugate(z)) - I, f(- I)
```

Hintergründe:

☞ Enthält z einen Teilausdruck der Form $f(u, \dots)$, wo f eine Funktionsumgebung ist, so ruft `conjugate` den Eintrag "conjugate" von f auf, um die Konjugierte von $f(u, \dots)$ zu bestimmen. Auf diese Weise kann die Funktionalität von `conjugate` auf benutzerdefinierte Funktionen ausgedehnt werden.

Der Eintrag "conjugate" wird mit den Argumenten u, \dots von f aufgerufen.

Hat f keinen Eintrag "conjugate", so wird der Teilausdruck $f(u, \dots)$ im Ergebnis durch einen symbolischen Aufruf `conjugate(f(u, ...))` ersetzt.

Siehe Beispiel ??.

☞ Ebenso versucht `conjugate` in dem Fall, dass ein Element `d` eines Domains `T` als Teilausdruck von `z` auftritt, den Eintrag "`conjugate`" dieses Domains mit `d` als Argument aufzurufen, um die Konjugierte von `d` zu berechnen.

Hat `T` keinen Eintrag "`conjugate`", so wird `d` im Ergebnis durch einen symbolischen Aufruf `conjugate(d)` ersetzt.

Dasselbe gilt für Objekte von Kern-Domains, die keine arithmetischen Ausdrücke sind, wie etwa Listen, Felder, Tabellen, Mengen oder Polynome.

Änderungen:

☞ `conjugate` berücksichtigt jetzt Eigenschaften von Bezeichnern.

`contains` – Test auf Vorhandensein eines Eintrags in einem Container

`contains(s, object)` testet, ob `object` ein Element der Menge `s` ist.

`contains(l, object)` liefert den Index von `object` in der Liste `l`.

`contains(t, object)` testet, ob das Domain, das Feld oder die Tabelle `t` einen Eintrag unter dem Index `object` hat.

Aufruf(e):

☞ `contains(s, object)`
☞ `contains(l, object <, i>)`
☞ `contains(t, object)`

Parameter:

<code>s</code>	— eine Menge
<code>l</code>	— eine Liste
<code>t</code>	— ein Domain, ein Feld oder eine Tabelle
<code>object</code>	— ein beliebiges MuPAD-Objekt
<code>i</code>	— eine ganze Zahl

Rückgabewert: Für Domains, Felder, Mengen oder Tabellen liefert `contains` einen der boolschen Werte `TRUE` oder `FALSE`. Bei Listen wird eine nicht-negative ganze Zahl geliefert.

Überladbar durch: `s`, `l`, `t`

Verwandte Funktionen: `_in`, `_index`, `has`, `op`, `slot`

Details:

- ⌘ `contains` ist ein schneller Test auf Vorhandensein von Einträgen in MuPADs Basis-Container-Datentypen. Für Listen und Mengen sucht `contains` nach einem entsprechenden Eintrag. Für Domains, Felder und Tabellen prüft `contains`, ob unter dem gegebenen Index ein Eintrag vorhanden ist.
- ⌘ `contains` arbeitet syntaktisch, d.h. mathematisch äquivalente Objekte werden nur dann als gleich betrachtet, wenn sie syntaktisch identisch sind. Siehe Beispiel ??.
- ⌘ `contains` steigt nicht rekursiv in Teilausdrücke ab; dies erreicht man mit `has`. Siehe Beispiel ??.
- ⌘ `contains(s, object)` liefert `TRUE`, wenn `object` ein Element der Menge `s` ist, ansonsten liefert es `FALSE`.
- ⌘ `contains(l, object)` liefert die Position von `object` in der Liste `l` als positive ganze Zahl zurück, wenn `object` ein Eintrag von `l` ist. Sonst ist der Rückgabewert 0. Wenn mehr als ein Eintrag von `l` gleich `object` ist, dann wird der Index des ersten Auftretens geliefert.
Durch die Angabe eines dritten Arguments `i` kann man eine Startposition in der Liste angeben, wo die Suche beginnen soll. Dann werden Einträge mit Index kleiner als `i` nicht berücksichtigt. Wenn `i` nicht im gültigen Bereich liegt, dann ist der Rückgabewert 0.
Siehe Beispiele ?? und ??.
- ⌘ `contains(t, object)` liefert `TRUE`, wenn das Domain, das Feld oder die Tabelle `t` einen Eintrag unter dem Index `object` hat und sonst `FALSE`. Siehe Beispiel ??.
- ⌘ `contains` ist eine Funktion des Systemkerns.

Beispiel 1. `contains` kann dazu verwendet werden, das Vorhandensein eines Elements in einer Menge zu testen:

```
>> contains({a, b, c}, a), contains({a, b, c}, 2)
      TRUE, FALSE
```

Beispiel 2. `contains` arbeitet syntaktisch, d.h. mathematisch äquivalente Objekte werden nur dann als gleich betrachtet, wenn sie syntaktisch identisch sind. In diesem Beispiel liefert `contains` `FALSE`, weil $y \cdot (x + 1)$ und $y \cdot x + y$ verschiedene Darstellungen desselben mathematischen Ausdrucks sind:

```
>> contains({y*(x + 1)}, y*x + y)

FALSE
```

Beispiel 3. `contains` macht keinen rekursiven Abstieg in die Operanden seines ersten Arguments. In diesem Beispiel ist `c` kein Element der Menge, und daher wird `FALSE` zurückgeliefert:

```
>> contains({a, b, c + d}, c)

FALSE
```

Um zu testen, ob ein Ausdruck *irgendwo* in einem komplexeren Ausdruck enthalten ist, ist `has` zu verwenden:

```
>> has({a, b, c + d}, c)

TRUE
```

Beispiel 4. Wenn `contains` auf eine Liste angewendet wird, so wird die Position des gesuchten Ausdrucks in der Liste zurückgeliefert:

```
>> contains([a, b, c], b)

2
```

Wenn der gesuchte Ausdruck nicht in der Liste enthalten ist, wird 0 zurückgeliefert:

```
>> contains([a, b, c], d)

0
```

Beispiel 5. `contains` liefert die Position des ersten Auftretens des gegebenen Objekts in der Liste, wenn es mehrfach vorhanden ist:

```
>> l := [a, b, a, b]: contains(l, b)

2
```

Für die Suche in Listen kann eine Start-Position als optionales drittes Argument angegeben werden:

```
>> contains(l, b, 1), contains(l, b, 2),  
      contains(l, b, 3), contains(l, b, 4)  
  
      2, 2, 4, 4
```

Wenn das dritte Argument nicht im Gültigkeitsbereich liegt, dann ist der Rückgabewert 0:

```
>> contains(l, b, -1), contains(l, b, 0), contains(l, b, 5)  
  
      0, 0, 0
```

Beispiel 6. Für Tabellen liefert `contains` `TRUE`, wenn das zweite Argument ein gültiger Index in der Tabelle ist. Die in der Tabelle abgespeicherten Werte werden nicht berücksichtigt:

```
>> t := table(13 = value): contains(t, 13), contains(t, value)  
  
      TRUE, FALSE
```

Auf die Weise ist es möglich zu testen, ob ein Feld einen Wert unter einem gegebenen Index enthält. Das Feld `a` hat einen Wert unter dem Index `(1, 1)`, aber nicht unter dem Index `(1, 2)`:

```
>> a := array(1..3, 1..2, (1, 1) = x, (2, 1) = PI):  
      contains(a, (1, 1)), contains(a, (1, 2))  
  
      TRUE, FALSE
```

`contains` ist nicht dazu gedacht, um zu testen, ob ein Feld einen gegebenen Wert enthält:

```
>> contains(a, PI)  
  
      Error: Index dimension mismatch [array]
```

Selbst wenn die Dimensionen übereinstimmen, muss der Index für das Feld gültig sein:

```
>> contains(a, (4, 4))  
  
      Error: Illegal argument [array]
```

Beispiel 7. `contains` kann verwendet werden, um zu prüfen, ob ein Domain einen Eintrag unter einem gegebenen slot enthält:

```
>> T := newDomain("T"): T::index := value:
      contains(T, index), contains(T, value)

FALSE, FALSE
```

Es ist kein Eintrag unter dem Slot `index` vorhanden. Beachten sie, dass die Schreibweise `T::index` äquivalent zu `slot(T, "index")` ist:

```
>> contains(T, "index")

TRUE
```

Beispiel 8. Anwender können `contains` für ihre eigenen Domains überladen. Zur Illustration erzeugen wir ein Domain `T` und definieren einen "`contains`"-Eintrag, der testet, ob die Menge der Einträge eines Elements den gegebenen Wert `idx` enthält:

```
>> T := newDomain("T"):
      T::contains := (e, idx) -> contains({extop(e)}, idx):

>> e := new(T, 1, 2): contains(e, 2), contains(e, 3)

TRUE, FALSE
```

Änderungen:

☞ Keine Änderungen.

content – der Inhalt eines Polynoms

`content(f)` berechnet den Inhalt von f , d. h. den größten gemeinsamen Teiler aller Koeffizienten von f .

Aufruf(e):

☞ `content(p)`
☞ `content(f <, vars>)`

Parameter:

- `p` — ein Polynom vom Typ `DOM_POLY`
- `f` — ein polynomialer Ausdruck
- `vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: ein Objekt gleichen Typs wie die Koeffizienten der Eingabe oder `FAIL`

Überladbar durch: `p`

Verwandte Funktionen: `coeff`, `factor`, `gcd`, `icontent`, `ifactor`, `igcd`, `ilcm`, `lcm`, `poly`, `polylib::primpart`

Details:

- ☞ Ist `p` das Nullpolynom, so liefert `content` den Wert 0.
- ☞ Ist `p` ein Polynom, das ungleich dem Nullpolynom ist, über dem Ring `IntMod(n)`, so liefert `content` den Wert 1, falls `n` eine Primzahl ist; andernfalls liefert `content` einen Fehler.
- ☞ Ist `p` ein Polynom mit einem auf Bibilotheksebene definierten Domain `R` als Koeffizientenring, so wird der ggT der Koeffizienten von dem Eintrag `gcd` des Koeffizientenrings berechnet. Existiert kein solcher Eintrag, so liefert `content` den Wert `FAIL`.
- ☞ Ist `p` ein Polynom mit Koeffizientenring `Expr`, so geschieht folgendes.
 Sind alle Koeffizienten von `p` ganze bzw. rationale Zahlen, so ist `content(p)` äquivalent zu `gcd(coeff(p))`, und der Rückgabewert ist eine natürliche Zahl bzw. eine rationale Zahl. Siehe Beispiel ??.
 Ist mindestens ein Koeffizient eine Gleitkommazahl oder eine komplexe Zahl, und sind die anderen Koeffizienten Zahlen, so liefert `content` den Wert 1. Siehe Beispiel ??.
 Ist wenigstens ein Koeffizient keine Zahl, und können alle Koeffizienten von `p` mittels `poly` in Polynome konvertiert werden, so ist `content(p)` äquivalent zu `gcd(coeff(p))`. Siehe Beispiel ??.
 Andernfalls liefert `content` den Wert 1 zurück.
- ☞ Ein polynomialer Ausdruck `f` wird in ein Polynom über dem Koeffizientenring `Expr` konvertiert, indem `p := poly(f <, vars>)` aufgerufen wird; danach wird `content` auf das Ergebnis angewendet. Siehe Beispiel ??.
- ☞ Weiß man, dass ein Polynom nur ganzzahlige oder rationale Koeffizienten hat, so ist es aus Geschwindigkeitsgründen besser, `icontent` zu verwenden.

☞ Division aller Koeffizienten von p durch dessen Inhalt ergibt den Primivteil von f . Diesen kann man mittels `polylib::primpart` auch direkt erhalten.

Beispiel 1. Für Polynome mit ganzzahligen Koeffizienten ist das Ergebnis gleich dem von `icontent`.

```
>> content(poly(6*x^3*y + 3*x*y + 9*y, [x, y]))
```

3

Der folgende Aufruf, mit einem polynomialen Ausdruck statt einem Polynom als erstem Argument, ist zum obigen äquivalent:

```
>> content(6*x^3*y + 3*x*y + 9*y, [x, y])
```

3

Wird keine Liste von Unbestimmten angegeben, so wandelt `poly` den Ausdruck in ein Polynom in allen vorkommenden Unbestimmten um, so dass auch der folgende Aufruf zu den vorigen äquivalent ist:

```
>> content(6*x^3*y + 3*x*y + 9*y)
```

3

Statt als Polynom in zwei Variablen mit ganzzahligen Koeffizienten können wir denselben Ausdruck auch als Polynom in einer Variable x auffassen, dessen Koeffizienten einen Parameter y enthalten. Dann sind die Koeffizienten und deren ggT polynomiale Ausdrücke in y :

```
>> content(poly(6*x^3*y + 3*x*y + 9*y, [x]))
```

3 y

Auch im folgenden Beispiel sind die Koeffizienten und der Inhalt polynomiale Ausdrücke:

```
>> content(poly(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x]))
```

3 y + 2

Der nachstehende Aufruf ist äquivalent zum vorigen:

```
>> content(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x])
```

3 y + 2

Beispiel 2. Hat ein Polynom Zahlen als Koeffizienten und ist unter diesen wenigstens eine Gleitkommazahl, so ist der Inhalt des Polynoms 1:

```
>> content(2.0*x+2.0)
```

1

Beispiel 3. Sind nicht alle Koeffizienten Zahlen, so wird der ggT der Koeffizienten zurückgegeben:

```
>> content(poly(x^2*y+x, [y]))
```

x

Änderungen:

☞ Keine Änderungen.

context – Auswertung eines Objekts im umgebenden Kontext

Innerhalb einer Prozedur wertet `context(object)` das Objekt `object` im Kontext der aufrufenden Prozedur aus.

Aufruf(e):

☞ `context(object)`

Parameter:


`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: das ausgewertete Objekt.

Seiteneffekte: Das Verhalten von `context` wird durch den Wert der Umgebungsvariablen `LEVEL` beeinflusst, die die maximale Substitutionstiefe von Bezeichnern festlegt.

Verwandte Funktionen: `DOM_PROC`, `eval`, `freeze`, `hold`, `LEVEL`, `level`, `MAXLEVEL`, `proc`

Details:

- ☞ Die meisten MuPAD Prozeduren evaluieren ihre Argumente bevor der Prozedurrumpf ausgeführt wird. Wenn die Prozedur jedoch mit der Option `hold` deklariert wurde, dann werden die Argumente unevaluiert an die Prozedur übergeben. `context` dient dazu, diese Argumente nachträglich innerhalb der Prozedur zu evaluieren.
 - ☞ Wie die meisten MuPAD Prozeduren evaluiert `context` sein Argument `object` wie üblich im Kontext der aktuellen Prozedur. Dann wird das Ergebnis erneut im dynamischen Kontext evaluiert, der gültig war, bevor die aktuelle Prozedur aufgerufen wurde. Der umgebende Kontext ist entweder die interaktive Ebene oder die Prozedur, die die aktuelle Prozedur aufgerufen hat.
 - ☞ "`func_call`"-Methoden von Domains evaluieren ihre Argumente nie, egal ob die Option `hold` verwendet wird oder nicht. Siehe Beispiel ??.
 - ☞ `context` reagiert auf den Wert der Umgebungsvariablen `LEVEL`, die die maximale Auswertungstiefe im rekursiven Prozess der Bezeichner-Ersetzung bei der Evaluierung bestimmt. Die Evaluierung des Arguments findet mit dem Wert von `LEVEL` statt, der in der aktuellen Prozedur gültig ist, das ist standardmäßig 1. Die zweite Evaluierung verwendet den Wert von `LEVEL`, der im umgebenden Kontext gültig ist; das ist üblicherweise 1, wenn der umgebende Kontext auch eine Prozedur ist, während er standardmäßig 100 ist, wenn der umgebende Kontext die interaktive Ebene ist. Siehe Beispiel ??.
 - ☞ Die Funktion `context` kann nicht interaktiv oder in der Funktion `context` selbst aufgerufen werden. Deshalb ist es nicht möglich, ein Objekt in einer höheren Ebene der Aufrufhierarchie zu evaluieren. Siehe Beispiel ??.
- 
- ☞ `context` ist eine Funktion des Systemkerns.
-

Beispiel 1. Wir definieren eine Prozedur `f` mit der Option `hold`. Wenn diese Prozedur mit einem Bezeichner als Argument aufgerufen wird, so wie `a` im folgenden Beispiel, dann ist dieser Bezeichner selbst das Argument innerhalb von `f`. `context` kann dann dazu verwendet werden, den Wert von `a` im umgebenden Kontext zu erhalten:

```
>> a := 2:
    f := proc(i)
        option hold;
        begin
            print(i, context(i), i^2 + 2, context(i^2 + 2));
        end_proc:
    f(a):
```

$$a, 2, a + 2, 6$$

Wenn eine Prozedur mit Option `hold` von einer anderen Prozedur aufgerufen wird, kann man sonderbare Effekte beobachten, wenn die Prozedur mit der Option `hold` ihre formalen Parameter nicht mit `context` evaluiert. Hier ist der Wert des formalen Parameters `j` in `g` die Variable `i`, die im Kontext der Prozedur `f` definiert ist und nicht ihr Wert 4. Wenn man auf den Wert dieser Variable zugreifen will, dann muss `context` verwenden, sonst sieht man die Ausgabe `DOM_VAR(0,2)`, was die Variable `i` aus `f` ist, die ihren Kontext verloren hat:

```
>> f := proc()
      local i;
      begin
        i := 4;
        g(i);
      end_proc:
    g := proc(j)
      option hold;
      begin
        print(j, eval(j), context(j));
        print(j + 1)
      end_proc:
    f()

      DOM_VAR(0,2), DOM_VAR(0,2), 4

      DOM_VAR(0,2) + 1
```

Beispiel 2. Die "`func_call`"-Methode eines Domains hat implizit die Option `hold` gesetzt. Wir definieren eine "`func_call`"-Methode für das Domain `DOM_STRING` für MuPAD Zeichenketten. Die definierte Methode wandelte alle verbleibenden Argumente in Zeichenketten um und hängt sie an das erste Argument an, welches identisch mit der Zeichenkette ist, die als 0-ter Operand des Funktionsaufrufs steht:

```
>> unprotect(DOM_STRING):
    DOM_STRING::func_call :=
      string -> _concat(string, map(args(2..args(0)), expr2text)):
    a := 1: "abc"(1, a, x)

      "abc1ax"
```

Man sieht, dass der Bezeichner `a` zu der Zeichenkette hinzugefügt wurde und nicht sein Wert 1. Durch die Verwendung von `context` kann man den Wert des Bezeichners `a` erhalten, den er vor dem Aufruf der "`func_call`"-Methode hatte:

```
>> DOM_STRING::func_call :=
  string -> _concat(string, map(context(args(2..args(0))),
                                expr2text)):

"abc"(1, a, x);
delete DOM_STRING::func_call:  protect(DOM_STRING, Error):

"abc11x"
```

Beispiel 3. Dieses Beispiel zeigt den Einfluss der Umgebungsvariablen `LEVEL` auf die Evaluierung von `context` und den Unterschied zu den Funktionen `eval` und `level`. `p` ist eine Funktion mit der Option `hold`. `x` ist ein formaler Parameter dieser Prozedur. Wenn sie ihre Argumente evaluieren, ersetzen `context`, `eval` und `level` alle `x` zuerst durch den Wert `a`. Dann evaluiert `eval` `a` im aktuellen Kontext mit `LEVEL = 1`, was den Wert `b` ergibt. `context` evaluiert `a` im umgebenden Kontext (was die interaktive Ebene ist) mit `LEVEL = 100`, was `c` ergibt.

Wenn `LEVEL` auf interaktiver Ebene gleich 1 ist, dann liefert `context` `b` wie `eval`, weil die zweite Evaluierung wie in `eval` auch mit `LEVEL = 1` durchgeführt wird.

Die lokale Variable `b` von `p` beeinflusst die Evaluierung in `context`, `eval` und `level` nicht, weil sie nur eine lokal deklarierte Variable vom Typ `DOM_VAR` ist, die nichts mit dem Bezeichner `b` zu tun hat, der der Wert von `a` ist:

```
>> delete a, b, c:  a := b:  b := c:
  p := proc(x)
    option hold;
    local b;
    begin
      b := 2;
      eval(x), context(x), level(x), level(x,2);
    end:
  p(a);
  LEVEL := 1: p(a);
  delete LEVEL:
```

`b, c, a, a`

`b, b, a, a`

Beispiel 4. Die Funktion `context` kann nicht interaktiv aufgerufen werden:

```
>> context(x)

Error: Function call not allowed on interactive level [context]
```

Änderungen:

- ⌘ Aufgrund des lexikalischen Scoping ist die Verwendung `context` nur in Prozeduren mit der Option `hold` sinnvoll.
-

debug – Ausführung von Prozeduren im Einzelschrittmodus

`debug(statement)` startet den MuPAD-Debugger und erlaubt es, `statement` schrittweise auszuführen.

Aufruf(e):

- ⌘ `debug(statement)`

Parameter:

`statement` — ein beliebiges MuPAD-Objekt; typischerweise ein Funktionsaufruf

Rückgabewert: der Rückgabewert von `statement`.

Verwandte Funktionen: `noDebug`, `Pref::ignoreNoDebug`, `prog::check`, `prog::profile`, `prog::trace`

Details:

- ⌘ `debug` versetzt den MuPAD-Kern in den *Debug-Modus* und startet MuPADs interaktiven Debugger, falls `statement` Prozeduraufrufe enthält, für die Debugging möglich ist.
- ⌘ In MuPAD-Versionen mit einer grafischen Benutzeroberfläche öffnet sich ein eigenes Debugger-Fenster. In der Terminalversion unter UNIX wird der Kommandozeilen-Debugger aktiviert.
Der Debugger bietet Einzelschrittausführung, Untersuchung von Variablenwerten und des Prozedurstacks, Haltepunkte, etc. Weitere Dokumentation ist im Hilfe-Menü des Debugger-Fensters zu finden.
- ⌘ Debugging ist nur für Prozeduren möglich, die in der MuPAD-Sprache geschrieben sind und nicht die Option `noDebug` haben. Insbesondere ist Debugging von Kernfunktionen nicht möglich.
Nach dem Aufruf `Pref::ignoreNoDebug(TRUE)` wird die Prozeduroption `noDebug` ignoriert.
- ⌘ Debugging einer Folge von durch Semikolons getrennten Anweisungen ist möglich, wenn diese in ein zusätzliches Paar runder Klammern eingeschlossen wird.

- ☞ `debug(statement)` liefert das selbe Ergebnis wie `statement`, falls die Ausführung nicht vom Benutzer im Debugger-Fenster abgebrochen wird.
 - ☞ `debug` ist eine Funktion des Systemkerns.
-

Beispiel 1. Dieses Beispiel startet den Debugger für die schrittweise Ausführung des Kommandos `int(cos(x), x)`, welches die Kosinusfunktion integriert:

```
>> debug(int(cos(x), x)):
```

Hintergründe:

- ☞ Im Debug-Modus wird der MuPAD-Parser umkonfiguriert. Er fügt dann so genannte *Debug-Knoten* in Prozeduren ein, die aus Dateien gelesen werden. Diese Debug-Knoten, die Datei-Identifikationen und Zeilennummern enthalten, erlauben es dem Debugger, ein auszuführendes Stück Code der zugehörigen Quelltext-Datei zuzuordnen.
- ☞ Wenn der Debug-Modus aktiviert ist und MuPAD auf eine Prozedur ohne Debug-Knoten stößt, schreibt das System diese Prozedur in eine temporäre Datei und fügt gleichzeitig Debug-Knoten in die Prozedur ein. Dies ermöglicht das Debugging interaktiv eingegebener Prozeduren genauso, als wenn sie aus einer Datei eingelesen worden wären. Die temporäre Datei wird am Ende der Sitzung gelöscht.
Das soeben beschriebene Verhalten trifft auch auf Prozeduren zu, die vor dem Eintritt in den Debug-Modus eingelesen wurden. Es ist daher empfehlenswert, für das Debugging einer größeren Anwendung den Kern bereits im Debug-Modus zu starten (siehe weiter unten).
- ☞ Wenn der MuPAD-Kern nicht im Debug-Modus gestartet wurde, so wird er beim ersten Aufruf von `debug` eingeschaltet. Der Debug-Modus bleibt bis zum Ende der Sitzung aktiviert.

Man kann den Kern bereits im Debug-Modus starten. Auf Windows-Plattformen erreicht man diese Konfigurationseinstellung über „Optionen“ im Menü „Ansicht“ durch anschließendes Klicken auf „Kern“. In der grafischen Benutzungsoberfläche unter UNIX wird die entsprechende Option durch Klicken auf „Kernel Debug Mode“ im Menü „Options“ ein- und ausgeschaltet. Auf einem Macintosh wählt man „Voreinstellungen“ im „Ablage“-Menü und anschließend „Kern“.

Änderungen:

- ☞ Die neue Prozedur-Option `noDebug` verhindert das Debuggen der entsprechenden Prozedur.

degree – der Grad eines Polynoms

`degree(p)` gibt den totalen Grad des Polynoms `p` zurück.

`degree(p, x)` gibt den Grad von `p` bezüglich der Unbestimmten `x` zurück.

Aufruf(e):

- ⌘ `degree(p)`
- ⌘ `degree(p, x)`
- ⌘ `degree(f <, vars>)`
- ⌘ `degree(f <, vars>, x)`

Parameter:

- `p` — ein Polynom vom Typ `DOM_POLY`
- `f` — ein polynomialer Ausdruck
- `vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- `x` — eine Unbestimmte

Rückgabewert: eine nicht-negative Zahl. `FAIL` wird zurückgeliefert, wenn die Eingabe nicht als Polynom interpretiert werden kann.

Überladbar durch: `p`, `f`

Verwandte Funktionen: `coeff`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- ⌘ Wenn das erste Argument `f` nicht Element eines Polynom-Domains ist, so konvertiert `degree` diesen Ausdruck intern mittels `poly(f)` in ein Polynom vom Typ `DOM_POLY`. Ist eine Liste von Unbestimmten angegeben, so wird `poly(f, vars)` betrachtet.
 - ⌘ `degree(f, vars, x)` liefert das Ergebnis 0, falls `x` nicht in der Liste `vars` enthalten ist.
 - ⌘ Der Grad des Nullpolynoms ist als 0 definiert.
 - ⌘ `degree` ist eine Funktion des Systemkerns.
-

Beispiel 1. Der totale Grad des folgenden polynomialen Ausdrucks wird berechnet:

```
>> degree(x^3 + x^2*y^2 + 2)
```

4

Beispiel 2. `degree` kann auf Polynome von Typ `DOM_POLY` angewendet werden:

```
>> degree(poly(x^2*z + x*z^3 + 1, [x, z]))
```

4

Beispiel 3. Der nächste Ausdruck wird als bivariates Polynom in `x` und `z` aufgefasst. Der Grad bezüglich `z` wird berechnet:

```
>> degree(x^2*z + x*z^3 + 1, [x, z], z)
```

3

Beispiel 4. Der Grad des Nullpolynoms ist als 0 definiert:

```
>> degree(0, [x, y])
```

0

Änderungen:

☞ Keine Änderungen.

`degreevec` – **die Exponenten des führenden Terms eines Polynoms**

`degreevec(p)` liefert eine Liste mit den Exponenten des führenden Terms des Polynoms `p`.

Aufruf(e):

☞ `degreevec(p <, order>)`

☞ `degreevec(f <, vars> <, order>)`

Parameter:

- `p` — ein Polynom vom Typ `DOM_POLY`
`order` — die Termordnung; entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.
`f` — ein polynomialer Ausdruck
`vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: eine Liste nicht-negativer ganzer Zahlen. `FAIL` wird zurückgeliefert, wenn die Eingabe nicht als Polynom interpretiert werden kann.

Überladbar durch: `p`, `f`

Verwandte Funktionen: `coeff`, `degree`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- ⌘ Wenn das erste Argument `f` nicht Element eines Polynom-Domains ist, so konvertiert `degreevec` diesen Ausdruck intern mittels `poly(f)` in ein Polynom vom Typ `DOM_POLY`. Ist eine Liste von Unbestimmten angegeben, so wird `poly(f, vars)` betrachtet.
- ⌘ Für ein Polynom in den Unbestimmten x_1, x_2, \dots, x_n mit dem Leitterm $x_1^{e_1} \times x_2^{e_2} \times \dots \times x_n^{e_n}$ wird der Exponentenvektor $[e_1, e_2, \dots, e_n]$ geliefert.
- ⌘ Für das Null-Polynom liefert `degreevec` eine Liste von Nullen.
- ⌘ Für die Ordnungen *LexOrder*, *DegreeOrder* und *DegInvLexOrder* ruft `degreevec` eine schnelle Kernfunktion auf. Andere Ordnungen werden durch langsamere Bibliotheksfunktionen behandelt.

Beispiel 1. Der führende Term des folgenden polynomialen Ausdrucks (bezüglich der Hauptvariablen `x`) ist x^4 :

```
>> degreevec(x^4 + x^2*y^3 + 2, [x, y])
[4, 0]
```

Mit der Hauptvariablen `y` ist der führende Term x^2y^3 :

```
>> degreevec(x^4 + x^2*y^3 + 2, [y, x])
[3, 2]
```

Bei Polynomen vom Typ DOM_POLY sind die Unbestimmten ein Bestandteil des Datentyps:

```
>> degreevec(poly(x^4 + x^2*y^3 + 2, [x, y])),
      degreevec(poly(x^4 + x^2*y^3 + 2, [y, x]))
      [4, 0], [3, 2]
```

Beispiel 2. Für univariate Polynome betrachten die Standard-Termordnungen denselben Term als „führend“:

```
>> degreevec(poly(x^2*z + x*z^3 + 1, [x]), LexOrder),
      degreevec(poly(x^2*z + x*z^3 + 1, [x]), DegreeOrder),
      degreevec(poly(x^2*z + x*z^3 + 1, [x]), DegInvLexOrder)
      [2], [2], [2]
```

Im multivariaten Fall können unterschiedliche Termordnungen zu unterschiedlichen führenden Exponentenvektoren führen:

```
>> degreevec(poly(x^2*z + x*z^3 + 1, [x, z])),
      degreevec(poly(x^2*z + x*z^3 + 1, [x, z]), DegreeOrder)
      [2, 1], [1, 3]

>> degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], LexOrder),
      degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], DegreeOrder),
      degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], DegInvLexOrder)
      [3, 0, 0], [1, 2, 1], [0, 4, 0]
```

Beispiel 3. Der Exponentenvektor des Null-Polynoms ist eine Liste von Nullen:

```
>> degreevec(0, [x, y, z])
      [0, 0, 0]
```

Änderungen:

- ☞ Selbstdefinierte Termordnungen können nun vorgegeben werden.
 - ☞ In früheren MuPAD-Versionen war degreevec eine Kernfunktion.
-

delete – Löschen des Wertes eines Bezeichners

Die Anweisung `delete x` löscht den Wert des Bezeichners `x`.

Aufruf(e):

```

⇨ delete x1, x2, ...
⇨ _delete(x1, x2, ...)

```

Parameter:

x1, x2, ... — Bezeichner oder indizierte Bezeichner

Rückgabewert: das leere Objekt vom Typ DOM_NULL.

Verwandte Funktionen: :=, _assign, assign, assignElements, evalassign

Details:

- ⇨ Für viele Berechnungen werden symbolische Variablen benötigt. Beispielsweise wird beim Lösen einer Gleichung nach einer Unbestimmten x ein Bezeichner x benötigt, der keinen Wert trägt. Hat x einen Wert, so kann er mittels `delete x` gelöscht werden, und x steht als symbolische Variable zur Verfügung.
- ⇨ Die Anweisung `delete x1, x2, ...` ist äquivalent zum Funktionsaufruf `_delete(x1, x2, ...)`. Die Werte aller angegebenen Bezeichner werden gelöscht.
- ⇨ Die Anweisung `delete x[j]` löscht den Eintrag j einer Liste, eines Arrays oder einer Tabelle mit dem Namen x . Listen und Tabellen werden durch das Löschen einzelner Elemente bzw. Einträge verkleinert.
- ⇨ Für einen Bezeichner x , dem mittels `assume` Eigenschaften zugeordnet wurden, löscht `delete x` diese Eigenschaften, d.h., `delete x` wirkt wie `unassume(x)`. Siehe Beispiel ??.
- ⇨ `_delete` ist eine Funktion des Systemkerns.

Beispiel 1. Den Bezeichnern x und y werden Werte zugewiesen. Nach dem Löschen werden die Bezeichner ohne Wert zurückgegeben:

```

>> x := 42: y := 7: delete x: x, y
                                x, 7

>> delete y: x, y
                                x, y

```

Mehrere Bezeichner können gemeinsam mit einem Aufruf gelöscht werden:

```

>> a := b := c := 42: a, b, c

```

```

42, 42, 42

>> delete a, b, c: a, b, c

a, b, c

```

Beispiel 2. `delete` kann auch zum Löschen von Elementen in Listen, Arrays und Tabellen verwendet werden:

```

>> L := [7, 13, 42]

[7, 13, 42]

>> delete L[2]: L

[7, 42]

>> A := array(1..3, [7, 13, 42])

+-          +-
| 7, 13, 42 |
+-          +-

>> delete A[2]: A, A[2]

+-          +-
| 7, ?[2], 42 |, A[2]
+-          +-

>> T := table(1 = 7, 2 = 13, 3 = 42)

table(
  3 = 42,
  2 = 13,
  1 = 7
)

>> delete T[2]: T

table(
  3 = 42,
  1 = 7
)

```

Man beachte, dass `delete` die zu löschenden Objekte nicht evaluiert. Im Folgenden wird ein Element aus der Liste `U` entfernt, wobei der ursprüngliche Wert von `U` (die Liste `L`) nicht verändert wird:

```

>> U := L: delete U[1]: U, L

```

[42], [7, 42]

Zuletzt werden alle zugewiesenen Werte gelöscht:

```
>> delete U, L, A, T: U, L, A, T  
  
U, L, A, T
```

Beispiel 3. `delete` kann auch zum Löschen von mittels `assume` gesetzten Eigenschaften verwendet werden. Mit der Annahme `,x > 1'` hat der Ausdruck `ln(x)` die Eigenschaft `,ln(x) > 0'` und damit das Vorzeichen 1:

```
>> assume(x > 1): sign(ln(x))  
  
1
```

Ohne eine Eigenschaft von `x` kann die Funktion `sign` das Vorzeichen von `ln(x)` nicht bestimmen:

```
>> delete x: sign(ln(x))  
  
sign(ln(x))
```

Änderungen:

- ⌘ `delete` ist ein neues Schlüsselwort.
- ⌘ In früheren MuPAD-Versionen wurde der Wert eines Bezeichners gelöscht, indem dem Bezeichner das spezielle Objekt `NIL` zugewiesen wurde. In der vorliegenden MuPAD-Version ist `NIL` ein gewöhnliches Objekt, das Bezeichnern als gültiger Wert zugewiesen werden kann. Werte können nur noch mittels `delete` gelöscht werden.

denom – der Nenner eines rationalen Ausdrucks

`denom(f)` liefert den Nenner des Ausdrucks `f` zurück.

Aufruf(e):

⌘ `denom(f)`

Parameter:

`f` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `f`

Verwandte Funktionen: `gcd, factor, normal, numer`

Details:

☞ `denom` behandelt die Eingabe als einen rationalen Ausdruck: nicht-rationale Teilausdrücke wie z. B. `sin(x)`, `x^(1/2)` etc. werden intern durch „temporäre Variablen“ ersetzt. Der Nenner dieses rationalisierten Ausdrucks wird berechnet, die temporären Variablen werden zuletzt wieder durch die ursprünglichen Teilausdrücke ersetzt.

☞ Es wird nicht immer gekürzt: der von `denom` berechnete Nenner ist im allgemeinen nicht teilerfremd zu dem mittels `numer` bestimmten Zähler. Mittels `normal` kann erreicht werden, dass Zähler und Nenner gekürzt werden. Siehe Beispiel ??.



Beispiel 1. Die Nenner einiger Ausdrücke werden berechnet:

```
>> denom(-3/4)
```

4

```
>> denom(x + 1/(2/3*x - 2/x))
```

$$\frac{2}{2x^2 - 6}$$

```
>> denom((cos(x)^2 - 1)/(cos(x) - 1))
```

$$\cos(x) - 1$$

Beispiel 2. Ein Bruch wird nicht gekürzt, wenn er in der Form „Zähler/Nenner“ gegeben ist:

```
>> r := (x^2 - 1)/(x^3 - x^2 + x - 1): denom(r)
```

$$x^2 - x^3 + x^2 - 1$$

Dieser Nenner ist nicht teilerfremd zum Zähler von `r`. Mit `normal` wird das Kürzen gemeinsamer Faktoren in Zähler und Nenner erzwungen:

```
>> denom(normal(r))
```

$$x^2 + 1$$

Ist dagegen der rationale Ausdruck eine Summe rationaler Ausdrücke, so wird automatisch normalisiert:

```
>> denom(r + x/(x + 1) + 1/(x + 1) - 1)
```

$$x^2 + 1$$

```
>> delete r:
```

Änderungen:

⌘ Keine Änderungen.

diff – Differenzieren eines Ausdrucks oder eines Polynoms

`diff(f, x)` berechnet die (partielle) Ableitung $\partial f / \partial x$ der Funktion `f` nach `x`.

Aufruf(e):

```
⌘ diff(f)
⌘ diff(f, x)
⌘ diff(f, x1, x2, ...)
```

Parameter:

`f` — ein arithmetischer Ausdruck oder ein Polynom vom Typ `DOM_POLY`
`x, x1, x2, ...` — Unbestimmte: Bezeichner oder indizierte Bezeichner

Rückgabewert: ein arithmetischer Ausdruck oder ein Polynom.

Überladbar durch: `f`

Weitere Dokumentation: Abschnitt 7.1 des MuPAD Tutoriums.

Verwandte Funktionen: `D`, `int`, `limit`, `poly`, `taylor`

Details:

- ☞ `diff(f, x)` berechnet die Ableitung des arithmetischen Ausdrucks (oder Polynoms) `f` nach der Unbestimmten `x`.
 - ☞ `diff(f, x1, x2, ...)` ist äquivalent zu `diff(...diff(diff(f, x1), x2)...) , d.h., f wird erst nach x1 differenziert, dann wird das Ergebnis nach x2 differenziert, usw., d.h., die partielle Ableitung $\cdots \frac{\partial}{\partial x_2} \frac{\partial}{\partial x_1} f$ wird berechnet.`
 - ☞ `diff(f)` liefert sein evaluiertes Argument zurück, d.h., die „null-te“ Ableitung von `f` wird berechnet.
 - ☞ Höhere Ableitungen können durch die Verwendung des Sequenz-Operators berechnet werden: Wenn `n` eine nicht-negative ganze Zahl ist, dann liefert `diff(f, x $ n)` die *n*-te Ableitung von `f` nach `x` zurück. Siehe Beispiel ??.
 - ☞ Die Unbestimmten `x`, `x1`, `x2`, ... müssen entweder Bezeichner (vom Domain-Typ `DOM_IDENT`) oder indizierte Bezeichner sein, d.h., von der Form `x[n]`, wobei `x` ein Bezeichner und `n` eine ganze Zahl ist. Wenn eine von ihnen eine andere Form hat, dann wird in symbolischer `diff`-Aufruf zurückgeliefert. Siehe Beispiel ??.
 - ☞ Wenn `f` ein arithmetischer Ausdruck ist, so liefert `diff` einen arithmetischen Ausdruck zurück. Wenn `f` ein Polynom ist, so liefert `diff` auch ein Polynom zurück; siehe Beispiel ?? . Eine Ausnahme von diesen Regeln tritt auf, wenn die Ableitung nicht berechnet werden kann, in diesem Fall wird ein symbolischer `diff`-Aufruf geliefert. Siehe Beispiel ??.
 - ☞ MuPAD macht keine Annahme, dass Ableitungen nach mehreren verschiedenen Unbestimmten kommutieren, d.h., im Allgemeinen repräsentieren `diff(f, x1, x2)` und `diff(f, x2, x1)` verschiedene Objekte. Siehe Beispiel ??.
 - ☞ Anwender können die Funktionalität von `diff` für ihre eigenen speziellen mathematischen Funktionen durch Überladung erweitern. Dazu macht man aus der entsprechenden Funktion eine Funktionsumgebung und implementiert die Ableitungsregel für die Funktion als "diff"-Slot der Funktionsumgebung. Siehe Beispiel ??.
 - ☞ MuPAD hat zwei Differentiations-Funktionen: `D` und `diff`. `D` darf nur auf Funktionen angewendet werden, wohingegen `diff` dazu verwendet wird, um Ausdrücke zu differenzieren. `D`-Ausdrücke können mit `rewrite` in `diff`-Ausdrücke umgeschrieben werden. Siehe Beispiel ??.
 - ☞ `diff` ist eine Funktion des Systemkerns.
-

Beispiel 1. Wir berechnen die Ableitung von x^2 nach x :

```
>> diff(x^2, x)
2 x
```

Beispiel 2. Indizierte Bezeichner dürfen als Differentiationsvariablen benutzt werden, wenn der Index eine ganze Zahl ist:

```
>> diff(x[1]*y + x[1]*x[r], x[1])
y + x[r]
```

Wenn der Index keine ganze Zahl ist, dann wird ein symbolischer `diff`-Aufruf zurückgeliefert:

```
>> diff(x[1]*y + x[1]*x[r], x[r])
diff(y x[1] + x[r] x[1], x[r])
```

Beispiel 3. Man kann nach mehr als einer Variablen in einem einzigen `diff`-Aufruf differenzieren. Im folgenden Beispiel wird zuerst nach x und dann nach y differenziert:

```
>> diff(x^2*sin(y), x, y) = diff(diff(x^2*sin(y), x), y)
2 x cos(y) = 2 x cos(y)
```

Beispiel 4. Durch Verwendung des Sequenz-Operators `$` berechnet man die dritte Ableitung des folgenden Ausdrucks nach x :

```
>> diff(sin(x)*cos(x), x $ 3)
4 sin(x) - 4 cos(x)
```

Beispiel 5. Polynome können nach den Polynom-Unbestimmten (hier x) und nach freien Parametern (hier a) differenziert werden:

```
>> diff(poly(sin(a)*x^3 + 2*x, [x]), x)
poly((3 sin(a)) x^2 + 2, [x])
>> diff(poly(sin(a)*x^3 + 2*x, [x]), a)
poly(cos(a) x^3, [x])
```

Beispiel 6. Die Ableitung einer unbekannte Funktion wird als symbolischer `diff`-Aufruf zurückgeliefert:

```
>> diff(f(x) + x, x)

diff(f(x), x) + 1
```

Beispiel 7. Geschachtelte `diff`-Aufrufe werden intern in einen `diff`-Aufruf mit mehreren Argumenten konvertiert:

```
>> diff(diff(f(x, y), x), y)

diff(f(x, y), x, y)
```

Beispiel 8. Die Ableitung nach mehreren Unbestimmten kommutiert nicht:

```
>> diff(f(x, y), x, y) = diff(f(x, y), y, x);
    bool(%)

diff(f(x, y), x, y) = diff(f(x, y), y, x)

FALSE
```

Beispiel 9. `D` darf nur auf Funktionen angewendet werden, wohingegen `diff` auf Ausdrücke angewendet wird:

```
>> D(sin), diff(sin(x), x)

cos, cos(x)
```

Die Anwendung von `D` auf Ausdrücke und von `diff` auf Funktionen macht keinen Sinn:

```
>> D(sin(x)), diff(sin, x)

D(sin(x)), 0
```

Mit `rewrite` kann man Ausdrücke mit `D` in `diff`-Ausdrücke umschreiben:

```
>> rewrite(D(f)(y), diff), rewrite(D(D(f))(y), diff)

diff(f(y), y), diff(f(y), y, y)
```

Beispiel 10. Fortgeschrittene Benutzer können die Funktion `diff` für ihre eigenen speziellen mathematischen Funktionen erweitern (siehe „Hintergründe“ unten). Zu diesem Zweck bettet man die mathematische Funktion in eine Funktionsumgebung `g` ein und implementiert deren Slot `"diff"`, der das Verhalten der Funktion `diff` für die Funktionsumgebung `g` beschreibt.

Enthält nun ein Ausdruck `f` einen Teilausdruck der Form `g(u, ...)`, so bewirkt die Funktion `diff` den Aufruf `g::diff(g(u, ...), x)` der Slot-Routine, um die Ableitung von `g(u, ...)` bezüglich `x` zu bestimmen.

Wir zeigen diese Funktionsweise am Beispiel der Exponential-Funktion. Die Funktionsumgebung `exp` besitzt natürlich schon den Slot `"diff"`. Wir nennen unsere Funktionsumgebung daher `Exp`, um nicht die Systemfunktion `exp` zu überschreiben.

Dieser Beispiel-`"diff"`-Slot implementiert die Kettenregel für die Exponential-Funktion. Die Ableitung ist der ursprüngliche Aufruf der Exponential-Funktion mal der inneren Ableitung:

```
>> Exp := funcenv(Exp):
      Exp::diff := (f, x) -> f*diff(op(f, 1), x):
      delete x: diff(Exp(x^2), x)
```

$$2 \quad 2$$

$$2 \times \text{Exp}(x)$$

`prog::trace` zeigt hier, dass die Funktion nur mit zwei Argumenten aufgerufen wird. `Exp::diff` wird nur einmal aufgerufen, weil der zweite notwendige Aufruf aus einem internen Cache für Zwischenergebnisse in `diff` gelesen wird:

```
>> prog::trace(Exp::diff):
      diff(Exp(x^2), x, x)

enter 'Exp::diff'                                with args    : Exp(x^2), x
leave 'Exp::diff'                                with result  : 2*x*Exp(x^2)
```

$$2 \quad 2 \quad 2$$

$$2 \text{Exp}(x) + 4 x \text{Exp}(x)$$

```
>> prog::untrace(Exp::diff): delete f, Exp:
```

Hintergründe:

☞ Wenn in `f` ein Teilausdruck `g(u, ...)` vorkommt, wobei `g` eine Funktionsumgebung ist, so versucht `diff` den Slot `"diff"` von `g` aufzurufen, um `g(u, ...)` zu differenzieren. Damit läßt sich die Funktionalität von `diff` für benutzereigene mathematische Funktionen erweitern.

Der Slot `"diff"` von `g` wird mit den Argumenten `g(u, ...)`, `x` aufgerufen.

Falls g keinen Slot "diff" hat, so gibt `diff` das Objekt `diff(g(u, . . .), x)` für den entsprechenden Teilausdruck zurück.

Der "diff"-Slot wird immer mit genau zwei Argumenten aufgerufen. Wenn im `diff`-Aufruf mehr Unbestimmte gegeben sind, dann werden mehrere Aufrufe des "diff"-Slots durchgeführt. Die Ergebnisse der Aufrufe von "diff"-Slots werden in `diff` zwischengespeichert, um redundante Funktionsaufrufe einzusparen. Siehe Beispiel ??.

- ☞ Entsprechendes gilt für Domainelemente: Tritt ein Domainelement d eines Bibliothekdomains T als Teilausdruck in f auf, so versucht `diff` den Slot "diff" dieses Domains mit dem Element d und der Unbestimmten x als Argumenten aufzurufen, um die Ableitung von d bezüglich x zu bestimmen.

Falls das Domain T keinen Slot "diff" hat, so betrachtet `diff` das Objekt als konstant und gibt für den entsprechenden Teilausdruck 0 zurück.

Änderungen:

- ☞ `diff` akzeptiert keine Prozeduren mehr als erstes Argument.
-

dilog – die Dilogarithmusfunktion

`dilog(x)` stellt die Dilogarithmusfunktion $\int_1^x \ln(t)/(1-t) dt$ dar.

Aufruf(e):

- ☞ `dilog(x)`

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `ln`, `polylog`

Details:

☞ Wenn x eine Gleitkommazahl ist, dann liefert `dilog(x)` den numerischen Wert der Dilogarithmusfunktion. Die speziellen Werte

$$\text{dilog}(-1) = \pi^2/4 - i\pi \ln(2),$$

$$\text{dilog}(0) = \pi^2/6,$$

$$\text{dilog}(1/2) = \pi^2/12 - \ln(2)^2/2,$$

$$\text{dilog}(1) = 0,$$

$$\text{dilog}(2) = -\pi^2/12,$$

$$\text{dilog}(I) = \pi^2/16 - i \text{CATALAN} - i\pi \ln(2)/4,$$

$$\text{dilog}(-I) = \pi^2/16 + i \text{CATALAN} + i\pi \ln(2)/4,$$

$$\text{dilog}(1+I) = -\pi^2/48 - i \text{CATALAN},$$

$$\text{dilog}(1-I) = -\pi^2/48 + i \text{CATALAN},$$

$$\text{dilog}(\text{infinity}) = -\text{infinity}$$

sind implementiert. Für alle anderen Argumente wird ein symbolischer Funktionsaufruf zurückgeliefert.

☞ Für exakte numerische Argumente vom Typ `Type::Numeric` mit negativem Realteil oder vom Betrag größer als 1 wird das Ergebnis mittels funktionaler Identitäten umgeschrieben. Siehe Beispiel ??.

☞ `dilog(x)` stimmt mit `polylog(2, 1-x)` überein.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> dilog(0), dilog(2/3), dilog(sqrt(2)), dilog(1 + I), dilog(x)
```

$$\begin{array}{ccc} \frac{2}{6} \pi^2 & \frac{1}{2} & \frac{2}{48} \pi^2 \\ \text{---}, \text{dilog}(2/3), \text{dilog}(2^{1/2}), -I \text{CATALAN} - \text{---}, \text{dilog}(x) & & \end{array}$$

Für Gleitkommazahlen wird der numerische Wert von `dilog` berechnet:

```
>> dilog(-1.2), dilog(3.4 - 5.6*I)
```

$$2.458586602 - 2.477011851 I, - 2.529187195 + 2.25273709 I$$

Beispiel 2. Aus ganzen oder rationalen Zahlen aufgebaute Argumente werden umgeschrieben, wenn sie in der linken komplexen Halbebene liegen oder ihr Betrag größer als 1 ist. Die folgenden Argumente haben einen negativen Realteil:

```
>> dilog(-400/3), dilog(-1/2 + I)
```

$$\frac{\pi^2}{6} + \operatorname{dilog}\left(\frac{3}{403}\right) + \frac{\ln^2(403/3)}{2} - \ln(403/3) (\pi + \ln(400/3))$$

$$, \frac{\pi^2}{6} + \frac{\ln^2(3/2 - i)}{2} + \operatorname{dilog}\left(\frac{6}{13} + \frac{4}{13}i\right) - \ln(-1/2 + i) \ln(3/2 - i)$$

Die folgenden Argumente haben einen Betrag größer als 1:

```
>> dilog(31/30), dilog(1 + 2/3*I)
```

$$- \operatorname{dilog}(30/31) - \frac{\ln^2(31/30)}{2}, - \frac{\ln^2(1 + 2/3 i)}{2} - \operatorname{dilog}(9/13 - 6/13 i)$$

Beispiel 3. Die negative reelle Halbachse ist ein Verzweigungsschnitt von `dilog`. Beim Überschreiten des Schnitts am reellen Punkt $x < 0$ springen die Funktionswerte um $2\pi i \ln(1 - x)$:

```
>> dilog(-1.2), dilog(-1.2 + I/10^100), dilog(-1.2 - I/10^100)
```

$$2.458586602 - 2.477011851 i, 2.458586602 - 2.477011851 i,$$

$$2.458586602 + 2.477011851 i$$

Beispiel 4. Die Funktionen `diff`, `float`, `limit` und `series` verarbeiten `dilog`:

```
>> diff(dilog(x), x, x, x), float(ln(3 + dilog(sqrt(PI))))
```


Aufruf(e):

⌘ `dirac(x)`
 ⌘ `dirac(x, n)`

Parameter:

`x` — ein arithmetischer Ausdruck
`n` — ein arithmetischer Ausdruck, der eine nichtnegative ganze Zahl repräsentiert

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Seiteneffekte: `dirac` reagiert auf Eigenschaften von Bezeichnern.

Verwandte Funktionen: `heaviside`

Details:

- ⌘ Die Aufrufe `dirac(x, 0)` und `dirac(x)` sind äquivalent.
 - ⌘ Für nicht-verschwindende reelle Argumente `x` wird 0 zurückgeliefert. Für nicht-reelles `x` vom Domain-Typ `DOM_COMPLEX` wird `undefined` zurückgeliefert. Für alle anderen Argumente wird ein symbolischer Funktionsaufruf zurückgeliefert.
 - ⌘ `dirac` hat keinen vordefinierten Wert am Ursprung. Mit

```
unprotect(dirac):  dirac(0) := myValue:
```

und

```
dirac(float(0)) := myFloatValue:  protect(dirac):
```

können gewünschte Werte wie z. B. `infinity` gesetzt werden.
 - ⌘ Für univariate lineare Ausdrücke ist die Vereinfachung
$$\delta^{(n)}(ax - b) = \frac{\text{sign}(a)}{a^{n+1}} \delta^{(n)}\left(x - \frac{b}{a}\right)$$
implementiert, falls `a` ein reeller numerischer Wert ist.
 - ⌘ Der Integrierer `int` verarbeitet `dirac` als die übliche Delta-Distribution. Siehe Beispiel ??.
-

Beispiel 1. `dirac` liefert 0 für Argumente, die nicht-verschwindende reelle Zahlen darstellen:

```
>> dirac(-3), dirac(3/2), dirac(2.1, 1),
    dirac(3*PI), dirac(sqrt(3), 3)

0, 0, 0, 0, 0
```

Argumente vom Domain-Typ `DOM_COMPLEX` liefern `undefined`:

```
>> dirac(1 + I), dirac(2/3 + 7*I), dirac(0.1*I, 1)

undefined, undefined, undefined
```

Für andere Argumente wird ein symbolischer Funktionsaufruf zurückgeliefert:

```
>> dirac(0), dirac(x), dirac(ln(-5)), dirac(x + I, 2), di-
rac(x, n)

dirac(0), dirac(x), dirac(I PI + ln(5)), dirac(x + I, 2),

dirac(x, n)
>> dirac(2*x - 1, n)
```

$$\frac{\text{dirac}(x - 1/2, n)}{n + 1}$$

Eine natürlicher „Funktionswert“ für `dirac(0)` ist `infinity`:

```
>> unprotect(dirac): dirac(0) := infinity: dirac(0)

infinity
>> delete dirac(0): protect(dirac): dirac(0)

dirac(0)
```

Beispiel 2. `dirac` berücksichtigt durch `assume` gesetzte Annahmen:

```
>> assume(x < 0): dirac(x)

0
>> assume(x, Type::Real): assume(x <> 0, _and): dirac(x)

0
>> unassume(x):
```

Beispiel 3. Der symbolische Integrierer `int` verarbeitet `dirac` als Delta-Distribution:

```
>> int(f(x)*dirac(x - y^2), x = -infinity..infinity)

      2
    f(y )

>> int(int(f(x, y)*dirac(x - y^2), x = -infinity..infinity),
      y = -1..1)

      2
    int(f(y , y), y = -1..1)
```

Die Stammfunktion von `dirac` wird mittels der Sprungfunktion `heaviside` dargestellt:

```
>> int(f(x)*dirac(x), x), int(f(x)*dirac(x, 1), x)

heaviside(x) f(0), dirac(x) f(0) - heaviside(x) D(f)(0)
```

Das Resultat enthält einen symbolischen Aufruf `heaviside(0)`, wenn die Delta-Spitze auf dem Rand des Integrationsbereichs liegt:

```
>> int(f(x)*dirac(x - 3), x = -1..3)

f(3) heaviside(0)
```

Man beachte, dass `int` die Distribution nur verarbeiten kann, wenn das Argument von `dirac` linear in der Integrationsvariablen ist:

```
>> int(f(x)*dirac(2*x - 3), x = -10..10),
    int(f(x)*dirac(x^2), x = -10..10)

      f(3/2)                2
    -----, int(f(x) dirac(x ), x = -10..10)
      2
```

Man beachte weiterhin, dass `dirac` nicht bei numerischer Integration eingesetzt werden darf. Numerisch arbeitende Algorithmen können die Delta-Spitze meist nicht entdecken:

```
>> numeric::int(dirac(x - 3), x = -10..10)

0.0
```

Änderungen:

- ⌘ Durch das zweite Argument können nun auch Ableitungen der Distribution dargestellt werden. Durch `assume` gesetzte Eigenschaften von Bezeichnern werden nun berücksichtigt. Das Zusammenspiel zwischen `dirac` und `int` wurde verbessert.
-

`discont` – Unstetigkeitsstellen einer Funktion

`discont(f, x)` berechnet die Menge aller Unstetigkeitsstellen der Funktion $f(x)$.

`discont(f, x=a..b)` berechnet die Menge aller Unstetigkeitsstellen von $f(x)$ im Intervall $[a, b]$.

Aufruf(e):

- ⌘ `discont(f, x)`
- ⌘ `discont(f, x, F)`
- ⌘ `discont(f, x = a..b)`
- ⌘ `discont(f, x = a..b, F)`

Parameter:

- `f` — eine arithmetischer Ausdruck, der eine Funktion in `x` repräsentiert
- `x` — ein Bezeichner
- `F` — `Dom::Real` oder `Dom::Complex`
- `a, b` — Intervallgrenzen: arithmetische Ausdrücke

Rückgabewert: `discont` liefert eine Menge zurück – siehe die Hilfeseite von `solve` für einen Überblick über alle Mengentypen – oder den unevaluierten Aufruf von `discont`.

Seiteneffekte: `discont` reagiert auf Eigenschaften freier Parameter in `f` sowie `a` und `b`. `discont` reagiert in manchen Fällen auf Eigenschaften von `x`.

Überladbar durch: `f`

Verwandte Funktionen: `limit`, `solve`

Details:

- ⌘ `discont(f, x, F)` liefert eine Menge von Zahlen zurück, die alle Unstetigkeitsstellen von f enthält, wobei f als Funktion von x aufgefasst

wird, die auf F definiert ist. Zu beachten ist, dass nicht jede reelle Zahl, die Unstetigkeitsstelle einer komplexen Funktion ist, auch Unstetigkeitsstelle der Einschränkung dieser Funktion auf die reellen Zahlen sein muss, z. B. kann ja der Verzweigungsschnitt einer Funktion auf der reellen Achse liegen, wie etwa im Beispiel ?? unten.

- ☞ Der Begriff „Unstetigkeitsstellen“ umfasst auch Definitionslücken.
- ☞ Wird der Parameter F nicht angegeben, so wird f als auf den komplexen Zahlen definiert angesehen; dies gilt nicht, falls vorher die globale Annahme `assume(Global, Type::Real)` gemacht wurde.
- ☞ Wird ein Bereich $a..b$ angegeben, so wird der Durchschnitt der Menge der Unstetigkeitsstellen mit dem Intervall $[a, b]$ zurückgeliefert.
- ☞ Die von `discont` zurückgelieferte Menge kann eine echte Obermenge der Menge aller Unstetigkeitsstellen von f sein. Siehe Beispiel ??.
- ☞ Kann `discont` die Menge der Unstetigkeitsstellen nicht berechnen, so liefert es einen symbolischen Aufruf von `discont` zurück; siehe Beispiel ??.
- ☞ `discont` kann erweitert werden, so dass es auch die Unstetigkeitsstellen benutzerdefinierter Funktionen erkennt. Zu diesem Zweck muss die benutzerdefinierte Funktion in eine Funktionsumgebung eingebettet und die Menge der reellen bzw. komplexen Diskontinuitäten dem Attribut `"realDiscont"` bzw. `"complexDiscont"` zugewiesen werden. Siehe `solve` für einen Überblick über die verschiedenen Mengentypen. Siehe auch Beispiel ??.

Beispiel 1. Die Gammafunktion hat Polstellen bei allen ganzen Zahlen kleiner oder gleich Null. Daher hat $x \rightarrow \text{gamma}(x/2)$ Polstellen bei allen geraden Zahlen, die kleiner oder gleich Null sind:

```
>> discont(gamma(x/2), x)
      { 2*x3 | x3 in Z_ } intersect ]-infinity, 0]
```

Beispiel 2. Die Logarithmusfunktion hat auf der negativen reellen Achse einen Verzweigungsschnitt, ist dort also unstetig. Ihre Einschränkung auf die reellen Zahlen hingegen ist an jeder Stelle (außer Null) stetig.

```
>> discont(ln(x), x), discont(ln(x), x, Dom::Real)
      ]-infinity, 0], {0}
```

Beispiel 3. Wird ein Bereich angegeben, so werden nur die Unstetigkeitsstellen in diesem Bereich zurückgeliefert.

```
>> discount(1/x/(x - 1), x = 0..1/2)

{0}
```

Beispiel 4. Ein Bereich kann beliebige arithmetische Ausdrücke als Grenzen haben. Es wird nicht implizit angenommen, dass die rechte Grenze größer oder gleich der linken ist.

```
>> discount(1/x, x = a..b)

piecewise({0} if a <= 0 and 0 <= b,

          {} if (not a <= 0 or not 0 <= b))
```

Beispiel 5. Wie aus dem vorigen Beispiel ersichtlich, reagiert `discount` auf die Eigenschaften freier Bezeichner (weil `piecewise` dies tut). Das Ergebnis hängt auch von den Eigenschaften von `x` ab: Werte, die `x` auf Grund seiner Eigenschaften nicht annehmen kann, werden manchmal weggelassen:

```
>> assume(x>0):
    discount(1/x, x)

{}

>> delete x:
```

Beispiel 6. Manchmal liefert `discount` eine echte Obermenge der Menge aller Unstetigkeitsstellen:

```
>> discount(piecewise([x<>0, x*sin(1/x)], [x=0, 0]), x)

{0}
```

Beispiel 7. `discount` liefert einen unevaluierten Aufruf von sich selbst zurück, falls es die Unstetigkeitsstellen einer gegebenen Funktion nicht bestimmen kann.

```
>> delete f: discount(f(x), x)
```

```
discont(f(x), x)
```

Die nötige Information kann zur Verfügung gestellt werden, indem zu f ein Eintrag hinzugefügt wird. `discont` sorgt dafür, dass f auch innerhalb komplizierterer Ausdrücke korrekt behandelt wird.

```
>> f := funcenv(x->procname(x)): f::complexDiscont:={1}:
      discont(f(sin(x)), x=-4..34)
```

```

{ PI   5 PI   9 PI   13 PI   17 PI   21 PI }
{ --, ----, ----, -----, -----, ----- }
{ 2    2    2    2    2    2    }
```

Beispiel 8. Wir wollen eine eigene Funktion schreiben, die den Logarithmus zur Basis 2 implementiert. Der Einfachheit halber legen wir fest, dass sie immer den unevaluierten Aufruf zurückliefert. Der Logarithmus hat einen Verzweigungsschnitt auf der negativen reellen Achse; seine Einschränkung auf die reelle Achse ist außerhalb des Nullpunkts stetig:

```
>> binlog := funcenv(x -> procname(x)):
      binlog::realDiscont := {0}:
      binlog::complexDiscont := Dom::Interval(-infinity, [0]):
      discont(binlog(x), x=-2..2);
      discont(binlog(x), x=-2..2, Dom::Real)
```

```
[-2, 0]
```

```
{0}
```

Änderungen:

- ⌘ Ein drittes Argument wurde eingeführt, um reelle und komplexe Unstetigkeitsstellen unterscheiden zu können.

div – der ganzzahlige Anteil eines Quotienten

$x \operatorname{div} m$ repräsentiert die ganze Zahl q , die $x = qm + r$ mit $0 \leq r < |m|$ erfüllt.

Aufruf(e):

- ⌘ $x \operatorname{div} m$
- ⌘ `_div(x, m)`

Parameter:

x, m — ganze Zahlen oder symbolische arithmetische Ausdrücke; m darf nicht 0 sein.

Rückgabewert: eine ganze Zahl oder ein arithmetischer Ausdruck vom Typ `"_div"`.

Überladbar durch: x, m

Verwandte Funktionen: `_mod, /, divide, mod, modp, mods`

Details:

☞ Für positives x und m ist $q = x \text{ div } m$ der ganzzahlige Anteil des Quotienten x/m , d. h., $q = \text{trunc}(x/m)$.

☞ $x \text{ div } m$ ist äquivalent zum Funktionsaufruf `_div(x, m)`.

☞ Sind sowohl x und m ganze Zahlen, so wird eine ganze Zahl zurückgeliefert. Evaluiert x oder m nicht zu einer Zahl, so wird ein symbolischer Ausdruck vom Typ `"_div"` zurückgeliefert. Ist x oder m eine Zahl, aber keine ganze Zahl, so wird ein Fehler ausgelöst.

☞ `div` operiert nicht auf Polynomen. Man benutze `divide`.

☞ `div` ist eine Funktion des Systemkerns.

Beispiel 1. Mit der Voreinstellung von `mod` gilt die Identität $(x \text{ div } m) * m + (x \text{ mod } m) = x$ für ganze Zahlen x und m :

```
>> 43 div 13 = trunc(43/13), 43 mod 13 = frac(43/13) * 13
```

$$3 = 3, 4 = 4$$

```
>> (43 div 13) * 13 + (43 mod 13) = 43
```

$$43 = 43$$

Beispiel 2. Ist x oder m nicht zu einer Zahl, so ergibt sich ein symbolischer Ausdruck vom Typ `"_div"`:

```
>> 43 div m, x div 13, x div m
```

$$43 \text{ div } m, x \text{ div } 13, x \text{ div } m$$

```
>> type(x div m)
```

"_div"

Sind x und m Zahlen, so müssen dies ganze Zahlen sein:

```
>> 1/2 div 2
```

```
Error: Illegal argument in div or mod
```

```
>> x div 2.0
```

```
Error: Illegal operand [_mod]
```

Änderungen:

☞ Keine Änderungen.

divide – Division von Polynomen

`divide(p, q)` dividiert die univariaten Polynome p und q . Der Quotient s und der Rest r mit $p = sq + r$, $\text{degree}(r) < \text{degree}(q)$ werden zurückgeliefert.

Aufruf(e):

```
☞ divide(p1, q1 <, mode>)
☞ divide(f1, g1 <, mode>)
☞ divide(f, g <, [x]> <, mode>)
☞ divide(p, q, Exact)
☞ divide(f, g, <[x1, x2, ...],> Exact)
```

Parameter:

$p1, q1$	— univariate Polynome vom Typ <code>DOM_POLY</code> .
$f1, g1$	— univariate polynomiale Ausdrücke
p, q	— univariate oder multivariate Polynome vom Typ <code>DOM_POLY</code> .
f, g	— univariate oder multivariate polynomiale Ausdrücke
x	— ein Bezeichner oder ein indizierter Bezeichner. Ausdrücke werden als univariate Polynome in der Unbestimmten x aufgefasst.
$x1, x2, \dots$	— Bezeichner oder indizierte Bezeichner. Multivariate Ausdrücke werden als multivariate Polynome in diesen Unbestimmten aufgefasst.

Optionen:

- `mode` — entweder *Quo* or *Rem*. Mit *Quo* wird nur der Quotient *s* geliefert, mit *Rem* nur der Rest *r*.
- Exact* — exakte Division von multivariaten Polynomen. Nur der Quotient *s* wird zurückgeliefert. Ist keine exakte Division ohne Rest möglich, wird `FAIL` zurückgeliefert.

Rückgabewert: ein Polynom, ein polynomialer Ausdruck, eine Folge von zwei Polynomen bzw. polynomialen Ausdrücken oder der Wert `FAIL`.

Überladbar durch: *p*, *q*, *p1*, *q1*, *f*, *g*, *f1*, *g1*

Verwandte Funktionen: `/`, `content`, `degree`, `div`, `factor`, `gcd`, `gcdex`, `groebner::normalf`, `ground`, `mod`, `multcoeffs`, `pdivide`, `poly`, `powermod`

Details:

- ☞ `divide(p, q)` dividiert die univariaten Polynome *p* und *q*. Der Quotient *s* und der Rest *r* mit $p = s \cdot q + r$ und $\text{degree}(r) < \text{degree}(q)$ werden berechnet. Ist keine Option angegeben, wird die Folge *s*, *r* zurückgegeben.
 - ☞ Die ersten beiden Argumente können Polynome oder polynomialen Ausdrücke sein.

Polynome müssen vom gleichen Typ sein, d. h., ihre Unbestimmten und ihre Koeffizientenringe müssen übereinstimmen.

Ausdrücke werden intern in Polynome konvertiert (siehe die Funktion `poly`). Wird keine Liste von Unbestimmten angegeben, so werden automatisch alle symbolischen Variablen in den Ausdrücken als Unbestimmte gewählt. `FAIL` wird zurückgeliefert, wenn die Ausdrücke nicht in Polynome umgewandelt werden können.

Die Rückgabepolynome sind vom gleichen Typ wie die ersten beiden Argumente, d. h., es werden entweder Polynome vom Typ `DOM_POLY` oder polynomialen Ausdrücke zurückgegeben.
 - ☞ Der Koeffizientenring der Polynome muss eine `"_divide"`-Methode implementieren. Intern werden Koeffizienten mit dieser Methode dividiert. Können Koeffizienten nicht dividiert werden, so muss diese Methode `FAIL` liefern.
 - ☞ `divide` ist eine Funktion des Systemkerns.
-

Beispiel 1. Ohne weitere Optionen liefert `divide` den Quotienten und den Rest der Division von univariaten Polynomen:

```
>> divide(poly(x^3 + x + 1, [x]), poly(x^2 + x + 1, [x]))
      poly(x - 1, [x]), poly(x + 2, [x])
>> divide(x^3 + x + 1, x^2 + x + 1)
      x - 1, x + 2
```

Beispiel 2. Wenn Ausdrücke mehr als eine Variable enthalten, dann müssen die Unbestimmten angegeben werden. Andere symbolische Objekte werden als Parameter betrachtet. Die Option *Quo* weist `divide` an, nur den Quotienten zurück zu liefern:

```
>> divide(a*x^3 + x + 1, x^2 + x + 1, [x], Quo)
      a x - a
```

Die Option *Rem* weist `divide` an, nur den Rest zurück zu liefern:

```
>> divide(a*x^3 + x + 1, x^2 + x + 1, [x], Rem)
      a + x + 1
```

Beispiel 3. Für multivariate Ausdrücke, die als univariate Polynome in einer angegebenen Unbestimmten betrachtet werden, hängt das Ergebnis der Division von der Unbestimmten ab:

```
>> divide(x^2 - 2*x - y, y*x - 1, [x]);
      1      1
      - - 2 - - 2
      x  y      y
      - + ----, ---- - y
      y      y      y
>> divide(x^2 - 2*x - y, y*x - 1, [y])
      1  2  1
      - -, x - - - 2 x
      x      x
```

Beispiel 4. Multivariate Polynome und polynomiale Ausdrücke können nur mit der Option *Exact* dividiert werden. Wenn eine Division ohne Rest möglich ist, wird der Quotient zurückgeliefert. Diese Operation ist äquivalent zur Division von Polynomen mit dem */*-Operator:

```
>> p := poly(x^2 - x*y - x + y, [x, y]): q := poly(x - 1, [x, y]):
    p/q = divide(p, q, Exact)
```

```
poly(x - y, [x, y]) = poly(x - y, [x, y])
```

FAIL wird zurückgeliefert, wenn exakte Division ohne Rest nicht möglich ist:

```
>> p := poly(x^2 + y, [x, y]): q := poly(x - 1, [x, y]):
    divide(p, q, Exact) = p/q
```

```
FAIL = FAIL
```

```
>> delete p, q:
```

Änderungen:

⌘ Keine Änderungen.

domtype – der Datentyp eines Objekts

`domtype(object)` liefert den Domain-Typ (Datentyp) des Objektes `object`.

Aufruf(e):

⌘ `domtype(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: der Datentyp, d. h., ein Objekt vom Typ `DOM_DOMAIN`.

Überladbar durch: `object`

Verwandte Funktionen: `coerce`, `DOM_DOMAIN`, `domain`, `hastype`, `testtype`, `type`, `Type`

Details:

- ⌘ Für die meisten Datentypen stimmt der von `domtype` gelieferte Domain-Typ mit dem von der Funktion `type` gelieferten Typ überein. Lediglich für „Ausdrücke“ vom Domain-Typ `DOM_EXPR` liefert `type` eine verfeinerte Unterscheidung gemäß des 0-ten Operanden. Siehe Beispiel ??.
 - ⌘ Im Gegensatz zu den meisten anderen Funktionen werden als Argument an `domtype` übergebene Ausdrucksfolgen nicht ausgeglichen.
 - ⌘ `domtype` ist eine Funktion des Systemkerns.
-

Beispiel 1. Reelle Gleitpunktzahlen sind vom Domain-Typ `DOM_FLOAT`:

```
>> domtype(12.345)
```

```
DOM_FLOAT
```

Komplexe Zahlen sind vom Domain-Typ `DOM_COMPLEX`. Die Operanden können ganze Zahlen (`DOM_INT`), rationale Zahlen (`DOM_RAT`) oder Gleitpunktzahlen (`DOM_FLOAT`) sein. Auf die Operanden kann mit `op` zugegriffen werden:

```
>> domtype(1 - 2*I), op(1 - 2*I);  
domtype(1/2 - I), op(1/2 - I);  
domtype(2.0 - 3.0*I), op(2.0 - 3.0*I)
```

```
DOM_COMPLEX, 1, -2
```

```
DOM_COMPLEX, 1/2, -1
```

```
DOM_COMPLEX, 2.0, -3.0
```

Beispiel 2. Ausdrücke sind Objekte vom Domain-Typ `DOM_EXPR`. Der Typ von Ausdrücken kann mit der Funktion `type` genauer bestimmt werden:

```
>> domtype(x + y), type(x + y);  
domtype(x - 1.0*I), type(x - 1.0*I);  
domtype(x*I), type(x*I);  
domtype(x^y), type(x^y);  
domtype(x[i]), type(x[i])
```

```

DOM_EXPR, "_plus"

DOM_EXPR, "_plus"

DOM_EXPR, "_mult"

DOM_EXPR, "_power"

DOM_EXPR, "_index"

```

Beispiel 3. `domtype` evaluiert sein Argument. In diesem Beispiel wird erst die Zuweisung evaluiert, dann wird `domtype` auf den Rückgabewert der Zuweisung angewendet. Das ist die rechte Seite der Zuweisung, also 5:

```

>> domtype((a := 5))

DOM_INT

>> delete a:

```

Beispiel 4. Hier wird der Bezeichner `a` erst zu der Ausdrucksfolge `3, 4` evaluiert. Ihr Domain-Typ ist `DOM_EXPR`, ihr Typ ist `"_exprseq"`:

```

>> a := 3, 4: domtype(a), type(a)

DOM_EXPR, "_exprseq"

>> delete a:

```

Beispiel 5. `factor` erzeugt Objekte vom Datentyp `Factored`:

```

>> domtype(factor(x^2 - x))

Factored

```

Beispiel 6. `matrix` erzeugt Objekte vom Datentyp `Dom::Matrix()`:

```

>> domtype(matrix([[1, 2], [3, 4]]))

Dom::Matrix()

```

Beispiel 7. Domains sind vom Datentyp `DOM_DOMAIN`:

```
>> domtype(DOM_INT), domtype(DOM_DOMAIN)

DOM_DOMAIN, DOM_DOMAIN
```

Beispiel 8. `domtype` ist überladbar, d. h., dass ein Domain vorgeben kann, von einem anderen Datentyp zu sein. Der spezielle Slot "dom" liefert immer das wirkliche Domain:

```
>> d := newDomain("d"): d::domtype := x -> "domain type d":
    e := new(d, 1): e::dom, type(e), domtype(e)

d, d, "domain type d"

>> delete d, e:
```

Änderungen:

☞ Keine Änderungen.

end – Abschließen einer Block-Anweisung

`end` ist ein Schlüsselwort, das der Parser je nach Kontext wie eines der Folgenden behandelt:

- `end_case`
- `end_for`
- `end_if`
- `end_proc`
- `end_repeat`
- `end_while`

Verwandte Funktionen: `end_case`, `end_for`, `end_if`, `end_proc`, `end_repeat`, `end_while`

Beispiel 1. Die Schlüsselworte `proc`, `case`, `if`, `for`, `repeat` und `while` öffnen in der MuPAD-Sprache Block-Konstrukte. Diese semantischen Blöcke können mit `end` oder mit den spezialisierten Kommandos `end_proc`, `end_case` etc. geschlossen werden:

```
>> f :=
  proc(a, b)
    local i;
    begin
      for i from a to b do
        if isprime(i) then
          print(Unquoted, expr2text(i)." is a prime")
        end
      end
    end:
  end:

>> f(20, 30):

                23 is a prime

                29 is a prime
```

Der Parser übersetzt `end` zum dem Block entsprechenden Schlüsselwort:

```
>> expose(f)

      proc(a, b)
        name f;
        local i;
        begin
          for i from a to b do
            if isprime(i) then
              print(Unquoted, expr2text(i)." is a prime")
            end_if
          end_for
        end_proc
      end:

>> delete f:
```

Änderungen:

☞ `end` ist ein neues Schlüsselwort.

erf, erfc – die Fehlerfunktion und die komplementäre Fehlerfunktion

$\text{erf}(x)$ stellt die Fehlerfunktion $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ dar. Die komplementäre Fehlerfunktion ist $\text{erfc}(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$.

Aufruf(e):

- ☞ `erf(x)`
- ☞ `erfc(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Seiteneffekte: Für Gleitpunktargumente reagieren die Funktionen auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Details:

- ☞ Diese Funktionen sind für alle komplexen Argumente definiert.
- ☞ Für Gleitpunktargumente werden Gleitpunktwerte berechnet. Die exakten Werte
 $\text{erf}(0) = 0, \text{erf}(\text{infinity}) = 1, \text{erf}(-\text{infinity}) = -1,$
 $\text{erfc}(0) = 1, \text{erfc}(\text{infinity}) = 0, \text{erfc}(-\text{infinity}) = 2$
sind implementiert. Für alle anderen Argumente werden symbolische Funktionsaufrufe zurückgeliefert.
- ☞ Für betragsmäßig große Argumente kann intern numerischer Unterlauf auftreten. Das durch $|\text{Im}(x)| \leq |\text{Re}(x)|/10$ gegebene Segment der komplexen Ebene ist hiergegen geschützt: für große positive Realteile kann `erfc` das Ergebnis zu 0.0 abschneiden, für große negative Realteile zu 2.0 runden. Mit $\text{erf}(x) = 1 - \text{erfc}(x)$ kann auch `erfc` entsprechend gerundete Werte liefern. Siehe Beispiel ??.
- ☞ Die float-Attribute sind Kernfunktionen, d. h., die Gleitpunktauswertung ist schnell.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> erf(0), erf(3/2), erf(sqrt(2)), erf(infinity)
                                1/2
0, erf(3/2), erf(2    ), 1
```



```
>> erfc(0), erfc(x + 1), erfc(-infinity)

1, erfc(x + 1), 2
```

Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> erf(-7.2), erf(2.0 + 3.5*I), erfc(100.0 + 100.0*I)

-1.0, 421.8123327 + 343.6612334 I,

0.0006523436638 - 0.003935726363 I
```

Beispiel 2. Für große Gleitpunktargumente mit positivem Realteil kann `erfc` das Ergebnis auf 0.0 abrunden:

```
>> erfc(2411.3), erfc(2411.4)

3.678326052e-2525152, 0.0
```

Ein solcher Schutz gegen numerischen Unterlauf ist für Argumente mit $|\operatorname{Im}(x)| \leq |\operatorname{Re}(x)|/10$ eingebaut:

```
>> erfc(2500.0 + 250.0*I)

0.0
```

Außerhalb dieser Region können Fehler ausgelöst werden:

```
>> erfc(2500.0 + 250.1*I)

Error: Overflow/underflow in arithmetical operation;
during evaluation of 'erfc::float'
```

Beispiel 3. Die Funktionen `diff`, `float`, `limit` und `series` verarbeiten die Fehlerfunktionen:

```
>> diff(erf(x), x, x, x), float(ln(3 + erfc(sqrt(PI)*I)))

      2      2      2
      8 x  exp(- x )  4 exp(- x )
      ----- - -----, 2.309003461 - 1.16207002 I
      1/2      1/2
      PI      PI

>> limit(x/(1 + x)*erf(x), x = infinity)

1
```

```
>> series(erfc(x), x = infinity, 4)
```

$$\frac{\exp(-x^2)}{\sqrt{\pi}} - \frac{\exp(-x^2)}{2\sqrt{\pi}} x^2 + O\left(\frac{\exp(-x^2)}{x^4}\right)$$

Hintergründe:

☞ erf und erfc sind ganze Funktionen.

Änderungen:

☞ Gleitpunktauswertung ist nun für alle komplexen Argument möglich. Für betragsmäßig große Argumente können gerundete Werte geliefert werden, um numerischen Unterlauf zu vermeiden.

error – Auslösen einer benutzerspezifisierten Fehlermeldung

`error(message)` bricht die aktuelle Prozedur ab, kehrt auf interaktive Ebene zurück und gibt die Fehlermeldung `message` aus.

Aufruf(e):

☞ `error(message)`

Parameter:

`message` — die Fehlermeldung: eine Zeichenkette

Seiteneffekte: Die Ausgabeformatierung von `error` ist von der Umgebungsvariablen `TEXTWIDTH` abhängig.

Verwandte Funktionen: `lasterror`, `prog::error`, `traperror`, `warning`

Details:

☞ Der Aufruf `error(message)` bricht die aktuelle Prozedur mit einem Fehler ab. Wenn der Fehler nicht mittels `traperror` von einer anderen Prozedur abgefangen wird, die die aktuelle Prozedur direkt oder indirekt aufgerufen hat, dann kehrt MuPAD auf die interaktive Ebene zurück und gibt die Zeichenkette `message` als Fehlertext aus.

- ⌘ Der ausgegebene Fehlertext hat die Form `Error: message [name]`, wobei `name` der Name der Prozedur ist, die den Aufruf von `error` enthält. Siehe Beispiele.
- ⌘ Fehler können mit der Funktion `traperror` abgefangen werden. Falls während der Evaluation der Argumente eines `traperror`-Aufrufs ein Fehler auftritt, so kehrt die Kontrolle zu der Prozedur zurück, die den Aufruf von `traperror` enthält, und nicht zur interaktiven Ebene. Eine Fehlermeldung wird nicht ausgegeben. Der Rückgabewert von `traperror` ist 1028, wenn ein mittels `error` ausgelöster Fehler abgefangen wird; siehe Beispiel ??.
- ⌘ Die Funktion `error` kann dazu verwendet werden, um in einem Typprüfungs-Abschnitt zu Beginn einer Prozedur einen Fehler auszulösen, wenn diese Prozedur mit falschen Argumenten aufgerufen wird.
- ⌘ `error` ist eine Funktion des Systemkerns.

Beispiel 1. Wenn der Divisor der folgenden einfachen Divisionsroutine 0 ist, dann wird ein Fehler ausgelöst:

```
>> mydivide := proc(n, d) begin
    if iszero(d) then
        error("Division by 0")
    end_if;
    n/d
end_proc:
mydivide(2, 0)
```

```
Error: Division by 0 [mydivide]
```

Beispiel 2. Wenn in der folgenden Prozedur `p` der Fehler ausgelöst wird, kehrt die Kontrolle unmittelbar zur interaktiven Ebene zurück. Der zweite Aufruf von `print` wird nie ausgeführt. Der Prozedurname wird mit der Fehlermeldung ausgegeben:

```
>> p := proc() begin
    print("entering procedure p");
    error("oops");
    print("leaving procedure p")
end_proc:
p()
```

```
"entering procedure p"
```

```
Error: oops [p]
```

In der folgenden Prozedur `q` wird die Prozedur `p` aufgerufen. Dabei werden alle innerhalb von `p` auftretenden Fehler abgefangen:

```
>> q := proc() begin
    print("entering procedure q");
    print("caught error: ", traperror(p()));
    print("leaving procedure q")
end_proc:
q()
```

"entering procedure q"

"entering procedure p"

"caught error: ", 1028

"leaving procedure q"

Änderungen:

☞ Keine Änderungen.

eval – Auswertung eines Objekts

`eval(object)` evaluiert sein Argument `object`, indem es rekursiv alle vorkommenden Bezeichner durch ihre Werte ersetzt und Funktionsaufrufe ausführt und das Ergebnis anschließend noch einmal evaluiert.

Aufruf(e):

☞ `eval(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: das evaluierte Objekt.

Seiteneffekte: Das Verhalten von `eval` wird durch den Wert der Umgebungsvariablen `LEVEL` beeinflusst, die die maximale Substitutionstiefe von Bezeichnern festlegt.

Weitere Dokumentation: Kapitel 5 des MuPAD Tutoriums.

Verwandte Funktionen: `context`, `evalassign`, `evalp`, `freeze`, `hold`, `indexval`, `LEVEL`, `level`, `MAXLEVEL`, `MAXDEPTH`, `val`

Details:

- ☞ `eval` dient zur Evaluierung von unevaluierten oder nur teilweise evaluierten Objekten. *Evaluierung* bedeutet dabei, dass Bezeichner durch ihre Werte ersetzt werden und Funktions-Aufrufe ausgeführt werden.
Üblicherweise evaluiert jede System-Funktion automatisch ihre Argumente und liefert vollständig evaluierte Objekte zurück und die Verwendung von `eval` ist nur in Ausnahmefällen notwendig. So können die Funktionen `map`, `op` und `subs` zum Beispiel Objekte zurückliefern, die nicht vollständig evaluiert sind. Siehe Beispiel ??.
- ☞ Wie die meisten anderen MuPAD-Funktionen evaluiert `eval` zuerst seine Argumente. Dann evaluiert es das Ergebnis noch einmal. Auf interaktiver Ebene hat die zweite Evaluierung üblicherweise keine Wirkung, aber dies ist anders innerhalb von Prozeduren; siehe Beispiele ?? und ??.
- ☞ Das Verhalten von `eval` hängt vom Wert der Umgebungsvariablen `LEVEL` ab, die die maximale Tiefe des rekursiven Prozesses bestimmt, mit der Bezeichner durch ihre Werte ersetzt werden. Die Auswertung des Arguments und die nachfolgende Evaluierung des Ergebnisses werden beide mit der Auswertungstiefe `LEVEL` durchgeführt. Siehe Beispiel ??.
- ☞ Wenn eine lokale Variable oder ein formaler Parameter (vom Typ `DOM_VAR`) von einer Prozedur in `object` auftritt, dann wird sie/er durch ihren/seinen Wert ersetzt, unabhängig vom Wert von `LEVEL`. Bei der folgenden zweiten Evaluierung wird der Wert der lokalen Variablen mit der durch `LEVEL` gegebenen Substitutionstiefe evaluiert, die üblicherweise 1 ist. Siehe Beispiel ??.
- ☞ Das Verhalten von `eval` innerhalb einer Prozedur ist manchmal anders, als man es erwartet, weil die Standard-Substitutionstiefe in Prozeduren 1 ist und `eval` mit dieser Substitutionstiefe evaluiert. Eine vollständige Evaluierung innerhalb einer Prozedur erhält man mit `level`; siehe die entsprechende Hilfeseite zu Details.
- ☞ `eval` erzwingt die Evaluierung von Ausdrücken der Form `hold(x)`: `eval(hold(x))` ist äquivalent zu `x`. Siehe Beispiel ??.
- ☞ `eval` akzeptiert Ausdrucks-Sequenzen als Argumente, siehe Beispiel ??.
Insbesondere liefert der Aufruf `eval()` die leere Sequenz `null()` zurück.
- ☞ `eval` steigt nicht rekursiv in Felder ab. Die Einträge eines Feldes lassen sich mit `map(object, eval)` evaluieren. Siehe Beispiel ??.

- ⌘ `eval` steigt nicht rekursiv in Tabellen ab. Die Einträge einer Tabelle lassen sich mit `map(object, eval)` evaluieren.
Die Indizes einer Tabelle kann man jedoch nicht weiter evaluieren. Benötigt man dies, so muss man eine neue Tabelle aus den evaluierten Operanden der ersten Tabelle erzeugen. Siehe Beispiel ??.
- ⌘ Polynome werden von `eval` nicht weiter evaluiert. Wenn man Werte für Unbestimmte des Polynoms einsetzen möchte, so sollte man `evalp` verwenden. Ferner kann `mapcoeffs(object, eval)` benutzt werden, um die Koeffizienten eines Polynoms auszuwerten. Siehe Beispiel ??.
- ⌘ Die Auswertung von Elementen eines benutzerdefinierten Domains hängt von der Implementation des Domains ab. Normalerweise werden die Domain-Elemente nicht weiter evaluiert. Wenn das Domain einen Slot "evaluate" besitzt, wird die entsprechende Slot-Routine mit dem Domain-Element als Argument bei jeder Evaluierung aufgerufen, also zweimal bei einer Ausführung von `eval`. Siehe Beispiel ??.
- ⌘ `eval` ist eine Funktion des Systemkerns.

Beispiel 1. `subs` macht eine Ersetzung, aber evaluiert sein Ergebnis nicht:

```
>> subs(ln(x), x = 1)

ln(1)
```

Ein expliziter Aufruf von `eval` ist notwendig, um die Evaluierung der Ergebnisse zu erhalten:

```
>> eval(subs(ln(x), x = 1))

0
```

Auch `text2expr` evaluiert sein Ergebnis nicht:

```
>> a := c:
text2expr("a + a"), eval(text2expr("a + a"))

a + a, 2 c
```

Beispiel 2. Die Funktion `hold` verhindert die Evaluierung ihres Arguments. Eine spätere Evaluierung kann mit `eval` erzwungen werden:

```
>> hold(1 + 1); eval(%)

1 + 1

2
```

Beispiel 3. Wenn ein Objekt evaluiert wird, werden Bezeichner rekursiv durch ihre Werte ersetzt. Die maximale Rekursionstiefe in diesem Prozess ist durch die Umgebungsvariable `LEVEL` gegeben:

```
>> delete a0, a1, a2, a3, a4:
      a0 := a1:  a1 := a2 + 2:  a2 := a3 + a4:  a3 := a4^2:  a4 := 5:

>> LEVEL := 1:  a0, a0 + a2;
      LEVEL := 2:  a0, a0 + a2;
      LEVEL := 3:  a0, a0 + a2;
      LEVEL := 4:  a0, a0 + a2;
      LEVEL := 5:  a0, a0 + a2;

                        a1, a1 + a3 + a4

                                2
                        a2 + 2, a2 + a4  + 7

                        a3 + a4 + 2, a3 + a4 + 32

                                2          2
                        a4  + 7, a4  + 37

                        32, 62
```

`eval` evaluiert zuerst sein Argument und evaluiert das Ergebnis noch einmal. Beide Evaluierungen geschehen mit der Substitutionstiefe, die durch `LEVEL` gegeben ist:

```
>> LEVEL := 1:  eval(a0, a0 + a2);
      LEVEL := 2:  eval(a0, a0 + a2);
      LEVEL := 3:  eval(a0, a0 + a2);

                                2
                        a2 + 2, a2 + a4  + 7

                                2          2
                        a4  + 7, a4  + 37

                        32, 62
```

Weil der Standard-Wert von `LEVEL` 100 ist, hat `eval` üblicherweise keine Auswirkung auf der interaktiven Ebene:

```
>> delete LEVEL:
      a0, eval(a0), a0 + a2, eval(a0 + a2)

                        32, 32, 62, 62
```

Beispiel 4. Diese Beispiel zeigt den Unterschied zwischen der Evaluierung von Bezeichnern und lokalen Variablen. Standardmäßig ist der Wert von `LEVEL` innerhalb einer Prozedur 1, d.h. ein globaler Bezeichner wird bei der Evaluierung durch seinen Wert ersetzt, aber es gibt keine weitere rekursive Evaluierung. Dies ändert sich, wenn man `LEVEL` innerhalb der Prozedur einen größeren Wert zuweist:

```
>> delete a0, a1, a2, a3:
      a0 := a1 + a2:  a1 := a2 + a3:  a2 := a3^2 - 1:  a3 := 5:
      p := proc()
        save LEVEL;
        begin
          print(a0, eval(a0)):
          LEVEL := 2:
          print(a0, eval(a0)):
        end_proc:
      end
```

```
>> p()

          2
a1 + a2, a2 + a3 + a3  - 1

          2
a2 + a3 + a3  - 1, 53
```

Im Gegensatz dazu wird eine lokale Variable bei der Evaluierung ohne weitere Evaluierung durch ihren Wert ersetzt. Wenn `eval` auf ein Objekt angewendet wird, das eine lokale Variable enthält, dann ist die Auswirkung eine Evaluierung des Wertes der lokalen Variablen mit Substitutionstiefe `LEVEL`:

```
>> q := proc()
      save LEVEL;
      local x;
      begin
        x := a0:
        print(x, eval(x)):
        LEVEL := 2:
        print(x, eval(x)):
      end_proc:
    end
```

```
>> q()

          2
a1 + a2, a2 + a3 + a3  - 1

          2
a1 + a2, a3  + 28
```

Das Kommando `x := a0` weist den Wert des Bezeichners `a0`, also den unevaluierten Ausdruck `a1 + a2`, an die lokale Variable `x` zu und `x` wird, un-

abhängig vom Wert von LEVEL, jedes Mal durch diesen Wert ersetzt, wenn es evaluiert wird:

Beispiel 5. Im Gegensatz zu Listen und Mengen, werden bei der Evaluierung von Feldern die Einträge nicht evaluiert. Daher hat eval auch keinen Einfluss auf Felder. Durch die Verwendung von map lassen sich die Einträge eines Feldes evaluieren:

```
>> delete a, b:
  L := [a, b]:  A := array(1..2, L):  a := 1:  b := 2:
  L, A, eval(A), map(A, eval)
```

```

      +-      -+ +-      -+ +-      -+
[1, 2], | a, b |, | a, b |, | 1, 2 |
      +-      -+ +-      -+ +-      -+
```

Der Aufruf map(A, gamma) evaluiert die Einträge des Feldes A nicht, bevor die Funktion gamma auf sie angewendet wird. Man kann die Evaluierung erzwingen, indem man die Funktion gamma@eval auf die Einträge anwendet:

```
>> map(A, gamma), map(A, gamma@eval)
```

```

      +-      -+ +-      -+
| gamma(a), gamma(b) |, | 1, 1 |
      +-      -+ +-      -+
```

Beispiel 6. Ebenso werden durch die Evaluierung einer Tabelle ihre Einträge nicht evaluiert und man erreicht dies durch die Verwendung von map. Dies hat jedoch keinen Einfluss auf die Indizes:

```
>> delete a, b:
  T := table(a = b):  a := 1:  b := 2:
  T, eval(T), map(T, eval)
```

```

      table(      table(      table(
        a = b ,   a = b ,   a = 2
      )          )          )
```

Benötigt man eine Tabelle, in der auch die Indizes evaluiert sind, so kann man eine neue Tabelle aus den evaluierten Operanden der alten Tabelle erzeugen. Dabei muss eval verwendet werden, da die Operandenfunktion op die zurückgelieferten Operanden nicht auswertet:

```
>> op(T), table(eval(op(T)))
```

```

      table(
a = b,   1 = 2
      )
```

Beispiel 7. Polynome werden bei Evaluierung nicht verändert, und auch eval bleibt ohne Wirkung:

```
>> delete a, x:  p := poly(a*x, [x]):  a := 2:  x := 3:
      p, eval(p), map(p, eval)

      poly(a x, [x]), poly(a x, [x]), poly(a x, [x])
```

Die Koeffizienten lassen sich mittels mapcoeffs evaluieren:

```
>> mapcoeffs(p, eval)

      poly(2 x, [x])
```

Die Unbestimmte x läßt sich mit evalp durch einen Wert ersetzen:

```
>> delete x:  evalp(p, x = 3)

      3 a
```

Wie man sieht kann das Ergebnis eines evalp-Aufrufs unevaluierte Bezeichner enthalten und diese kann man durch die Anwendung von eval evaluieren:

```
>> eval(evalp(p, x = 3))

      6
```

Beispiel 8. Die Evaluierung eines Elements eines benutzerdefinierten Domains hängt von der Implementation des Domains ab. Normalerweise wird es nicht weiter evaluiert:

```
>> delete a:  T := newDomain("T"):
      e := new(T, a):  a := 1:
      e, eval(e), map(e, eval), val(e)

      new(T, a), new(T, a), new(T, a), new(T, a)
```

Wenn der Slot "evaluate" existiert, so wird die entsprechende Slot-Routine jedesmal aufgerufen, wenn ein Domain-Element ausgewertet wird. Wir implementieren die Routine T::evaluate, die einfach alle internen Operanden ihres Arguments auswertet, für das Domain T. Auf das unevaluierte Domain-Element kann immer noch mit val zugegriffen werden:

```
>> T::evaluate := x -> new(T, eval(extop(x))):
      e, eval(e), map(e, eval), val(e)

      new(T, 1), new(T, 1), new(T, 1), new(T, a)
```

Änderungen:

- ☞ `eval` arbeitet nun auf allen Funktionen. Bis zur Version 1.4.2 wurden nur die Funktionen `args`, `coeff`, `evalp`, `expr`, `hold`, `input`, `last`, `lcoeff`, `nthcoeff`, `subs`, `subsex`, `subsop`, `tcoeff` und `text2expr` von `eval` evaluiert.
 - ☞ Die Evaluierung von lokalen Variablen und formalen Parametern (vom neuen Datentyp `DOM_VAR`) hat sich in `eval` geändert. Details werden in den Abschnitten "Das LEVEL-Problem" und "Symbole, Bezeichner und Variablen" im Dokument "Von MuPAD 1.4 zu MuPAD 2.0" beschrieben.
-

`evalassign` – Zuweisung unter Evaluierung der linken Seite

`evalassign(x, value, i)` weist `value` an das Ergebnis der Evaluierung von `x` mit Substitutionstiefe `i` zu.

Aufruf(e):

- ☞ `evalassign(x, value, i)`
- ☞ `evalassign(x, value)`

Parameter:

- `x` — ein Objekt, dass sich zu einer gültigen linken Seite einer Zuweisung evaluiert
- `value` — ein beliebiges MuPAD-Objekt
- `i` — nichtnegative ganze Zahl kleiner als 2^{31}

Rückgabewert: `value`.

Verwandte Funktionen: `:=`, `_assign`, `assign`, `assignElements`, `delete`, `eval`, `LEVEL`, `level`

Details:

- ☞ `evalassign(x, value, i)` wertet erst `value` aus, wertet dann `x` mit Substitutionstiefe `i` aus, und weist schließlich das Ergebnis der ersten Auswertung dem Ergebnis der zweiten Auswertung zu.
Der Unterschied zwischen `evalassign` und dem Zuweisungsoperator `:=` besteht darin, dass letzterer die linke Seite überhaupt nicht auswertet.
- ☞ Wie üblich, wird `value` mit Substitutionstiefe `LEVEL` ausgewertet. Diese ist innerhalb von Prozeduren per Voreinstellung gleich 1.
- ☞ Näheres zum Begriff der Substitutionstiefe und zur Evaluierung findet man auf den Hilfeseiten von `LEVEL` und `level`.

- ☞ Das dritte Argument ist optional. Die Aufrufe `evalassign(x, value)`, `evalassign(x, value, 0)`, `x := value` und `_assign(x, value)` haben alle die gleiche Wirkung.
- ☞ Das Ergebnis der Auswertung von `x` muss auf der linken Seite einer Zuweisung erlaubt sein. Näheres dazu findet man auf der Hilfeseite von `:=`.
- ☞ Das zweite Argument wird *nicht* ausgeglichen. Es darf daher auch eine Folge sein. Siehe Beispiel ??.

Beispiel 1. `evalassign` kann in Situationen wie der folgenden angewendet werden. Nehmen wir an, ein Bezeichner `a` hat als Wert einen anderen Bezeichner `b`, und wir wollen diesem *Wert* von `a` etwas zuweisen, nicht `a` selbst:

```
>> delete a, b: a := b:
      evalassign(a, 100, 1): level(a, 1), a, b

                        b, 100, 100
```

Wir hätten hier nicht den Zuweisungsoperator `:=` verwenden dürfen, da dieser seine linke Seite nicht auswertet:

```
>> delete a, b: a := b:
      a := 100: level(a, 1), a, b

                        100, 100, b
```

Beispiel 2. Das zweite Argument darf auch eine Folge sein:

```
>> a := b:
      evalassign(a, (3,5), 1):
      b

                        3, 5
```

Hintergründe:

- ☞ Intern wird `level` zur Auswertung von `x` verwendet, d. h., `i` darf den Wert von `LEVEL` überschreiten.
- ☞ Alle in der Hilfe zu `_assign` aufgeführten Spezialfälle gelten auch hier: dort findet man Näheres zu indizierten Zuweisungen, Zuweisungen an Slots, und zum `protect`-Mechanismus.

Änderungen:

☞ Keine Änderungen.

evalp – Auswertung eines Polynoms an einer Stelle

`evalp(p, x = v)` wertet das Polynom `p` in der Unbestimmten `x` an der Stelle `v` aus.

Aufruf(e):

☞ `evalp(p, x = v, ...)`
☞ `evalp(f, <vars,> x = v, ...)`

Parameter:

`p` — ein Polynom vom Typ `DOM_POLY`
`x` — eine Unbestimmte
`v` — die Stelle, an der evaluiert wird: ein Element des Koeffizientenrings des Polynoms
`f` — ein polynomialer Ausdruck
`vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: ein Element des Koeffizientenrings oder ein Polynom oder ein polynomialer Ausdruck oder `FAIL`

Überladbar durch: `p`, `f`

Verwandte Funktionen: `eval`, `poly`

Details:

- ☞ `evalp(p, x = v)` wertet das Polynom `p` in der Unbestimmten `x` an der Stelle `v` aus. Ein Fehler wird ausgelöst, falls `x` keine der Unbestimmten von `p` ist. Der Wert `v` muss ein Objekt sein, das als Koeffizient des Polynoms gültig ist. Das Ergebnis ist ein Element des Koeffizienten-Rings von `p`, falls `p` univariat ist. Ist `p` multivariat, so ist das Ergebnis ein Polynom in den verbleibenden Unbestimmten.
- ☞ Werden mehrere Stellen angegeben, an denen ausgewertet werden soll, so werden sie nacheinander von links nach rechts abgearbeitet. Jede Auswertung erfolgt gemäß der obigen Beschreibung.
- ☞ Für ein Polynom `p` in den Unbestimmten `x1`, `x2`, ... kann auch die Syntax `p(v1, v2, ...)` anstelle von `evalp(p, x1 = v1, x2 = v2, ...)` verwendet werden.

- ⌘ Bei `evalp(f, vars, x = v, ...)` wird zuerst der polynomiale Ausdruck `f` mittels `poly` in ein Polynom in den Unbestimmten `vars` konvertiert. Werden keine Unbestimmten angegeben, so werden diese in `f` gesucht. Siehe `poly` zu Details bezüglich der Konvertierung. Falls `f` nicht in ein Polynom konvertiert werden kann, so liefert `evalp` den Wert `FAIL`. Ein erfolgreich konvertiertes Polynom wird wie oben beschrieben ausgewertet. Das Ergebnis wird wieder in einen Ausdruck konvertiert.
- ⌘ Das Horner-Schema wird zur Auswertung des Polynoms verwendet. Die Auswertung von Unbestimmten an der Stelle 0 ist am effizientesten und sollte daher zuerst erfolgen. Danach sollte die verbleibende Hauptvariable zuerst ausgewertet werden.
- ⌘ Das Ergebnis von `evalp` wird nicht weiter evaluiert. Man kann `eval` verwenden, um ein vollständig evaluiertes Ergebnis zu erhalten.
- ⌘ `evalp` ist eine Funktion des Systemkerns.

Beispiel 1. `evalp` wird verwendet, um den polynomialen Ausdruck $x^2 + 2x + 3$ an der Stelle $x = a + 2$ auszuwerten. Der Form des sich ergebenden Ausdrucks sieht man an, dass die Horner-Regel verwendet wurde:

```
>> evalp(x^2 + 2*x + 3, x = a + 2)
      (a + 2) (a + 4) + 3
```

Beispiel 2. `evalp` wird verwendet, um ein Polynom in den Unbestimmten x und y an der Stelle $x = 3$ auszuwerten. Zurückgeliefert wird ein Polynom in der verbleibenden Unbestimmten y :

```
>> p := poly(x^2 + x*y + 2, [x, y]): evalp(p, x = 3)
      poly(3 y + 11, [y])

>> delete p:
```

Beispiel 3. Polynome können wie Funktionen aufgerufen werden, wenn alle Variablen durch Werte ersetzt werden sollen:

```
>> p := poly(x^2 + x*y, [x, y]): evalp(p, x = 3, y = 2) = p(3, 2)
      15 = 15

>> delete p:
```

Beispiel 4. Das Ergebnis ist ein Polynom in den verbleibenden Variablen, wenn nicht alle Variablen durch Werte ersetzt werden:

```
>> evalp(poly(x*y*z + x^2 + y^2 + z^2, [x, y, z]), x = 1, y = 1)
      2
    poly(z  + z + 2, [z])
```

Beispiel 5. Das Ergebnis von `evalp` wird nicht vollständig evaluiert. Um dies zu sehen, definieren wir ein Polynom `p` mit einem Koeffizienten `a` und ändern anschließend den Wert von `a`. Diese Änderung spiegelt sich nicht in `p` wieder, da die Koeffizienten von Polynomen nicht implizit evaluiert werden. Man muss die Funktion `eval` auf die Koeffizienten anwenden, um sie zu evaluieren:

```
>> p := poly(x^2 + a*y + 1, [x,y]): a := 2:
    p, mapcoeffs(p, eval)
      2                2
    poly(x  + a y + 1, [x, y]), poly(x  + 2 y + 1, [x, y])
```

Wenn man `evalp` verwendet, um `p` an der Stelle $x = 1$ auszuwerten, so ist das Ergebnis nicht vollständig evaluiert: der symbolische Koeffizient `a` verbleibt. Man muss wieder `eval` verwenden, um vollständig evaluierte Koeffizienten zu erhalten:

```
>> r := evalp(p, x = 1):
    r, mapcoeffs(r, eval)
      poly(a y + 2, [y]), poly(2 y + 2, [y])

>> delete p, a, r:
```

Änderungen:

☞ Keine Änderungen.

exp – die Exponentialfunktion

`exp(x)` stellt den Wert der Exponentialfunktion am Punkt x dar.

Aufruf(e):

☞ `exp(x)`

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

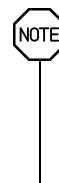
Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `ln`, `log`

Details:

- ☞ Die Exponentialfunktion ist für alle komplexen Argumente definiert.
- ☞ Für die meisten exakten Argumente werden unevaluierte Funktionsaufrufe zurückgegeben. Einige Vereinfachungen werden durchgeführt:
 - Aufrufe der Form $\exp(q \pi i)$ mit ganzzahligem oder rationalem q werden so umgeschrieben, dass q im Intervall $[0, 2)$ liegt. Es werden explizite Zahlenergebnisse geliefert, wenn q den Nenner 1, 2, 3, 4, 5, 6, 8, 10 oder 12 hat.
 - Weiterhin sind folgende speziellen Werte implementiert: $\exp(0) = 1$, $\exp(\text{infinity}) = \text{infinity}$, $\exp(-\text{infinity}) = 0$.
 - Ein Aufruf der Form $\exp(c \ln(y))$ mit unevaluiertem $\ln(y)$ und c vom Typ `Type::Constant` liefert y^c .
 - Der Aufruf $\exp(f(y))$ liefert $y/f(y)$ für $f = \text{lambertV}$ bzw. lambertW .
- ☞ Für Gleitpunktargumente werden Gleitpunktwerte berechnet.
 Man beachte, dass numerischer Über- oder Unterlauf auftreten kann, wenn der Realteil des Gleitpunktarguments x betragsmäßig groß ist. Folgender Schutz gegen Unterlauf ist implementiert: falls $\text{Re}(x) < -10^6$ gilt, so kann $\exp(x)$ das abgeschnittene Ergebnis 0.0 zurückliefern. Siehe Beispiel ??.
- ☞ Das Schlüsselwort `E` ist gleichbedeutend mit `exp(1)`.



Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> E, exp(2), exp(-3), exp(1/4), exp(1 + I), exp(x^2)
                                     2
exp(1), exp(2), exp(-3), exp(1/4), exp(1 + I), exp(x )
```


Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> exp(1.23), exp(4.5 + 6.7*I), exp(1.0/10^20), exp(123456.7)
3.421229536, 82.31014791 + 36.44342846 I, 1.0,
3.660698702e53616
```

Einige spezielle symbolische Vereinfachungen sind implementiert:

```
>> exp(I*PI), exp(x - 22*PI*I), exp(3 + I*PI)
-1, exp(x), -exp(3)
>> exp(ln(-2)), exp(ln(x)*PI), exp(lambertW(5))
-2, xPI,  $\frac{5}{\text{lambertW}(5)}$ 
```

Beispiel 2. Als Schutz gegen numerischen Unterlauf kann `exp` für Gleitpunktargumente mit betragsmäßig großem negativen Realteil das abgeschnittene Ergebnis `0.0` zurückliefern:

```
>> exp(-5.81*10^6), exp(-5.82*10^6)
1.148529374e-2523251, 0.0
>> exp(-5.81*10^6 + 10^10*I), exp(-5.82*10^6 + 10^10*I)
1.002803534e-2523251 - 5.599149896e-2523252 I, 0.0
```

Ein solcher Schutz existiert nicht für numerischen Überlauf:

```
>> exp(5.81*10^6)
8.706786458e2523250
>> exp(5.82*10^6)
Error: Overflow/underflow in arithmetical operation;
during evaluation of 'exp::float'
```

Beispiel 3. Systemfunktionen wie `limit`, `series`, `expand`, `combine` etc. verarbeiten `exp`:

```
>> limit(x*exp(-x), x = infinity), series(exp(x/(x + 1)), x = 0)
```

$$0, 1 + x - \frac{x^2}{2} + \frac{x^3}{6} - \frac{x^4}{24} + \frac{19x^5}{120} + O(x^6)$$

```
>> expand(exp(x + y + (sqrt(2) + 5)*PI*I))
```

$$- \exp(x) \exp(y) \exp(I \pi 2^{1/2})$$

```
>> combine(%, exp)
```

$$- \exp(x + y + I \pi 2^{1/2})$$

Änderungen:

- ⌘ Einige der Vereinfachungsregeln wurden geändert. Zur Vermeidung von numerischem Unterlauf wird nun der abgeschnittene Wert 0.0 für Gleitpunktargumente mit betragsmäßig großem negativem Realteil zurückgeliefert. Die Werte `exp(infinity) = infinity` und `exp(-infinity) = 0` wurden implementiert.

`expand` – Expansion eines Ausdrucks

`expand(f)` expandiert den arithmetischen Ausdruck `f`.

Aufruf(e):

- ⌘ `expand(f)`
- ⌘ `expand(f, g1, g2, ...)`

Parameter:

`f, g1, g2, ...` — arithmetische Ausdrücke

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `f`

Überladbar durch:

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `collect`, `combine`, `denom`, `factor`, `normal`, `numer`, `partfrac`, `rationalize`, `rectform`, `rewrite`, `simplify`

Details:

- ☞ Das wichtigste Einsatzgebiet von `expand` ist die Anwendung des Distributivgesetzes, um Produkte von Summen in Summen von Produkten umzuschreiben. In diesem Sinne ist `expand` die Umkehrfunktion von `factor`.

Potenzen von Summen mit positiven ganzen Zahlen als Exponenten werden auch expandiert, aber Potenzen von Summen mit negativen ganzen Zahlen als Exponenten werden nicht expandiert; siehe Beispiel ??.

Der Zähler eines Bruchs wird expandiert und dann wird der Bruch umgeschrieben in eine Summe von Brüchen mit einfacheren Zählern; siehe Beispiel ??. In gewissem Sinne ist dies die umgekehrte Funktionalität von `normal`. Brüche lassen sich auch mit der mächtigeren Funktion `partfrac` in eine Summe von einfacheren Brüchen umschreiben.

- ☞ `expand(f)` wendet die folgenden Umformungsregeln auf Potenzen in Teilausdrücken von `f` an:

- $x^{a+b} = x^a x^b$
- $(xy)^b = x^b y^b$
- $(x^a)^b = x^{ab}$

Die beiden letzten Regeln gelten nur unter bestimmten zusätzlichen Einschränkungen, d. h. wenn `b` eine ganze Zahl ist. Außer der dritten Regel ist dieses Verhalten von `expand` die umgekehrte Funktionalität von `combine`. Siehe Beispiel ??.

- ☞ `expand` arbeitet rekursiv auf den Teilausdrücken eines Ausdrucks `f`. Wenn `f` einer der Container-Typen `Feld`, `Liste`, `Menge` oder `Tabelle` ist, dann liefert `expand` nur `f` zurück und wird nicht rekursiv auf die Einträge angewendet. Wenn man alle Einträge von einem der Container expandieren will, muss man `map` verwenden. Siehe Beispiel ??.

- ☞ Wenn optionale Argumente `g1`, `g2`, ... gegeben sind, dann werden alle Teilausdrücke von `f`, die gleich einem dieser zusätzlichen Argumente sind, nicht expandiert; siehe Abschnitt „Hintergründe“ zu einer Beschreibung dieses Verhaltens.

☞ Eigenschaften von Bezeichnern werden berücksichtigt (siehe `assume`). Bezeichner ohne Eigenschaften werden als komplexwertig angenommen. Siehe Beispiel ??.

☞ `expand` reagiert auch auf einige spezielle mathematische Funktionen. Es schreibt einen einzelnen Aufruf einer speziellen Funktion mit einem kompliziertem Argument in eine Summe oder ein Produkt von mehreren Aufrufen derselben Funktion oder verwandten Funktionen mit einfacheren Argumenten um. In diesem Sinne ist `expand` die Umkehrfunktion von `combine`.

Insbesondere implementiert `expand` die Funktional-Gleichungen der Exponentialfunktion und des Logarithmus, der Gamma-Funktion und der Polygamma-Funktion und die Additionstheoreme der trigonometrischen Funktionen und der hyperbolischen Funktionen. Siehe Beispiel ??.

☞ `expand` ist eine Funktion des Systemkerns.

Beispiel 1. `expand` expandiert Produkte von Summen durch Ausmultiplizieren:

```
>> expand((x + 1)*(y + z)^2)
```

$$2 y z + 2 x y z + y^2 + z^2 + x y^2 + x z^2$$

Nach der Expandierung des Zählers wird ein Bruch in eine Summe von Brüchen umgeschrieben:

```
>> expand((x + 1)^2*y/(y + z)^2)
```

$$\frac{y}{(y + z)^2} + \frac{2 x y}{(y + z)^2} + \frac{x^2 y}{(y + z)^2}$$

Beispiel 2. Potenzen von Summen mit positiven ganzzahligen Exponenten werden expandiert:

```
>> expand((x + y)^2)
```

$$2 x y + x^2 + y^2$$

Potenzen von Summen mit negativen ganzzahligen Exponenten werden als Nenner von Brüchen betrachtet und nicht expandiert:

```
>> expand((x + y)^(-2))
```

$$\frac{1}{(x + y)^2}$$

Beispiel 3. Eine Potenz mit einer Summe im Exponenten wird in ein Produkt von Potenzen umgeschrieben:

```
>> expand(x^(y + z + 2))
```

$$x^2 x^y x^z$$

Wenn einer der Summanden im Exponent negativ ist, dann wird die Potenz in einen Bruch aus Potenzen expandiert:

```
>> expand((x + y)^(z - 2))
```

$$\frac{(x + y)^z}{(x + y)^2}$$

Beispiel 4. `expand` arbeitet rekursiv. Im folgenden Beispiel wird die Potenz $(x + y)^{z+2}$ zuerst in ein Produkt aus zwei Potenzen expandiert. Dann wird die Potenz $(x + y)^2$ in eine Summe expandiert. Schließlich wird das Produkt aus dieser Summe und der verbliebenen Potenz $(x + y)^z$ ausmultipliziert:

```
>> expand((x + y)^(z + 2))
```

$$x^z y^z (x + y)^2 + x^z (x + y)^z + y^z (x + y)^z$$

Hier ist ein weiteres Beispiel:

```
>> expand(2^((x + y)^2))
```

$$\frac{x^2}{2} \frac{y^2}{2} + \frac{2xy}{2}$$

`expand` wird nicht rekursiv auf die Einträge eines Container-Typs angewendet:

```
>> expand([(a + b)^2, c]), expand({(a + b)^2, c})
```

$$\begin{matrix} & 2 & & 2 \\ & & & \\ (a + b)^2, & c, & \{c, & (a + b)^2\} \end{matrix}$$

Mit map kann man alle Einträge eines Containers expandieren:

```
>> map([(a + b)^2, c], expand), map({(a + b)^2, c}, expand)
```

$$\begin{matrix} & 2 & & 2 & & 2 & & 2 \\ & & & & & & & \\ [2 a b + a^2 + b^2, & c], & \{c, & 2 a b + a^2 + b^2\} \end{matrix}$$

Beispiel 5. Wenn zusätzliche Argumente angegeben werden, dann nimmt expand nur eine teilweise Expandierung vor. Diese weiteren Ausdrücke, wie $x + 1$ im folgenden Beispiel, werden dann nicht expandiert:

```
>> expand((x + 1)*(y + z))
```

$$y + z + x y + x z$$

```
>> expand((x + 1)*(y + z), x + 1)
```

$$y (x + 1) + z (x + 1)$$

Beispiel 6. Die folgenden Expandierungen sind nicht für alle Werte von a und b in der komplexen Ebene gültig. Deshalb wird keine Expandierung durchgeführt:

```
>> expand(ln(a^2)), expand(ln(a*b))
```

$$\begin{matrix} & 2 \\ \ln(a^2), & \ln(a b) \end{matrix}$$

Die Expandierungen sind gültig unter der Annahme, dass a eine positive reelle Zahl ist:

```
>> assume(a > 0): expand(ln(a^2)), expand(ln(a*b))
```

$$2 \ln(a), \ln(a) + \ln(b)$$

```
>> unassume(a):
```

Beispiel 7. Die Additionstheoreme der Trigonometrie sind als "expand"-Slots der trigonometrischen Funktionen sin und cos implementiert:

```
>> expand(sin(a + b)), expand(sin(2*a))
cos(a) sin(b) + cos(b) sin(a), 2 cos(a) sin(a)
```

Dasselbe gilt für die hyperbolischen Funktionen sinh und cosh:

```
>> expand(cosh(a + b)), expand(cosh(2*a))
cosh(a) cosh(b) + sinh(a) sinh(b), 2 cosh(a)2 - 1
```

Die Exponential-Funktion mit einer Summe als Argument wird mittels `exp::expand` expandiert:

```
>> expand(exp(a + b))
exp(a) exp(b)
```

Hier sind weitere Expandierungsbeispiele für die Funktionen `sum`, `fact`, `abs`, `coth`, `sign`, `binomial`, `beta`, `gamma`, `log`, `cot`, `tan`, `exp` und `psi`:

```
>> sum(x + exp(x), x); expand(%)
sum(x + exp(x), x)
x2 - x + exp(x)
-- - - + -----
2 2 exp(1) - 1
```

```
>> fact(x + 1); expand(%)
fact(x + 1)
fact(x) (x + 1)
```

```
>> abs(a*b); expand(%)
abs(a b)
abs(a) abs(b)
```

```
>> coth(a + b); expand(%)
coth(a + b)
```

```
cosh(a) cosh(b)
----- +
cosh(a) sinh(b) + cosh(b) sinh(a)
sinh(a) sinh(b)
-----
cosh(a) sinh(b) + cosh(b) sinh(a)
```

```

>> coth(a*b); expand(%)

coth(a b)

cosh(a b)
-----
sinh(a b)

>> sign(a*b); expand(%)

sign(a b)

sign(a) sign(b)

>> tan(a); expand(%)

tan(a)

sin(a)
-----
cos(a)

>> binomial(n, m); expand(%)

binomial(n, m)

n gamma(n)
-----
m gamma(m) (n - m) gamma(n - m)

>> beta(n, m); expand(%)

beta(m, n)

gamma(m) gamma(n)
-----
gamma(m + n)

>> gamma(x+1); expand(%)

gamma(x + 1)

x gamma(x)

>> log(10, x); expand(%)

log(10, x)

ln(x)
-----
ln(10)

```



```

>> cot(x);  expand(%)

cot(x)

cos(x)
-----
sin(x)

>> exp(x + y);  expand(%)

exp(x + y)

exp(x) exp(y)

>> psi(x + 2);  expand(%)

psi(x + 2)

psi(x) + 1/x + 1/(x + 1)

```

Beispiel 8. Dieses Beispiel illustriert, wie man die Funktionalität von `expand` für Nutzer-definierte mathematische Funktionen erweitert. Als Beispiel betrachten wir die Sinus-Funktion. (Natürlich hat die Systemfunktion `sin` bereits einen "expand"-Slot; siehe Beispiel ??.)

Zuerst betten wir unsere Funktion in eine Funktionsumgebung ein, die wir `Sin` nennen, um die Systemfunktion `sin` nicht zu überschreiben. Dann implementieren wir das Additionstheorem $\sin(x + y) = \sin(x)\cos(y) + \sin(y)\cos(x)$ im "expand"-Slot der Funktionsumgebung, d. h. in der Routine `Sin::expand`:

```

>> Sin := funcenv(Sin):
Sin::expand := proc(u) // Berechnung von expand(Sin(u))
  local x, y;
  begin
    // rekursiv das Argument u expandieren
    u := expand(u);

    if type(u) = "_plus" then // u ist eine Summe

      x := op(u, 1); // der erste Term
      y := u - x;    // die restlichen Terme

      // Anwendung des Additionstheorems und
      // Expandierung des Ergebnisses
      expand(Sin(x)*cos(y) + cos(x)*Sin(y))
    end if;
  end begin;
end proc;

```

```

else
  Sin(u)
end_if
end_proc:

```

Wenn nun `expand` auf einen Teilausdruck der Form `Sin(u)` stößt, ruft es `Sin::expand(u)` auf, um `Sin(u)` zu expandieren. Das folgende Kommando expandiert zuerst das Argument `a*(b+c)` über einen rekursiven Aufruf in `Sin::expand`, wendet dann das Additionstheorem an und schließlich expandiert `expand` selbst das Produkt aus dem Ergebnis mit `z`:

```

>> expand(z*Sin(a*(b + c)))

      z Sin(a b) cos(a c) + z Sin(a c) cos(a b)

```

Die Expandierung nach der Anwendung des Additionstheorems in `Sin::expand` ist notwendig, um den Fall zu behandeln, wenn `u` eine Summe mit mehr als zwei Termen ist: dann ist `y` wieder eine Summe und `cos(y)` und `Sin(y)` werden rekursiv expandiert:

```

>> expand(Sin(a + b + c))

Sin(a) cos(b) cos(c) + Sin(b) cos(a) cos(c) +
      Sin(c) cos(a) cos(b) - Sin(a) sin(b) sin(c)

```

Hintergründe:

- ☞ Mit optionalen Argumenten `g1, g2, ...` kann die Expandierung gewisser Teilausdrücke von `f` unterdrückt werden. Dies funktioniert wie folgt: Jedes Auftreten von `g1, g2, ...` in `f` wird durch vor der Expandierung durch eine Hilfsvariable ersetzt und anschließend werden die Hilfsvariablen wieder durch die originalen Teilausdrücke ersetzt.
- ☞ Anwender können die Funktionalität von `expand` für ihre eigenen speziellen mathematischen Funktionen durch Überladung erweitern. Dazu bettet man die Funktion in eine Funktionsumgebung `g` ein und implementiert das Verhalten von `expand` für diese Funktion im "expand"-Slot der Funktionsumgebung.
 Wann immer `expand` auf einen Teilausdruck der Form `g(u, ...)` stößt, wird der Aufruf `g::expand(u, ...)` der Slot-Routine ausgeführt, um den Teilausdruck zu expandieren, wobei die noch nicht expandierten Argumente `u, ...` von `g` als Argumente übergeben werden. Das Ergebnis wird nicht weiter von `expand` expandiert. Siehe Beispiel ?? oben.
- ☞ Gleichermäßen kann ein "expand"-Slot für ein Nutzer-definiertes Bibliotheks-Domain `T` definiert werden. Wann immer `expand` auf einen Teilausdruck `d` vom Domain-Typ `T` stößt, so wird der Aufruf `T::expand(d)`

der Slot-Routine ausgeführt, um `d` zu expandieren. Das Ergebnis dieses Aufrufs wird nicht weiter von `expand` expandiert. Wenn `T` keinen "expand"-Slot hat, dann bleibt `d` unverändert.

Änderungen:

⌘ `expand` berücksichtigt jetzt Eigenschaften von Bezeichnern.

`export`, `unexport` – Exportieren von Bibliotheksfunktionen bzw. Rückgängigmachen des Exports

`export(L, f)` exportiert die öffentliche Funktion `L : f` der Bibliothek `L`, so dass anschließend auf diese mittels `f` ohne das Präfix `L` zugegriffen werden kann.

`export(L)` exportiert alle öffentlichen Funktionen der Bibliothek `L`.

`unexport(L, f)` macht den Export der öffentlichen Funktion `L : f` der Bibliothek `L` rückgängig, so dass auf diese nicht mehr mittels `f` zugegriffen werden kann.

`unexport(L)` macht den Export aller bisher exportierten öffentlichen Funktionen der Bibliothek `L` rückgängig.

Aufruf(e):

⌘ `export(L, f1, f2, ...)`
⌘ `export(L)`
⌘ `unexport(L, f1, f2, ...)`
⌘ `unexport(L)`

Parameter:

`L` — die Bibliothek: ein Domain
`f1, f2, ...` — öffentliche Funktionen von `L`: Bezeichner

Rückgabewert: das leere Objekt `null()` vom Typ `DOM_NULL`.

Seiteneffekte: Wenn eine Funktion exportiert wird, so wird sie dem entsprechenden globalen Bezeichner zugewiesen. Wenn der Export rückgängig gemacht wird, so wird der entsprechende Bezeichner gelöscht.

Weitere Dokumentation: Kapitel „Die MuPAD-Bibliotheken“ des Tutoriums.

Verwandte Funktionen: `:=`, `delete`, `info`, `loadmod`, `loadproc`, `package`, `unloadmod`

Details:

- ⇒ Eine Bibliothek enthält *öffentliche* Funktionen, die der Benutzer aufrufen kann. Diese Funktionen bilden das *Interface* der Bibliothek. (Es kann daneben noch private Funktionen in einer Bibliothek geben, die nicht dafür gedacht sind, direkt vom Benutzer aufgerufen zu werden, und die undokumentiert sind.) Eine Funktion f in einer Bibliothek L wird normalerweise mittels $L : : f$ angesprochen. Wenn die Funktion *exportiert* wird, so ist sie danach in der Kurzform f verfügbar. Technisch bedeutet exportieren, dass dem globalen Bezeichner f die Funktion $L : : f$ als Wert zugewiesen wird.
 - ⇒ Der Export wird rückgängig gemacht, indem der Wert des globalen Bezeichners f gelöscht wird. Die Bibliotheksfunktion steht danach nur unter dem Namen $L : : f$ zur Verfügung.
 - ⇒ `export(L, f1, f2, ...)` exportiert die angegebenen Funktionen $f1, f2, \dots$ von L . Hat einer der Bezeichner bereits einen Wert, so wird die entsprechende Funktion nicht exportiert; statt dessen wird eine Warnung ausgegeben. Es ist ein Fehler, wenn einer der Bezeichner keine öffentliche Funktion von L benennt.
 - ⇒ `export(L)` exportiert alle öffentlichen Funktionen von L .
 - ⇒ Eine Funktion, die bereits exportiert ist, wird nicht nochmals exportiert; statt dessen wird eine Warnung ausgegeben.
 - ⇒ `unexport(L, f1, f2, ...)` macht die Exporte der genannten Funktionen von L rückgängig. Übrigens evaluiert `unexport` die Bezeichner nicht. Man muss also nicht `hold` verwenden, um die Evaluation der Bezeichner zu verhindern.
 - ⇒ `unexport(L)` macht jegliche Exporte der Bibliothek L rückgängig.
 - ⇒ `export` und `unexport` werten ihr erstes Argument L aus. Die weiteren Argumente $f1, f2, \dots$, falls vorhanden, werden nicht ausgewertet.
 - ⇒ Die Funktion `info` gibt Informationen über die öffentlichen und die exportierten Funktionen einer Bibliothek aus.
 - ⇒ Einige Bibliotheken haben Funktionen, die automatisch zu jeder Zeit exportiert sind. Diese Exporte können nicht mit `unexport` rückgängig gemacht werden. Die Funktion `append` aus der Bibliothek `listlib` ist ein Beispiel dafür.
-

Beispiel 1. Die öffentliche Funktion `powerset` der Bibliothek `combinat` wird exportiert, anschließend wird der Export wieder zurückgenommen:

```
>> combinat::powerset(2)

      {{}}, {2}, {1}, {1, 2}}

>> export(combinat, powerset):

>> powerset(2)

      {{}}, {2}, {1}, {1, 2}}

>> unexport(combinat, powerset):

>> powerset(2)

      powerset(2)
```

Nun werden alle öffentlichen Funktionen von `combinat` exportiert, anschließend wird der Export wieder zurückgenommen:

```
>> export(combinat):
      permute([1, 2])

      [[1, 2], [2, 1]]

>> unexport(combinat):
      permute([1, 2])

      permute([1, 2])
```

Beispiel 2. `export` gibt eine Warnung aus, wenn eine Funktion nicht exportiert werden kann, weil der entsprechende Bezeichner schon einen Wert hat:

```
>> powerset := 17:
      export(combinat, powerset)

Warning: 'powerset' already has a value, not exported.
```

Eine Funktion wird nicht doppelt exportiert, und `export` gibt eine Meldung aus, wenn man es dennoch versucht:

```
>> delete powerset:
      export(combinat, powerset):
      export(combinat, powerset):
      unexport(combinat, powerset):

Info: 'combinat::powerset' already is exported.
```

Hintergründe:

- ⌘ Die Namen der öffentlichen Funktionen einer Bibliothek `L` sind in der Menge `L::interface` gespeichert. Diese Menge wird für den Export und für die Funktion `info` verwendet.
- ⌘ Die Namen der aktuell exportierten Funktionen der Bibliothek `L` sind in der Menge `L::exported` gespeichert.

Änderungen:

- ⌘ `unexport` ist eine neue Funktion.
-

expose – Ausgabe des Quellcodes einer Prozedur oder der Methoden eines Domains

`expose(f)` gibt den Quellcode der MuPAD-Prozedur `f` bzw. die Einträge des Domains `f` aus.

Aufruf(e):

- ⌘ `expose(f)`

Parameter:

- `f` — beliebiges Objekt; typischerweise eine Prozedur, eine Funktionsumgebung oder ein Domain

Rückgabewert:

- ⌘ Falls `f` eine Prozedur ist, liefert `expose` den Quellcode von `f` als Objekt vom Typ `stdlib::Exposed` zurück (siehe den Abschnitt „Hintergrund“).
- ⌘ Ist `f` eine Funktionsumgebung, so ist das Ergebnis gleich dem Ergebnis der Anwendung von `expose` auf den ersten Operanden von `f`.
- ⌘ Ist `f` ein Domain, so liefert `expose` einen symbolischen Aufruf von `newDomain` (Details siehe unten).
- ⌘ In allen anderen Fällen ist das Ergebnis `f` selbst, falls `expose` nicht überladen ist.

Seiteneffekte: Die Ausgabeformatierung von `expose` ist von der Umgebungsvariablen `TEXTWIDTH` abhängig.

Überladbar durch: `f`

Verwandte Funktionen: `print`

Details:

- ⌘ Normalerweise werden Prozeduren und Domains in abgekürzter Form ausgegeben. `expose` bietet die Möglichkeit, den vollständigen Quellcode einer Prozedur bzw. alle Einträge eines Domains angezeigt zu bekommen.
- ⌘ Ist `f` ein Domain, so liefert `expose` einen symbolischen Aufruf der Funktion `newDomain`. Die Argumente dieses Aufrufs sind Gleichungen der Form `index = value`, wobei `value` gleich dem Eintrag `f :: index` ist. `expose` wird nicht rekursiv auf `f :: index` angewandt, d. h. der Quellcode von Domainmethoden wird nicht angezeigt.
- ⌘ `expose` liefert ein syntaktisch korrektes MuPAD-Objekt zurück, das jedoch nur zur Ausgabe dient und nicht zur Weiterverarbeitung geeignet ist.

Beispiel 1. Mit Hilfe von `expose` kann man sich den Quellcode von Prozeduren der MuPAD-Bibliothek ansehen:

```
>> sin

sin

>> expose(%)

proc(x)
  name sin;
  local f, y;
  option noDebug;
begin
  if args(0) = 0 then
    error("no arguments given")
  else
    ...
end_proc
```

Beispiel 2. Der Quellcode von Kernfunktionen hingegen kann nicht angezeigt werden:

```
>> expose(_plus)

builtin(817, NIL, "_plus", NIL)
```

Beispiel 3. Wird `expose` auf ein Domain angewandt, so zeigt es die Einträge dieses Domains an:

```
>> expose(DOM_INT)

newDomain("coerce" = proc DOM_INT::coerce(x) ... end,

        "phi" = phi, "new_extelement" =

        proc new_extelement(d) ... end, "new" = proc new() ... end,

        "D" = 0, "key" = "DOM_INT")
```

Beispiel 4. `expose` kann auch auf andere Objekte angewandt werden, dies ist aber im allgemeinen nicht besonders sinnvoll:

```
>> expose(3)
```

3

Hintergründe:

- ⌘ Zusätzlich zum üblichen Überladungsmechanismus für Elemente eines Domains muss eine Domain-Methode `"expose"` auch den Fall behandeln, dass das Domain selbst dargestellt werden soll: in diesem Fall wird sie ohne Argumente aufgerufen.
- ⌘ Ist `f` eine Prozedur, so ist der Rückgabewert ein Objekt des Domains `stdlib::Exposed`. Dieser Datentyp dient nur Zwecken der Ausgabe mittels einer eigenen `"print"`-Methode; es sollte niemals nötig sein, auf Objekte dieses Typs zuzugreifen. Daher bleibt er undokumentiert.

Änderungen:

- ⌘ Der Rückgabewert von `expose` ist jetzt der neu eingeführte Datentyp `stdlib::Exposed`, falls das Argument eine Prozedur ist.

`expr` – Konvertierung in ein Element eines Basis-Domains

`expr(object)` konvertiert `object` in ein Element eines Basis-Domains, so dass auch alle Teilausdrücke Elemente von Basis-Domains sind.

Aufruf(e):

⌘ `expr(object)`

Parameter:

`object` — ein beliebiges Objekt

Rückgabewert: ein Element eines Basis-Domains.

Überladbar durch: `object`

Verwandte Funktionen: `coerce, domtype, eval, testtype, type`

Details:

⌘ `expr` ist eine Typ-Konvertierungs-Funktion zur Konvertierung eines Elements eines komplexeren Bibliotheks-Domains, wie ein Polynom oder eine Matrix, in ein Element eines Basis-Kern-Domains.

`expr` geht dabei rekursiv vor, so dass alle Teil-Objekte des zurückgelieferten Objekts auch Elemente von Basis-Domains sind. Siehe Beispiel ??.

⌘ Die beiden speziellen Objekte `infinity` und `complexInfinity` werden durch `expr` in Bezeichner mit den selben Namen übersetzt. Die Evaluierung dieser Bezeichner liefert wieder die originalen Objekte. Siehe Beispiel ??.

⌘ Wenn `object` bereits zu einem Basis-Domain außer `DOM_POLY` gehört, dann wird `expr` nur rekursiv auf alle Operanden von `object` angewendet, wenn es welche gibt.

⌘ Wenn `object` ein Polynom vom Domain-Typ `DOM_POLY` ist, dann wird `expr` rekursiv auf alle Koeffizienten von `object` angewendet und anschließend wird das Ergebnis in einen Bezeichner, eine Zahl oder einen Ausdruck umgewandelt. Siehe Beispiel ??.

⌘ Wenn `object` zu einem Bibliotheks-Domain `T` mit einem "`expr`"-Slot gehört, dann wird die zugehörige Slot-Routine `T::expr` mit `object` als Argument aufgerufen und das Ergebnis wird zurückgeliefert.

Dies kann dazu verwendet werden, um die Funktionalität von `expr` auf Elemente von Nutzer-definierten Domains zu erweitern. Wenn die Slot-Routine die Konvertierung nicht durchführen kann, muss sie `FAIL` zurückliefern. Siehe Beispiel ??.

Wenn das Domain `T` keinen "`expr`"-Slot hat, dann liefert `expr` `FAIL` zurück.

⌘ Das Resultat von `expr` wird ohne Evaluierung zurückgegeben; zur Evaluierung kann `eval` benutzt werden. Siehe Beispiel ??.

Beispiel 1. `expr` konvertiert ein Polynom in einen Ausdruck, einen Bezeichner oder eine Zahl:

```
>> expr(poly(x^2 + y, [x])), expr(poly(x)), expr(poly(2, [x]));
      map(%, domtype)
```

$$y + x^2, x, 2$$

DOM_EXPR, DOM_IDENT, DOM_INT

Die Objekte `infinity` und `complexInfinity` werden in die gleichnamigen Bezeichner konvertiert:

```
>> expr(infinity), expr(complexInfinity);
      map(%, domtype)
```

infinity, complexInfinity

DOM_IDENT, DOM_IDENT

Wenn diese Bezeichner mit `eval` evaluiert werden, sind die Ergebnisse die originalen Objekte der Typen `stdlib::Infinity` und `stdlib::CInfinity`:

```
>> expr(infinity), expr(complexInfinity);
      map(eval(%), domtype)
```

infinity, complexInfinity

stdlib::Infinity, stdlib::CInfinity

Beispiel 2. Dieses Beispiel zeigt, dass `expr` rekursiv auf Ausdrücken arbeitet. Alle Teilausdrücke, die Domainelemente sind, werden in Ausdrücke konvertiert. In früheren Versionen von MuPAD (bis Version 1.4.2) wäre das Ergebnis `x + (1 mod 7)` gewesen. Die Konstruktion mit `hold(_plus)(...)` ist notwendig, weil `x + i(1)` sich sonst zu FAIL evaluieren würde:

```
>> i := Dom::IntegerMod(7):
      hold(_plus)(x, i(1)); expr(%)
```

x + (1 mod 7)

x + 1

Beispiel 3. Die Funktion `series` liefert ein Element des Domains `Series::Puisseux` zurück, bei dem es sich nicht um ein Basisdomain handelt:

```
>> s := series(sin(x), x);
      domtype(s)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^6)$$

`Series::Puisseux`

Man verwende `expr`, um das Ergebnis in ein Element vom Domain-Typ `DOM_EXPR` zu konvertieren:

```
>> e := expr(s); domtype(e)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120}$$

`DOM_EXPR`

Man beachte, dass die Information über den Ordnungsterm bei der Konvertierung verloren geht.

Beispiel 4. `expr` evaluiert sein Ergebnis nicht. In diesem Beispiel hat das Polynom `p` einen Parameter `a` und die globale Variable `a` hat einen Wert. Wird `expr` auf das Polynom `p` angewendet, so wird ein Ausdruck zurückgeliefert, der `a` enthält. Will man den Wert von `a` einsetzen, so muss man `eval` verwenden:

```
>> p := poly(a*x, [x]); a := 2; expr(p); eval(%)
```

`a x`

`2 x`

Beispiel 5. `A` ist ein Element vom Typ `Dom::Matrix(Dom::Integer)`:

```
>> A := Dom::Matrix(Dom::Integer)([[1, 2], [3, 2]]);
      domtype(A)
```

$$\begin{array}{cc} + - & - + \\ | & 1, 2 \\ | & \\ | & 3, 2 \\ | & \\ + - & - + \end{array}$$

`Dom::Matrix(Dom::Integer)`

`expr` wandelt dann die Matrix `A` in ein Element vom Typ `DOM_ARRAY` um:

```
>> a := expr(A); domtype(a)
```

$$\begin{array}{cc} + - & - + \\ | & 1, 2 \\ | & \\ | & 3, 2 \\ | & \\ + - & - + \end{array}$$

`DOM_ARRAY`

Es ist jedoch nicht garantiert, dass das Ergebnis auch in zukünftigen Versionen von **MuPAD** vom Typ `DOM_ARRAY` ist. So könnte sich zum Beispiel die interne Repräsentation von Matrizen in Zukunft ändern. Daher sollte man `coerce` verwenden, um die Konvertierung in einen bestimmten Datentyp zu erhalten:

```
>> coerce(A, DOM_ARRAY)
```

$$\begin{array}{cc} + - & - + \\ | & 1, 2 \\ | & \\ | & 3, 2 \\ | & \\ + - & - + \end{array}$$

Eine alternative Darstellung für eine Matrix ist eine geschachtelte Liste:

```
>> coerce(A, DOM_LIST)
```

```
[[1, 2], [3, 2]]
```

Beispiel 6. Wenn in Teilobjekt zu einem Domain ohne einen "expr"-Slot gehört, dann liefert `expr` `FAIL`:

```
>> T := newDomain("T");
    d := new(T, 1, 2);
    expr(d)
```

```
new(T, 1, 2)
```

```
FAIL
```

Man kann die Funktionalität von `expr` für eigene Domains erweitern. Wir demonstrieren dies für das Domain `T` durch die Implementation eines "`expr`"-Slots, der eine Liste mit den internen Operanden seines Arguments zurückliefert:

```
>> T::expr := x -> [extop(x)]:
```

Wenn `expr` nun während des rekursiven Durchlaufs auf ein Teilobjekt des Typs `T` stößt, ruft es die Slot Routine `T::expr` mit dem Teilobjekt als Argument auf:

```
>> expr(d), expr([d, 3])  
  
[1, 2], [[1, 2], 3]
```

Änderungen:

- ⌘ `expr` ist keine Kern-Funktion mehr.
- ⌘ `expr` arbeitet jetzt rekursiv. Siehe Beispiel ??.

`expr2text` – Umwandlung von Objekten in Zeichenketten

`expr2text(object)` konvertiert `object` in eine Zeichenkette.

Aufruf(e):

- ⌘ `expr2text(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: eine Zeichenkette.

Überladbar durch: `object`

Verwandte Funktionen: `coerce`, `fprint`, `int2text`, `tbl2text`, `text2expr`, `text2int`, `text2list`, `text2tbl`, `print`

Details:

- ☞ `expr2text(object)` wandelt `object` in eine Zeichenkette um, die gewöhnlich der Bildschirmausgabe von `object` entspricht, wenn `PRETTYPRINT` auf `FALSE` gesetzt ist.
- ☞ Wird die Funktion ohne Argument aufgerufen, ist die Ausgabe eine leere Zeichenkette. Sind mehrere Argumente angegeben, so werden sie als eine Folge von Ausdrücken behandelt und in eine einzige Zeichenkette umgewandelt.
- ☞ Wie die meisten anderen MuPAD-Funktionen auch wertet `expr2text` seine Argumente vor der Konvertierung aus.
- ☞ Kommen in `object` Zeichenketten vor, so werden diese im Ergebnis in Anführungszeichen eingeschlossen.
- ☞ `expr2text` ist eine Funktion des Systemkerns.

Beispiel 1. Ausdrücke werden in Zeichenketten umgewandelt:

```
>> expr2text(a + b)
```

```
"a + b"
```

`expr2text` schließt Zeichenketten in Anführungszeichen ein, denen bei der Bildschirmausgabe ein Backslash vorangestellt wird:

```
>> expr2text(["text", 2])
```

```
"[\"text\", 2]"
```

Beispiel 2. Ist mehr als ein Argument gegeben, so werden die Argumente wie eine Ausdrucksfolge behandelt:

```
>> expr2text(a, b, c)
```

```
"a, b, c"
```

Ist kein Argument gegeben, so wird eine leere Zeichenkette erzeugt:

```
>> expr2text()
```

```
" "
```

Beispiel 3. `expr2text` evaluiert seine Argumente:

```
>> a := b; c := d; expr2text(a, c)

      "b, d"
```

Man verwende `hold`, um die Auswertung zu verhindern:

```
>> expr2text(hold(a, c));
      delete a, c:

      "a, c"
```

Ein weiteres Beispiel:

```
>> expr2text((a := b; c := d));
      delete a, c:

      "d"

>> e := expr2text(hold((a := b; c := d)))

      "(a := b; \nc := d)"
```

Die letzte Zeichenkette enthält ein Zeilenumbruchszeichen `,\n'`. Die Funktion `print` mit der Option `Unquoted` gibt die Zeichenkette mit expandierten Zeilenumbrüchen aus:

```
>> print(Unquoted, e):

      (a := b;
      c := d)
```

Beispiel 4. `expr2text` ist überladbar. Ist in einem Domain weder eine `"print"`- noch eine `"expr2text"`-Methode vorhanden, so wird eine Standardausgabe für dessen Elemente verwendet:

```
>> T := newDomain("T"); e := new(T, 1):
      e;
      print(e):
      expr2text(e)

      new(T, 1)

      new(T, 1)

      "new(T, 1)"
```

Gibt es eine "print"-Methode, so wird diese von `expr2text` aufgerufen, um die Ausgabe zu erzeugen:

```
>> T::print := proc(x) begin
    _concat("foo: ", expr2text(extop(x)))
end_proc:
e;
print(e):
expr2text(e)
```

foo: 1

foo: 1

"foo: 1"

Soll sich die `expr2text`-Ausgabe von der normalen `print`-Ausgabe unterscheiden, so kann man eine "expr2text"-Methode definieren:

```
>> T::expr2text := proc(x) begin
    _concat("bar: ", expr2text(extop(x)))
end_proc:
e;
print(e):
expr2text(e)
```

foo: 1

foo: 1

"bar: 1"

Hintergründe:

- Bei der Bearbeitung eines Domain-Elements `e` versucht `expr2text` zuerst, die "expr2text"-Methode des entsprechenden Domains `T` aufzurufen. Wenn sie existiert, so wird `T::expr2text(e)` aufgerufen und das Ergebnis davon zurückgeliefert. Wenn keine "expr2text"-Methode existiert, so versucht `expr2text` in derselben Weise, die "print"-Methode aufzurufen. Wenn keine der beiden Methoden existiert, so generiert `expr2text` eine Standard-Ausgabe. Siehe Beispiel ??.

Eine "expr2text"- oder "print"-Methode kann ein beliebiges MuPAD-Objekt zurückliefern, welches dann rekursiv von `expr2text` behandelt wird.

Das zurückgelieferte Objekt darf das Domainelement `e` nicht enthalten, da sonst der MuPAD-Kern in eine Endlosrekursion gerät.



- ☞ Das Ergebnis von `expr2text` für einen Ausdruck stimmt immer mit der durch `print` erzeugten Bildschirmausgabe überein. Ist der nullte Operand des Ausdrucks eine Funktionsumgebung, so berechnet deren zweiter Operand das Ergebnis von `expr2text`.

Änderungen:

- ☞ Aufgrund einiger Ausgabeänderungen liefert `expr2text` manchmal eine Zeichenkette, die anders formatiert ist als in früheren MuPAD-Versionen. Die neue Ausgabeformatierung ist schöner als zuvor. Zum Beispiel liefert `expr2text(b-1/a)` nun " $b - 1/a$ " an Stelle von " $b + a^{(-1)}*(-1)$ ".
-

external – Erzeugen einer Modul-Funktionsumgebung

`external("mstring", "fstring")` liefert die Funktionsumgebung der Modulfunktion mit dem Namen `mstring::fstring`.

Aufruf(e):

- ☞ `external("mstring", "fstring")`

Parameter:

- "mstring" — der Modulname: eine Zeichenkette
- "fstring" — der Name einer Modulfunktion: eine Zeichenkette

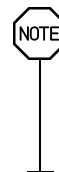
Rückgabewert: eine Funktionsumgebung vom Typ `DOM_FUNC_ENV`.

Verwandte Funktionen: `loadmod`, `module::new`, `unloadmod`

Details:

- ☞ `external("mstring", "fstring")` erzeugt die Funktionsumgebung der Modulfunktion `mstring::fstring` und liefert diese als Rückgabewert.
- ☞ Eine Datei `mstring.mdg` kann existieren, die gegebenenfalls MuPAD-Objekte enthält, welche geladen und in die Modul-Funktionsumgebung eingebunden werden. Tritt hierbei ein Fehler auf, so wird eine Warnung ausgegeben. MuPAD versucht bei jedem Aufruf der hiervon betroffenen Modulfunktionen, diese Objekte erneut zu laden.
- ☞ Mittels `external` kann auf eine Modulfunktion zugegriffen werden, ohne das Modul explizit zu laden und ohne das Modul-Domain zu erstellen. Beim Ausführen der Modulfunktion wird der Maschinencode bei Bedarf automatisch geladen.

☞ Manche Modulfunktionen arbeiten nur korrekt, wenn ihr Modul-Domain vorher erzeugt wurde. Derartige Module müssen mittels `loadmod` geladen werden, bevor eine ihrer Modulfunktionen ausgeführt wird. Bitte beachten Sie entsprechende Hinweise in der Dokumentation des Moduls.



☞ `external` ist eine Funktion des Systemkerns.

Beispiel 1. Modul-Funktionsumgebungen können in lokalen oder globalen Variablen gespeichert werden. Mit ihnen können Modulfunktionen ohne vorheriges Laden des Moduls verwendet werden:

```
>> where := external("stdmod", "which"): where("stdmod")  
      "/usr/local/mupad/linux/modules/stdmod.mdm"  
  
>> delete where:
```

Hintergründe:

☞ Die Kernfunktionen `external`, `loadmod` und `unloadmod` sind Basisfunktionen zum Zugriff auf Module. Weitere Funktionen stehen in der Bibliothek `module` zur Verfügung.

Änderungen:

☞ Keine Änderungen.

`extnops` – die Anzahl der Operanden eines Domain-Elementes

`extnops(object)` liefert die Anzahl der Operanden der internen Darstellung des Objekts.

Aufruf(e):

☞ `extnops(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: eine nichtnegative ganze Zahl.

Verwandte Funktionen: `DOM_DOMAIN`, `extop`, `extsubsop`, `new`, `nops`, `op`, `subsop`

Details:

- ⌘ Für Objekte der elementaren Datentypen wie z. B. Ausdrücke, Mengen, Listen, Tabellen, Arrays usw. liefert `extnops` dasselbe Resultat wie die Funktion `nops`. Der einzige Unterschied zur Funktion `nops` ist, dass `extnops` nicht von in der MuPAD-Sprache implementierten „Bibliotheksdatentypen“ überladen werden kann.
 - ⌘ Domainelemente können intern aus einer beliebigen Anzahl von Datenobjekten bestehen; `extnops` liefert die tatsächliche Anzahl *interner* Operanden. Jedes Domain sollte Schnittstellenfunktionen für den Zugriff von außen besitzen, und `extnops` sollte nur von internen Methoden des Domains benutzt werden. Äußerer Zugriff sollte mittels `nops` erfolgen.
 - ⌘ `extnops` ist eine Funktion des Systemkerns.
-

Beispiel 1. `extnops` liefert die Anzahl der Einträge eines Domainelements:

```
>> d := newDomain("demo"): e := new(d, 1, 2, 3, 4): extnops(e)
4

>> delete d, e:
```

Beispiel 2. Für Datentypen des MuPAD-Kerns ist `extnops` äquivalent zu `nops`:

```
>> extnops([1, 2, 3, 4]), nops([1, 2, 3, 4])
4, 4
```

Beispiel 3. Es soll ein Domain für Listen implementiert werden. Die interne Darstellung besteht aus einem einzigen Objekt, nämlich einer Liste vom Kerntyp `DOM_LIST`:

```
>> myList := newDomain("lists"):
    myList::new := proc(l : DOM_LIST) begin new(myList, l) end_proc:
```

Das Domain soll sich ähnlich verhalten wie der Kerndatentyp `DOM_LIST`. Um dies zu erreichen, wird die Funktion `nops` überladen. Dabei wird mittels `extop(l, 1)` auf die interne Liste zugegriffen:

```
>> myList::nops := l -> nops(extop(l, 1)):
```

Ein Element des neuen Listen-Domains wird erzeugt:

```
>> mylist := myList([1, 2, 3])  
  
new(lists, [1, 2, 3])
```

Da `nops` überladen wurde, bietet `extnops` an dieser Stelle die einzige Möglichkeit, die Anzahl der Operanden der internen Darstellung von `mylist` zu bestimmen. Im Gegensatz zu `nops` liefert `extnops` stets 1, da die interne Darstellung aus einer einzigen Liste besteht:

```
>> nops(mylist), extnops(mylist)  
  
3, 1  
  
>> delete myList, mylist:
```

Änderungen:

☞ `extnops` arbeitet nun auch auf Elementen der Kern-Domains.

`extop` – die Operanden eines Domain-Elementes

`extop(object)` liefert alle Operanden des Domain-Elements `object`.

`extop(object, i)` liefert den *i*-ten Operanden.

`extop(object, i..j)` liefert den *i*-ten bis *j*-ten Operanden.

Aufruf(e):

☞ `extop(object)`
☞ `extop(object, i)`
☞ `extop(object, i..j)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt
`i, j` — nichtnegative ganze Zahlen

Rückgabewert: eine Folge von Operanden oder der angegebene Operand.
`FAIL` wird geliefert, wenn kein entsprechender Operand existiert.

Verwandte Funktionen: `DOM_DOMAIN`, `extnops`, `extsubsop`, `new`, `nops`, `op`, `subsop`

Details:

- ⇒ Für Objekte der elementaren Datentypen wie z. B. Ausdrücke, Mengen, Listen, Tabellen, Arrays usw. liefert `extop` dieselben Operanden wie die Funktion `op`. Der wichtigste Unterschied zur Funktion `op` ist, dass `extop` nicht durch Überladung umdefiniert werden kann. Damit ermöglicht `extop` einen sicheren Zugriff auf die Operanden der *internen Darstellung* von Bibliotheksdatentypen. Typischerweise wird `extop` in der Implementation der "`op`"-Methode solcher Datentypen verwendet.
- ⇒ Ein Domain-Element enthält einen Verweis auf sein Domain und eine Folge von Werten, die seinen Inhalt beschreiben. `extop` ermöglicht den Zugriff auf das Domain und die Operanden dieser internen Datenfolge.
- ⇒ `extop(object)` gibt eine Folge aller internen Operanden (außer dem 0-ten) zurück. Dieser Aufruf ist äquivalent zu `extop(object, 1..ext-nops(object))`.
- ⇒ `extop(object, i)` gibt den *i*-ten internen Operanden zurück. Insbesondere liefert `extop(object, 0)` das Domain des Objekts zurück, wenn `object` ein Element eines Bibliotheksdatentypen ist. Wenn `object` ein Element eines elementaren Datentypen ist, ist der Aufruf `extop(object, 0)` äquivalent zu `op(object, 0)`.
- ⇒ `extop(object, i..j)` gibt eine aus dem *i*-ten bis zum *j*-ten internen Operanden bestehende Folge zurück; *i* und *j* müssen dabei nichtnegative ganze Zahlen und *i* darf nicht größer als *j* sein. Dieser Aufruf ist äquivalent zu `extop(object, k) $ k = i..j`.
- ⇒ `extop` liefert `FAIL`, wenn ein angeforderter Operand nicht existiert. Siehe Beispiel ??.
- ⇒ Die Operanden einer Ausdrucksfolge sind deren Elemente. Man beachte, dass Folgen nicht durch `extop` ausgeglichen werden.
- ⇒ `extop` ist eine Funktion des Systemkerns.

Beispiel 1. `extop` liefert alle Operanden eines Domain-Elements:

```
>> d := newDomain("demo"): e := new(d, 1, 2, 3): extop(e)
1, 2, 3
```

Einzelne Operanden können ausgewählt werden:

```
>> extop(e, 2)
```

Bereiche von Operanden können ausgewählt werden:

```
>> extop(e, 1..2)
```

1, 2

Der 0-te Operand eines Domain-Elements ist sein Domain:

```
>> extop(e, 0)
```

demo

```
>> delete d, e:
```

Beispiel 2. Zunächst wird mittels `newDomain` ein neuer Datentyp `d` definiert. Die `"new"`-Methode dient zur Erzeugung von Elementen dieses Typs. Die interne Darstellung des Datentyps ist die Folge der Argumente der `"new"`-Methode:

```
>> d := newDomain("d"): d::new := () -> new(dom, args()):
```

Die `op`-Funktion des Systems wird durch die folgende `"op"`-Methode dieses Datentyps überladen. Sie soll die Elemente einer sortierten Kopie der internen Datenfolge zurückliefern. In der Implementation wird dabei mittels `extop` auf die internen Daten zugegriffen:

```
>> d::op := proc(x, i = null())
    local internalData;
    begin internalData := extop(x);
           op(sort([internalData]), i)
    end_proc:
```

Mit dieser Überladung liefert `op` andere Operanden als `extop`:

```
>> e := d(3, 7, 1): op(e); extop(e)
```

1, 3, 7

3, 7, 1

```
>> delete d, e:
```

Beispiel 3. Für die Kerndatentypen wie z. B. Mengen, Listen etc. liefert `extop` immer dieselben Operanden wie `op`:

```
>> extop([a, b, c]) = op([a, b, c])

      (a, b, c) = (a, b, c)
```

Ausdrücke sind vom Kerndatentyp `DOM_EXPR`, folglich ist `extop(sin(x), 0)` äquivalent zu `op(sin(x), 0)`:

```
>> domtype(sin(x)), extop(sin(x), 0) = op(sin(x), 0)

      DOM_EXPR, sin = sin
```

Ausdrucksfolgen werden nicht ausgeglichen:

```
>> extop((1, 2, 3), 0), extop((1, 2, 3))

      _exprseq, 1, 2, 3
```

Beispiel 4. Nicht existierende Operanden werden als `FAIL` zurückgeliefert:

```
>> extop([1, 2], 4), extop([1, 2], 1..4)

      FAIL, FAIL
```

Änderungen:

☞ `extop` arbeitet nun auf Elementen der Kern-Domains genauso wie `op`.

`extsubsop` – Substitution von Operanden eines Domain-Elementes

`extsubsop(d, i = new)` liefert eine Kopie des Domain-Elements `d`, in der der `i`-te Operand der internen Darstellung durch den Wert `new` ersetzt ist.

Aufruf(e):

☞ `extsubsop(d, i1 = new1, i2 = new2, ...)`

Parameter:

<code>d</code>	— ein Element eines Bibliotheks-Domains
<code>i1, i2, ...</code>	— nichtnegative ganze Zahlen
<code>new1, new2, ...</code>	— beliebige MuPAD-Objekte

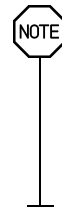
Rückgabewert: das Eingabeobjekt mit ersetzten Operanden.

Verwandte Funktionen: `DOM_DOMAIN`, `extnops`, `extop`, `new`, `nops`, `op`, `subs`, `subsex`, `subsop`

Details:

- ☞ Intern kann jedes Domain-Element aus einer beliebigen Anzahl von Objekten bestehen. `extsubsop` ersetzt einige dieser Objekte, ohne darauf zu achten, ob die Ersetzung sinnvoll ist.

Die Elemente der Domains der MuPAD-Bibliothek müssen hinsichtlich ihrer Operanden bestimmten undokumentierten Konventionen genügen; `extsubsop` sollte daher nur auf Elemente benutzerdefinierter Domains angewendet werden. Es ist guter Programmierstil, diese Funktion nur innerhalb von Domain-Methoden einzusetzen.



- ☞ `extsubsop` liefert eine substituierte Kopie des Objektes ohne das Objekt selbst zu verändern.
- ☞ Die Nummerierung der Operanden ist dieselbe, die auch von `extop` verwendet wird.
- ☞ Falls ein 0-ter Operand substituiert werden soll, so muss der entsprechende neue Wert ein Domain vom Typ `DOM_DOMAIN` sein. In diesem Fall ersetzt `extsubsop` das Domain von `d` durch dieses neue Domain.
- ☞ Soll der *i*-te Operand verändert werden, wobei *i* größer ist als die tatsächliche Anzahl der Operanden, so erhöht `extsubsop` zunächst die Anzahl der Operanden durch Hinzufügen entsprechend vieler NILs und führt dann die Substitution durch. Siehe Beispiel ??.
- ☞ Wird der *i*-te Operand durch eine Ausdrucksfolge mit *k* Elementen ersetzt, so wird jedes dieser Elemente ein eigener Operand der substituierten Kopie. Sie tragen die Indizes von *i* bis *i+k-1*, weitere Operanden von *d* werden entsprechend nach rechts verschoben. Die neue Nummerierung gilt schon für weitere Substitutionen im selben Aufruf von `extsubsop`. Siehe Beispiel ??.
- ☞ Das leere Objekt `null()` wird zum Operanden des Resultats, wenn es in ein Objekt hineinsubstituiert wird: es wird nicht entfernt.
- ☞ Nach Durchführung der Substitution wird das Ergebnis nicht noch ein weiteres Mal ausgewertet. Siehe Beispiel ??.
- ☞ Im Unterschied zu `subsop` kann `extsubsop` nicht überladen werden.
- ☞ Im Gegensatz zu `extop` und `extnops` kann `extsubsop` nicht auf Elemente eines Kerntyps angewendet werden.

☞ extsubsop ist eine Funktion des Systemkerns.

Beispiel 1. Wir erzeugen ein Domain-Element und ersetzen dann seinen ersten Operanden:

```
>> d := newDomain("1st"): e := new(d, 1, 2, 3): extsubsop(e, 1 = 5)

new(1st, 5, 2, 3)
```

Dies ändert den Wert von e nicht:

```
>> e

new(1st, 1, 2, 3)

>> delete d, e:
```

Beispiel 2. Durch Ersetzung des 0-ten Operanden kann der Domain-Typ eines Objekts geändert werden.

```
>> d := newDomain("some_domain"): e := new(d, 2):
extsubsop(e, 0 = Dom::IntegerMod(5))

2 mod 5

>> delete d, e:
```

Beispiel 3. Der sechste Operand eines Domain-Elements mit weniger als sechs Operanden soll ersetzt werden. In solchen Fällen wird eine entsprechende Anzahl von NILs eingefügt:

```
>> d := newDomain("example"): e := new(d, 1, 2, 3, 4):
extsubsop(e, 6 = 8)

new(example, 1, 2, 3, 4, NIL, 8)

>> delete d, e:
```

Beispiel 4. Im folgenden Beispiel wird der erste Operand eines Domain-Elements durch eine Folge mit drei Elementen ersetzt. Diese werden zu den ersten drei Operanden des Ergebnisses, der zweite Operand von `e` wird zum vierten Operanden, usw. Die zweite Substitution im selben Aufruf bezieht sich schon auf die neue Numerierung:

```
>> d := newDomain("example"): e := new(d, 1, 2, 3, 4):
      extsubsup(e, 1 = (11, 13, 17), 2 = (29, 99))

      new(example, 11, 29, 99, 17, 2, 3, 4)

>> delete d, e:
```

Beispiel 5. Wir definieren ein Domain mit eigener Auswertungsfunktion. Diese Funktion gibt ihr Argument auf dem Bildschirm aus, sodass man sehen kann, wenn sie aufgerufen wird. Danach definieren wir ein Element des Domains.

```
>> d := newDomain("anotherExample"):
      d::evaluate := x -> (print("Argument:", x); x):
      e := new(d, 3)

      new(anotherExample, 3)
```

Nun kann man alle Auswertungen verfolgen: `extsubsup` wertet seine Argumente aus und führt die gewünschte Ersetzung durch; das Ergebnis wird nicht erneut ausgewertet:

```
>> extsubsup(e, 1 = 0)

      "Argument:", new(anotherExample, 3)

      new(anotherExample, 0)

>> delete d, e:
```

Änderungen:

☞ Keine Änderungen.

fact – die Fakultätsfunktion

`fact(n)` stellt die Fakultät $n! = 1 \times 2 \times 3 \times \cdots \times n$ einer ganzen Zahl dar.

Aufruf(e):

⌘ `fact(n)`
⌘ `n!`

Parameter:

`n` — ein arithmetischer Ausdruck, der eine nicht-negative ganze Zahl repräsentiert

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `n`

Verwandte Funktionen: `gamma`, `igamma`, `psi`

Details:

- ⌘ Der Aufruf `n!` ist äquivalent zu `fact(n)`.
 - ⌘ Für nicht-negative ganze Zahlen `n` wird eine ganze Zahl zurückgeliefert, für andere numerische Argumente wird ein Fehler ausgelöst. Ist `n` ein symbolischer Ausdruck, so wird ein symbolischer Aufruf von `fact` zurückgeliefert.
 - ⌘ Ganzzahlige Argumente müssen auf 32-Bit-Systemen kleiner als 2^{31} sein, auf 64-Bit-Systemen kleiner als 2^{63} . Größere Argumente lösen einen Fehler aus.
 - ⌘ Die `gamma`-Funktion verallgemeinert die Fakultätsfunktion auf beliebige komplexe Argumente. Sie erfüllt `gamma(n+1) = n!` für nicht-negative ganze Zahlen `n`. Mittels `rewrite(Ausdruck, gamma)` kann ein Ausdruck umgeschrieben werden, der symbolische Aufrufe von `fact` enthält. Siehe Beispiel ??.
 - ⌘ Der Operator `!` kann auch in Präfix-Notation (mit anderer Bedeutung) eingesetzt werden: `!command` ist äquivalent zu `system("command")`.
 - ⌘ `fact` ist eine Funktion des Systemkerns.
-

Beispiel 1. Für nicht-negative ganzzahlige Argumente wird die Fakultät berechnet:

```
>> fact(0), fact(5), fact(2^5)
1, 120, 263130836933693530167218012160000000
```

Für symbolische Ausdrücke wird ein symbolischer Aufruf von `fact` zurückgeliefert:

```
>> fact(n), fact(n - sin(x)), fact(3.0*n + I)

      2
fact(n), fact(n - sin(x)), fact(3.0 n + I)
```

Die Aufrufe `fact(n)` und `n!` sind äquivalent:

```
>> 5! = fact(5), (n^2 + 3)!

      2
120 = 120, fact(n + 3)
```

Ein numerisches Argument löst einen Fehler aus, wenn es keine ganze Zahl ist:

```
>> fact(3/2 + I)

Error: Non-negative integer expected [specfunc::fact];
during evaluation of 'fact'
```

Beispiel 2. Eine numerische Näherung sollte mittels `gamma(float(n+1))` statt `float(fact(n))` angefordert werden, um die zeitaufwendige Berechnung sehr großer ganzer Zahlen zu vermeiden:

```
>> float(fact(2^13)) = gamma(float(2^13 + 1))

1.275885799e28503 = 1.275885799e28503
```

Beispiel 3. Die Funktionen `expand`, `limit`, `rewrite` und `series` verarbeiten symbolische `fact`-Ausdrücke:

```
>> expand(fact(n^2 + 4))

      2      2      2      2      2
fact(n ) (n + 1) (n + 2) (n + 3) (n + 4)

>> limit(fact(n)/exp(n), n = infinity)

infinity

>> rewrite(fact(2*n^2 + 1)/fact(n - 1), gamma)

      2
gamma(2 n + 2)
-----
gamma(n)
```

Die Stirling-Formel erhält man durch asymptotische Entwicklung:

```
>> series(fact(n), n = infinity, 2)
```

$$\begin{aligned}
 & \frac{(n+1)^{n+1} \exp(-n-1)}{\sqrt{n+1}} \left(\frac{\sqrt{2\pi}}{\sqrt{n+1}} \right)^{1/2} + \\
 & \frac{(n+1)^{n+1} \exp(-n-1)}{\sqrt{n+1}} \left(\frac{\sqrt{2\pi}}{\sqrt{n+1}} \right)^{1/2} + \\
 & \frac{1}{12n} + \\
 & \frac{1}{n^2} \left(\frac{(n+1)^{n+1} \exp(-n-1)}{\sqrt{n+1}} \left(\frac{\sqrt{2\pi}}{\sqrt{n+1}} \right)^{1/2} \right)^2
 \end{aligned}$$

Änderungen:

☞ Keine Änderungen.

factor – Faktorisierung von Polynomen und Ausdrücken

`factor(f)` liefert eine Faktorisierung des Polynoms f in irreduzible Faktoren, d.h. eine Darstellung von f in der Form $f = u \cdot f_1^{e_1} \cdot \dots \cdot f_r^{e_r}$, wobei u der Inhalt von f und f_i ($1 \leq i \leq r$) die irreduziblen Faktoren von f sind.

Aufruf(e):

☞ `factor(f)`

Parameter:

f — ein Polynom oder ein arithmetischer Ausdruck

Rückgabewert: ein faktorisiertes Objekt, d.h. ein Objekt vom Domain-Typ `Factored`.

Überladbar durch: f

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `collect`, `content`, `denom`, `div`, `divide`,
`expand`, `Factored`, `gcd`, `icontent`, `ifactor`, `igcd`, `ilcm`, `indets`,
`irreducible`, `isprime`, `lcm`, `normal`, `numer`, `partfrac`,
`polylib::decompose`, `polylib::divisors`, `polylib::primpart`,
`polylib::sqrfree`, `rationalize`, `simplify`

Details:

☞ `factor` schreibt sein Argument in ein Produkt von möglichst vielen Termen um. In gewisser Weise handelt es sich um eine Art Umkehrfunktion von `expand`, das sein Argument als Summe so vieler Terme wie möglich schreibt.

☞ Ist f ein Polynom mit Koeffizientenring Expr , so wird f über dem kleinsten Ring faktorisiert, der alle Koeffizienten enthält. Mathematisch gesehen enthält dieser implizite Koeffizientenring immer die ganzen Zahlen. Siehe Beispiel ??.

Ist der Koeffizientenring R von f nicht Expr , so ist der implizite Koeffizientenring gleich R selbst. Elemente dieses Ringes werden nicht weiter in Faktoren zerlegt. Insbesondere ist der Inhalt u ein Element des impliziten Koeffizientenrings.

☞ Ist das Argument f ein arithmetischer Ausdruck, jedoch keine Zahl, so wird es als rationaler Ausdruck aufgefasst. Vor dem Faktorisieren werden irrationale Teilausdrücke wie $\sin(x)$, $\exp(1)$, $x^{(1/3)}$ durch Hilfsvariablen ersetzt; dies gilt nicht für konstante algebraische Teilausdrücke wie 1 und $(\sqrt{2}+1)^3$. Bei der Ersetzung werden algebraische Abhängigkeiten der irrationalen Teilausdrücke untereinander, wie z. B. $\cos(x)^2 = 1 - \sin(x)^2$, in vielen Fällen nicht berücksichtigt; siehe Beispiel ??.

Der sich so ergebende Ausdruck wird dann als Quotient zweier polynomialer Ausdrücke in den ursprünglichen Unbestimmten sowie den neu hinzugekommenen Hilfsvariablen aufgefasst. Zähler und Nenner werden in Polynome über expr umgewandelt; impliziter Koeffizientenring ist der kleinste Ring, der alle Koeffizienten des Zähler- und des Nennerpolynoms umfasst; dies ist meist der Ring der ganzen Zahlen. Danach werden Zähler- und Nennerpolynom über diesem Ring faktorisiert, wobei den Faktoren des Nenners negative Vielfachheit e_i zugeschrieben wird; siehe Beispiel ??. Nach der Faktorisierung werden die Hilfsvariablen wieder durch die ursprünglichen Teilausdrücke ersetzt, siehe Beispiel ??.

☞ Ist f eine ganze Zahl, so wird sie in Primfaktoren zerlegt; das Ergebnis ist dasselbe, das auch `ifactor` liefert. Ist f eine rationale Zahl, so werden ihr Zähler und ihr Nenner in Primfaktoren zerlegt; die Faktoren des Nenners haben negative Vielfachheit. Siehe Beispiel ??.

☞ Ist f eine Gleitkommazahl oder eine komplexe Zahl, so liefert `factor` eine Faktorisierung mit f als einzigem Faktor.

☞ Das Ergebnis von `factor` ist ein Objekt vom Domain-Typ `Factored`. Ist $g := \text{factor}(f)$ ein solches Objekt, so ist dessen interne Darstellung die Liste $[u, f_1, e_1, \dots, f_r, e_r]$ von ungerader Länge ($= 2r + 1$).

Hierbei sind f_1 bis f_r vom selben Typ wie die Eingabe (Polynome oder arithmetische Ausdrücke), e_1 bis e_r sind Zahlen, und u ist ein arithmetischer Ausdruck.

Auf den Inhalt u , die Faktoren f_i sowie die Exponenten e_i lässt sich u. a. über den gewöhnlichen Indexoperator `[]` zugreifen, d. h. $g[1] = u$, $g[2] = f_1$, $g[3] = e_1$,

Die Eingabe $g[2*i] \ \$ \ i = 1..nops(g) \ \text{div} \ 2$ liefert beispielsweise alle irreduziblen Faktoren von f . Das kann auch – etwas komfortabler – über den Aufruf `Factored::factors(g)` erreicht werden. Entsprechend liefert `Factored::exponents(g)` eine Liste der Exponenten e_i ($1 \leq i \leq r$) der Faktorisierung von f .

Die Eingabe `coerce(g, DOM_LIST)` liefert die interne Darstellung eines faktorisierten Objektes, d. h. die Liste wie oben beschrieben.

Man beachte, dass das Ergebnis von `factor` wie ein Ausdruck ausgegeben wird und sich im Prinzip auch als solcher verhält. Beispielsweise ist das Ergebnis $q := \text{factor}(x^2+2*x+1)$ ein Objekt, das in der Form $(x+1)^2$ ausgegeben wird und vom Ausdruckstyp `"_power"` ist.

Siehe Beispiel ?? . Zu Details siehe auch die Hilfeseite zu `Factored`.

☞ Ist f keine Zahl, so ist jedes der Polynome p_1, \dots, p_r primitiv, d. h. der größte gemeinsame Teiler seiner Koeffizienten über dem impliziten Koeffizientenring (Definition siehe oben) ist eins. Siehe dazu auch `content` und `gcd`.

☞ Faktorisieren von Polynomen ist gegenwärtig nur über den folgenden Koeffizientenringen möglich: Ganze und rationale Zahlen, endliche Körper (repräsentiert durch `IntMod(n)` oder `Dom::IntegerMod(n)`, wo n Primzahl, oder durch ein `Dom::GaloisField`), und alle daraus durch folgende Konstruktionen erhaltenen Ringe: Polynomringe (`Dom::DistributedPolynomial`, `Dom::MultivariatePolynomial`, `Dom::Polynomial` und `Dom::UnivariatePolynomial`), Quotientenkörper (`Dom::Fraction`), sowie algebraische Erweiterungen (`Dom::AlgebraicExtension`). Insbesondere ist es *nicht* möglich, über den reellen und komplexen Zahlen zu faktorisieren.

☞ Ist die Eingabe f ein arithmetischer Ausdruck, jedoch keine Zahl, so werden alle darin auftretenden Gleitkommazahlen durch mittels Kettenbruchentwicklung ermittelte Näherungsbrüche ersetzt. Das Ergebnis hängt von dem Wert der Umgebungsvariablen `DIGITS` ab, siehe `numeric::rationalize` zu Details.

Beispiel 1. Der folgende Aufruf liefert eine Faktorisierung des Polynoms $x^3 + x$:

```
>> g := factor(x^3+x)
```

$$x (x^2 + 1)$$

Im Regelfall wird über den rationalen Zahlen faktorisiert, so dass Faktoren wie $x - 1$ nicht in Betracht kommen.

Auf die interne Darstellung dieser Faktorisierung kann dann u. a. über den gewöhnlichen Indexoperator zugegriffen werden:

```
>> g[1]; // der Inhalt
      g[2*i]      $ i = 1..nops(g) div 2; // die Faktoren
      g[2*i + 1] $ i = 1..nops(g) div 2; // die Exponenten
```

$$x, x^2 + 1, 1$$

Die vollständige interne Repräsentation von g , wie oben beschrieben, liefert die folgende Eingabe:

```
>> coerce(g, DOM_LIST)
```

$$[1, x, 1, x^2 + 1, 1]$$

Das Ergebnis der Faktorisierung ist ein Objekt des Domains `Factored`:

```
>> domtype(g)
```

`Factored`

Dieses Domain stellt einige Funktionalität zum Arbeiten mit solchen Objekten zur Verfügung, auf die wir an dieser Stelle nur kurz eingehen.

So lassen sich die Faktoren und Exponenten auch direkt wie folgt erfragen:

```
>> Factored::factors(g), Factored::exponents(g)
```

$$[x, x^2 + 1], [1, 1]$$

Es läßt sich die Art der Faktorisierung erfragen, was sich auf die Faktoren bezieht:

```
>> Factored::getType(g)
```


"irreducible"

Diese Ausgabe bedeutet, dass die f_i irreduzibel sind. Andere Typen von Faktorisierung sind "squarefree" (siehe hierzu die Funktion `polylib::sqrfree`) und "unknown".

Solche Objekte lassen sich multiplizieren, wobei das Ergebnis dann ebenfalls in faktorisierter Form vorliegt:

```
>> g2 := factor(x^2 + 2*x + 1)
```

$$(x + 1)^2$$

```
>> g * g2
```

$$x^2 (x + 1)^2 (x + 1)^2$$

Es lässt sich (nahezu) jede Funktion, die als Eingabe arithmetische Ausdrücke erwartet, auf ein solches Objekt anwenden. Das Ergebnis wird in der Regel nicht mehr ein Objekt des Domains `Factored` sein:

```
>> expand(g);  
domtype(%)
```

$$x^3 + x^2$$

DOM_EXPR

Für eine detaillierte Beschreibung solcher Objekte sei auf die Hilfeseite des Domains `Factored` verwiesen.

Beispiel 2. `factor` zerlegt ganze Zahlen in ihre Primfaktoren:

```
>> factor(8)
```

$$2^3$$

Bei rationalen Zahlen werden Zähler und Nenner faktorisiert:

```
>> factor(10/33)
```

$$\frac{2^2 \cdot 5}{3 \cdot 11}$$

Im Gegensatz dazu werden konstante Polynome *nicht* in irreduzible Faktoren zerlegt:

```
>> factor(poly(8, [x]))
```

$$8$$

Beispiel 3. Faktoren des Nenners sind an ihren negativen Vielfachheiten zu erkennen:

```
>> factor((z^2 - 1)/z^2)
```

$$\frac{(z + 1)(z - 1)}{z^2}$$

```
>> Factored::factors(%), Factored::exponents(%)
```

```
[z, z + 1, z - 1], [-2, 1, 1]
```

Beispiel 4. Sind einige Koeffizienten irrational, aber algebraisch, so wird über der kleinsten Körpererweiterung der rationalen Zahlen faktorisiert, die alle Koeffizienten enthält. Daher wird x^2+1 als irreduzibel betrachtet, das 1-fache davon aber als reduzibel:

```
>> factor(x^2 + 1), factor(I*x^2 + I)
```

$$x^2 + 1, I(x - I)(x + I)$$

MuPAD faktorisiert nicht über dem Körper der algebraischen Zahlen; vielmehr werden nur die Koeffizienten der Eingabe zu den rationalen Zahlen adjungiert:

```
>> factor(sqrt(2)*x^4 - sqrt(2)*x^2 - sqrt(2)*2)
```

$$2^{1/2} (x^2 + 2^{1/2}) (x^2 - 2^{1/2}) (x^2 + 1)$$

```
>> factor(I*x^4 - I*x^2 - I*2)
```

$$I(x - I)(x + I)(x^2 - 2)$$

```
>> factor(sqrt(2)*I*x^4 - sqrt(2)*I*x^2 - sqrt(2)*I*2)
```

$$(I 2^{1/2}) (x + I) (x + 2^{1/2}) (x - I) (x - 2^{1/2})$$

Beispiel 5. Transzendente Objekte werden wie Unbestimmte behandelt; sie sind irreduzibel, aber keine Einheiten:

```
>> delete x:
      factor(7*(exp(x)^2 - 1)*sin(1)^3)

              3
      7 (exp(x) + 1) (exp(x) - 1) sin(1)
```

```
>> Factored::factors(%), Factored::exponents(%)

      [exp(x) + 1, exp(x) - 1, sin(1)], [1, 1, 3]
```

Beispiel 6. `factor` behandelt voneinander verschiedene transzendente Teilausdrücke als algebraisch unabhängig. Daher wird im folgenden Fall nicht die erste binomische Formel angewandt:

```
>> factor(x + 2*sqrt(x) + 1)

              1/2
      x + 2 x      + 1
```

Beispiel 7. `factor` ersetzt Gleitkommazahlen durch Näherungsbrüche, die mittels Kettenbruchentwicklung ermittelt werden, faktorisiert das sich ergebende Polynom, und wendet schließlich `float` auf die Koeffizienten des Ergebnisses an:

```
>> factor(x^2 + 2.0*x - 8.0)

      (x + 4.0) (x - 2.0)
```

Beispiel 8. Polynome über anderen Ringen als `Expr` werden einfach über ihrem Koeffizientenring faktorisiert. Das folgende Polynom wird z. B. modulo 17 faktorisiert:

```
>> R := Dom::IntegerMod(17): f:= poly(x^3 + x + 1, R):
      factor(f)

      poly(x + 6, [x], Dom::IntegerMod(17))

              2
      poly(x  + 11 x + 3, [x], Dom::IntegerMod(17))
```

Statt `Dom::IntegerMod(p)` (wo `p` Primzahl ist) darf auch `IntMod(p)` verwendet werden:

```
>> R := IntMod(17): f:= poly(x^3 + x + 1, R):
    factor(f)

poly(x + 6, [x], IntMod(17)) poly(x^2 - 6 x + 3, [x], IntMod(17))
)
```

Beispiel 9. Auch komplexere Domains sind als Koeffizientenringe geeignet, wenn sie aus den rationalen Zahlen oder einem endlichen Körper durch aufeinanderfolgende Bildung von algebraischen Erweiterungen, Polynomringen und Quotientenkörpern entstehen. Im folgenden Beispiel wird das univariate Polynom $u^2 - x^3$ in der Variablen u über dem Körper $F = \mathbb{Q}(x, \sqrt{x})$ faktorisiert:

```
>> Q := Dom::Rational:
    Qx := Dom::Fraction(Dom::DistributedPolynomial([x], Q)):
    F := Dom::AlgebraicExtension(Qx, poly(z^2 - x, [z])):
    f := poly(u^2 - x^3, [u], F)

poly(u^2 - x^3, [u], Dom::AlgebraicExtension(
    Dom::Fraction(Dom::DistributedPolynomial([x],
        Dom::Rational, LexOrder)), - x + z^2 = 0, z))
>> factor(f)

poly(u - x z, [u], Dom::AlgebraicExtension(
    Dom::Fraction(Dom::DistributedPolynomial([x],
        Dom::Rational, LexOrder)), - x + z^2 = 0, z)) poly(u + x z,
    [u], Dom::AlgebraicExtension(Dom::Fraction(
        Dom::DistributedPolynomial([x], Dom::Rational, LexOrder)),
        - x + z^2 = 0, z))
```

Hintergründe:

- ⌘ Die verwendeten Algorithmen befinden sich in einem besonderen Bibliotheksdomain `facLib`; sie sollten nicht direkt aufgerufen werden.
- ⌘ Verwendet werden u. a. folgende Algorithmen: Cantor-Zassenhaus (über endlichen Körpern), Hensel Lifting (über \mathbb{Q} und für multivariate Polynome).

Änderungen:

- ⌘ Der Rückgabewert von `factor` ist ein Objekt des Domains `Factored`.
 - ⌘ Ganze und rationale Zahlen als Eingabe werden in Primfaktoren zerlegt.
-

`fclose` – Schließen einer Datei

`fclose(n)` schließt die durch den Dateibezeichner `n` bezeichnete Datei.

Aufruf(e):

- ⌘ `fclose(n)`

Parameter:

- `n` — ein von `fopen` erzeugter Dateibezeichner: eine positive ganze Zahl

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Verwandte Funktionen: `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

- ⌘ Die Datei muss zuvor mit `fopen` geöffnet worden sein. Der Aufruf an `fopen` liefert dabei den Dateibezeichner `n`.
 - ⌘ Dem System steht nur eine begrenzte Anzahl von Dateibezeichnern zur Verfügung. Eine nicht mehr benutzte Datei sollte daher umgehend mit `fclose` geschlossen werden, um den Dateibezeichner wieder freizugeben. Die genaue Anzahl der zur Verfügung stehenden Dateibezeichner ist betriebssystemabhängig.
 - ⌘ `fclose` ist eine Funktion des Systemkerns.
-

Beispiel 1. Die Datei „test“ wird im Schreibmodus geöffnet. Dies liefert den Dateibezeichner n:

```
>> n := fopen("test", Write)
```

17

Die Datei wird geschlossen:

```
>> fclose(n): delete n:
```

Änderungen:

⌘ Keine Änderungen.

finput – Einlesen von MuPAD-Objekten aus einer Datei

finput(filename, x) liest ein MuPAD-Objekt aus der Datei filename und weist es dem Bezeichner x zu.

finput(n, x) liest aus der mit dem Dateibezeichner n verknüpften Datei.

Aufruf(e):

```
⌘ finput(filename)
⌘ finput(filename, x1, x2, ...)
⌘ finput(n)
⌘ finput(n, x1, x2, ...)
```

Parameter:

filename	—	der Dateiname: eine Zeichenkette
n	—	ein von fopen erzeugter Dateibezeichner: eine positive ganze Zahl
x1, x2, ...	—	Bezeichner

Rückgabewert: das zuletzt aus der Datei eingelesene Objekt.

Verwandte Funktionen: fclose, fopen, fprintf, fread, ftextinput, input, loadproc, pathname, print, protocol, read, READPATH, textinput, write, WRITEPATH

Details:

- ☞ `finput` kann sowohl MuPAD-Binärdateien als auch ASCII-Textdateien lesen. Das Format der Datei wird dabei automatisch erkannt.

Binärdateien können mittels `fprint` oder `write` erzeugt werden. Textdateien können ebenfalls mit diesen Funktionen aus einer MuPAD-Sitzung heraus erzeugt werden (mit der `Text`-Option; siehe die entsprechenden Hilfeseiten für Details). Alternativ können Textdateien auch direkt mittels eines Editors erzeugt oder verändert werden. Die Dateien müssen syntaktisch korrekte MuPAD-Objekte oder -Anweisungen enthalten, die jeweils durch ein Semikolon oder einen Doppelpunkt abgeschlossen sein müssen. Ein Objekt kann sich dabei über mehrere Textzeilen erstrecken.
- ☞ `finput(filename)` liest das erste Objekt aus der Datei ein und liefert es als Rückgabewert an die MuPAD-Sitzung.
- ☞ `finput(filename, x1, x2, ...)` liest nacheinander die einzelnen Objekte aus der Datei und weist dabei das i -te Objekt dem Bezeichner x_i zu. Die Bezeichner werden von `finput` nicht evaluiert. Eventuell vorher zugewiesene Werte werden überschrieben. Die Objekte werden nicht evaluiert. Die Evaluierung kann mittels der Funktion `eval` erzwungen werden. Siehe Beispiel ??.
- ☞ Anstatt eines Dateinamens kann auch der Dateibezeichner `n` einer mittels `fopen` geöffneten Datei übergeben werden. Die Funktionalität ist in beiden Fällen dieselbe. Es gibt jedoch einen Unterschied: Bei Angabe eines Dateinamens wird die Datei nach dem Lesen der Daten automatisch wieder geschlossen. Ein erneutes Lesen beginnt wieder am Anfang der Datei. Bei Angabe eines Dateibezeichners bleibt die Datei nach dem Lesen geöffnet (sie muss explizit mittels `fclose` geschlossen werden), und ein erneutes Lesen setzt an der aktuellen Position fort. Sollen mehrere Daten aus einer Datei durch aufeinanderfolgende Aufrufe von `finput` gelesen werden, so ist dementsprechend ein Dateibezeichner anstelle eines Dateinamens zu benutzen. Siehe Beispiel ??.
- ☞ Ist die Anzahl der an `finput` übergebenen Bezeichner größer als die Anzahl der Objekte in der Datei, so werden den überschüssigen Bezeichnern keine Werte zugewiesen. Der Rückgabewert von `finput` ist in einem solchen Fall das leere Objekt vom Typ `DOM_NULL`.
- ☞ `finput` interpretiert den Dateinamen als Pfadnamen relativ zum „aktuellen Arbeitsverzeichnis“.

Man beachte, dass die Bedeutung des „aktuellen Arbeitsverzeichnisses“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.

Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann.

Auch absolute Pfadnamen werden von `finput` verarbeitet.

⌘ Ausdrucksfolgen werden von `finput` nicht ausgeglichen und können daher nicht benutzt werden, um mehrere Bezeichner an `finput` zu übergeben. Siehe Beispiel ??.

⌘ `finput` ist eine Funktion des Systemkerns.

Beispiel 1. Zuerst schreiben wir die Zahlen 11, 22, 33 und 44 in eine Datei:

```
>> fprintf("test", 11, 22, 33, 44):
```

Die soeben erstellte Datei wird mit `finput` eingelesen:

```
>> finput("test", x1, x2, x3, x4)
```

```
44
```

```
>> x1, x2, x3, x4
```

```
11, 22, 33, 44
```

Wird versucht, mehr Objekte aus der Datei zu lesen als vorhanden sind, so liefert `finput` das leere Objekt vom Typ `DOM_NULL`:

```
>> finput("test", x1, x2, x3, x4, x5); domtype(%)
```

```
DOM_NULL
```

```
>> x1, x2, x3, x4, x5
```

```
11, 22, 33, 44, x5
```

```
>> delete x1, x2, x3, x4:
```

Beispiel 2. Von `finput` eingelesene Objekte werden nicht evaluiert:

```
>> fprintf("test", x1): x1 := 23: finput("test")
```

```
x1
```

```
>> eval(%)
```

```
23
```

```
>> delete x1:
```


Beispiel 3. Hier werden mehrere Daten durch aufeinanderfolgende `finput` Aufrufe aus einer Datei gelesen. Hierbei muss man mit einem Dateibezeichner auf die Datei zugreifen. Die Datei wird mittels `fopen` im Lesemodus geöffnet:

```
>> fprintf("test", 11, 22, 33, 44): n := fopen("test"):
```

Der von `fopen` gelieferte Dateibezeichner kann an `finput` übergeben werden, um die Daten nacheinander einzulesen:

```
>> finput(n, x1, x2): x1, x2
                        11, 22

>> finput(n, x3, x4): x3, x4
                        33, 44
```

Die Datei wird geschlossen, und die benutzten Bezeichner werden gelöscht:

```
>> fclose(n): delete n, x1, x2, x3, x4:
```

Alternativ kann der Inhalt einer Datei in der folgenden Weise in eine MuPAD-Sitzung eingelesen werden:

```
>> n := fopen("test"):
    for i from 1 to 4 do
        x.i := finput(n)
    end_for:
    x1, x2, x3, x4
                        11, 22, 33, 44

>> fclose(n): delete n, i, x1, x2, x3, x4:
```

Beispiel 4. Ausdrucksfolgen werden von `finput` nicht ausgeglichen und können daher nicht benutzt werden, um Bezeichner an `finput` zu übergeben:

```
>> fprintf("test", 11, 22, 33): finput("test", (x1, x2), x3)

Error: Illegal argument [finput]
```

Der folgende Aufruf führt zu keinem Fehler, weil der Bezeichner `x12` nicht evaluiert wird. Dementsprechend wird nur ein einziges Objekt ausgelesen und `x12` zugewiesen:

```
>> x12 := x1, x2: finput("test", x12): x1, x2, x12
                        x1, x2, 11

>> delete x12:
```

Änderungen:

☞ Keine Änderungen.

float – Konvertierung in eine Gleitpunktzahl

`float(object)` konvertiert das Objekt oder numerische Teilausdrücke des Objekts in Gleitpunktzahlen.

Aufruf(e):

☞ `float(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: eine Gleitpunktzahl vom Typ `DOM_FLOAT` oder `DOM_COMPLEX` oder das Eingabeobjekt, in dem exakte Zahlen durch Gleitpunktzahlen ersetzt sind.

Überladbar durch: `object`

Seiteneffekte: Die Funktion reagiert auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `DIGITS`, `Pref::floatFormat`,
`Pref::trailingZeroes`

Details:

- ☞ `float` konvertiert Zahlen und numerische Ausdrücke wie `sqrt(sin(2))` oder `sqrt(3) + sin(PI/17)*I` zu reellen oder komplexen Gleitpunktzahlen. Enthält `object` außer den speziellen Konstanten `CATALAN`, `E`, `EULER` und `PI` noch weitere symbolische Objekte, so werden nur die *numerischen* Teilausdrücke zu Gleitpunktzahlen konvertiert. Insbesondere werden Bezeichner und indizierte Bezeichner nicht durch `float` verändert. Siehe Beispiel ??.
- ☞ Der `float` Aufruf wird rekursiv auf alle Operanden eines Ausdrucks angewendet. Zahlen oder Konstanten wie `PI` werden zu Gleitpunktzahlen konvertiert. Die Genauigkeit der erzeugten Gleitpunktzahlen wird durch die Umgebungsvariable `DIGITS` gesteuert. `DIGITS` legt die Anzahl der signifikanten Dezimalstellen der Gleitpunktzahl fest, die Voreinstellung für `DIGITS` ist 10. Die konvertierten Operanden werden gemäß der gegebenen Struktur des Ausdrucks miteinander verknüpft.

Ein Aufruf wie z. B. `float(PI - 314/100)` kann damit als Hintereinanderausführung der folgenden Operationen angesehen werden:

```
t1 := float(PI); t2 := float(314/100); result := t1 - t2
```

Die numerische Auswertung durch `float` kann daher zu Fehlerfortpflanzungen in numerischen Rechnungen führen. Siehe Beispiel ??.

- ☞ `float` wird automatisch an die Elemente von Mengen und Listen weitergeleitet. Für Arrays oder Tabellen geschieht dies nicht. Der schnellste Weg, alle Einträge von Arrays und Tabellen in Gleitpunktzahlen zu konvertieren, ist der Aufruf `map(object, float)`. Für ein Polynom `p` vom Typ `DOM_POLY` ist `mapcoeffs(p, float)` aufzurufen, um die Koeffizienten numerisch zu approximieren. Siehe Beispiel ??.
- ☞ Durch Aufruf der Präferenzen `Pref::floatFormat` und `Pref::trailingZeroes` kann die Bildschirmausgabe von Gleitpunktzahlen nach eigenen Wünschen eingestellt werden.
- ☞ Rationale Approximationen von Gleitpunktzahlen können mittels `numeric::rationalize` berechnet werden.
- ☞ Spezielle MuPAD-Funktionen wie `sin`, `exp`, `besselJ` usw. sind als Funktionsumgebungen implementiert. Durch Überladung des "float"-Attributs (Slot) einer Funktionsumgebung `f` wird die numerische Auswertung des symbolischen Aufrufs `f(x1, x2, ...)` innerhalb von Ausdrücken gesteuert.
Die Systemfunktion `float` kann durch den Benutzer auf eigene Funktionen erweitert werden. Hierzu muss die benutzerdefinierte Funktion `f` mittels `funcenv` als Funktionsumgebung definiert werden. Weiterhin ist ein "float"-Attribut `f::float` zu definieren. Die Systemfunktion `float` ruft dann `f::float(x1, x2, ...)` für jedes Auftreten von `f(x1, x2, ...)` innerhalb von Ausdrücken auf. Die Argumente, die an `f::float` übergeben werden, sind noch nicht zu Gleitpunktzahlen konvertiert, der Rückgabewert von `f::float` wird nicht weiter evaluiert. Es ist also einzig die Slot-Routine `f::float`, welche die Gleitpunkttauswertung von Funktionsaufrufen von `f` ausführt. Siehe Beispiel ??.
- ☞ Ein MuPAD-Domain `d` kann ebenso die `float`-Funktion überladen und damit die numerische Auswertung seiner Elemente selbst vornehmen. Hierzu muss der Slot `d::float` implementiert werden. Wird nun ein Domainelement `x` numerisch durch `float(x)` ausgewertet, so wird `d::float(x)` aufgerufen. Wie bei Funktionsumgebungen wird hierbei weder das an die Slot-Routine übergebene Argument vorher numerisch evaluiert, noch wird der von `d::float(x)` gelieferte Rückgabewert weiter numerisch verändert.

Besitzt ein Domain keinen "float"-Slot, so liefert der Aufruf `float(x)` das Element `x` unverändert zurück.

☞ Man beachte, dass MuPADs Gleitpunktzahlen in ihrer Größe beschränkt sind. Auf 32-Bit-Architekturen passiert ein numerischer Über-/Unterlauf, sobald Zahlen einer absoluten Größe von ungefähr $10.0^{\pm 2525222}$ auftreten. Auf 64-Bit-Architekturen liegen die Grenzen bei etwa $10.0^{\pm 42366205509363}$.

☞ Weitere Informationen sind auf der Hilfeseite von DIGITS zu finden.

☞ `float` ist eine Funktion des Systemkerns.

Beispiel 1. Einige Zahlen und numerische Ausdrücke werden in Gleitpunktzahlen verwandelt:

```
>> float(17), float(PI/7 + I/4), float(4^(1/3) + sin(7))
17.0, 0.4487989505 + 0.25 I, 2.244387651
```

`float` reagiert auf die Umgebungsvariable DIGITS:

```
>> DIGITS := 20:
float(17), float(PI/7 + I/4), float(4^(1/3) + sin(7))
17.0, 0.4487989505128276055 + 0.25 I, 2.2443876506869885652
```

Symbolische Objekte wie Bezeichner bleiben unverändert:

```
>> DIGITS := 10: float(2*x + sin(3))
2.0 x + 0.141120008
```

Beispiel 2. Die Fehlerfortpflanzung in numerischen Rechnungen wird demonstriert. Die folgende rationale Zahl approximiert $\exp(2)$ auf 17 Dezimalstellen:

```
>> r := 738905609893065023/1000000000000000000:
```

Der folgende `float`-Aufruf approximiert $\exp(2)$ und `r` durch Gleitpunktzahlen. Die Approximationsfehler werden durch Auslöschung in der Differenz extrem verstärkt:

```
>> DIGITS := 10: float(10^20*(r - exp(2)))
320.0
```

Keine Ziffer dieses Resultats ist korrekt! Ein besseres Ergebnis erhält man durch Erhöhung von DIGITS:

```
>> DIGITS := 20: float(10^20*(r - exp(2)))
276.95725394785404205
```

```
>> delete r, DIGITS:
```

Beispiel 3. `float` wird an die Elemente von Mengen und Listen weitergeleitet:

```
>> float([PI, 1/7, [1/4, 2], {sin(1), 7/2}])

[3.141592654, 0.1428571429, [0.25, 2.0], {0.8414709848, 3.5}]
```

Bei Tabellen und Arrays muss die Funktion `map` benutzt werden, um `float` auf die Einträge anzuwenden:

```
>> T := table("a" = 4/3, 3 = PI): float(T), map(T, float)

      table(      table(
        3 = PI,    ,    3 = 3.141592654,
        "a" = 4/3   "a" = 1.333333333
      )           )
```

```
>> A := array(1..2, [1/7, PI]): float(A), map(A, float)
```

```
+-      +-  +-      +-
| 1/7, PI |, | 0.1428571429, 3.141592654 |
+-      +-  +-      +-
```

Matrizendomains überladen die Funktion `float`. Im Gegensatz zum Verhalten bei Arrays wird `float` automatisch auf jedes Matricelement angewendet:

```
>> float(matrix(A))
```

```
+-      +-
| 0.1428571429 |
|              |
| 3.141592654  |
+-      +-
```

Um `float` auf die Koeffizienten eines durch `poly` erzeugten Polynoms anzuwenden, muss die Funktion `mapcoeffs` benutzt werden:

```
>> p := poly(9/4*x^2 + PI, [x]): float(p), mapcoeffs(p, float)
```

```
      2              2
poly(9/4 x  + PI, [x]), poly(2.25 x  + 3.141592654, [x])
```

```
>> delete A, T, p:
```

Beispiel 4. Hier wird die Überladung der `float`-Funktion am Beispiel einer Funktionsumgebung gezeigt. Die hier definierte Funktion `Sin` repräsentiert die Sinusfunktion. Im Gegensatz zu der bereits existierenden `sin`-Funktion misst `Sin` seine Argumente in Grad statt im Bogenmaß, d. h., es gilt $\text{Sin}(x) = \sin(\text{PI}/180 * x)$. Die einzige Funktionalität von `Sin` soll sein, ein numerisches Ergebnis zu liefern, falls das Argument eine reelle Gleitpunktzahl ist. In allen anderen Fällen wird ein unevaluierter Funktionsaufruf als Ergebnis geliefert:

```
>> Sin := proc(x)
      begin
        if domtype(x) = DOM_FLOAT then
          return(Sin::float(x));
        else return(procname(args()))
        end_if;
      end_proc;
```

Die gerade definierte Funktion wird zu einer Funktionsumgebung gemacht:

```
>> Sin := funcenv(Sin):
```

Zuletzt wird nun noch das "`float`"-Attribut implementiert. Es wird hierbei ein numerisches Ergebnis berechnet, sofern das Argument in eine reelle Gleitpunktzahl konvertiert werden kann. In allen anderen Fällen wird der unevaluierte Funktionsaufruf `Sin` als Ergebnis zurückgegeben:

```
>> Sin::float := proc(x)
      begin x := float(x):
        if domtype(x) = DOM_FLOAT then
          return(float(sin(PI/180*x)));
        else return(Sin(x))
        end_if;
      end_proc;
```

Die numerische Auswertung eines beliebigen Ausdrucks, der die `Sin`-Funktion enthält, ist nun möglich:

```
>> Sin(x), Sin(x + 0.3), Sin(120)

      Sin(x), Sin(x + 0.3), Sin(120)

>> Sin(120.0), float(Sin(120)), float(Sin(x + 120))

      0.8660254038, 0.8660254038, Sin(x + 120.0)

>> float(sqrt(2) + Sin(120 + sqrt(3)))

      2.264730594

>> delete Sin;
```

Änderungen:

☞ Keine Änderungen.

fopen – Öffnen einer Datei

`fopen(filename)` öffnet die Datei namens `filename`.

Aufruf(e):

☞ `fopen(filename)`

☞ `fopen(<format,> filename, writemode)`

Parameter:

`filename` — der Dateiname: eine Zeichenkette

Optionen:

`writemode` — entweder *Write* oder *Append*. Mit beiden Optionen wird die Datei zum Schreiben geöffnet. Existiert keine Datei mit dem angegebenen Namen, so wird eine neue Datei erzeugt. Mit *Write* werden existierende Dateien überschrieben. Mit *Append* können mittels `fprint` oder `write` neue Daten an schon vorhandene Daten einer bereits existierenden Datei angehängt werden. Das angegebene Format muss hierbei mit dem Format der existierenden Datei übereinstimmen, anderenfalls kann die Datei nicht geöffnet werden, und `fopen` liefert FAIL.

`format` — das Schreibformat: entweder *Bin* oder *Text*. Mit *Bin* werden die Daten in MuPADs Binärformat gespeichert, mit *Text* im üblichen ASCII-Format. Die Voreinstellung ist *Bin*.

Rückgabewert: eine positive ganze Zahl: der Dateibezeichner. Kann die Datei nicht geöffnet werden, so wird FAIL zurückgeliefert.

Seiteneffekte: Die Funktion reagiert auf die Umgebungsvariable `WRITEPATH`. Hat diese Variable einen Wert, so wird im Schreibmodus (Optionen *Write* oder *Append*) die Datei in dem entsprechenden Verzeichnis angelegt, anderenfalls im „aktuellen Arbeitsverzeichnis“.

Verwandte Funktionen: `fclose`, `finput`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

☞ `fopen(filename)` öffnet eine existierende Datei zum Lesen. `fopen` erkennt hierbei automatisch, ob es sich um eine Text- oder Binärdatei handelt. Ein Fehler wird ausgelöst, wenn keine entsprechende Datei gefunden wird.

☞ `fopen(<format,> filename, writemode)` öffnet die Datei zum Schreiben im angegebenen Format. Existiert noch keine Datei mit dem angegebenen Namen, so wird eine neue Datei erzeugt.

☞ Im Schreibmodus (mit einer der Optionen *Write* oder *Append*), wird die Umgebungsvariable `WRITEPATH` betrachtet. Hat sie einen Wert, so wird eine neue Datei im entsprechenden Verzeichnis erzeugt (bzw. eine existierende Datei wird dort gesucht), ansonsten geschieht dies im „aktuellen Arbeitsverzeichnis“.

Man beachte, dass das „aktuelle Arbeitsverzeichnis“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.

Man beachte, dass `fopen` im Lesemodus nicht in den durch `READPATH` und `LIBPATH` gegebenen Verzeichnissen sucht.

Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann. Weiterhin warnt MacMuPAD den Benutzer vor dem Überschreiben existierender Dateien.

Auch absolute Pfadnamen werden von `fopen` verarbeitet.

☞ Der von `fopen` gelieferte Dateibezeichner kann von Funktionen wie `fclose`, `fread`, `fprint`, `read`, `write` etc. benutzt werden.

☞ Eine mit `fopen` geöffnete Datei sollte nach Gebrauch mit `fclose` wieder geschlossen werden.

☞ `fopen` ist eine Funktion des Systemkerns.

Beispiel 1. Zuerst wird die Datei „test“ zum Schreiben geöffnet. Mit der Option *Write* braucht diese Datei noch nicht zu existieren. Standardmäßig wird die Datei als Binärdatei geöffnet:

```
>> n := fopen("test", Write)
```

17

Eine Zeichenkette wird in die Datei geschrieben, und diese wird wieder geschlossen:


```
>> fprintf(n, "a string"): fclose(n):
```

Eine weitere Zeichenkette wird in die Datei geschrieben:

```
>> n := fopen("test", Append)
```

18

```
>> fprintf(n, "another string"): fclose(n):
```

Die Binärdatei kann nicht als Textdatei zum Anhängen von Daten geöffnet werden:

```
>> n := fopen(Text, "test", Append)
```

FAIL

Sie kann jedoch mit der Option *Write* als Textdatei geöffnet werden. Die existierende Binärdatei wird als Textdatei überschrieben:

```
>> n := fopen(Text, "test", Write)
```

19

```
>> fclose(n): delete n:
```

Beispiel 2. `fopen` kann nicht existierende Dateien nicht zum Lesen öffnen. Es wird hier angenommen, dass die Datei „xyz“ nicht existiert:

```
>> n := fopen("xyz")
```

FAIL

Es wird angenommen, dass die in Beispiel ?? erzeugte Datei „test“ existiert. Sie kann erfolgreich zum Lesen geöffnet werden:

```
>> n := fopen("test")
```

20

```
>> fclose(n): delete n:
```

Änderungen:

☞ Keine Änderungen.

for – for-Schleife

`for` – `end_for` stellt ein Schleifenkonstrukt zur Verfügung, mit dem automatisch über einem Bereich von Zahlen oder Objekten iteriert werden kann.

Aufruf(e):

```
# for i from start to stop <step stepwidth> do
    body
end_for
```

```
# _for(i, start, stop, stepwidth, body)
```

```
# for i from start downto stop <step stepwidth> do
    body
end_for
```

```
# _for_down(i, start, stop, stepwidth, body)
```

```
# for x in object do
    body
end_for
```

```
# _for_in(x, object, body)
```

Parameter:

i, x	— die Schleifenvariable: ein Bezeichner oder eine lokale Variable (DOM_VAR) einer Prozedur
start	— der Startwert für i: eine reelle Zahl. Dies kann eine ganze Zahl, eine rationale Zahl oder eine Gleitpunktzahl sein.
stop	— der Endwert für i: eine reelle Zahl. Dies kann eine ganze Zahl, eine rationale Zahl oder eine Gleitpunktzahl sein.
stepwidth	— die Schrittweite: eine positive reelle Zahl. Dies kann eine ganze Zahl, eine rationale Zahl oder eine Gleitpunktzahl sein. Der Standardwert ist 1.
object	— ein beliebiges MuPAD-Objekt
body	— der Schleifenrumpf: eine beliebige Folge von Anweisungen

Rückgabewert: der Wert des letzten innerhalb der Schleife ausgeführten Kommandos. Wurde kein Kommando innerhalb der Schleife ausgeführt, wird `NIL` zurückgegeben. Ist der Iterationsbereich der Schleife leer, wird das leere Objekt vom Typ `DOM_NULL` zurückgegeben.

Weitere Dokumentation: Kapitel 16 des MuPAD-Tutoriums.

Verwandte Funktionen: `break`, `next`, `repeat`, `while`

Details:

```
# Beim Eintritt in eine aufwärts zählende Schleife
```

```
for i from start to stop step stepwidth do body end_for
```

wird die Zuweisung `i := start` durchgeführt. Mit diesem Wert von `i` wird der Schleifenrumpf `body` durchgeführt (hierbei darf innerhalb des Rumpfs der Wert von `i` verändert werden). Nachdem alle Anweisungen innerhalb des Rumpfs ausgeführt wurden, kehrt die Schleife an den Beginn des Rumpfs zurück. Dort wird `i := i + stepwidth` gesetzt und das Abbruchkriterium `i > stop` überprüft. Liefert dies `FALSE`, so wird der Rumpf erneut mit dem neuen Wert von `i` durchlaufen. Gilt `TRUE`, so wird die Schleife sofort beendet ohne den Rumpf erneut zu durchlaufen.

⌘ Die abwärts zählende Schleife

```
for i from start downto stop step stepwidth do body end_for
```

implementiert ein entsprechendes Verhalten. Der einzige Unterschied besteht darin, dass beim Rücksprung zum Beginn des Schleifenrumpfs die Schleifenvariable durch `i := i - stepwidth` heruntergezählt wird, bevor das Abbruchkriterium `i < stop` überprüft wird.

⌘ Die Schleife `for x in object do body end_for` lässt `x` über alle Operanden des Objekts `object` laufen. Diese Schleife ist äquivalent zu

```
for i from 1 to nops(object) do
  x := op(object, i);
  body
end_for
```

Typischerweise wird `object` eine Liste, eine Ausdrucksfolge oder ein Array sein. Man beachte, dass andere „Container-Objekte“ wie z. B. endliche Mengen oder Tabellen keine natürliche interne Ordnung haben. Solche Objekte sollten in einer Iterationsschleife nur eingesetzt werden, wenn die Reihenfolge der Iterationsschritte irrelevant ist.

⌘ Der Schleifenrumpf kann aus beliebig vielen Anweisungen bestehen, die durch einen Doppelpunkt `:` oder ein Semikolon `;` zu trennen sind. Höchstens das letzte evaluierte Resultat innerhalb des Rumpfs wird als Rückgabewert der Schleife auf dem Bildschirm ausgegeben. Zwischenergebnisse können durch `print`-Anweisungen innerhalb der Schleife ausgegeben werden.

⌘ Die Schleifenvariable `i` bzw. `x` kann vor Beginn der Schleife einen Wert haben, der innerhalb der Schleife überschrieben wird. Nach Beendigung der Schleife hat sie den zuletzt in der Schleife zugewiesenen Wert. Typischerweise ist dies in auf- bzw. abwärts zählenden Schleifen mit ganzzahligen Werten von `start`, `stop` und `stepwidth` der Wert `i = stop ± stepwidth`.

⌘ Die Argumente `start`, `stop`, `stepwidth` und `object` werden nur einmal beim Schleifenbeginn und nicht nach jedem Schritt evaluiert. d. h.,

falls beispielsweise `object` während eines Schritts verändert wird, läuft `x` weiterhin durch die Operanden des ursprünglichen Objekts.

- ⇒ Schleifen können mit einer `break`-Anweisung vorzeitig beendet werden. Schleifenschritte können mittels der `next`-Anweisung übersprungen werden. Siehe Beispiel ??.
 - ⇒ Statt des Schlüsselworts `end_for` kann auch das Schlüsselwort `end` benutzt werden. Siehe Beispiel ??.
 - ⇒ Alternativ zur imperativen Form der Schleifen können auch Aufrufe der äquivalenten Funktionen `_for`, `_for_down` bzw. `_for_in` verwendet werden. Siehe Beispiel ??.
 - ⇒ `_for`, `_for_down` und `_for_in` sind Funktionen des Systemkerns.
-

Beispiel 1. Der Rumpf der folgenden Schleife besteht aus mehreren Anweisungen. Der ursprüngliche Wert der Schleifenvariable `i` wird beim Eintritt in die Schleife überschrieben:

```
>> i := 20:
    for i from 1 to 3 do
        a := i;
        b := i^2;
        print(a, b)
    end_for:
```

1, 1

2, 4

3, 9

Die Schleifenvariable hat nun den Wert, der das Abbruchkriterium `i > 3` erfüllt:

```
>> i
```

4

Der Iterationsbereich ist nicht auf ganze Zahlen eingeschränkt:

```
>> for i from 2.2 downto 1 step 0.5 do
    print(i)
end_for:
```

2.2

1.7

1.2

Die folgende Schleife summiert alle Elemente einer Liste. Der Rückgabewert der Schleife ist die Summe, die unmittelbar einer Variablen zugewiesen werden kann:

```
>> s := 0: S := for x in [c, 1, d, 2] do s := s + x end_for  
c + d + 3
```

Man beachte, dass für Mengen die interne Anordnung der Elemente nicht mit der auf dem Bildschirm ausgegebenen Anordnung übereinzustimmen braucht:

```
>> S := {c, d, 1}  
{c, d, 1}  
  
>> for x in S do print(x) end_for:  
1  
d  
c  
  
>> delete a, b, i, s, S, x:
```

Beispiel 2. Schleifen können mit einer `break`-Anweisung vorzeitig beendet werden:

```
>> for i from 1 to 3 do  
    print(i);  
    if i = 2 then break end_if  
end_for:  
1  
2
```

Mit der `next`-Anweisung kann die Ausführung von Kommandos innerhalb eines Schleifenschritts übersprungen werden. Die Evaluierung wird am Beginn des Schleifenrumpfs mit dem nächsten Wert des Schleifenparameters fortgesetzt:

```
>> a := 0:  
for i from 1 to 3 do  
    a := a + 1;  
    if i = 2 then next end_if;  
    print(i, a)  
end_for:
```

1, 1

3, 3

```
>> delete i, a:
```

Beispiel 3. Schleifen können mit dem Schlüsselwort `end` abgeschlossen werden. Der Parser erkennt automatisch den Gültigkeitsbereich von `end`-Anweisungen:

```
>> s:= 0:
    for i from 1 to 3 do
        for j from 1 to 3 do
            s := i + j;
            if i + j > 4 then
                break;
            end
        end
    end
end
```

5

```
>> delete s, i, j:
```

Beispiel 4. Das Beispiel demonstriert den Bezug zwischen der funktionalen und der imperativen Form von `for`-Schleifen:

```
>> hold(
    _for(i, start, stop, stepwidth, (statement1; statement2))
)

    for i from start to stop step stepwidth do
        statement1;
        statement2
    end_for
```

Die optionale `step`-Angabe wird weggelassen, wenn für die Schrittweite der Wert `NIL` angegeben wird:

```
>> hold(
    _for_down(i, 10, 1, NIL, (x := i^2; x := x - 1))
)

    for i from 10 downto 1 do
        x := i^2;
        x := x - 1
    end_for
```

```
>> hold(
    _for_in(x, object, body)
)

for x in object do
    body
end_for
```

Änderungen:

⌘ end kann nun als Alternative zu end_for benutzt werden.

fprint – Schreiben in eine Datei

fprint(filename, objekte) schreibt MuPAD-Objekte in die Datei filename.

fprint(n, objekte) schreibt in die mit dem Dateibezeichner n verknüpfte Datei.

Aufruf(e):

```
⌘ fprint(<style,> <format,> filename, object1, ob-
        ject2, ...)
⌘ fprint(<style,> n, object1, object2, ...)
```

Parameter:

filename	— der Dateiname: eine Zeichenkette
object1, object2, ...	— beliebige MuPAD-Objekte
n	— ein von fopen erzeugter Dateibezeichner: eine nichtnegative ganze Zahl

Optionen:

- `style` — entweder *Unquoted* oder *NoNL*. Diese Optionen sind nur für Textdateien relevant. Beide Optionen veranlassen `fprint`, Zeichenketten ohne Anführungsstriche zu speichern. Die gespeicherten Objekte werden ohne trennende Doppelpunkte in die Datei geschrieben. Mit *Unquoted* wird die durch `fprint` erzeugte Zeile durch einen Zeilenvorschub beendet, mit *NoNL* wird kein Zeilenvorschub angehängt.
- `format` — das Schreibformat: entweder *Bin* oder *Text*. Mit *Bin* werden die Daten in MuPADs Binärformat gespeichert, mit *Text* im üblichen ASCII-Format. Die Voreinstellung ist *Bin*.

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Seiteneffekte: Die Funktion reagiert auf die Umgebungsvariable `WRITEPATH`. Hat diese Variable einen Wert, so wird die Datei in dem entsprechenden Verzeichnis angelegt, anderenfalls im „aktuellen Arbeitsverzeichnis“.

Verwandte Funktionen: `expr2text`, `fclose`, `finput`, `fopen`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

- ☞ Mit `fprint` können MuPAD-Objekte in eine Datei geschrieben werden. Die Objekte werden evaluiert, die Resultate werden gespeichert. Diese Daten können mittels der Funktionen `finput` bzw. `ftextinput` in eine andere MuPAD-Sitzung eingelesen werden.
- ☞ Die Datei kann direkt durch ihren Namen spezifiziert werden. Hierbei wird entweder eine neue Datei geöffnet oder eine existierende Datei dieses Namens wird überschrieben. Dabei öffnet und schließt `fprint` die Datei automatisch.

Wenn `WRITEPATH` keinen Wert hat, interpretiert `fprint` den Dateinamen als Pfadnamen relativ zum „aktuellen Arbeitsverzeichnis“.

Man beachte, dass die Bedeutung des „aktuellen Arbeitsverzeichnisses“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.

Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann. Weiterhin warnt MacMuPAD den Benutzer vor dem Überschreiben existierender Dateien.

Auch absolute Pfadnamen werden von `fprint` verarbeitet.

- ☞ Anstatt eines Dateinamens kann auch der Dateibezeichner einer durch `fopen` geöffneten Datei übergeben werden. Siehe Beispiel ?? . Hierbei werden alle durch `fprint` geschriebenen Daten an das Ende der entsprechenden Datei angehängt. Die Datei wird nicht automatisch von `fprint` geschlossen und muss durch einen folgenden Aufruf von `fclose` ‚per Hand‘ geschlossen werden.

Man beachte, dass `fopen(filename)` die Datei nur im Lesemodus öffnet, d. h., ohne Schreibberechtigung. Ein folgender `fprint`-Befehl auf diese Datei liefert einen Fehler. Man verwende die *Write*- bzw. *Append*-Option von `fopen` um die Datei im Schreibmodus zu öffnen.

Der Dateibezeichner 0 stellt den Bildschirm dar.

- ☞ Die Ausgabe von Text erfolgt ohne Pretty-Printer. Ein Aufruf von `fprint` schreibt alle übergebenen Objekte in eine einzige Zeile der Textdatei. Diese Zeile wird mit einem Zeilenvorschub beendet, falls dies nicht mittels *NoNL* unterdrückt wird. Standardmäßig werden die geschriebenen Objekte durch Doppelpunkte (ohne zusätzliche Leerzeichen) getrennt. Die so entstehenden Textdaten stellen syntaktisch korrekten MuPAD-Code dar und können mittels `finput` wiedereingelesen werden. Mit den Optionen *Unquoted* und *NoNL* werden die Objekte in der Textdatei ohne trennende Doppelpunkte oder Leerzeichen hintereinandergeschrieben. Die so entstehenden Textdaten stellen in der Regel keinen syntaktisch korrekten MuPAD-Code mehr dar und können nicht mittels `finput` wiedereingelesen werden. Siehe Beispiel ?? .

Man beachte, dass die Textversion eines MuPAD-Objekts nicht notwendigerweise die interne Datenstruktur des Objekts anzeigt. Ein im Text-Modus abgespeichertes Domain-Element kann von `finput` als entsprechendes Objekt einer anderen Datenstruktur wiedereingelesen werden. Man benutze den Binär-Modus, wenn gespeicherte Daten originalgetreu wiedereingelesen werden sollen. Siehe Beispiel ?? .



- ☞ MuPAD-Anweisungen wie z. B. Zuweisungen können als Objekte gespeichert werden. Sie müssen dazu aber in zusätzlichen Klammern eingeschlossen werden wie z. B. `fprint("test", (a := 2))`.

- ☞ `fprint` ist eine Funktion des Systemkerns.

Option *<Unquoted>*:

- ☞ Diese Option ist zur Erstellung formatierter Textdateien sinnvoll. Im Allgemeinen können mit dieser Option gespeicherte Daten nicht wieder mittels `finput` eingelesen werden.
- ☞ Mit dieser Option werden Zeichenketten ohne Anführungszeichen geschrieben. Zusätzlich werden die Steuerzeichen `'\n'` und `'\t'` expandiert. Weiterhin werden Objekte nicht durch Doppelpunkte getrennt.

Die von `fprint` geschriebene Zeile wird mit einem Zeilenvorschub abgeschlossen.

Option `<NoNL>`:

- ☞ Diese Option hat die gleiche Funktionalität wie *Unquoted*, nur dass die von `fprint` geschriebene Zeile nicht mit einem Zeilenvorschub beendet wird.
-

Beispiel 1. Zuerst werden einige Daten in die Datei „test“ geschrieben, die standardmäßig als Binärdatei erzeugt wird. Die Anweisung `d := 5` muss aus syntaktischen Gründen extra geklammert werden:

```
>> fprint("test", (d := 5), d*3):
```

Die Datei wird wieder eingelesen, wobei die gespeicherte Zuweisung `d := 5` ausgeführt wird. Der von dieser Zuweisung gelieferte Wert wird dabei gleichzeitig dem Bezeichner `e` zugewiesen. Der Wert `d*3` wird dem Bezeichner `f` zugewiesen:

```
>> finput("test", e, f): d, e, f;
                        5, 5, 15
```

```
>> delete d, e, f:
```

Beispiel 2. Hier wird über einen Dateibezeichner auf die Datei „test“ zugegriffen. Die Datei bleibt hierbei solange geöffnet, bis sie mit `fclose` geschlossen wird. Es können daher mit mehreren aufeinanderfolgenden `fprint`-Aufrufen Daten hintereinandergehängt werden:

```
>> n := fopen("test", Write):
    fprint(n, (d := 5), d*3):
    fprint(n, "more data"):
```

Die Datei wird geschlossen:

```
>> fclose(n):
```

Die gespeicherten Werte werden wieder eingelesen und den Bezeichnern `e`, `f` und `g` zugewiesen:

```
>> finput("test", e, f, g ): e, f, g;
                        5, 15, "more data"
>> delete n, d, e, f, g:
```

Beispiel 3. Mit der Option *Unquoted* werden Zeichenketten ohne Anführungszeichen geschrieben:

```
>> fprintf(Text, "test1", "Hello World!", MuPAD + 1):
      fprintf(Unquoted, Text, "test2", "Hello World!", MuPAD + 1):
```

Die Dateien „test1“ und „test2“ besitzen nun folgenden Inhalt:

```
test1:
"Hello World!":MuPAD + 1:
```

```
test2:
Hello World!MuPAD + 1
```

Mit `finput` oder `ftextinput` können diese Daten dann wieder aus der Datei gelesen werden:

```
>> finput("test1", a, b): a, b;
      "Hello World!", MuPAD + 1
>> ftextinput("test2", c): c
      "Hello World!MuPAD + 1"
>> delete a, b, c:
```

Beispiel 4. Die Textversion eines MuPAD-Objekts zeigt nicht notwendigerweise die interne Datenstruktur des Objekts an. Beispielsweise liefert die Funktion `matrix` eine Matrix vom Domain-Typ `Dom::Matrix()`. Die Textversion ist jedoch ein Array:

```
>> fprintf(Text, "test", matrix([1, 2])):
      finput("test")
      array(1..2, 1..1, (1, 1) = 1, (2, 1) = 2)
```

Man benutze den Binär-Modus, um sicherzustellen, dass Objekte originalgetreu wiedereingelesen werden können:

```
>> fprintf("test", matrix([1, 2])):
      finput("test"); domtype(%)
```

```
+ -   - +
|  1  |
|     |
|  2  |
+ -   - +
```

```
Dom::Matrix()
```

Änderungen:

☞ Keine Änderungen.

`frac` – der „nicht-ganzzahlige Anteil“ einer Zahl

`frac(x)` stellt den „nicht-ganzzahligen Anteil“ $x - \text{floor}(x)$ der Zahl x dar.

Aufruf(e):

☞ `frac(x)`

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Die Funktion reagiert auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `floor`

Details:

- ☞ Für komplexe Argumente wird `frac` getrennt auf den Real- und den Imaginärteil angewendet.
- ☞ Für reelle Argumente ist der von `frac(x)` dargestellte Wert $x - \text{floor}(x)$ eine Zahl aus dem Intervall $[0, 1)$. Für positive Argumente hat `frac` folgende intuitive Interpretation: diese Funktion schneidet alle Stellen vor dem Dezimalpunkt ab.
- ☞ Für ganzzahlige Argumente wird 0 zurückgeliefert. Für rationale Argumente werden rationale Zahlen zurückgeliefert. Für Argumente, die symbolische Bezeichner enthalten, werden symbolische Funktionsaufrufe zurückgeliefert. Für Gleitpunktzahlen sowie nicht-rationale exakte numerische Ausdrücke werden Gleitpunktzahlen zurückgeliefert.
- ☞ Für Gleitpunktzahlen mit einem Betrag größer als 10^{DIGITS} wird das Resultat von den internen nicht-signifikanten Stellen mitbestimmt! Siehe Beispiel ??.



☞ Intern werden exakte numerische Ausdrücke durch Gleitpunkt-
werte approximiert. Dementsprechend hängt das Ergebnis vom
momentanen Wert von DIGITS ab! Siehe Beispiel ??.



Beispiel 1. Die „nicht-ganzzahligen Anteile“ einiger reeller und komplexer Zahlen:

```
>> frac(1234), frac(123/4), frac(1.234)
      0, 3/4, 0.234
>> frac(-1234), frac(-123/4), frac(-1.234)
      0, 1/4, 0.766
>> frac(3/2 + 7/4*I), frac(4/3 + 1.234*I)
      1/2 + 3/4 I, 0.3333333333 + 0.234 I
```

Für symbolische numerische Ausdrücke werden Gleitpunktwerte zurückge-
liefert:

```
>> frac(exp(123)), frac(3/4*sin(1) + I*tan(3))
      0.7502040792, 0.6311032386 + 0.8574534569 I
```

Ausdrücke mit symbolischen Bezeichnern führen zu symbolischer Rückgabe:

```
>> frac(x), frac(sin(1) + x^2), frac(exp(-x))
      2
      frac(x), frac(sin(1) + x ), frac(exp(-x))
```

Beispiel 2. Die Verwendung von frac kann bei betragsmäßig großen Gleit-
kommazahlen problematisch sein:

```
>> 10^13/3.0
      3.333333333e12
```

Man beachte, dass nur die ersten 10 Dezimalstellen „signifikant“ sind. Die
weiteren Stellen werden durch die interne Binärdarstellung erzeugt. Diese
„unsignifikanten“ Stellen fließen in das Resultat ein:

```
>> frac(10^13/3.0)
      0.3333332539
```

Die Mantisse der folgenden Gleitpunktzahl ist nicht lang genug, um Stellen
nach dem Dezimalpunkt zu speichern:

```
>> floor(10^25/9.0), ceil(10^25/9.0), frac(10^25/9.0)
      11111111111111111111081984, 111111111111111111111081984, 0.0
```


Optionen:

- Plain* — mit dieser Option benutzt `fread` einen eigenen Parser-Kontext
- Quiet* — unterdrückt Ausgaben während der Ausführung von `fread`

Rückgabewert: der Rückgabewert der letzten Anweisung der Datei.

Verwandte Funktionen: `fclose`, `fininput`, `fopen`, `fprint`, `ftextinput`, `input`, `loadproc`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `textinput`, `write`, `WRITEPATH`

Details:

- ☞ `fread(filename)` liest den Inhalt der Datei `filename` und führt die darin enthaltenen MuPAD-Anweisungen aus.
 - ☞ `fread` arbeitet wie `read`, nur sucht es die gegebene Datei nicht in den durch `READPATH` und `LIBPATH` gegebenen Verzeichnissen, sondern nur relativ zum „aktuellen Arbeitsverzeichnis“.
Man beachte, dass die Bedeutung des „aktuellen Arbeitsverzeichnisses“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.
Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann. Weiterhin warnt MacMuPAD den Benutzer vor dem Überschreiben existierender Dateien.
Auch absolute Pfadnamen werden von `fread` verarbeitet.
 - ☞ Mit der Funktion `fread` können sowohl (mittels `fprint` oder `write` erzeugte) Binärdateien als auch ASCII-Textdateien eingelesen werden. `fread` erkennt das Format der Datei dabei automatisch.
 - ☞ Anstatt eines Dateinamens kann auch der Dateibezeichner einer durch `fopen` geöffneten Datei übergeben werden. Siehe Beispiel ??.
 - ☞ `fread` ist eine Funktion des Systemkerns.
-

Option <Quiet>:

- ☞ Mit dieser Option werden mögliche Ausgaben während des Lesens und Ausführens einer Datei unterdrückt. Warnungen, Fehlermeldungen und durch `print` erzeugte Ausgaben sind jedoch weiterhin sichtbar.

Option *<Plain>*:

- ☞ Mit dieser Option wird die Datei in einem eigenen Parser-Kontext eingelesen. d. h., die History wird nicht verändert, und vom Benutzer außerhalb der Datei definierte Aliase und Operatoren werden hierbei ignoriert. Die Option kann z. B. eingesetzt werden, um Initialisierungen in einer wohldefinierten Umgebung ohne Seiteneffekte durchzuführen.

Beispiel 1. Das folgende Beispiel funktioniert so nur unter UNIX und Linux; auf einem anderen Betriebssystem müssen die Dateinamen entsprechend geändert werden. Zunächst wird mittels `fprint` eine Datei mit drei MuPAD-Befehlen erzeugt:

```
>> fprint(Unquoted, Text, "/tmp/test", "a := 3; b := 5; a + b;");
```

Nun wird die gerade erstellte Datei eingelesen. Die Anweisungen werden ausgeführt und erzeugen jeweils eine Ausgabe. Die zweite 8 ist der Rückgabewert der Funktion `fread`:

```
>> delete a, b: fread("/tmp/test")
```

3

5

8

8

Nun besitzen die eingelesenen Variablen die in der Datei zugewiesenen Werte:

```
>> a, b
```

3, 5

Mit der Option *Quiet* wird lediglich der Rückgabewert der Funktion `fread` ausgegeben:

```
>> delete a, b: fread("/tmp/test", Quiet)
```

8

```
>> delete a, b:
```


Beispiel 2. Für das nächste Beispiel, welches näher auf die Option *Plain* eingeht, wird zunächst eine Eingabedatei erstellt:

```
>> fprintf(Unquoted, Text, "/tmp/test",
           "f := proc(x) begin x^2 end_proc:",
           "a := f(3): b := f(4):"):
```

Ein Alias für *f* wird definiert:

```
>> alias(f = "ein Text"):
```

Wird die Datei nun ohne die Option *Plain* eingelesen, so erhält man eine Fehlermeldung. Im Kontext der MuPAD-Sitzung wird in der Zuweisung *f := ...* die linke Seite durch die Zeichenkette des Alias ersetzt. Allerdings ist eine Zuweisung von Werten an Zeichenketten nicht möglich:

```
>> fread("/tmp/test"):

Error: Invalid left-hand side [_assign];
while reading file '/tmp/test'
```

Mit der Option *Plain* ignoriert *fread* den Alias für *f*:

```
>> fread("/tmp/test", Plain): a, b

9, 16

>> unalias(f): delete f, a, b:
```

Beispiel 3. Der Wert der Variable *a* wird mittels *write* in der Datei „/tmp/test“ gespeichert:

```
>> a := PI + 1: write("/tmp/test", a): delete a:
```

Diese Datei wird mittels *fopen* im Lesemodus geöffnet:

```
>> n := fopen("/tmp/test")

17
```

Der von *fopen* gelieferte Dateibezeichner kann an *fread* übergeben werden. Durch Einlesen der Datei wird der Wert von *a* wiederhergestellt:

```
>> fread(n): a

PI + 1

>> fclose(n): delete a:
```

Änderungen:

- ☞ Die neue Option *Plain* wurde eingeführt.
-

`freeze`, `unfreeze` – Erzeugung einer inaktiven bzw. aktiven Kopie einer Funktion

`freeze(f)` liefert eine inaktive Form der Funktion `f`.

`unfreeze(object)` reaktiviert alle in `object` vorkommenden inaktive Funktionen und evaluiert das Ergebnis.

Aufruf(e):

- ☞ `freeze(f)`
- ☞ `unfreeze(object)`

Parameter:

- `f` — eine Prozedur oder eine Funktionsumgebung
- `object` — ein beliebiges MuPAD-Objekt

Rückgabewert: `freeze` liefert ein Objekt vom selben Typ wie `f` zurück. `unfreeze` re-aktiviert alle in `object` enthaltenen inaktiven Funktionen und liefert das Ergebnis evaluiert zurück.

Verwandte Funktionen: `eval`, `hold`, `MAXDEPTH`

Details:

- ☞ `ff := freeze(f)` liefert eine Funktion, die eine so genannte „inaktive“ Kopie der Funktion `f` darstellt. Das bedeutet, dass

1. `ff` ausschließlich seine Argumente evaluiert, aber keine Berechnungen durchführt,
2. `ff` genauso wie `f` auf dem Bildschirm dargestellt wird,
3. symbolische `ff`-Aufrufe denselben Typ wie symbolische `f`-Aufrufe haben,
4. `ff` alle Funktionsattribute von `f` besitzt, wenn `f` eine Funktionsumgebung ist.

Man beachte, dass `ff` seine Eingabeparameter auch dann evaluiert, falls die Funktion `f` die Prozeduroption `hold` gesetzt hat.

- ☞ `freeze` ist dazu gedacht, um mehrere Operationen mit `f` ausschließlich in seiner symbolischen Form durchzuführen, ohne `f` dabei selbst auszuführen. Siehe Beispiel ??.

☞ Weder `eval` noch `level` bewirken die Evaluierung einer inaktiven Funktion; siehe Beispiel ??.

☞ `unfreeze(object)` re-aktiviert alle in `object` enthaltenen inaktiven Funktionen und evaluiert das Ergebnis. `unfreeze` arbeitet dabei rekursiv auf der Struktur von `object`.

☞ `unfreeze` verwendet `misc::maprec` zum rekursiven Arbeiten auf der Struktur von `object`. Das bedeutet, dass die Funktion `unfreeze` für Basisdomains wie beispielsweise Felder, Tabellen, Listen oder Polynome auf alle Operanden dieser Objekte angewendet wird.

Falls `object` ein Element eines Bibliotheks-Domains ist, so wird das Verhalten von `unfreeze` von der Methode "`maprec`" bestimmt, die die Funktion `misc::maprec` überlädt. Ist diese Methode nicht implementiert, so hat die Eingabe `unfreeze(object)` keine Wirkung auf das Objekt `object`. Siehe Beispiel ??.

☞ `unfreeze` wirkt nicht auf Prozeduren. Es wird daher nicht empfohlen, Ausdrücke, die inaktive Funktionen enthalten, in Prozeduren zu schachteln.

Beispiel 1. Nach der folgenden Anweisung ist `_int` die inaktive Form der Funktionsumgebung `int`:

```
>> _int := freeze(int): F := _int(x*exp(x^2), x = 0..1)
                                     2
                                     int(x exp(x ), x = 0..1)
```

Die inaktive Form von `int` bewahrt jede Information, die über die Funktion `int` bekannt ist, wie z.B. die Ausgabe, den Type und das "`float`"-Attribut für numerische Auswertung:

```
>> F, type(F), float(F)
                                     2
                                     int(x exp(x ), x = 0..1), "int", 0.8591409142
>> int(x*exp(x^2), x = 0..1)
                                     exp(1)
                                     ----- - 1/2
                                     2
```

Zum Re-aktivieren der inaktiven Funktion `_int` steht die Funktion `unfreeze` zur Verfügung, die das Ergebnis auch auswertet:

```
>> unfreeze(F), unfreeze(F + 1/2)
                                     exp(1)          exp(1)
                                     ----- - 1/2,  -----
                                     2                2
```

Beispiel 2. Die Funktion `student::riemann` macht Gebrauch von der Funktion `freeze`, um ihr Ergebnis in einer Form zurück zu liefern, in der die Funktion `sum` in symbolischer Form enthalten ist:

```
>> a:= student::riemann(sin(x), x = 0..PI)
```

$$\frac{\text{PI} \sum_{i=1}^{\text{PI}} \sin\left(\frac{i-1/2}{4}\right)}{4}, \quad i=0..3$$

Erst durch die Anwendung der Funktion `unfreeze` wird die Summe berechnet und das Ergebnis ausgewertet:

```
>> unfreeze(a)
```

$$\frac{\text{PI} \left(\left(2^{1/2} + 2^{1/2}\right) + \left(2^{1/2} - 2^{1/2}\right) \right)}{4}$$

```
>> float(%)
```

2.052344306

Beispiel 3. Dieses Beispiel demonstriert den Unterschied zwischen den Funktionen `hold` und `freeze`. Das Ergebnis der Eingabe `S := hold(sum)(...)` enthält nicht die inaktive Version der Funktion `sum`, sondern den nicht-evaluierten Bezeichner `sum`:

```
>> S := hold(sum)(1/n^2, n = 1..infinity)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

Zum Zeitpunkt der Evaluierung von `S` wird der Bezeichner `sum` durch seinen Wert, der Funktionsumgebung `sum`, ersetzt und die Berechnung der Reihe ausgelöst:

```
>> S
```

$$\frac{\text{PI}^2}{6}$$

Im Gegensatz zu `hold` führt eine Evaluation des Ergebnisses von `freeze` nicht zu einer entsprechenden Evaluation der inaktiven Funktion:

```
>> S := freeze(sum)(1/n^2, n = 1..infinity)
```

$$\text{sum} \left| \begin{array}{c} / 1 \\ --, n = 1..infinity \\ | 2 \\ \backslash n \end{array} \right|$$

```
>> S
```

$$\text{sum} \left| \begin{array}{c} / 1 \\ --, n = 1..infinity \\ | 2 \\ \backslash n \end{array} \right|$$

Selbst die Funktion `eval` bewirkt keine Evaluation einer inaktiven Funktion:

```
>> eval(S)
```

$$\text{sum} \left| \begin{array}{c} / 1 \\ --, n = 1..infinity \\ | 2 \\ \backslash n \end{array} \right|$$

Der einzige Weg, um die Anwendung von `freeze` rückgängig zu machen, ist die Funktion `unfreeze`, die jede inaktive Funktion in `S` re-aktiviert und das Ergebnis evaluiert:

```
>> unfreeze(S)
```

$$\frac{\pi^2}{6}$$

Beispiel 4. Man beachte, dass `freeze(f)` keine Änderungen an dem Objekt `f` vornimmt, sondern eine Kopie von `f` in einer inaktiven Form zurückliefert. Das bedeutet, dass Berechnungen mit der inaktiven Version von `f` die Originalfunktion `f` enthalten kann.

Als Beispiel erzeugen wir die inaktive Version der Sinusfunktion:

```
>> Sin := freeze(sin):
```

Die Anwendung der Funktion `expand` auf den Term `Sin(x+y)` liefert ein Ergebnis, das die (ursprüngliche) Sinusfunktion `sin` enthält:

```
>> expand(Sin(x + y))
```

$$\cos(x) \sin(y) + \cos(y) \sin(x)$$

Beispiel 5. Die Funktion `unfreeze` verwendet `misc::maprec`, um rekursiv auf der Struktur von `object` zu arbeiten. Ist `object` beispielsweise ein Feld, dessen Einträge inaktive Funktionen enthalten:

```
>> a := array(1..2,
  [freeze(int)(sin(x), x = 0..2*PI), freeze(sum)(k^2, k = 1..n)]
)
```

```

+-                                     2                                     -+
| int(sin(x), x = 0..2 PI), sum(k  , k = 1..n) |
+-                                     -+

```

so bewirkt der Aufruf `unfreeze(a)` die Anwendung der Funktion `unfreeze` auf die Operanden des Feldes `a`:

```
>> unfreeze(a)
```

```

+-                                     -+
|                                     |
|          2          3          |
|          n          n          |
|    0, - + -- + --          |
|          6          2          3          |
+-                                     -+

```

Für Bibliotheks-Domains wird die Wirkung von `unfreeze` daher über die Methode "`maprec`" festgelegt. Falls das Domain diese Methode nicht implementiert, so hat `unfreeze` keine Wirkung auf die Objekte dieses Domains.

Als Beispiel erzeugen wir ein Domain und ein Objekt dieses Domains, dessen Operand eine inaktive Funktion enthält:

```
>> dummy := newDomain("dummy"):
  o := new(dummy, freeze(int)(sin(x), x = 0..2*PI))
      new(dummy, int(sin(x), x = 0..2 PI))
```

Die Funktion `unfreeze` hat keine Wirkung auf das Objekt `o`:

```
>> unfreeze(o)

      new(dummy, int(sin(x), x = 0..2 PI))
```

Überladen wir nun die Funktion `misc::maprec`, indem wir die Funktion `unfreeze` auf den ersten Operanden von Objekten des Domains `dummy` wirken lassen, so erhalten wir das gewünschte Resultat:

```
>> dummy::maprec :=
  x -> extsubsop(x,
    1 = misc::maprec(extop(x,1), args(2..args(0)))
  ):
  unfreeze(o)

      new(dummy, 0)
```

Änderungen:

- ⌘ `freeze` hieß früher `misc::freeze`.
 - ⌘ `unfreeze` hieß früher `misc::unfreeze`.
-

`ftextinput` – Einlesen einer Textdatei

`ftextinput(filename, x)` liest eine Zeile aus der Textdatei namens `filename`, interpretiert die Zeile als Zeichenkette und weist diese dem Bezeichner `x` zu.

`ftextinput(n, x)` liest aus der mit dem Dateibezeichner `n` verknüpften Datei.

Aufruf(e):

- ⌘ `ftextinput(filename)`
- ⌘ `ftextinput(filename, x1, x2, ...)`
- ⌘ `ftextinput(n)`
- ⌘ `ftextinput(n, x1, x2, ...)`

Parameter:

- `filename` — der Dateiname: eine Zeichenkette
- `n` — ein von `fopen` erzeugter Dateibezeichner: eine positive ganze Zahl
- `x1, x2, ...` — Bezeichner

Rückgabewert: die zuletzt eingelesene Zeile: eine Zeichenkette.

Verwandte Funktionen: `fclose`, `finput`, `fopen`, `fprint`, `fread`, `input`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `textinput`, `write`, `WRITEPATH`

Details:

- ⌘ `ftextinput(filename)` liest die erste Zeile der Textdatei und liefert sie als Zeichenkette an die MuPAD-Sitzung.
- ⌘ `ftextinput(filename, x1, x2, ...)` liest nacheinander die einzelnen Zeilen der Datei, wandelt sie in Zeichenketten um und weist die i -te Zeile dem Bezeichner x_i zu. Die Bezeichner werden von `ftextinput` nicht evaluiert. Eventuell vorher zugewiesene Werte werden überschrieben.

- ☞ Anstatt eines Dateinamens kann auch der Dateibezeichner `n` einer mittels `fopen` geöffneten Datei übergeben werden. Die Funktionalität ist in beiden Fällen dieselbe. Es gibt jedoch einen Unterschied: Bei Angabe eines Dateinamens wird die Datei nach dem Lesen der Daten automatisch wieder geschlossen. Ein erneutes Lesen beginnt wieder am Anfang der Datei. Bei Angabe eines Dateibezeichners bleibt die Datei nach dem Lesen geöffnet (sie muss explizit mittels `fclose` geschlossen werden), und ein erneutes Lesen setzt an der aktuellen Position fort. Sollen mehrere Daten aus einer Datei durch aufeinanderfolgende Aufrufe von `ftextinput` gelesen werden, so ist dementsprechend ein Dateibezeichner anstelle eines Dateinamens zu benutzen. Siehe Beispiel ??.
- ☞ Ist die Anzahl der an `ftextinput` übergebenen Bezeichner größer als die Anzahl der Zeilen in der Datei, so werden den überschüssigen Bezeichnern keine Werte zugewiesen. Der Rückgabewert von `ftextinput` ist in einem solchen Fall das leere Objekt vom Typ `DOM_NULL`.
- ☞ `ftextinput` interpretiert den Dateinamen als Pfadnamen relativ zum „aktuellen Arbeitsverzeichnis“.
 Man beachte, dass die Bedeutung des „aktuellen Arbeitsverzeichnisses“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.
 Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann.
 Auch absolute Pfadnamen werden von `ftextinput` verarbeitet.
- ☞ Ausdrucksfolgen werden von `ftextinput` nicht ausgeglichen und können daher nicht benutzt werden, um mehrere Bezeichner an `ftextinput` zu übergeben. Siehe Beispiel ??.
- ☞ `ftextinput` ist eine Funktion des Systemkerns.

Beispiel 1. Zuerst wird mittels `fprint` eine Textdatei mit drei Zeilen erzeugt:

```
>> fprint(Unquoted, Text, "test", "x + 1\n2nd line\n3rd line");
```

Die beiden ersten Zeilen der Datei werden gelesen und den Bezeichnern `x1` und `x2` zugewiesen:

```
>> ftextinput("test", x1, x2): x1, x2
                                "x + 1", "2nd line"
```

Wird versucht, mehr Zeilen als wirklich vorhanden auszulesen, so liefert `ftextinput` das leere Objekt vom Typ `DOM_NULL`:


```
>> ftextinput("test", x1, x2, x3, x4); domtype(%)
```

```
DOM_NULL
```

```
>> x1, x2, x3, x4
```

```
"x + 1", "2nd line", "3rd line", x4
```

```
>> delete x1, x2, x3:
```

Beispiel 2. Hier werden mehrere Daten durch aufeinanderfolgende `ftextinput` Aufrufe aus einer Datei gelesen. Hierbei muss man mit einem Dateibezeichner auf die Datei zugreifen. Die Datei wird mittels `fopen` im Lesemodus geöffnet:

```
>> fprintf(Unquoted, Text, "test",
           "x + 1\nx + 2\n3rd line\n4th line");
```

```
>> n := fopen("test");
```

Der von `fopen` gelieferte Dateibezeichner kann an `ftextinput` übergeben werden, um die Daten nacheinander einzulesen:

```
>> ftextinput(n, x1, x2): x1, x2
```

```
"x + 1", "x + 2"
```

```
>> ftextinput(n, x3, x4): x3, x4
```

```
"3rd line", "4th line"
```

Die Datei wird geschlossen, und die benutzten Bezeichner werden gelöscht:

```
>> fclose(n): delete n, x1, x2, x3, x4:
```

Alternativ kann der Inhalt einer Datei in der folgenden Weise in eine MuPAD-Sitzung eingelesen werden:

```
>> n := fopen("test"):
  for i from 1 to 4 do
    x.i := ftextinput(n)
  end_for:
  x1, x2, x3, x4

  "x + 1", "x + 2", "3rd line", "4th line"

>> fclose(n): delete n, i, x1, x2, x3, x4:
```

Beispiel 3. Ausdrucksfolgen werden von `ftextinput` nicht ausgeglichen und können daher nicht benutzt werden, um Bezeichner an `ftextinput` zu übergeben:

```
>> fprintf(Unquoted, Text, "test", "1st line\n2nd line\n3rd line"):
      ftextinput("test", (x1, x2), x3)
```

```
Error: Illegal argument [ftextinput]
```

Der folgende Aufruf führt nicht zu einem Fehler, weil der Bezeichner `x12` nicht evaluiert wird. Dementsprechend wird nur eine einzige Zeile ausgelesen und `x12` zugewiesen:

```
>> x12 := x1, x2: ftextinput("test", x12): x1, x2, x12

      x1, x2, "1st line"

>> delete x12:
```

Änderungen:

⌘ Keine Änderungen.

funcenv – Erzeugen einer Funktionsumgebung

`funcenv` erzeugt eine Funktionsumgebung. Eine Funktionsumgebung verhält sich wie eine gewöhnliche Funktion, für die jedoch Funktionsattribute definiert werden können. Diese können benutzt werden, um Systemfunktionen wie z. B. `diff` oder `float` zu überladen, d. h., sie definieren das Verhalten der Funktion, wenn sie von Systemfunktionen aufgerufen wird.

Aufruf(e):

⌘ `funcenv(f1 <, f2> <, slotTable>)`

Parameter:

- `f1` — ein beliebiges MuPAD-Objekt. Typischerweise eine Prozedur. Sie legt die Evaluierung eines Funktionsaufrufs der Funktionsumgebung fest.
- `f2` — eine Prozedur, die die Bildschirmausgabe symbolischer Funktionsaufrufe festlegt
- `slotTable` — eine Tabelle von Funktionsattributen (slots)

Rückgabewert: eine Funktionsumgebung vom Typ `DOM_FUNC_ENV`.

Weitere Dokumentation: Kapitel „Funktionsumgebungen“ des Tutoriums.

Verwandte Funktionen: `slot`

Details:

☞ `funcenv` erzeugt Funktionsumgebungen des Domaintyps `DOM_FUNC_ENV`.

Aus Benutzersicht verhalten sich Funktionsumgebungen ähnlich wie Prozeduren und können wie Funktionen benutzt werden.

Im Gegensatz zu einfachen Prozeduren ermöglichen Funktionsumgebungen jedoch eine weitgehende Integration in das MuPAD System. Es kann festgelegt werden, wie sich bestehende Systemfunktionen wie `diff`, `expand`, `float` usw. beim Auswerten eines symbolischen Aufrufs einer Funktionsumgebung verhalten sollen.

Hierfür speichert eine Funktionsumgebung spezielle Funktionsattribute (Slots) in einer internen Tabelle. Wenn eine überladbare Systemfunktion wie `diff`, `expand` oder `float` nun bei der Auswertung auf ein Objekt vom Typ `DOM_FUNC_ENV` trifft, so wird die Funktionsumgebung nach dem entsprechenden Slot durchsucht. Wird dieser gefunden, so wird der Slot aufgerufen und sein Ergebnis zurückgeliefert.

Slots können der Funktionsumgebung angeheftet werden, indem sie als Tabelle `slotTable` bei der Erzeugung der Funktionsumgebung an `funcenv` übergeben werden. Alternativ kann die Funktion `slot` zum Einfügen weiterer Slots in bereits existierende Funktionsumgebungen benutzt werden.

Das Beispiel ?? gibt hierzu weitere Informationen.

☞ Das erste Argument `f1` von `funcenv` bestimmt die Auswertung des Funktionsaufrufs einer Funktionsumgebung. Mit `f := funcenv(f1)` liefert der Aufruf `f(x)` das Ergebnis `f1(x)`. Man beachte, dass Aufrufe der Form `f := funcenv(f)` möglich und auch typisch sind. Die Prozedur `f` wird hiermit in eine Funktionsumgebung des gleichen Namens eingebettet, wobei die ursprüngliche Prozedur intern gespeichert wird. Nachdem die Funktionsumgebung `f` so erzeugt wurde, können weitere Funktionsattribute durch die `slot`-Funktion hinzugefügt werden.

☞ Das zweite Argument `f2` von `funcenv` bestimmt die Bildschirmausgabe des Funktionsaufrufs einer Funktionsumgebung. Ist die Funktionsumgebung durch `f := funcenv(f1, f2)` definiert und liefert `f1` einen symbolischen Funktionsaufruf `f(x)` mit dem 0-ten Operanden `f` zurück, so wird zur Bildschirmausgabe von `f(x)` das Ergebnis von `f2(f(x))` benutzt.

Man beachte, dass `f2(f(x))` kein Ergebnis erzeugen darf, das seinerseits wieder einen symbolischen Aufruf von `f` enthält, da dies dann zu einer unendlichen Rekursion der Ausgabefunktion führt.



- ☞ Das dritte Argument `slotTable` von `funcenv` ist eine Tabelle, welche die Funktionsattribute (Slots) enthält. Die Tabelle muss Zeichenketten als Indizes zur Bestimmung der zu überladenden Systemfunktion benutzen. So fügt z. B.

```
slotTable := table("diff" = mydiff, "float" = myfloat):
f := funcenv(f1, f2, slotTable):
```

der Funktionsumgebung `f` die Funktionen `mydiff` und `myfloat` als Slot "diff" und "float" hinzu. Die Routinen `mydiff` und `myfloat` werden durch die Systemfunktion `diff` bzw. `float` aufgerufen, wenn diese auf einen symbolischen Ausdruck $f(x)$ mit dem 0-ten Operanden `f` treffen. Die interne Slottabelle kann mit der Funktion `slot` geändert und erweitert werden.

- ☞ Auf den Hilfeseiten von `float`, `print` und `slot` finden sich weitere Beispiele zum Umgang mit Funktionsumgebungen.
- ☞ `funcenv` ist eine Funktion des Systemkerns.

Beispiel 1. Es soll eine Funktion `f` eingeführt werden, welche eine Lösung der Differentialgleichung $f'(x) = x + \sin(x)f(x)$ repräsentieren soll. Dazu wird zunächst eine Funktion `f` definiert, die jeden Aufruf $f(x)$ symbolisch zurückliefert:

```
>> f := proc(x) begin procname(args()) end_proc: f(x), f(3 + y)

f(x), f(y + 3)
```

Wegen der Differentialgleichung $f'(x) = x + \sin(x)f(x)$ können Ableitungen von `f` mittels `f` ausgedrückt werden. Wie kann dem MuPAD-System mitgeteilt werden, wie $f(x)$ zu differenzieren ist? Dazu wird zunächst die Prozedur `f` in eine Funktionsumgebung gleichen Namens eingebettet:

```
>> f := funcenv(f):
```

Diese Funktionsumgebung verhält sich wie die ursprüngliche Prozedur:

```
>> f(x), f(3 + y)

f(x), f(y + 3)
```

Systemfunktionen wie z. B. `diff` behandeln symbolische Aufrufe von `f` zunächst noch wie Aufrufe unbekannter Funktionen:

```
>> diff(f(x + 3), x)

D(f)(x + 3)
```

Als Funktionsumgebung kann `f` nun Funktionsattribute erhalten, welche die Systemfunktionen überladen können. Der folgende `slot`-Aufruf heftet ein "diff"-Attribut an `f`:

```
>> f := slot(f, "diff", mydiff): diff(2*f(x^2) + x, x)
```

$$2 \operatorname{mydiff}(f(x^2), x) + 1$$

Unter Ausnutzung von $f'(x) = x + \sin(x)f(x)$ wird `mydiff` durch ein sinnvol-
leres "diff"-Attribut ersetzt. Es muss beliebige Aufrufe der Form `diff(f(y),
x1, x2, ...)` verarbeiten:

```
>> fdiff := proc(fcall) local y; begin
    y:= op(fcall, 1);
    (y + sin(y)*f(y))*diff(y, args(2..args(0)))
end_proc:
f := slot(f, "diff", fdiff):
```

Soweit es die Differentiation betrifft, ist die Funktion `f` nun vollständig in Mu-
PAD integriert:

```
>> diff(f(x), x), diff(f(x), x, x)
```

$$x + f(x) \sin(x), f(x) \cos(x) + \sin(x) (x + f(x) \sin(x)) + 1$$

```
>> diff(sin(x)*f(x^2), x)
```

$$\cos(x) f(x^2) + 2 x \sin(x) (x^2 + f(x^2) \sin(x^2))$$

Da für eine Taylor-Entwicklung um einen endlichen Punkt nur Ableitungs-
werte berechnet werden müssen, können auch Taylor-Entwicklungen von `f`
berechnet werden:

```
>> taylor(f(x^2), x = 0, 9)
```

$$f(0) + x \left| \frac{4}{2} \frac{f(0)}{2} + \frac{1}{2} \right| + x \left| \frac{8}{12} \frac{f(0)}{12} + \frac{1}{8} \right| + O(x^9)$$

```
>> delete f, fdiff:
```

Hintergründe:

- ⌘ Mathematische Funktion wie `exp`, `ln` etc. oder auch `abs`, `Re`, `Im` etc.
sind als Funktionsumgebungen implementiert.

Änderungen:

☞ `funcenv` hieß früher `func_env`.

`gamma` – die Gammafunktion

`gamma(x)` stellt die Gammafunktion $\int_0^\infty e^{-t} t^{x-1} dt$ dar.

Aufruf(e):

☞ `gamma(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `beta`, `fact`, `igamma`, `psi`

Details:

☞ Die Gammafunktion ist für alle komplexen Argumente außer den Polstellen $0, -1, -2, \dots$ definiert.

☞ Für positive ganze Zahlen ist sie mit der Fakultätsfunktion verwandt:
`gamma(x) = fact(x-1) = (x-1)!`.

☞ Ist `x` eine Gleitpunktzahl, so wird eine Gleitpunktzahl zurückgeliefert. Ist `x` eine positive ganze Zahl < 1000 , so wird eine ganze Zahl zurückgeliefert. Ist `x` eine rationale Zahl vom Domain-Typ `DOM_RAT` im Bereich $1 < x < 500$, so wird das Resultat mittels der Rekursion `gamma(x) = (x-1) * gamma(x-1)` „normalisiert“. Die Beziehung `gamma(x) * gamma(1-x) = PI * csc(PI * x)` wird benutzt, falls `x` $< 1/2$ eine rationale Zahl vom Domain-Typ `DOM_RAT` und ein ganzzahliges Vielfaches von $1/4$ oder $1/6$ ist. Der Aufruf `gamma(1/2)` liefert `sqrt(PI)`; `gamma(infinity)` liefert `infinity`.

Für alle anderen Argumente wird ein symbolischer Aufruf von `gamma` zurückgeliefert.

- ☞ Das float-Attribut von gamma ist eine Kernfunktion, d. h., die numerische Auswertung ist schnell.
- ☞ Die logarithmische Ableitung von gamma ist durch die Digammafunktion psi implementiert.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> gamma(15), gamma(23/2), gamma(sqrt(2)), gamma(x + 1)

13749310575 PI 1/2
87178291200, -----, gamma(2  ), gamma(x + 1)
2048
```

Für Gleitkommazahlen wird der numerische Wert von gamma berechnet:

```
>> gamma(11.5), gamma(2.0 + 10.0*I)

11899423.08, - 0.00001089258677 + 0.00000504737724 I
```

Beispiel 2. gamma ist singulär für nicht-positive ganze Zahlen:

```
>> gamma(-2)

Error: singularity [gamma]
```

Beispiel 3. Die Funktionen diff, expand, float, limit und series verarbeiten symbolische gamma-Ausdrücke:

```
>> diff(gamma(x^2 + 1), x), float(ln(3 + gamma(sqrt(PI))))

2 2
2 x psi(x + 1) gamma(x + 1), 1.367203476

>> expand(gamma(3*x - 4))

gamma(3 x)
-----
(3 x - 1) (3 x - 2) (3 x - 3) (3 x - 4)

>> limit(1/gamma(x), x = infinity),
limit(gamma(x - 4)/gamma(x - 10), x = 0)

0, 151200
```

```
>> series(gamma(x), x = 0, 4)
```

$$\frac{1}{x} - \frac{\text{EULER}}{6} + x \frac{\text{PI}^2}{12} + \frac{x^2 \text{PI}^2 \text{EULER}}{288} + O(x^3)$$

Die Stirling Formel erhält man durch asymptotische Entwicklung:

```
>> series(gamma(x), x = infinity, 3)
```

$$\frac{x^x \exp(-x) \sqrt{2\pi x}}{12x} + \frac{x^x \exp(-x) \sqrt{2\pi x}}{288x^2} + O\left(\frac{x^x \exp(-x) \sqrt{2\pi x}}{x^3}\right)$$

Änderungen:

☞ Keine Änderungen.

gcd – der größte gemeinsame Teiler von Polynomen

`gcd(p, q, ...)` berechnet den größten gemeinsamen Teiler der Polynome p, q, \dots

Aufruf(e):

☞ `gcd(p, q, ...)`

☞ `gcd(f, g, ...)`

Parameter:

p, q, \dots — Polynome vom Typ `DOM_POLY`
 f, g, \dots — polynomiale Ausdrücke

Rückgabewert: Ein Polynom, ein polynomialer Ausdruck oder der Wert FAIL.

Überladbar durch: p , q , f , g

Verwandte Funktionen: `content`, `div`, `divide`, `factor`, `gcdex`,
`icontent`, `ifactor`, `igcd`, `igcdex`, `ilcm`, `lcm`, `mod`, `poly`

Details:

- ☞ `gcd(p, q, ...)` berechnet den größten gemeinsamen Teiler einer beliebigen Anzahl von Polynomen. Der Koeffizientenring der Polynome kann aus ganzen oder rationalen Zahlen bestehen oder *Expr*, ein Restklassenring *IntMod*(n) mit einer Primzahl n oder ein Domain sein. Alle Polynome müssen dieselben Unbestimmten und denselben Koeffizientenring haben.
 - ☞ Polynomiale Ausdrücke werden in Polynome konvertiert. Siehe `poly` zu Details der Konvertierung. `gcd` liefert den Wert FAIL, falls ein Argument nicht konvertiert werden kann.
 - ☞ Der Rückgabewert ist vom selben Typ wie die Eingabepolynome, d. h., entweder ein Polynom vom Typ DOM_POLY oder ein polynomialer Ausdruck.
 - ☞ Wenn alle Argumente 0 sind oder kein Argument angegeben wird, dann gibt `gcd` 0 zurück. Wenn mindestens ein Argument gleich 1 oder -1 ist, dann gibt `gcd` 1 zurück.
 - ☞ Wenn alle Argumente ganzzahlig sind, so wird empfohlen, die deutlich schnellere Funktion `igcd` zu verwenden.
-

Beispiel 1. Der größte gemeinsame Teiler zweier polynomialer Ausdrücke kann wie folgt berechnet werden:

```
>> gcd(6*x^3 + 9*x^2*y^2, 2*x + 2*x*y + 3*y^2 + 3*y^3)
```

$$2x^2 + 3y^2$$

```
>> f := (x - sqrt(2))*(x^2 + sqrt(3)*x-1):  
g := (x - sqrt(2))*(x - sqrt(3)):  
gcd(f, g)
```

$$x - \sqrt{2}$$

Als Argumente sind auch Polynome zulässig:

```
>> p := poly(2*x^2 - 4*x*y - 2*x + 4*y, [x, y], IntMod(17)):
    q := poly(x^2*y - 2*x*y^2, [x, y], IntMod(17)):
    gcd(p, q)

        poly(x - 2 y, [x, y], IntMod(17))

>> delete f, g, p, q:
```

Hintergründe:

- ⌘ Sind die Argumente Polynome und gehören deren Koeffizienten zu einem Domain, so muss dieses die Methoden "gcd" und "_divide" haben. Die Methode "gcd" muss den größten gemeinsamen Teiler beliebig vieler Domain-Elemente liefern; "_divide" muss ein Domain-Element durch ein anderes teilen. Ist dies nicht möglich, muss diese Methode FAIL zurückgeben.

Änderungen:

- ⌘ Beliebige Ausdrücke werden nun als Koeffizienten akzeptiert.
-

gcdex – der erweiterte Euklidische Algorithmus für Polynome

gcdex(p, q, x) behandelt p und q als univariate Polynome in x und liefert ihren größten gemeinsamen Teiler als Linearkombination von p und q.

Aufruf(e):

- ⌘ gcdex(p, q, x)
- ⌘ gcdex(f, g, x)

Parameter:

- p, q — Polynome vom Typ DOM_POLY
- f, g — polynomiale Ausdrücke
- x — eine Unbestimmte: ein Bezeichner oder ein indizierter Bezeichner

Rückgabewert: eine Folge von drei Polynomen oder eine Folge von drei polynomialen Ausdrücken oder FAIL.

Überladbar durch: p, q

Verwandte Funktionen: factor, div, divide, gcd, ifactor, igcd, igcdex, ilcm, lcm, mod, poly

Details:

☞ `gcdex(p, q, x)` liefert die Folge g, s, t . Hierbei ist das Polynom g der größte gemeinsame Teiler von p und q . Die Polynome s und t erfüllen $g = sp + tq$ mit $\deg(s) < \deg(q)$, $\deg(t) < \deg(p)$. Diese Daten werden mittels des erweiterten Euklidischen Algorithmus berechnet.

☞ `gcdex` verarbeitet nur univariate Polynome:

- Wird eine Unbestimmte x angegeben, so werden die Eingabepolynome als univariate Polynome in x behandelt.
- Ist keine Unbestimmte angegeben, so wird intern nach der Unbestimmten der Polynome gesucht. Ein Fehler wird ausgelöst, wenn mehr als eine Unbestimmte gefunden wird.

Man beachte, dass eine Unbestimmte x angegeben werden muss, wenn polynomiale Ausdrücke als Eingabe benutzt werden.

☞ Polynomiale Ausdrücke werden in Polynome konvertiert. Siehe `poly` zu Details der Konvertierung. `gcdex` liefert `FAIL`, falls ein Argument nicht konvertiert werden kann,

☞ Die Rückgabepolynome sind arithmetische Ausdrücke, wenn die Eingabe aus polynomialen Ausdrücken besteht. Anderenfalls werden Polynome vom Typ `DOM_POLY` zurückgegeben.

☞ Der Koeffizientenring der Polynome muss die Methode "`_divide`" implementieren. Diese Methode muss `FAIL` liefern, wenn Domain-Elemente nicht dividiert werden können.

☞ Ist der Koeffizientenring der Polynome kein Körper, so kann ein größter gemeinsamer Teiler nicht immer als Linearkombination der Eingabepolynome dargestellt werden. In einem solchen Fall liefert `gcdex` einen Fehler.



Beispiel 1. Der größte gemeinsame Teiler zweier univariater Polynome in erweiterter Form kann wie folgt berechnet werden:

```
>> gcdex(poly(x^3 + 1), poly(x^2 + 2*x + 1))  
  
poly(x + 1, [x]), poly(1/3, [x]), poly(- 1/3 x + 2/3, [x])
```

Bei multivariaten polynomialen Ausdrücken muss eine Unbestimmte angegeben werden:

```
>> gcdex(poly(x^2*y), poly(x + y), x)
```

```

poly(1, [x]), poly| / 1 \ / / 1 \ 1 \
| 3 | | - -- | x + -, [x] |
\ y / \ \ y /
>> gcdex(poly(x^2*y), poly(x + y), y)

poly(1, [y]), poly| / 1 \ / 1 \
| 3 | | - -- | \ x /
\ x / /
>> gcdex(x^3 + a, x^2 + 1, x)

2
a + x 1 - x - a x
1, -----, -----
2 2
a + 1 a + 1

```

Änderungen:

☞ Keine Änderungen.

genident – Erzeugung eines unbenutzten Bezeichners

`genident()` erzeugt einen Bezeichner, der in der aktuellen Sitzung noch nicht verwendet wurde.

Aufruf(e):

☞ `genident()`
☞ `genident(S)`

Parameter:

`S` — eine Zeichenkette

Rückgabewert: ein Bezeichner.

Verwandte Funktionen: `delete`, `hold`

Details:

- ⌘ `genident()` erzeugt einen Bezeichner mit einem Namen der Form x_i , wobei i eine natürliche Zahl ist. Es ist garantiert, dass der Bezeichner in der aktuellen MuPAD-Sitzung noch nicht verwendet wurde.
 - ⌘ Wird als Argument eine Zeichenkette S angegeben, so gibt `genident` einen Bezeichner mit einem Namen der Form S_i zurück, wobei i eine natürliche Zahl ist.
 - ⌘ Der zurückgegebene Bezeichner hat keinen Wert.
 - ⌘ `genident` ist eine Funktion des Systemkerns.
-

Beispiel 1. Drei Bezeichner werden erzeugt. Der zweite Bezeichner bekommt ein anderes Präfix:

```
>> genident(), genident("Y"), genident()  
  
x1, y1, x2
```

Im nächsten Beispiel weisen wir dem Bezeichner x_4 einen Wert zu. Nachfolgende Aufrufe von `genident` überspringen dann den Namen x_4 :

```
>> x4 := 5:  
    genident(), genident()  
  
x3, x5
```

Änderungen:

- ⌘ Keine Änderungen.
-

genpoly – Erzeugung eines Polynoms mit Hilfe „b“-adischer Entwicklung

`genpoly(n , b , x)` erzeugt ein Polynom p in der Unbestimmten x , so dass $p(b) = n$ gilt.

Aufruf(e):

- ⌘ `genpoly(n , b , x)`

Parameter:

- n — eine ganze Zahl, ein Polynom vom Typ `DOM_POLY`, oder ein polynomialer Ausdruck
- b — eine ganze Zahl größer als 1
- x — die Unbestimmte des erzeugten Polynoms: ein Bezeichner

Rückgabewert: ein Polynom, wenn das erste Argument ein Polynom oder eine ganze Zahl ist. Sonst ein polynomialer Ausdruck.

Verwandte Funktionen: `genident`, `indets`, `int2text`, `mods`, `numlib::g_adic`, `numeric::lagrange`, `poly`, `text2int`

Details:

- ☞ `genpoly(n, b, x)` erzeugt ein Polynom p in der Variablen x aus der b -adischen Entwicklung von n , so dass $p(b) = n$ gilt. Die ganzzahligen Koeffizienten des resultierenden Polynoms sind größer als $-b/2$ und kleiner oder gleich $b/2$.
- ☞ Die b -adische Entwicklung einer ganzen Zahl n ist als $n = \sum_{i=0}^m c_i b^i$ definiert, wobei die c_i symmetrische Reste modulo b sind, d. h. $-b/2 < c_i \leq b/2$ gilt für alle i (siehe `mods`). Es wird das Polynom $\sum_{i=0}^m c_i x^i$ erzeugt. Das Polynom ist über dem Koeffizientenring `Expr` definiert.
- ☞ Ist das erste Argument von `genpoly` ein (multivariate) Polynom, muss es über dem Koeffizientenring `Expr` definiert sein und darf nur ganze Zahlen als Koeffizienten haben. Das dritte Argument x darf keine Variable des Polynoms sein. In diesem Fall wird jeder Koeffizient b -adisch zu einem Polynom in x expandiert, wie oben beschrieben. Das Ergebnis ist ein Polynom in x , gefolgt von den Variablen des gegebenen Polynoms (x ist die Haupt-Variable des neuen Polynoms).
- ☞ Das erste Argument kann auch ein polynomialer Ausdruck sein. Dann wird dieser Ausdruck in ein Polynom umgewandelt (siehe Funktion `poly`), `genpoly` wird, wie oben beschrieben, auf das Polynom angewandt, und das Ergebnis wird wieder in einen polynomialen Ausdruck zurückkonvertiert.
- ☞ Ist das erste Argument eine ganze Zahl oder ein Polynom des Typs `DOM_POLY`, wird ein Polynom zurückgegeben, sonst ein polynomialer Ausdruck.
- ☞ `genpoly` ist eine Funktion des Systemkerns.

Beispiel 1. Wir erzeugen ein Polynom p in der Unbestimmten x , so dass $p(7) = 15$ gilt. Die Koeffizienten von p liegen zwischen -3 und 3 :

```
>> p := genpoly(15, 7, x)
```

```
poly(2 x + 1, [x])
```

```
>> p(7)
```

15

Es folgt ein Beispiel mit einem polynomialen Ausdruck als Eingabe:

```
>> p := genpoly(15*y^2 - 6*y + 3*z, 7, x)
```

$$y^2 + 3z^2 - xy^2 + y^2 + 2xy^2$$

Der Rückgabewert hat den selben Typ wie das erste Argument:

```
>> p := genpoly(poly(15*y^2 + 8*z, [y, z]), 7, x)
```

$$\text{poly}(2xy^2 + xz^2 + y^2 + z, [x, y, z])$$

Wir überprüfen das Ergebnis:

```
>> p(7, y, z)
```

$$8z^2 + 15y^2$$

Änderungen:

☞ Keine Änderungen.

getpid – die Prozess-ID des laufenden MuPAD-Kerns

Auf UNIX- bzw. Linux-Systemen liefert getpid() die Prozess-ID des laufenden MuPAD-Kerns.

Aufruf(e):

☞ getpid()

Rückgabewert: eine nichtnegative ganze Zahl.

Verwandte Funktionen: sysname, system

Details:

- ⇒ Auf anderen Betriebssystemen als UNIX oder Linux liefert `getpid` den Wert 0.
 - ⇒ Die Prozess-ID kann eine nützliche Information sein, wenn mit anderen Prozessen kommuniziert oder mittels `system` UNIX-Befehle abgesetzt werden sollen.
 - ⇒ Unter anderen Betriebssystemen gibt `getpid` 0 zurück.
 - ⇒ `getpid` ist eine Funktion des Systemkerns.
-

Beispiel 1. Die Prozess-ID des laufenden Kerns ist eine ganze Zahl:

```
>> getpid()  
  
16184
```

Änderungen:

- ⇒ Keine Änderungen.
-

`getprop` – Abfragen von Eigenschaften

`getprop(f)` liefert eine mathematische Eigenschaft des Ausdrucks `f`.

Aufruf(e):

- ⇒ `getprop(f)`
- ⇒ `getprop()`

Parameter:

`f` — ein arithmetischer Ausdruck

Rückgabewert: `getprop(f)` gibt eine Eigenschaft vom Typ `Type::Property` oder den Ausdruck `f` zurück. `getprop()` gibt eine Eigenschaft oder den Bezeichner `Global` zurück.

Verwandte Funktionen: `assume`, `is`, `property::hasprop`, `Type::Property`, `unassume`

Details:

- Der „Property“-Mechanismus ermöglicht Vereinfachungen mathematischer Ausdrücke, die Bezeichner mit mathematischen Eigenschaften enthalten. Mit der Funktion `assume` können Eigenschaften (Annahmen) wie „ x ist reell“ oder „ x ist eine gerade Zahl“ an einen Bezeichner x „geheftet“ werden. Arithmetische Ausdrücke mit x tragen dann entsprechende Eigenschaften. Z. B. ist $1 + x^2$ positiv, wenn x eine reelle Zahl ist.

`getprop(f)` leitet die Eigenschaften des Ausdrucks f aus den Eigenschaften aller Bezeichner in f ab.

Eine Auflistung aller verfügbaren Eigenschaften ist mittels `?property` erhältlich.

- Wenn die Bezeichner innerhalb eines Ausdrucks keine Eigenschaften besitzen, gibt `getprop` den Ausdruck unverändert zurück. Insbesondere ergibt `getprop(x)` wieder x , wenn x ein Bezeichner ohne Eigenschaften ist. Siehe Beispiel ??.

Eine Ausnahme bilden die speziellen Funktionen `abs`, `Re` und `Im` mit symbolischen Argumenten. Unabhängig von den Argumenten repräsentieren solche Ausdrücke immer reelle Zahlen, sodass für diese Ausdrücke immer eine Eigenschaft abgeleitet werden kann. Siehe Beispiel ??.

- Der Aufruf `getprop()` gibt die globale Eigenschaft zurück, die mit `assume` definiert werden kann.

Der geschützte Bezeichner `Global` trägt die globale Eigenschaft. Wenn keine globale Eigenschaft definiert wurde, wird der Bezeichner `Global` zurückgegeben. Siehe Beispiel ??.

- Da nur grundlegende mathematische Eigenschaften vom System genau dargestellt werden können, führt `getprop` während der Ableitung der Eigenschaften eines Ausdrucks Vereinfachungen durch. Dabei kann es passieren, dass schwächere Eigenschaften abgeleitet werden, als es prinzipiell mathematisch möglich wäre. Siehe Beispiel ??.

- Die Funktion `is` erlaubt es, die Eigenschaften eines Ausdrucks mit einer vorgegebenen Eigenschaft zu vergleichen.

Beispiel 1. Wenn x ganzzahlig ist, muss $x^2 + 1$ eine positive ganze Zahl sein:

```
>> assume(x, Type::Integer):  
    getprop(x^2 + 1)
```

Type::PosInt

Liegt x im Intervall $[1, \text{infinity}]$, so hat der Ausdruck $1 - x$ die folgende Eigenschaft:

```
>> assume(x, Type::Interval([1], infinity)):
      getprop(1 - x)

      ]-infinity, 0] of Type::Real

>> unassume(x):
```

Beispiel 2. Ein Ausdruck wird unverändert zurückgegeben, wenn er konstant ist oder den enthaltenen Bezeichnern keine Eigenschaften zugeordnet sind:

```
>> getprop(x), getprop(x + 2*y), getprop(sin(3))

      x, x + 2 y, sin(3)
```

Beispiel 3. Globale Eigenschaften, die für alle Bezeichner gelten sollen, werden in der globalen Variablen `Global` gespeichert. Momentan ist keine globale Eigenschaft gesetzt:

```
>> getprop()

      Global
```

Im folgenden wird eine globale Eigenschaft gesetzt. Nun haben alle Bezeichner diese Eigenschaft:

```
>> assume(Type::Real):
      getprop(x), getprop(y), getprop((x + y)^2 + 1/2)

      Type::Real, Type::Real, Type::Positive
```

Die Funktionen `getprop` und `is` verknüpfen die globale Eigenschaft und die individuellen Eigenschaften von Bezeichnern mittels des logischen „und“:

```
>> assume(Type::Positive):
      assume(x, Type::Integer):
      getprop(x)

      Type::PosInt
```

Die globale Eigenschaft und die individuellen Eigenschaften können sich ausschließen. Dann wird die „leere Eigenschaft“ `property::Null` zurückgegeben:

```
>> assume(Type::Positive):
    assume(x < 0):
    getprop(x)

                                property::Null

>> delete x: unassume():
```

Beispiel 4. Die Funktionen `abs`, `Re` und `Im` haben eine „minimale Eigenschaft“: sie repräsentieren reelle Zahlen. `abs` repräsentiert nichtnegative reelle Zahlen:

```
>> delete x:
    getprop(abs(x)), getprop(Re(x)), getprop(Im(x))

                                Type::NonNegative, Type::Real, Type::Real
```

Beispiel 5. Die Menge der Quadrate aller Primzahlen kann nicht mit den in der `Type`-Bibliothek vorhandenen Eigenschaften dargestellt werden. Daher liefert `getprop` die schwächere Eigenschaft „ x^2 ist eine positive ganze Zahl“:

```
>> assume(x, Type::Prime):
    getprop(x^2)

                                Type::PosInt

>> unassume(x):
```

Änderungen:

- ⌘ Die Fähigkeiten des „property“-Mechanismus wurden verbessert.
-

ground – Absolutglied (konstanter Koeffizient) eines Polynoms

`ground(p)` liefert den konstanten Koeffizienten $p(0, 0, \dots)$ des Polynoms p .

Aufruf(e):

- ⌘ `ground(p)`
- ⌘ `ground(f)`
- ⌘ `ground(f, vars)`

Parameter:

- p — ein Polynom vom Typ `DOM_POLY`
 f — ein polynomialer Ausdruck
 $vars$ — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: ein Element des Koeffizientenrings von p , ein arithmetischer Ausdruck, oder `FAIL`.

Überladbar durch: p , f

Verwandte Funktionen: `coeff`, `collect`, `degree`, `degreevec`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- ☞ Das erste Argument kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `ground` überlädt.
- ☞ Wenn das erste Argument f nicht Element eines Polynom-Domains ist, so konvertiert `ground` diesen Ausdruck mittels `poly(f)` in ein Polynom. Ist eine Liste von Unbestimmten angegeben, so wird das Polynom `poly(f, vars)` betrachtet.
Der konstante Koeffizient wird als arithmetischer Ausdruck zurückgegeben.
- ☞ Das Ergebnis von `ground` wird nicht weiter evaluiert. Vollständige Evaluation kann mit der Funktion `eval` erzwungen werden. Siehe Beispiel ??.
- ☞ `ground` liefert `FAIL`, wenn f nicht in ein Polynom in den angegebenen Unbekannten konvertiert werden kann. Siehe Beispiel ??.

Beispiel 1. Es wird gezeigt, wie die Unbestimmten das Ergebnis beeinflussen:

```
>> f := 2*x^2 + 3*y + 1:
      ground(f), ground(f, [x]), ground(f, [y]),
      ground(poly(f)), ground(poly(f, [x])), ground(poly(f, [y]))
```

$$1, 3y + 1, 2x^2 + 1, 1, 3y + 1, 2x^2 + 1$$

Das Ergebnis stimmt mit der Auswertung am Nullpunkt überein:

```
>> subs(f, x = 0, y = 0), subs(f, x = 0), subs(f, y = 0)
```

$$1, 3y + 1, 2x^2 + 1$$

Man beachte den Unterschied zwischen `ground` und `tccoeff`:

```
>> g := 2*x^2 + 3*y:
      ground(g), ground(g, [x]);
      tccoeff(g), tccoeff(g, [x]);

      0, 3 y

      3, 3 y

>> delete f, g:
```

Beispiel 2. `ground` evaluiert sein Ergebnis nicht vollständig:

```
>> p := poly(27*x^2 + a, [x]): a := 5:
      ground(p), eval(ground(p))

      a, 5

>> delete p, a:
```

Beispiel 3. Der folgende Ausdruck ist syntaktisch kein polynomialer Ausdruck. Deswegen liefert `ground` FAIL:

```
>> f := (x^2 - 1)/(x - 1): ground(f)

      FAIL
```

Nach Kürzen mittels `normal` kann der konstante Koeffizient berechnet werden:

```
>> ground(normal(f))

      1

>> delete f:
```

Änderungen:

⌘ `ground` ist eine neue Funktion.

has – Existenz eines Objektes in einem anderen Objekt

`has(object1, object2)` testet, ob `object2` syntaktisch in `object1` vorkommt.

Aufruf(e):

```
# has(object1, object2)
# has(object1, l)
```

Parameter:

object1, object2 — beliebige MuPAD-Objekte
l — eine Liste oder eine Menge

Rückgabewert: entweder TRUE oder FALSE

Überladbar durch: object1

Verwandte Funktionen: _in, _index, contains, hastype, op, subs, subsex

Details:

- # has ist eine schnelle Funktion, um das Vorkommen von Teilausdrücken oder Objekten innerhalb anderer Objekte zu überprüfen. has arbeitet rein syntaktisch, d. h. mathematisch äquivalente Objekte werden nur dann als gleich angesehen, wenn sie syntaktisch identisch sind. Siehe Beispiel ??.
 - # Wenn object1 ein Ausdruck ist, dann testet has(object1, object2), ob object2 in object1 als Teilausdruck enthalten ist. Dabei werden nur vollständige Teilausdrücke und Objekte, die im nullten Operanden eines Teilausdrucks vorkommen, gefunden (siehe Beispiel ??).
 - # Wenn object1 ein „Container“ ist, dann überprüft has, ob object2 in einem Eintrag von object1 vorkommt. Siehe Beispiel ??.
 - # Wenn das zweite Argument eine Liste oder Menge ist, dann gibt has TRUE zurück, wenn mindestens ein Element von l in object1 enthalten ist (siehe Beispiel ??). has gibt immer FALSE zurück, wenn l keine Elemente hat.
 - # Wenn object1 ein Element eines Domains mit einem "has"-Slot ist, so wird die entsprechende Slot-Prozedur mit denselben Argumenten aufgerufen und deren Ergebnis zurückgegeben. Wenn das Domain keinen solchen Slot besitzt, wird FALSE zurückgegeben. Siehe Beispiel ??.
- Wird der Funktion has eine Liste oder Menge übergeben, so wird der "has"-Slot des Domains von object1 nacheinander für jedes Objekt der Liste oder Menge aufgerufen. Dies wird so lange gemacht, bis die Slot-Prozedur TRUE für eines der Objekte zurückgibt, und dann wird TRUE zurückgegeben. Andernfalls ist der Rückgabewert FALSE.
- # has ist eine Funktion des Systemkerns.
-

Beispiel 1. Der gegebene Ausdruck hat x als Operand:

```
>> has(x + y + z, x)
```

TRUE

Zu beachten ist, dass $x + y$ kein vollständiger Teilausdruck ist. Nur x , y , z und $x + y + z$ sind komplette Teilausdrücke:

```
>> has(x + y + z, x + y)
```

FALSE

`has` findet auch Objekte, die im nullten Operanden eines Teilausdrucks vorkommen:

```
>> has(x + sin(x), sin)
```

TRUE

Jedes Objekt enthält sich selbst:

```
>> has(x, x)
```

TRUE

Beispiel 2. `has` findet nur syntaktisch identische Teilausdrücke. Die Ausdrücke $y*(x + 1)$ und $y*x + y$ sind zwar mathematisch äquivalent, aber syntaktisch voneinander verschieden:

```
>> has(sin(y*(x + 1)), y*x + y),  
    has(sin(y*(x + 1)), y*(x + 1))
```

FALSE, TRUE

Komplexe Zahlen werden nicht als atomare Objekte behandelt:

```
>> has(2 + 5*I, 2), has(2 + 5*I, 5), has(2 + 5*I, I)
```

TRUE, TRUE, TRUE

Im Gegensatz dazu werden rationale Zahlen als atomar angesehen:

```
>> has(2/3*x, 2), has(2/3*x, 3), has(2/3*x, 2/3)
```

FALSE, FALSE, TRUE

Beispiel 3. Wenn das zweite Argument eine Liste oder Menge ist, überprüft `has`, ob mindestens ein Element im ersten Argument vorkommt:

```
>> has((x + y)*z, [x, t])

TRUE
```

Nullte Operanden werden ebenso überprüft:

```
>> has((a + b)*c, {_plus, _mult})

TRUE
```

Beispiel 4. `has` arbeitet auch mit Listen, Mengen, Tabellen und Feldern:

```
>> has([sin(f(a) + 2), cos(x), 3], {f, g})

TRUE

>> has({a, b, c, d, e}, {a, z})

TRUE

>> has(array(1..2, 1..2, [[1, 2], [3, 4]]), 2)

TRUE
```

In einem Feld `A` überprüft der Befehl `has(A, NIL)`, ob `A` undefinierte Einträge besitzt:

```
>> has(array(1..2, 1 = x), NIL),
    has(array(1..2, [2, 3]), NIL)

TRUE, FALSE
```

In Tabellen werden die Indizes, die Einträge und auch die Gleichungen `Index=Eintrag` (die Operanden einer Tabelle) überprüft:

```
>> T := table(a = 1, b = 2, c = 3):
    has(T, a), has(T, 2), has(T, b = 2)

TRUE, TRUE, TRUE
```

Beispiel 5. `has` arbeitet syntaktisch. Obwohl die Variable `x` mathematisch nicht in dem konstanten Polynom `p` des folgenden Beispiels vorkommt, enthält das Polynom den Bezeichner `x` als zweiten Operanden:

```
>> delete x: p := poly(1, [x]):
    has(p, x)

TRUE
```


Beispiel 6. Das zweite Argument kann ein beliebiges MuPAD-Objekt sein, z. B. auch ein beliebiges Element eines benutzerdefinierten Domains:

```
>> T := newDomain("T"):
      e := new(T, 1, 2);
      f := [e, 3];

                        new(T, 1, 2)

                    [new(T, 1, 2), 3]

>> has(f, e), has(f, new(T, 1))

                        TRUE, FALSE
```

Wenn das erste Argument von `has` zu einem Domain ohne "has"-Slot gehört, wird immer `FALSE` zurückgegeben:

```
>> has(e, 1)

                        FALSE
```

Die Funktion `has` kann durch den Benutzer für eigene Domains überladen werden. Das im vorletzten Beispiel konstruierte Domain `T` wird um eine "has"-Methode erweitert, die eine Liste mit den internen Operanden erzeugt und `has` mit dieser Liste aufruft:

```
>> T::has := (object1, object2) -> has([extop(object1)], object2):
```

Wenn man jetzt `has` mit dem Domainelement `e` des Domains `T` aufruft, so wird diese Methode `T::has` benutzt:

```
>> has(e, 1), has(e, 3)

                        TRUE, FALSE
```

Diese Methode wird auch aufgerufen, wenn ein Objekt des Domains `T` in einem anderen Objekt vorkommt:

```
>> has(f, 1), has(f, 3)

                        TRUE, TRUE
```

Änderungen:

☞ Keine Änderungen.

hastype – Test, ob ein Objekt eines bestimmten Typs in einem anderen Objekt vorkommt

`hastype(object, T)` testet, ob ein Objekt vom Typ `T` in `object` syntaktisch vorkommt.

Aufruf(e):

`hastype(object, T <, inspect>)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt
`T` — ein Typspezifizierer oder eine Liste oder Menge von Typspezifizierern
`inspect` — eine Menge von Domain-Typen

Rückgabewert: `TRUE` oder `FALSE`.

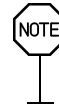
Überladbar durch: `object`

Verwandte Funktionen: `domtype`, `has`, `misc::maprec`, `testtype`, `Type`, `type`

Details:

- ☞ `hastype(object, T)` testet, ob `object` einen Teil-Operand `s` vom Typ `T` besitzt, für den `testtype(s, T)` `TRUE` ergibt.
- ☞ Der Typspezifizierer `T` kann entweder ein Domain-Typ sein wie `DOM_INT`, `DOM_EXPR` usw., eine Zeichenkette, die von der Funktion `type` zurückgegeben wird, oder ein Objekt der Bibliothek `Type`. Die vordefinierten Typspezifizierer der Bibliothek `Type` eignen sich meist besonders gut als Argumente für `T`.
Wenn `T` kein gültiger Typspezifizierer ist, gibt `hastype` `FALSE` zurück.
Siehe Beispiel ??.
- ☞ Wenn `object` ein Ausdruck ist, testet `hastype(object, T)`, ob `object` einen Teilausdruck vom Typ `T` enthält (siehe Beispiel ??).
Wenn `object` ein Behälter ist, testet `hastype`, ob ein Eintrag einen Teilausdruck vom Typ `T` enthält (siehe Beispiel ??).
- ☞ Ist das zweite Argument eine Liste oder eine Menge, so wird getestet, ob `object` ein Teilobjekt von einem der in `T` angegebenen Typen enthält.
Siehe Beispiel ??.
- ☞ `hastype` arbeitet rekursiv und analysiert dabei die Operanden der folgenden Objekte näher: Ausdrücke, Felder, Listen, Mengen und Tabellen (siehe Beispiel ??).
`hastype` zerlegt Objekte von Basis-Domains nicht weiter, wie rationale oder komplexe Zahlen, Polynome oder Prozeduren (siehe Beispiel ??).
- ☞ Ist das dritte Argument `inspect` angegeben, so werden zusätzlich alle Teilobjekte der in `inspect` angegebenen Domaintypen rekursiv untersucht (siehe Beispiel ??).

☞ `hastype` findet nur Teilobjekte, die syntaktisch vom Typ `T` ist. Eigenschaften von Bezeichnern, die durch `assume` definiert wurden, werden nicht berücksichtigt (siehe Beispiel ??).



Beispiel 1. In diesem Beispiel wird zuerst getestet, ob ein gegebener Ausdruck einen Teilausdruck vom Typ `DOM_FLOAT` besitzt:

```
>> hastype(1.0 + x, DOM_FLOAT)
```

TRUE

```
>> hastype(1 + x, DOM_FLOAT)
```

FALSE

Man kann ebenfalls testen ob ein Ausdruck einen Teilausdruck vom Typ `DOM_FLOAT` oder vom Typ `DOM_INT` enthält:

```
>> hastype(1.0 + x, {DOM_FLOAT, DOM_INT})
```

TRUE

Während der erste der folgenden beiden Tests `FALSE` ergibt, weil `tan` kein gültiger Typspezifizierer ist, liefert der zweite Test `TRUE`, denn es gibt einen Teilausdruck vom Typ `"tan"`:

```
>> hastype(sin(tan(x) + 1/exp(1 - x)), tan),  
      hastype(sin(tan(x) + 1/exp(1 - x)), "tan")
```

FALSE, TRUE

Es können ebenfalls Typspezifizierer der Type-Bibliothek benutzt werden:

```
>> hastype([-1, 10, -5, 2*I], Type::PosInt)
```

TRUE

Beispiel 2. Das Beispiel zeigt die Benutzung des optionalen dritten Arguments. Es soll überprüft werden, ob eine Prozedur einen Teilausdruck vom Typ `"float"` enthält. Normalerweise analysiert `hastype` Prozeduren nicht näher:

```
>> f := x -> float(x) + 3.0:  
      hastype(f, "float")
```

FALSE

Mit dem dritten Argument wird das Untersuchen von Prozeduren ausdrücklich gefordert:

```
>> hastype(f, "float", {DOM_PROC})
```

```
TRUE
```

Gleichermaßen werden normalerweise Objekte der Basisdomains DOM_COMPLEX und DOM_RAT nicht zerlegt:

```
>> hastype(1 + I, DOM_INT), hastype(2/3, DOM_INT)
```

```
FALSE, FALSE
```

Um diese Datentypen zu untersuchen, muss ein drittes Argument angegeben werden:

```
>> hastype(1 + I, DOM_INT, {DOM_COMPLEX}),  
    hastype(2/3, DOM_INT, {DOM_RAT})
```

```
TRUE, TRUE
```

Durch Angabe des dritten Arguments können auch Elemente beliebiger Domains analysiert werden. Als Beispiel wird eine Matrix definiert und überprüft, ob ein Eintrag vom Typ DOM_INT vorhanden ist:

```
>> A:=matrix([[1, 1], [1, 0]]):  
    hastype(A, DOM_INT), hastype(A, DOM_INT, {Dom::Matrix()})
```

```
FALSE, TRUE
```

Beispiel 3. Im nächsten Beispiel arbeitet hastype mit Behältern. Als erstes werden Tabellen untersucht:

```
>> hastype(table(1 = a), DOM_INT), hastype(table(a = 1), DOM_INT)
```

```
FALSE, TRUE
```

Wie zu sehen ist, untersucht hastype nicht die Indizes von Tabellen, aber alle Einträge einer Tabelle. Dasselbe gilt für Felder, Listen und Mengen:

```
>> hastype(array(1..4, [1, 2, 3, 4]), DOM_INT),  
    hastype([1, 2, 3, 4], DOM_INT),  
    hastype({1, 2, 3, 4}, DOM_INT),  
    hastype([a, [1]], b, c], DOM_INT)
```

```
TRUE, TRUE, TRUE, TRUE
```

hastype arbeitet nur syntaktisch, d.h. Eigenschaften von Bezeichnern werden nicht berücksichtigt:

```
>> assume(a, Type::Integer):  
    hastype([a, b], Type::Integer), hastype([a, b], DOM_INT)
```

```
FALSE, FALSE
```

```
>> delete a:
```

Änderungen:

- ⌘ Das zweite Argument kann auch eine Liste oder eine Menge sein.
 - ⌘ Das dritte optionale Argument ist eingeführt worden.
-

heaviside – die heavisidesche Sprungfunktion

`heaviside(x)` stellt die heavisidesche Sprungfunktion dar.

Aufruf(e):

- ⌘ `heaviside(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Verwandte Funktionen: `dirac`

Details:

- ⌘ Für positive reelle Argumente wird 1, für negative reelle Argumente wird 0 zurückgeliefert. Für komplexe Werte vom Domain-Typ `DOM_COMPLEX` wird `undefined` zurückgeliefert. Alle anderen Argumente ergeben einen unevaluierten Funktionsaufruf.
 - ⌘ `heaviside` hat keinen vordefinierten Wert am Ursprung. Mit
`sysassign(heaviside(0), meinWert)`
und
`sysassign(heaviside(float(0)), meinFloatWert)`
können gewünschte Werte gesetzt werden.
 - ⌘ Die Ableitung von `heaviside` ist die Delta-Distribution `dirac`.
-

Beispiel 1. `heaviside` liefert 1 bzw. 0 für Argumente, die positive bzw. negative reelle Zahlen darstellen:

```
>> heaviside(-3), heaviside(-sqrt(3)), heaviside(-2.1),  
    heaviside(PI - E), heaviside(sqrt(3))  
  
0, 0, 0, 1, 1
```

Argumente vom Domain-Typ `DOM_COMPLEX` liefern `undefined`:

```
>> heaviside(1 + I), heaviside(2/3 + 7*I), heaviside(0.1*I)  
  
undefined, undefined, undefined
```

Für andere Argumente wird ein unevaluierter Funktionsaufruf zurückgeliefert:

```
>> heaviside(0), heaviside(x), heaviside(ln(-5)), heaviside(x + I)  
  
heaviside(0), heaviside(x), heaviside(I PI + ln(5)),  
  
heaviside(x + I)
```

Natürliche Funktionswerte am Ursprung sind 0, 1/2 oder 1:

```
>> prev_protection:= unprotect(heaviside):  
    heaviside(0) := 1/2: heaviside(0)  
  
1/2  
  
>> delete heaviside(0):  
    protect(heaviside, prev_protection):  
    delete prev_protection:  
  
>> heaviside(0)  
  
heaviside(0)
```

Beispiel 2. `heaviside` berücksichtigt durch `assume` gesetzte Annahmen:

```
>> assume(x > 0): heaviside(x)  
  
1  
  
>> unassume(x):
```

Beispiel 3. Die Ableitung von heaviside ist die Delta-Distribution dirac:

```
>> diff(heaviside(x - 4), x)

dirac(x - 4)
```

Der Integrierer int verarbeitet heaviside:

```
>> int(exp(-x)*heaviside(x), x = -infinity..infinity)

1
```

Es ist nicht empfehlenswert, heaviside bei numerischer Integration zu verwenden. Es ist wesentlich effektiver, die Integration in mehrere Bereiche aufzuspalten, in denen der Integrand jeweils stetig ist:

```
>> DIGITS := 3: numeric::int(exp(-x)*heaviside(x^2 - 2), x=-
3..10)

16.2

>> numeric::int(exp(-x), x = -3..-2^(1/2)) +
numeric::int(exp(-x), x = 2^(1/2)..10)

16.2

>> delete DIGITS:
```

Änderungen:

- ⌘ Durch assume gesetzte Eigenschaften von Bezeichnern werden nun berücksichtigt.
-

help – Anzeigen von Hilfe-Texten

help("word") oder ?word zeigt die Online-Hilfe zu word an.

Aufruf(e):

- ⌘ help("word")
- ⌘ ?word

Parameter:

word — beliebiges Schlüsselwort

Rückgabewert: das leere Objekt `null()` vom Typ `DOM_NULL`.

Verwandte Funktionen: `info`, `Pref::ansi`

Details:

- ☞ `help("word")` zeigt eine Hilfe-Seite mit Informationen zu dem Schlüsselwort "word" an.
 - ☞ Die genaue Form der Ausgabe ist plattformabhängig. In der Terminal-Version wird ein ASCII-Text ausgegeben. In XMuPAD, unter MacOS oder unter Windows stehen die Informationen der Hypertext-Dokumentation in einem separaten Fenster zur Verfügung. Markierte Wörter sind Hypertext-Links, denen man durch Anklicken mit der Maus folgen kann.
 - ☞ Abkürzend für `help("keyword")` kann `?keyword` verwendet werden. `?` ist jedoch keine MuPAD-Funktion und kann somit nicht in Ausdrücken verwendet werden, sondern nur interaktiv allein in einer Zeile. `keyword` wird dabei weder in Anführungsstriche eingeschlossen noch durch Semikolon abgeschlossen.
 - ☞ Das Schlüsselwort kann die Platzhalterzeichen („wildcards“) `?` und `*` enthalten. `?` repräsentiert dabei ein beliebiges Zeichen, `*` eine beliebige, möglicherweise auch leere Zeichenkette. Es gibt 3 Ausnahmen: `?*` und `?`*`` führen direkt zur Hilfeseite von `_mult` und `?`**`` führt direkt zur Hilfeseite von `_power`.
 - ☞ Falls keine Hilfeseite für das angegebene Schlüsselwort existiert, so wird eine Liste von Schlüsselwörtern mit ähnlicher Schreibweise ausgegeben. Siehe Beispiel ??.
 - ☞ Der Aufruf `anames(All)` gibt einen Überblick über alle gegenwärtig geladenen Systemfunktionen. Der Aufruf `?**` listet alle verfügbaren Hilfeseiten auf.
 - ☞ Die Dokumentation zu Modulen ist nicht in dieses Hilfesystem integriert. Beispiel ?? erläutert, wie man auf die Dokumentation von Modul-Funktionen zugreifen kann.
 - ☞ `help` ist eine Funktion des Systemkerns.
-

Beispiel 1. `help` expandiert Platzhalterzeichen („wildcards“):

```
>> ?*type
```

```
Try: domtype hastype testtype type Type Type::AnyType
```

Eine Ausnahme: `?*` führt direkt zur Hilfeseite von `_mult`:


```
>> ?*
* -- multiply expressions

Introduction

a * b respectively _mult(a, b) computes the product a*b.

Call(s)

o a * b _mult( <a, b...>)

Parameters

a, b - arithmetical expressions

[...]
```

Beispiel 2. MuPAD kennt keine Funktion `worm` und hat daher auch keine Dokumentation dazu:

```
>> ?worm

Sorry, no help page available for 'worm' !

Try: norm
```

Beispiel 3. MuPAD unterstützt kompilierte C++-Erweiterungen des Kerns, so genannte dynamische Module. Die Dokumentation zu einem dynamischen Modul ist nicht in das Hypertext-System der Dokumentation integriert, sondern liegt als ASCII-Text vor, der mit Hilfe des Slots "doc" des jeweiligen Moduls angezeigt werden kann, so wie `util::doc` im folgenden Beispiel:

```
>> module(util): util::doc()

MODULE
  util - A collection of utility functions

INTRODUCTION
  The module provides a collection of useful utility functions.

INTERFACE
  util::busyWaiting, util::date,      util::doc,
  util::kernelPath,  util::kernelPid, util::sleep,
  util::time,        util::userName
```

```
>> util::doc("kernelPath")

NAME
    util::kernelPath - Returns the pathname of the MuPAD kernel

SYNOPSIS
    util::kernelPath()

DESCRIPTION
    This function returns the pathname of the MuPAD kernel.

EXAMPLES
    >> util::kernelPath()

        "C:\\\\PROGRA~1\\\\SCIFACE\\\\MUPADP~1.5\\\\BIN\\\\MUPKERN.EXE"

    >> util::kernelPath()

        "/usr/local/mupad/linux/bin/mupad"

SEE ALSO
    util::kernelPid, util::userName
```

Hintergründe:

- ☞ In der Terminal-Version wird das in der System-Variable `PAGER` vor-
eingestellte Programm zum Anzeigen einer ASCII-Hilfeseite verwendet.
Mit Hilfe von `Pref::ansi` lässt sich dieses Ausgabeformat beeinflus-
sen.

Änderungen:

- ☞ `?*` listet nicht mehr alle verfügbaren Hilfeseiten auf, sondern führt statt
dessen auf die Hilfeseite von `_mult`.

history – Zugriff auf einen Eintrag der History-Tabelle

`history(n)` gibt den n-ten Eintrag der History-Tabelle zurück.

`history()` gibt den Index des neuesten Eintrags in der History-Tabelle zu-
rück.

Aufruf(e):

- ☞ `history(n)`

⇒ `history()`

Parameter:

`n` — eine positive ganze Zahl

Verwandte Funktionen: `fread`, `HISTORY`, `last`, `read`

Details:

- ⇒ Die Befehle, die interaktiv während einer MuPAD-Sitzung eingegeben werden oder innerhalb einer Prozedur oder einer Datei ausgeführt werden, sowie die dazugehörigen Ergebnisse werden in einer internen Datenstruktur, der „History“-Tabelle, gespeichert. `history()` gibt den Index des neuesten Eintrags der History-Tabelle zurück. Auf interaktiver Ebene ist das die Anzahl aller Befehle, die seit dem Start der Sitzung oder dem letzten Neustart eingegeben wurden.
- ⇒ `history(n)` gibt den `n`-ten Eintrag der History-Tabelle als Liste mit zwei Elementen zurück. Das erste Element ist der eingegebene MuPAD-Befehl, das zweite Element das von MuPAD zurückgegebene Ergebnis dieses Befehls. Die Reihenfolge der Einträge in der History-Tabelle ist so, dass neuere Einträge einen größeren Index haben als ältere.
- ⇒ Der Befehl `last` erlaubt den direkten Zugriff auf die Ergebnisse der History-Tabelle. Der Aufruf `last(n)` ist auf interaktiver Ebene gleichbedeutend mit dem Aufruf `history(history() - n + 1)[2]`.
- ⇒ Die Umgebungsvariable `HISTORY` bestimmt die maximale Anzahl der Einträge der History-Tabelle auf interaktiver Ebene, die gespeichert werden. Der Standardwert von `HISTORY` ist 20. MuPAD behält nur die jeweils neuesten Einträge im Speicher. `history` kann mit ganzen Zahlen von `history() - HISTORY + 1` bis `history()` aufgerufen werden. Alle anderen Zahlen führen zu einer Fehlermeldung.
- ⇒ Die Rückgabewerte von `history` werden nicht noch einmal evaluiert (siehe Beispiel ??). Mit der Funktion `eval` kann eine erneute Evaluierung erzwungen werden.
- ⇒ Befehle und die zugehörigen Ergebnisse werden auch in der History-Tabelle gespeichert, wenn die Ausgabe des Ergebnisses durch `:` verhindert wird. Siehe Beispiel ??.
- ⇒ Zusammengesetzte Befehle wie `for`-, `repeat`- und `while`-Schleifen, `if`- und `case`-Verzweigungen sowie Prozedur-Definitionen mit `proc` werden in der History-Tabelle vollständig gespeichert. Die Hilfeseite von `last` enthält dazu Beispiele.

- ☞ Mehrere Befehle in einer Eingabezeile, die durch Semikolon oder Doppelpunkt getrennt sind, führen zu mehreren Einträgen in der History-Tabelle. Eine Anweisungsfolge wird aber als *ein* Befehl in die History-Tabelle eingetragen (siehe Beispiel ??).
- ☞ Befehle, die mit `fread` oder `read` aus einer Datei gelesen werden, werden alle in der History-Tabelle gespeichert, und erst danach wird der `fread`- oder `read`-Befehl in der History-Tabelle gespeichert (da der `fread`- oder `read`-Befehl erst nach dem Lesen beendet wird). Wird beim Einlesen aber die Option `Plain` angegeben, wird während des Einlesens eine neue History-Tabelle benutzt, die nach dem Einlesen wieder gelöscht wird. Die eingelesenen Befehle werden dadurch nicht in der aktuellen History-Tabelle gespeichert.
- ☞ Zu beachten ist, dass jeder Aufruf von `history` die History-Tabelle verändert und im allgemeinen den frühesten Eintrag löscht.
- ☞ Jede Prozedur hat eine eigene lokale History-Tabelle. Auf diese Tabelle kann aber nicht mit `history` zugegriffen werden (siehe `last`). Der Befehl `history` greift immer auf die History-Tabelle der interaktiven Ebene zu.
- ☞ `history` ist eine Funktion des Systemkerns.

Beispiel 1. Der Index des letzten Eintrags der History-Tabelle wird durch *jeden* Befehl erhöht, auch durch die Eingabe von `history()`. Zu beachten ist, dass auch Befehle mit den dazugehörigen Ergebnissen in die History-Tabelle eingetragen werden, deren Ausgabe durch `:` unterdrückt wird:

```
>> history(); sqrt(1764); history(): history()
```

3

42

6

`history(history())` gibt eine Liste mit zwei Elementen zurück. Das erste Element ist der letzte Befehl, das zweite Element ist das letzte von MuPAD berechnete Ergebnis, auf das auch mit `last(1)` oder `%` zugegriffen werden kann:

```
>> int(2*x*exp(x^2), x);
    history(history()), last(1)
```

2

$\exp(x^2)$

$$[\text{int}(2 \cdot x \cdot \exp(x^2), x), \exp(x^2)], \exp(x^2)$$

Der folgende Befehl gibt die vorletzte Eingabe und deren Ergebnis zurück:

```
>> history(history() - 1)

                2                2
      [int(2 x exp(x ), x), exp(x )]
```

Ein Neustart löscht die History-Tabelle:

```
>> reset():
      history()

                4
```

Die Ausgabe des Befehls `history()` im letzten Beispiel hängt von den Befehlen ab, die in der Datei `userinit.mu` gespeichert sind.

Beispiel 2. Zuerst soll `a` 0 sein:

```
>> a := 0:
      a

                0
```

Nun wird 1 an `a` zugewiesen:

```
>> a := 1:
      a

                1
```

Der Befehl `history(history()-2)` greift auf den Befehl `a` nach der Zuweisung von 0 und `a` zu. Der Rückgabewert von `history` ist nicht der neue Wert von `a`, da das Ergebnis von `history` nicht noch einmal evaluiert wird:

```
>> history(history() - 2)

      [a, 0]
```

Beispiel 3. Die beiden Befehle führen zu zwei Einträgen in der History-Tabelle. Der Befehl `history(history()-1)` gibt nur den letzten Befehl `b:=a` zurück, nicht beide Befehle:

```
>> a := 0: b := a:
      history(history() - 1)

      [(a := 0), 0]
```

Wenn die Befehle als Befehlsfolge geschrieben werden (eingeschlossen in ()), führt das zu einem Eintrag. Der Befehl `history(history())` gibt den letzten Befehl zurück, das ist die vollständige Ausdrucksreihe:

```
>> (a := 0; b := a):  
    history(history())  
  
[(a := 0;  
 b := a), 0]
```

Die letzte Eingabe ist eine Befehlsfolge:

```
>> type(op(%, 1))  
  
"_stmtseq"
```

Änderungen:

- ⌘ Das Argument kann leer oder eine nichtnegative ganze Zahl sein.
 - ⌘ `history` gibt den aktuellen Index der History-Tabelle oder eine Liste mit Ein- und Ausgabe zurück.
-

hold – verzögerte Evaluierung

`hold(object)` verhindert die Evaluierung von `object`.

Aufruf(e):

- ⌘ `hold(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: das unevaluierte Objekt.

Weitere Dokumentation: Kapitel 5 des Tutoriums.

Verwandte Funktionen: `context`, `delete`, `eval`, `freeze`, `genident`, `indexval`, `level`, `proc`, `val`

Details:

☞ Wenn ein MuPAD-Objekt interaktiv eingegeben wird, evaluiert MuPAD dieses Objekt und gibt das Ergebnis zurück. Beim Aufruf einer Prozedur werden die übergebenen Argumente normalerweise evaluiert, bevor sie verarbeitet werden. *Evaluierung* bedeutet, dass Bezeichner durch ihre Werte ersetzt werden und Funktionsaufrufe ausgeführt werden. `hold` verhindert diese Evaluierung.

☞ Eine typische Anwendung von `hold` ist, wenn eine Funktion gezeichnet oder an einen numerischen Algorithmus übergeben werden soll, die nur mit numerischen Argumenten aufgerufen werden darf, jedoch nicht mit symbolischen Argumenten. Siehe Beispiel ??.

☞ Ein weiterer Grund für die Anwendung von `hold` ist Effizienz. Beispielsweise kann ein Funktionsaufruf `f(x, y)` als Argument einer anderen Funktion auftreten, und es ist bekannt, dass das Ergebnis dieses Aufrufes wieder `f(x, y)` ist.

Um die möglicherweise zeitaufwendige Auswertung der „inneren“ Funktion `f` zu verhindern, übergibt man in diesem Fall den Ausdruck `hold(f)(x, y)` an die „äußere“ Funktion. Die Argumente `x, y` werden dann evaluiert, aber die Funktion `f` wird nicht ausgeführt. Siehe Beispiele ?? und ??.

☞ Die Benutzung von `hold` kann eigenartige Effekte zur Folge haben, daher wird dies nur bei absoluter Notwendigkeit empfohlen.

☞ `hold` verzögert die Auswertung eines Objektes nur, kann sie aber nicht dauerhaft verhindern (siehe Beispiel ??).

Man kann `freeze` verwenden, um die Auswertung einer Prozedur oder Funktionsumgebung vollständig zu unterbinden.

☞ Eine MuPAD-Prozedur kann mit der Option `hold` deklariert werden. Dann werden die Argumente beim Aufruf der Funktion nicht evaluiert. Die Hilfeseite von `proc` enthält eine ausführlichere Beschreibung.

☞ Mit den Funktionen `eval` oder `level` kann eine nachträgliche Evaluierung erzwungen werden (siehe Beispiel ??). In Prozeduren mit der Option `hold` verwendet man besser `context`.

☞ `hold` ist eine Funktion des Systemkerns.

Beispiel 1. In den folgenden beiden Beispielen wird die Evaluierung der eingegebenen MuPAD-Ausdrücke verhindert, indem `hold` benutzt wird:

```
>> x := 2:
    hold(3*0 - 1 + 2^2 + x)
```

```


$$3 \cdot 0 - 1 + 2^2 + x$$

>> hold(error("not really an error"))
error("not really an error")

```

Ohne hold würden die Ergebnisse wie folgt aussehen:

```

>> x := 2:

$$3 \cdot 0 - 1 + 2^2 + x$$

5
>> error("not really an error")

```

Error: not really an error

Der folgende Befehl verhindert die Evaluierung (und Ausführung) der Operation `_plus`, aber nicht die Evaluierung seiner Operanden:

```

>> hold(_plus)(3*0, -1, 2^2, x)

$$0 - 1 + 4 + 2$$


```

In solchen Anwendungen wie im letzten Beispiel ist darauf zu achten, dass die Argumente immer evaluiert werden, selbst wenn die Funktion mit der Option `hold` definiert wird:

```

>> f := proc(a)
    option hold;
    begin
        return(a + 1)
    end_proc:
x := 2:
hold(f)(x)
f(2)

```

Dies passiert aus folgendem Grund. Wenn `f` ausgewertet wird, verhindert die Option `hold` die Evaluierung des Arguments `x` von `f` (siehe nächstes Beispiel). Im letzten Beispiel wird jedoch die Auswertung von `f` durch `hold` verhindert, so dass die Option `hold` keine Auswirkung hat, und MuPAD evaluiert die Argumente, aber nicht den Funktionsaufruf.

Das folgende Beispiel zeigt das erwartete Verhalten:

```

>> f(x), hold(f(x))

$$x + 1, f(x)$$


```

Die Funktion `eval` hebt die Wirkung von `hold` auf. Die Ergebnisse sind abhängig von der Art der Anwendung von `hold`:

```

>> eval(f(x)), eval(hold(f)(x)), eval(hold(f(x))), eval(hold(f))(x)
3, 3, x + 1, x + 1

```


Beispiel 2. Aufrufe von `hold` können geschachtelt werden, um nachfolgende Evaluierungen zu verhindern:

```
>> x := 2:
      hold(x), hold(hold(x))

                        x, hold(x)
```

Das Ergebnis von `hold(hold(x))` ist der unevaluierte Operand des äußersten Aufrufs von `hold`, nämlich `hold(x)`. Anwenden von `eval` evaluiert das Ergebnis `hold(x)`, das ergibt den unevaluierten Bezeichner `x`:

```
>> eval(%)

                        2, x
```

Eine weitere Anwendung von `eval` ergibt den Wert, der `x` zugewiesen wurde:

```
>> eval(%)

                        2, 2
```

```
>> delete x, f:
```

Beispiel 3. Im folgenden Beispiel wird die Evaluierung der Operation `_plus` verhindert, `_plus` wird durch `_mult` ersetzt, und anschließend wird der Ausdruck evaluiert:

```
>> eval(subsop(hold(_plus)(2, 3), 0 = _mult))

                        6
```

Beispiel 4. Die Funktion `domtype` evaluiert ihre Argumente:

```
>> x := 0:
      domtype(x), domtype(sin), domtype(x + 2)

      DOM_INT, DOM_FUNC_ENV, DOM_INT
```

Durch die Benutzung von `hold` kann der Typ der unevaluierten Objekte festgestellt werden: `x` und `sin` sind Bezeichner, `x + 2` ist ein Ausdruck:

```
>> domtype(hold(x)), domtype(hold(sin)), domtype(hold(x + 2))

      DOM_IDENT, DOM_IDENT, DOM_EXPR
```

Beispiel 5. `hold` verzögert lediglich *einmal* die Evaluierung eines Objektes, aber es verhindert nicht eine nachfolgende Evaluierung. Daher ist die Verwendung von `hold`, um ein Symbol ohne Wert zu bekommen, keine gute Idee:

```
>> x := 2:
      y := hold(x);
      y

      x

      2
```

Durch Löschen des Wertes von `x` wird aus `x` ein Symbol, und die Anwendung von `hold` ist nicht notwendig:

```
>> delete x:
      y := x;
      y

      x

      x
```

Ein unbenutztes Symbol garantiert ohne Wert liefert am besten die Funktion `genident`:

```
>> y := genident("x");
      y

      x1

      x1

>> delete y:
```

Beispiel 6. Es soll der Graph der stückweise stetigen Funktion $f(x)$ gezeichnet werden, die für negative reelle Argumente identisch Null ist und für positive reelle Argument gleich $\exp(-x)$:

```
>> f := x -> if x < 0 then 0 else exp(-x) end_if:
```

Wenn der symbolische Ausdruck $f(x)$ an `plotfunc2d` übergeben wird, ergibt das einen Fehler:

```
>> delete x:
      plotfunc(f(x), x = -2..2)

Error: Can't evaluate to boolean [_less];
during evaluation of 'f'
```

Der Grund ist, dass `plotfunc2d` seine Argumente evaluiert und die Funktion `f` nicht mit symbolischen Argumenten aufgerufen werden darf:

```
>> f(x)

Error: Can't evaluate to boolean [_less];
during evaluation of 'f'
```

Eine mögliche Lösung dieses Problems ist die Anwendung von `hold`:

```
>> plotfunc2d(hold(f)(x), x = -2..2):
```

Das selbe Problem tritt auf, wenn `f` numerisch integriert werden soll:

```
>> numeric::int(f(x), x = -2..2)

Error: Can't evaluate to boolean [_less];
during evaluation of 'f'

>> numeric::int(hold(f)(x), x = -2..2)

0.8646647168
```

Beispiel 7. Die Funktion `int` kann für das folgende Integral keine geschlossene Form berechnen und gibt einen symbolischen Funktionsaufruf zurück:

```
>> int(sin(x)*sqrt(sin(x) + 1), x)

int(sin(x) (sin(x) + 1)1/2, x)
```

Nach der Variablentransformation $\sin(x)=t$ kann eine geschlossene Form berechnet werden:

```
>> t := time():
f := intlib::changevar(int(sin(x)*sqrt(sin(x) + 1), x), sin(x) = y);
time() - t;
eval(f)
```

$$\text{int} \left(\frac{y (y + 1)^{1/2}}{(1 - y)^{2 1/2}}, y \right)$$

$$\frac{(y - 1) (y + 1)^{1/2} \sqrt[3]{2 y} + 4/3}{(1 - y)^{2 1/2}}$$

Durch Messung der Laufzeit mit `time` ist zu sehen: Die meiste Zeit des Aufrufs von `intlib::changevar` wird für die erneute Evaluierung des Arguments verbraucht. Dies kann durch `hold` verhindert werden:

```
>> t := time():
    f := intlib::changevar(hold(int)(sin(x)*sqrt(sin(x) + 1), x),
                           sin(x) = y);
    time() - t;
```

$$\text{int} \left| \frac{y (y + 1)^{1/2}}{(1 - y)^{2 1/2}}, y \right|$$

20

Änderungen:

☞ Keine Änderungen.

`icontent` – der Inhalt eines Polynoms mit rationalen Koeffizienten

`icontent(p)` berechnet den Inhalt des Polynoms `p` mit ganzzahligen oder rationalen Koeffizienten, also den größten gemeinsamen Teiler seiner Koeffizienten.

Aufruf(e):

☞ `icontent(p)`

Parameter:

`p` — ein Polynom oder polynomialer Ausdruck mit ganzzahligen oder rationalen Koeffizienten

Rückgabewert: eine nicht-negative ganze oder rationale Zahl, oder `FAIL`

Verwandte Funktionen: `coeff`, `content`, `factor`, `gcd`, `ifactor`, `igcd`, `ilcm`, `lcm`, `poly`, `polylib::primpart`

Details:

- ⌘ `icontent` berechnet den Inhalt eines Polynoms oder polynomialen Ausdrucks mit ganzzahligen oder rationalen Koeffizienten, d. h. den größten gemeinsamen Teiler der Koeffizienten, so dass $p/\text{icontent}(p)$ ganzzahlige Koeffizienten hat, deren größter gemeinsamer Teiler 1 ist. Wenn x selbst eine ganze oder gebrochene Zahl ist, gibt `icontent(p)` den Ausdruck `abs(p)` zurück (siehe Beispiel ??).
- ⌘ Ein Polynom p mit ganzzahligen Koeffizienten hat als Inhalt den größten gemeinsamen Teiler der Koeffizienten. Ein Polynom p mit rationalen Koeffizienten hat als Inhalt den größten gemeinsamen Teiler der Koeffizienten dividiert durch das kleinste gemeinsame Vielfache der Nenner.
- ⌘ Wenn das erste Argument p von `icontent` ein polynomialer Ausdruck ist, wird p mit der Funktion `poly` in ein Polynom umgewandelt. `icontent` gibt `FAIL` zurück, wenn der Ausdruck nicht in ein Polynom umgeformt werden kann.
- ⌘ `icontent` liefert eine Fehlermeldung, falls das Polynom Koeffizienten hat, die weder ganzzahlig noch rational sind.
- ⌘ `icontent` ist eine Funktion des Systemkerns.

Beispiel 1. Das erste Argument kann ein Polynom oder ein polynomialer Ausdruck sein. Die folgenden beiden Aufrufe sind gleichbedeutend:

```
>> p := 6*x*y - 9*y^2 + 21:
      icontent(poly(p)), icontent(p)
```

3, 3

Das Ergebnis von `icontent` ist immer nichtnegativ:

```
>> icontent(2*x - 4), icontent(-2*x + 4)
```

2, 2

Der Inhalt eines konstanten Polynoms ist sein absoluter Betrag:

```
>> icontent(0), icontent(-2), icontent(poly(-2, [x]))
```

0, 2, 2

Beispiel 2. Der Inhalt eines Polynoms mit gebrochenen Koeffizienten ist im allgemeinen eine gebrochene Zahl:

```
>> q := 6/7*x*y - 9/4*y + 12:
      icontent(poly(q)), icontent(q)

3/28, 3/28
```

Ein Polynom dividiert durch seinen Inhalt hat ganzzahlige Koeffizienten, deren größter gemeinsamer Teiler 1 ist:

```
>> q/icontent(q)

8 x y - 21 y + 112

>> icontent(%)

1
```

Änderungen:

⌘ Keine Änderungen.

if – Verzweigungsanweisung

if-then-else-end_if erlaubt bedingte Verzweigungen innerhalb eines Programms.

Aufruf(e):

```
⌘ if condition1
    then casetrue1
    <elif condition2 then casetrue2>
    <elif condition3 then casetrue3>
    <...>
    <else casefalse>
    end_if
⌘ _if(condition1, casetrue1, casefalse)
```

Parameter:

condition1, condition2, ...	— Boolesche Ausdrücke
casetrue1, casetrue2, ..., casefalse	— beliebige Anweisungsfolgen

Rückgabewert: das Ergebnis des letzten innerhalb der `if`-Anweisung ausgeführten Kommandos. Das Ergebnis ist `NIL`, wenn kein Kommando ausgeführt wird.

Weitere Dokumentation: Kapitel 17 des MuPAD-Tutoriums.

Verwandte Funktionen: `case`, `piecewise`

Details:

- ☞ Falls der boolsche Ausdruck `condition1` zu `TRUE` ausgewertet werden kann, wird die Verzweigung `casetrue1` ausgeführt und ihr Ergebnis zurückgegeben. Anderenfalls, falls `condition2` zu `TRUE` ausgewertet werden kann, wird die Verzweigung `casetrue2` ausgeführt und ihr Ergebnis zurückgegeben. Usw. Falls alle Bedingungen zu `FALSE` evaluieren, wird die Verzweigung `casefalse` ausgeführt und ihr Ergebnis zurückgegeben.
 - ☞ Alle boolschen Bedingungen, die während der Ausführung einer `if`-Anweisung auszuwerten sind, müssen zu `TRUE` oder `FALSE` reduzierbar sein. Bedingungen können z. B. durch Gleichungen, Ungleichungen oder Vergleiche angegeben werden, die mit den logischen Operatoren `and`, `or` und `not` verknüpft werden können. Die boolsche Auswertung von Gleichungen etc. braucht dabei nicht explizit durch `bool` erzwungen zu werden. Die `if`-Anweisung erzwingt die boolsche Auswertung automatisch mittels „lazy evaluation“ durch die Funktionen `_lazy_and` bzw. `_lazy_or`. Eine Bedingung, die durch diese Funktionen nicht zu `TRUE` oder `FALSE` evaluierbar ist, führt zu einem Laufzeitfehler. Siehe Beispiel ??.
 - ☞ Statt des Schlüsselworts `end_if` kann auch das Schlüsselwort `end` benutzt werden.
 - ☞ Die Anweisung `if condition then casetrue else casefalse end_if` ist äquivalent zum Funktionsaufruf `_if(condition, casetrue, casefalse)`.
 - ☞ `_if` ist eine Funktion des Systemkerns.
-

Beispiel 1. Die `if`-Anweisung funktioniert wie folgt:

```
>> if TRUE then YES else NO end_if,  
    if FALSE then YES else NO end_if  
  
      YES, NO
```

Der `else`-Zweig ist optional:

```

>> if FALSE then YES end_if

NIL

>> if FALSE
    then if TRUE
        then NO_YES
        else NO_NO
    end_if
    else if FALSE
        then YES_NO
        else YES_YES
    end_if
end_if

YES_YES

```

Typischerweise werden die boolschen Bedingungen durch Gleichungen, Ungleichungen, Vergleiche oder boolsche Konstanten gegeben, die von Systemfunktionen wie z. B. `isprime` geliefert werden:

```

>> for i from 100 to 600 do
    if 105 < i and i^2 <= 17000 and isprime(i) then
        print(expr2text(i)." is a prime")
    end_if;
    if i < 128 then
        if isprime(2^i - 1) then
            print("2^".expr2text(i)." - 1 is a prime")
        end_if
    end_if
end_for:

"107 is a prime"

"2^107 - 1 is a prime"

"109 is a prime"

"113 is a prime"

"127 is a prime"

"2^127 - 1 is a prime"

```

Beispiel 2. Anstelle von geschachtelten `if`-Anweisungen kann die `elif`-Anweisung die Lesbarkeit des geschriebenen Quellcodes erhöhen. Intern werden solche Anweisungen allerdings vom Parser in äquivalente `if-then-else`-Anweisungen konvertiert:


```
>> hold(if FALSE then NO elif TRUE then YES_YES else YES_NO end_if)

      if FALSE then
        NO
      else
        if TRUE then
          YES_YES
        else
          YES_NO
        end_if
      end_if
```

Beispiel 3. Kann die Bedingung nicht zu einem der Werte TRUE oder FALSE ausgewertet werden, so wird ein Laufzeitfehler ausgelöst. Im folgenden Aufruf liefert `is(x > 0)` den Wert UNKNOWN, falls `x` keine entsprechende Eigenschaft mittels `assume` angeheftet wurde:

```
>> if is(x > 0) then
    1
  else
    2
  end_if
```

Error: Can't evaluate to boolean [if]

Vergleiche mittels `<`, `<=`, `>`, `>=` dürfen keine symbolischen Ausdrücke enthalten:

```
>> if 1 < sqrt(2) then print("1 < sqrt(2)"); end_if
```

Error: Can't evaluate to boolean [_less]

```
>> if 1 < float(sqrt(2)) then print("1 < float(sqrt(2))"); end_if

      "1 < float(sqrt(2))"
```

```
>> if PI < 3.1416 then print("PI < 3.1416"); end_if
```

Error: Can't evaluate to boolean [_less]

Beispiel 4. Dieses Beispiel demonstriert den Zusammenhang zwischen dem funktionalen und dem imperativen Einsatz der `if`-Anweisung:

```
>> condition := 1 > 0: _if(condition, casetrue, casefalse)

      casetrue
```

```
>> condition := 1 > 2: _if(condition, casetrue, casefalse)

                                casefalse

>> delete condition:
```

Änderungen:

☞ end kann nun als Alternative zu end_if benutzt werden.

id – die identische Abbildung

id(x) gibt x ausgewertet zurück.

Aufruf(e):

☞ id(x)
☞ id(x1, x2, ...)

Parameter:

x, x1, x2, ... — beliebige MuPAD-Objekte

Rückgabewert: die Folge der Eingabeparameter.

Details:

☞ id(x) liefert x ausgewertet zurück; id(x1, x2, ...) liefert die ausgewerteten Argumente als eine Folge zurück; id() liefert das leere Objekt null().
☞ id ist eine Funktion des Systemkerns.

Beispiel 1. id liefert die evaluierten Argumente zurück:

```
>> a := 2: id(a + 2)

                                4

>> id(a, b, 4 + 2)

                                2, b, 6

id( ) gibt null( ) zurück:
```

```
>> domtype(id())
```

DOM_NULL

```
>> delete a:
```

Beispiel 2. `id` ist beim Umgang mit funktionalen Ausdrücken nützlich:

```
>> f := 3*id + sin + 5*id^2 + exp@(-id^2): f(x)
```

$$3x + \sin(x) + 5x^2 + \exp(-x^2)$$

```
>> D(f)
```

$$10x + \cos - 2x \exp(-x^2) + 3$$

```
>> delete f:
```

Änderungen:

☞ Keine Änderungen.

`ifactor` – Zerlegung ganzer Zahlen in Primfaktoren

`ifactor(n)` berechnet die Primzahlzerlegung $n = s \cdot p_1^{e_1} \cdots p_r^{e_r}$ der ganzen Zahl n , dabei sind s das Vorzeichen von n , p_1, \dots, p_r verschiedene Primteiler von n und e_1, \dots, e_r positive ganze Zahlen.

Aufruf(e):

☞ `ifactor(n <, UsePrimeTab>)`

☞ `ifactor(PrimeLimit)`

Parameter:

n — ein arithmetischer Ausdruck, der eine ganze Zahl darstellt

Optionen:

UsePrimeTab — es wird nur nach Primfaktoren gesucht, die in der internen Primzahltable des Systems gespeichert sind

PrimeLimit — gib die Schranke für die der größten Primzahl der Primzahltable zurück

Rückgabewert: ein Objekt vom Domaintyp `Factored`, oder ein symbolischer `ifactor`-Aufruf.

Verwandte Funktionen: `content`, `factor`, `Factored`, `icontent`, `igcd`, `ilcm`, `isprime`, `ithprime`, `nextprime`, `numlib::divisors`, `numlib::ecm`, `numlib::mpqs`, `numlib::pollard`, `numlib::prevprime`, `numlib::primedivisors`

Details:

☞ Das Ergebnis von `ifactor` ist ein Objekt vom Domaintyp `Factored`. Ist `f:=ifactor(n)` ein solches Objekt, so ist dessen interne Darstellung die Liste `[s, p1, e1, ..., pr, er]` von ungerader Länge $2r+1$, wobei r die Anzahl verschiedener Primteiler von n ist. Die p_i sind im allgemeinen nicht nach Größe sortiert.

Auf das Vorzeichen s , die Primzahlen p_i sowie die Exponenten e_i lassen sich u. a. über den gewöhnlichen Indexoperator `[]` zugreifen, d. h. `f[1] = s`, `f[2] = p1`, `f[3] = e1`,

Die Eingabe `f[2*i] $ i = 1..nops(f) div 2` liefert beispielsweise alle Primzahlen von n . Das kann auch – etwas komfortabler – über den Aufruf `Factored::factors(f)` erreicht werden. Entsprechend liefert `Factored::exponents(f)` eine Liste der Exponenten e_i ($1 \leq i \leq r$) der Faktorisierung von n .

Die Faktorisierung von 0, 1 bzw. -1 liefert den einzigen Faktor 0, 1 bzw. -1 . In diesen Fällen ist die interne Darstellung die Liste `[0]`, `[1]` bzw. `[-1]`.

Die Eingabe `coerce(f, DOM_LIST)` liefert die interne Darstellung eines faktorierten Objektes, d. h. die Liste wie oben beschrieben.

Man beachte, dass das Ergebnis von `ifactor` wie ein Ausdruck ausgegeben wird und sich im Prinzip auch als solcher verhält. Beispielsweise ist das Ergebnis `ifactor(12)` ein Objekt, das in der Form $2^2 \cdot 3$ ausgegeben wird und ein Ausdruck vom Typ `"_mult"` ist.

Siehe Beispiel ?? und die Hilfeseite zu `Factored` für Details.

☞ Um zu erfahren, ob n eine Primzahl ist, sollte die wesentlich schnellere Funktion `isprime` verwendet werden, anstatt `ifactor` aufzurufen, wenn die Primzahlzerlegung nicht benötigt wird.

☞ `ifactor` gibt eine Fehlermeldung zurück, wenn das Argument eine Zahl aber keine ganze Zahl ist. Ist das Argument keine Zahl, so wird ein symbolischer `ifactor`-Aufruf zurückgegeben.

Option <UsePrimeTab>:

- ☞ MuPAD berechnet beim Systemstart eine Tabelle aller Primzahlen bis zu einer bestimmten Grenze. `ifactor(n, UsePrimeTab)` findet nur die Primfaktoren, die in dieser Primzahltable enthalten sind. Dieser Aufruf ist normalerweise wesentlich schneller, die Faktoren `pi` sind aber nicht notwendigerweise prim (siehe Beispiel ??).

Option <PrimeLimit>:

- ☞ `ifactor(PrimeLimit)` liefert eine natürliche Zahl `n`, die die Obergrenze für alle Zahlen in der Primzahltable darstellt.

Die Standard-Werte für `n` sind 1 000 000 unter Unix, 300 000 unter MacOS und Windows.

Auf UNIX-Systemen kann die Grenze für die Primzahltable durch die Kommandozeilen-Option `-L` geändert werden.

Beispiel 1. Die Primfaktorzerlegung der Zahl 120 lautet wie folgt:

```
>> f := ifactor(120)
```

$$2^3 \cdot 3 \cdot 5$$

Auf die interne Darstellung dieser Faktorisierung kann dann u. a. mit dem Indexoperator zugegriffen werden:

```
>> f[1]; // das Vorzeichen
f[2*i] $ i=1..nops(f) div 2; // die Faktoren
f[2*i + 1] $ i=1..nops(f) div 2; // die Exponenten
```

1

2, 3, 5

3, 1, 1

Die vollständige interne Repräsentation von `f`, wie oben beschrieben, liefert die folgende Eingabe:

```
>> coerce(f, DOM_LIST)
```

[1, 2, 3, 3, 1, 5, 1]

Das Ergebnis der Faktorisierung ist ein Objekt des Domains `Factored`:

```
>> domtype(f)
```

Factored

Diese Domain stellt einige Funktionalität zum Arbeiten mit solchen Objekten zur Verfügung, die an dieser Stelle nur kurz beschrieben werden.

So lassen sich die Faktoren und Exponenten auch direkt wie folgt erfragen:

```
>> Factored::factors(f), Factored::exponents(f)
```

[2, 3, 5], [3, 1, 1]

Der Typ der Faktorisierung kann bestimmt werden:

```
>> Factored::getType(f)
```

"irreducible"

Diese Ausgabe bedeutet, dass alle p_i prim sind. Andere mögliche Typen sind "squarefree" (siehe `polylib::sqrfree`) oder "unknown".

Solche Objekte lassen sich multiplizieren. Das Ergebnis liegt ebenfalls in faktorisierter Form vor:

```
>> f2 := ifactor(12)
```

2
2 3

```
>> f*f2
```

5 2
2 3 5

Es lässt sich fast jede Funktion auf ein solches Objekt anwenden (überwiegend Funktionen, die als Eingabe arithmetische Ausdrücke erwarten). Das Ergebnis ist meist nicht mehr ein Objekt des Domains Factored:

```
>> expand(f);  
domtype(%)
```

120

DOM_INT

Die Funktion `type` konvertiert ein Objekt des Domains Factored automatisch in einen Ausdruck:

```
>> type(f)
```

"_mult"

Eine detaillierte Beschreibung solcher Objekte enthält die Hilfeseite des Domains Factored.

Beispiel 2. Die Faktorisierungen von 0, 1 und -1 haben jeweils genau einen Faktor:

```
>> ifactor(0), ifactor(1), ifactor(-1)

0, 1, -1

>> map(%, coerce, DOM_LIST)

[0], [1], [-1]
```

Die interne Darstellung der Faktorisierung einer Primzahl p ist die Liste $[1, p, 1]$:

```
>> coerce(ifactor(5), DOM_LIST)

[1, 5, 1]
```

Beispiel 3. Die Schranke für die größte Zahl der Primzahltafel ist:

```
>> ifactor(PrimeLimit)

1000000
```

p wird eine große Primzahl zugewiesen:

```
>> p := nextprime(10^12)

10000000000039
```

Die Faktorisierung der 36stelligen Zahl $6 \cdot p^3$ dauert etwas länger (die zweite Ausgabezeile zeigt die benötigte Zeit in Sekunden):

```
>> t := time():
    f := ifactor(6*p^3);
    (time() - t)/1000.0

3
2 3 10000000000039

12.34

>> Factored::getType(f)
```

"irreducible"

Wenn zur Faktorisierung nur die Faktoren der Primzahltafel genutzt werden, ist die Berechnung sehr schnell; der Faktor p^3 wird jedoch unzerlegt zurückgegeben:

```

>> t := time():
    f := ifactor(6*p^3, UsePrimeTab);
    (time() - t)/1000.0

2 3 10000000000117000000004563000000059319

0.21

>> Factored::getType(f)

"unknown"

```

Hintergründe:

⌘ `ifactor` benutzt die elliptische Kurvenmethode.

`ifactor` ist eine Schnittstellenfunktion für die Kernfunktion `stdlib::ifactor`. Sie ruft die Funktion `stdlib::ifactor` mit den gegebenen Argumenten auf und konvertiert das Ergebnis, das die Liste `[s, p1, e1, ..., pr, er]` in der oben beschriebenen Form ist, in ein Objekt des Domain-typs `Factored`.

Um diese Konvertierung zu vermeiden und damit die Laufzeit zu reduzieren, kann (z. B. innerhalb eigener Routinen) die Kernfunktion `stdlib::ifactor` aufgerufen werden.

Änderungen:

⌘ Der Rückgabewert von `ifactor` ist ein Objekt des Domains `Factored`.

igamma – die unvollständige Gammafunktion

`igamma(a, x)` stellt die unvollständige Gammafunktion $\int_x^\infty e^{-t} t^{a-1} dt$ dar.

Aufruf(e):

⌘ `igamma(a, x)`

Parameter:

`a, x` — arithmetische Ausdrücke

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `a, x`

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `Ei`, `erfc`, `exp`, `fact`, `gamma`, `int`

Details:

- ☞ Eine Gleitpunktzahl wird berechnet, wenn mindestens eines der beiden Argumente eine Gleitpunktzahl ist, beide Werte numerisch sind und den unten aufgeführten Einschränkungen genügen. In allen anderen Fällen können symbolische Aufrufe von `igamma` und/oder anderen speziellen Funktionen zurückgeliefert werden.
- ☞ Gleitpunktauswertung steht für beliebige reelle positive Werte von a und x zur Verfügung.
- ☞ Darüberhinaus steht Gleitpunktauswertung für beliebige komplexe Werte x zur Verfügung, wenn a ein ganzzahliges Vielfaches von $1/2$ ist.
- ☞ In anderen Fällen kann eventuell keine Gleitpunktauswertung möglich sein. Insbesondere wird ein symbolischer Aufruf von `igamma` zurückgeliefert, falls a keine reelle Zahl ist.
- ☞ Die folgenden Vereinfachungen und Umschreiberegeln sind implementiert:

$$\begin{aligned}\text{igamma}(a, 0) &\rightarrow \text{gamma}(a), \\ \text{igamma}(0, x) &\rightarrow \text{Ei}(x), \\ \text{igamma}(1/2, x) &\rightarrow \sqrt{\text{PI}} * \text{erfc}(\sqrt{x}), \\ \text{igamma}(1, x) &\rightarrow \exp(-x).\end{aligned}$$

Für reelle numerische Werte von a vom Typ `Type::Real` wird die funktionale Beziehung

$$\text{igamma}(a, x) = x^{(a-1)} * \exp(-x) + (a-1) * \text{igamma}(a-1, x)$$

rekursiv benutzt, um das erste Argument in das Intervall $0 \leq a \leq 1$ zu verschieben. Dementsprechend wird das Ergebnis vollständig durch `Ei`, `erfc` und `exp` dargestellt, falls a ein ganzzahliges Vielfaches von $1/2$ ist. Siehe Beispiel ??.

- ☞ Der spezielle Wert `igamma(a, infinity)=0` ist implementiert.

- ☞ Die Gleitpunktauswertung ist schnell und numerisch stabil, wenn beide Argumente reell und positiv sind. In allen anderen Gleitpunktauswertungen kann es zu numerischer Instabilität durch Auslöschung kommen! Siehe Beispiel ??.



Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> igamma(2, 3), igamma(1/7, x), igamma(sqrt(2), 3)
                                     1/2
      4 exp(-3), igamma(1/7, x), igamma(2  , 3)
>> igamma(a, 4), igamma(1 + I, x^2 + 1), igamma(a, infinity)
                                     2
      igamma(a, 4), igamma(1 + I, x  + 1), 0
```

Wenn das erste Argument a ein reeller numerischer Wert ist, dann werden die funktionalen Relationen solange rekursiv benutzt, bis ein Aufruf mit einem ersten Argument aus dem Intervall $0 \leq a \leq 1$ auftritt:

```
>> igamma(-1/10, 1), igamma(7/4, 1)
                                     3 igamma(3/4, 1)
      10 exp(-1) - 10 igamma(9/10, 1), exp(-1) + -----
      ----
                                     4
```

Wenn das erste Argument ein ganzzahliges Vielfaches von $1/2$ ist, dann wird das Ergebnis vollständig mittels Ei , $erfc$, und exp ausgedrückt:

```
>> igamma(-3, x), igamma(-5/2, x), igamma(8, x), igamma(13/2, 4)
      / 1      1      2      \
      exp(-x) | -- - - - -- |
      |      2      x      3      |
      Ei(x)   \ x      x      /
      - - - - - -----,
      6              6

      1/2      1/2
      8 exp(-x) 4 exp(-x) 2 exp(-x) 8 PI  erfc(x )
      ----- - ----- + ----- - -----
      15 x      15 x      5 x      15

      1/2      3/2      5/2      15
      15 x      15 x      5 x      15

      /      2      3      4      5      6      7      \
      |      x      x      x      x      x      x      |
      5040 exp(-x) | x + -- + -- + -- + --- + --- + ---- + 1 |,
      \      2      6      24     120     720     5040    /

      1/2
      210979 exp(-4) 10395 PI  erfc(2)
      ----- + -----
      16              64
```

Für Gleitkommazahlen wird der numerische Wert von `igamma` berechnet:

```
>> igamma(0.1, 4.0), igamma(7, 0.5), igamma(100, 100.0)
0.004420083058, 719.9992783, 4.542198121e155
```

Beispiel 2. Schnelle und numerische stabile Gleitpunktauswertung steht für alle reellen positiven a und x zur Verfügung:

```
>> igamma(1.0, 4.0), igamma(7.0, 10.2), igamma(12.3, 34.5)
0.01831563889, 84.97892788, 361.781135
```

Gleitpunktauswertung ist nicht möglich, wenn a nicht reell ist:

```
>> igamma(1.0*I, 4.0), igamma(7.0 - 3.2*I, 10.2)
igamma(1.0 I, 4.0), igamma(7.0 - 3.2 I, 10.2)
```

Gleitpunktauswertung ist auch nicht möglich, wenn a kein ganzzahliges Vielfaches von $1/2$ und x nicht reell und positiv ist:

```
>> igamma(0.1, -4.0), igamma(3/4, 12.3 + 3.45*I)
igamma(0.1, -4.0), igamma(3/4, 12.3 + 3.45 I)
```

Gleitpunktauswertung ist für jedes komplexe x möglich, wenn a ein ganzzahliges Vielfaches von $1/2$ ist:

```
>> igamma(-3/2, -4.0), igamma(12, 12.3 + 3.45*I)
2.363271801 - 2.925061502 I, 13266196.93 - 17206446.91 I
```

Beispiel 3. Die funktionale Beziehung zwischen `igamma` mit unterschiedlichen ersten Argumenten wird benutzt, um eine „Normalisierung“ der zurückgelieferten Ausdrücke zu erreichen:

```
>> igamma(-8, x), igamma(7/3, x)
```

$$\begin{aligned}
 & \exp(-x) \left[\frac{1}{x} - \frac{1}{x^2} + \frac{2}{x^3} - \frac{6}{x^4} + \frac{24}{x^5} - \frac{120}{x^6} + \frac{720}{x^7} - \frac{5040}{x^8} \right] \\
 & - \frac{40320}{x^9} + \frac{40320}{x^{10}} \\
 & + \frac{4}{9} \operatorname{igamma}\left(\frac{1}{3}, x\right) + \frac{4}{3} x \exp(-x) + \frac{4}{3} x^2 \exp(-x) + \frac{4}{3} x^3 \exp(-x) + \frac{4}{3} x^4 \exp(-x) + \frac{4}{3} x^5 \exp(-x) + \frac{4}{3} x^6 \exp(-x) + \frac{4}{3} x^7 \exp(-x) + \frac{4}{3} x^8 \exp(-x) + \frac{4}{3} x^9 \exp(-x) + \frac{4}{3} x^{10} \exp(-x)
 \end{aligned}$$

Man beachte, dass solche Entwicklungen auch bei Gleitpunktauswertungen eingesetzt werden, falls a und x nicht reell und positiv sind. Die Reihendarstellung kann jedoch numerisch instabil sein, wenn $|a|$ groß ist:

```
>> DIGITS := 10: igamma(-100, 100.0)
                        8.139678825e-223
>> DIGITS := 20: igamma(-100, 100.0)
                        1.8951666599463620044e-232
>> delete DIGITS:
```

Änderungen:

- ⌘ Normalisierung des ersten Argumentes auf den Bereich $0 \leq a \leq 1$ wird nun für alle a vom Typ `Type::Real` eingesetzt. Der Aufruf `igamma(a, infinity)` liefert nun den Wert 0.
-

igcd – der größte gemeinsame Teiler ganzer Zahlen

`igcd(i1, i2, ...)` berechnet den größten gemeinsamen Teiler der ganzen Zahlen i_1, i_2, \dots

Aufruf(e):

- ⌘ `igcd(i1, i2, ...)`

Parameter:

- i_1, i_2, \dots — arithmetische Ausdrücke, die ganze Zahlen darstellen

Rückgabewert: eine nichtnegative ganze Zahl, oder ein symbolischer `igcd`-Aufruf.

Verwandte Funktionen: `content, div, divide, factor, gcd, gcdex, icontent, ifactor, igcdex, ilcm, lcm, mod`

Details:

- ⌘ `igcd` berechnet den größten gemeinsamen Teiler einer Folge von ganzen Zahlen. `igcd` aufgerufen mit nur einer Zahl gibt deren absoluten Betrag zurück. Das Ergebnis ist 0, wenn ein Argument 0 ist oder wenn kein Argument gegeben ist.

⌘ `igcd` gibt eine Fehlermeldung zurück, wenn ein Argument eine Zahl aber keine ganze Zahl ist. Wenn ein Argument keine Zahl ist, wird ein symbolischer `igcd`-Aufruf zurückgegeben.

⌘ `igcd` ist eine Funktion des Systemkerns.

Beispiel 1. Der größte gemeinsame Teiler einiger Zahlen wird berechnet:

```
>> igcd(-10, 6), igcd(6, 10, 15)

2, 1
```

```
>> a := 4420, 128, 8984, 488:
    igcd(a), igcd(a, 64)

4, 4
```

Das nächste Beispiel zeigt einige Spezialfälle:

```
>> igcd(), igcd(0), igcd(1), igcd(-1), igcd(2)

0, 0, 1, 1, 2
```

Wenn ein Argument keine Zahl ist, wird ein symbolischer `igcd`-Aufruf zurückgegeben, außer in Spezialfällen:

```
>> delete x:
    igcd(a, x), igcd(1, x), igcd(-1, x)

igcd(4420, 128, 8984, 488, x), 1, 1

>> type(igcd(a, x))

"igcd"
```

Änderungen:

⌘ Wenn eines der Argumente 1 oder -1 ist, dann ist der Rückgabewert 1.

`igcdex` – der erweiterte Euklidische Algorithmus für zwei ganze Zahlen

`igcdex(x, y)` berechnet den größten gemeinsamen Teiler g der ganzen Zahlen x und y und ganze Zahlen s und t , so dass $g = sx + ty$ gilt.

Aufruf(e):

```
# igcdex(x, y)
```

Parameter:

x, y — arithmetische Ausdrücke, die ganze Zahlen darstellen

Rückgabewert: eine Folge von drei ganzen Zahlen, oder ein symbolischer `igcdex`-Aufruf.

Verwandte Funktionen: `div, divide, factor, gcd, gcdex, ifactor, igcd, ilcm, lcm, mod, numlib::igcdmult`

Details:

`igcdex(x, y)` gibt eine Sequenz g, s, t zurück, wobei g der größte gemeinsame nichtnegative Teiler von x und y ist und s und t ganze Zahlen mit $g = sx + ty$ sind. Diese Daten werden mit dem erweiterten Euklidischen Algorithmus berechnet.

`igcdex(0, 0)` gibt die Sequenz $0, 1, 0$ zurück. Wenn x ungleich Null ist, dann wird für `igcdex(x, 0)` und $0, x$ jeweils die Sequenz $\text{abs}(x), 0, \text{sign}(x)$ und $\text{abs}(x), \text{sign}(x), 0$ zurückgegeben.

Wenn x und y ganze Zahlen ungleich Null sind, erfüllen die Zahlen s und t die Ungleichung $|s| < |y/g|$ und $|t| < |x/g|$.

`igcdex` gibt einen Fehler zurück, wenn ein Argument eine Zahl, aber keine ganze Zahl ist. Wenn ein Argument keine Zahl ist, wird ein symbolischer `igcdex`-Aufruf zurückgegeben.

Die Funktion `numlib::igcdmult` ist eine Erweiterung von `igcdex` für mehr als zwei Argumente.

`igcdex` ist eine Funktion des Systemkerns.

Beispiel 1. Der größte gemeinsame Teiler einiger Zahlen wird berechnet:

```
>> igcdex(-10, 6)
```

```
2, 1, 2
```

```
>> igcdex(3839882200, 654365735423132432848652680)
```

```
109710920, -681651885490791809, 4
```

Die zurückgegebenen Zahlen erfüllen die beschriebene Gleichung:

```
>> [g, s, t] := [igcdex(9, 15)];
g = s*9 + t*15
```

`[3, 2, -1]`

`3 = 3`

Wenn ein Argument keine Zahl ist, wird ein symbolischer `igcdex`-Aufruf zurückgegeben:

```
>> delete x:  
      igcdex(4, x)
```

`igcdex(4, x)`

Änderungen:

⌘ Keine Änderungen.

`ilcm` – das kleinste gemeinsame Vielfache ganzer Zahlen

`ilcm(i1, i2, ...)` berechnet das kleinste gemeinsame Vielfache der ganzen Zahlen i_1, i_2, \dots

Aufruf(e):

⌘ `ilcm(i1, i2, ...)`

Parameter:

`i1, i2, ...` — arithmetische Ausdrücke, die ganze Zahlen darstellen

Rückgabewert: eine nichtnegative ganze Zahl, oder ein symbolischer `ilcm`-Aufruf.

Verwandte Funktionen: `content, factor, gcd, gcdex, icontent, ifactor, igcd, igcdex, lcm`

Details:

- ⌘ `ilcm` berechnet das kleinste gemeinsame positive Vielfache einer Sequenz von ganzen Zahlen. `ilcm` aufgerufen mit nur einer Zahl gibt deren absoluten Betrag zurück. Das Ergebnis ist 0, wenn ein Argument 0 ist oder wenn kein Argument gegeben ist.
- ⌘ `ilcm` gibt eine Fehlermeldung zurück, wenn ein Argument eine Zahl aber keine ganze Zahl ist. Wenn ein Argument keine Zahl ist, wird ein symbolischer `ilcm`-Aufruf zurückgegeben.

⌘ `ilcm` ist eine Funktion des Systemkerns.

Beispiel 1. Das kleinste gemeinsame Vielfache einiger Zahlen wird berechnet:

```
>> ilcm(-10, 6), ilcm(6, 10, 15)
30, 30

>> a := 4420, 128, 8984, 488:
    ilcm(a), ilcm(a, 64)
9689064320, 9689064320
```

Das nächste Beispiel zeigt einige Spezialfälle:

```
>> ilcm(), ilcm(0), ilcm(1), ilcm(-1), ilcm(2)
1, 0, 1, 1, 2
```

Wenn ein Argument keine Zahl ist, wird ein symbolischer `ilcm`-Aufruf zurückgegeben, außer in Spezialfällen:

```
>> delete x:
    ilcm(a, x), ilcm(0, x)
    ilcm(4420, 128, 8984, 488, x), 0

>> type(ilcm(a, x))
"ilcm"
```

Änderungen:

⌘ Wenn eines der Argumente 0 ist, dann ist der Rückgabewert 0.

`in` – „ist Element von“

`x in set` ist die „Element von“-Relation. Darüberhinaus kann `in` in Kombination mit `for` und `$` als Schlüsselwort der MuPAD-Sprache verwendet werden; in diesem Kontext bedeutet es „iteriere über alle Operanden“.

Aufruf(e):

```
⌘ x in set
⌘ _in(x, set)
⌘ for y in object do ... end_for
⌘ f(y) $ y in object
```


Parameter:

- | | |
|---------------------------|---|
| <code>x</code> | — ein beliebiges MuPAD-Objekt |
| <code>set</code> | — eine Menge oder ein Objekt von einem mengenartigen Typ |
| <code>y</code> | — ein Bezeichner oder eine lokale Variable (DOM_VAR) einer Prozedur |
| <code>object, f(y)</code> | — beliebige MuPAD-Objekte |

Überladbar durch: `x, set`

Rückgabewert: `x in set` liefert einen Ausdruck vom Typ `"_or"`, `"_and"`, `"_equal"` oder `"_in"`.

Verwandte Funktionen: `_seqin, bool, contains, for, has, is`

Details:

- ☞ `x in set` ist die MuPAD-Notation der Aussage „`x` ist ein Element von `set`“.
 - ☞ Wird `in` mit einem der Schlüsselworte `for` oder `$` verwendet, verändert sich die Bedeutung zu „iteriere über alle Operanden“. Siehe die Hilfeseiten zu `for` und `$` sowie Beispiel ??).
 - ☞ Außer im Kontext von `for` und `$` ist die Anweisung `x in object` äquivalent zum Funktionsaufruf `_in(x, object)`.
 - ☞ Wenn `set` vom Typ `DOM_SET` oder eine mengentheoretische Vereinigung, Restmenge oder Schnittmenge ist, evaluiert `x in set` zu einer äquivalenten logischen Aussage, die Gleichungen und Aufrufe von `in` enthalten kann. Siehe Beispiel ??.
 - ☞ Ist `set` eine durch einen symbolischen `solve`-Aufruf repräsentierte Lösungsmenge einer Gleichung in einer Unbekannten, so liefert `in` einen logischen Ausdruck, der äquivalent dazu ist, daß `x` eine Lösung der Gleichung ist. Siehe Beispiel ??.
 - ☞ Wenn `set` ein Ausdruck vom Typ `RootOf` ist, liefert `in` einen logischen Ausdruck zurück, der äquivalent dazu ist, daß `x` eine Lösung der entsprechenden Gleichung ist. Siehe Beispiel ??.
 - ☞ Die Funktion `is` kann mit einer Reihe logischer Aussagen umgehen, in denen `in` benutzt wird. Dazu gehören auch viele Typen für den Parameter `set`, die `in` selbst nicht behandelt. Beispiel ?? zeigt einige typische Aufrufe.
 - ☞ `in` kann neben der normalen Überladung durch das erste Argument auch vom zweiten Argument überladen werden. Dafür muss dieses Argument den Slot `"set2expr"` definieren, der dann mit den Argumenten `set, x` aufgerufen wird.
-

Beispiel 1. $x \in \{1, 2, 3\}$ wird in eine äquivalente Aussage umgeformt, die = und or verwendet:

```
>> x in {1, 2, 3}
```

$$x = 1 \text{ or } x = 2 \text{ or } x = 3$$

Auch $1 \in \{1, 2, 3\}$ ergibt eine Ausgabe dieser Form, da boolsche Ausdrücke nur innerhalb bestimmter Funktionen wie bool oder is ausgewertet werden:

```
>> 1 in {1, 2, 3}, bool(1 in {1, 2, 3}), is(1 in {1, 2, 3})
```

$$1 = 1 \text{ or } 1 = 2 \text{ or } 1 = 3, \text{ TRUE}, \text{ TRUE}$$

Kann nur ein Teil der Menge in dieser Art behandelt werden, so enthält der zurückgegebene Ausdruck Aufrufe von in:

```
>> x in {1, 2, 3} union A
```

$$x \in A \text{ or } x = 1 \text{ or } x = 2 \text{ or } x = 3$$

Beispiel 2. Bei einem symbolischen solve-Aufruf, der die Lösungsmenge einer einzelnen Gleichung in einer Unbekannten darstellt, kann mit in überprüft werden, ob ein Wert in der Lösungsmenge liegt:

```
>> solve(x^2 = 2^x, x); 2 in %, bool(2 in %)
```

$$\text{solve}(x^2 - 2^x = 0, x)$$

$$0 = 0, \text{ TRUE}$$

Beispiel 3. Ein Ausdruck vom Typ RootOf stellt konzeptionell die Lösungsmenge einer Gleichung dar; in kann verwendet werden, um zu überprüfen, ob ein Wert in dieser Menge liegt:

```
>> r := RootOf(x^2 - 1, x);
    1 in r, bool(1 in r), 2 in r, bool(2 in r)
```

$$\text{RootOf}(x^2 - 1, x)$$

$$0 = 0, \text{ TRUE}, 3 = 0, \text{ FALSE}$$

```
>> (y - 1) in RootOf(x^2 - 1 - y^2 + 2*y, x)
```

$$2y^2 - y^2 + (y^2 - 1)^2 - 1 = 0$$

```
>> expand(%)
```

$$0 = 0$$

```
>> delete r:
```

Beispiel 4. Die MuPAD-Funktion `is` kann Elemente in unendlichen Mengen finden und berücksichtigt dabei Eigenschaften von Bezeichnern:

```
>> is(123 in Q_), is(2/3 in Q_)
```

```
TRUE, TRUE
```

```
>> assume(p, Type::Prime): is(p in Z_), is(p in Type::NonNegative)
```

```
TRUE, TRUE
```

```
>> unassume(p):
```

Beispiel 5. In Verbindung mit den Schlüsselworten `for` und `$` iteriert die Anweisung `y in object` die Variable `y` über alle Operanden des Objekts:

```
>> for y in [1, 2] do
    print(y)
end_for:
```

```
1
```

```
2
```

```
>> y^2 + 1 $ y in a + b*c + d^2
```

$$a^2 + 1, b^2 c^2 + 1, d^4 + 1$$

```
>> delete y:
```

Änderungen:

☞ `_in` ist eine neue Funktion.

`indets` – die Unbestimmten eines Ausdrucks

`indets(object)` gibt die in `object` enthaltenen Unbestimmten zurück.

Aufruf(e):

```

⌘ indets(object)
⌘ indets(object, PolyExpr)
⌘ indets(object, RatExpr)

```

Parameter:

`object` — ein beliebiges Objekt

Optionen:

PolyExpr — die Rückgabe ist eine Menge von arithmetischen Ausdrücken, so dass `object` ein polynomialer Ausdruck in den zurückgegebenen Ausdrücken ist

RatExpr — die Rückgabe ist eine Menge von arithmetischen Ausdrücken, so dass `object` ein rationaler Ausdruck in den zurückgegebenen Ausdrücken ist

Rückgabewert: eine Menge von arithmetischen Ausdrücken.

Überladbar durch: `object`

Verwandte Funktionen: `collect`, `domtype`, `op`, `poly`, `rationalize`, `type`, `Type::PolyExpr`, `Type::RatExpr`

Details:

- ⌘ `indets(object)` gibt die Unbestimmten von `object` als Menge zurück. Unbestimmte sind Bezeichner ohne Wert, mit Ausnahme derjenigen Bezeichner, die im 0-ten Operanden eines Teilausdrucks vorkommen (siehe Beispiel ??).
- ⌘ `indets` gibt die speziellen Bezeichner `PI`, `EULER` und `CATALAN` als Unbestimmte zurück, obwohl diese Bezeichner konstante reelle Zahlen repräsentieren. Zum Ausschluss dieser Konstanten kann die Anweisung `indets(object) minus Type::ConstantIdents` verwendet werden (siehe Beispiel ??).
- ⌘ Wenn `object` ein Polynom, eine Funktionsumgebung, eine Prozedur oder eine Funktion des Systemkerns ist, gibt `indets` die leere Menge zurück (siehe Beispiel ??).
- ⌘ `indets` ist eine Funktion des Systemkerns.

Option <PolyExpr>:

- ☞ Mit *PolyExpr* wird *expr* als ein polynomialer Ausdruck interpretiert. Nicht-polynomiale Teilausdrücke wie $\sin(x)$, $x^{(1/3)}$, $1/(x+1)$ oder $f(a, b)$ werden dabei als Unbestimmte betrachtet und mit zurückgegeben. Im Gegensatz dazu gilt der Teilausdruck $f(2, 3)$ als Konstante, auch wenn f ein Bezeichner ohne Wert ist, denn seine Operanden sind konstant (siehe Beispiel ??).
- ☞ Wenn *object* ein Feld, eine Liste, eine Menge oder eine Tabelle ist, gibt *indets* eine Menge arithmetischer Ausdrücke zurück, so dass jeder Eintrag von *object* ein polynomialer Ausdruck in diesen Ausdrücken ist (siehe Beispiel ??).

Option <RatExpr>:

- ☞ Mit dieser Option wird *object* als rationaler Ausdruck interpretiert. Analog zu *PolyExpr* werden alle nicht-rationalen Teilausdrücke als Unbestimmte betrachtet (siehe Beispiel ??).

Beispiel 1. Der folgende Ausdruck enthält viele Unbestimmte:

```
>> delete g, h, u, v, x, y, z:
      e := 1/(x[u] + g^h) - f(1/3) + (sin(y) + 1)^2*PI^3 + z^(-
3) * v^(1/2)
```

$$\frac{1}{g^h + x[u]} + \text{PI}^3 (\sin(y) + 1)^2 - f(1/3) + \frac{v^{1/2}}{z^3}$$

```
>> indets(e)
```

{g, h, u, v, x, y, z, PI}

Zu beachten ist, dass die zurückgegebene Menge x und u enthält, aber nicht, wie man vielleicht erwartet, $x[u]$, da dieser Ausdruck intern in der funktionalen Form `_index(x, u)` dargestellt wird. Der Bezeichner f ist nicht enthalten, da f der 0-te Operand des Teilausdrucks $f(1/3)$ ist.

Obwohl PI eine mathematische Konstante repräsentiert, gibt *indets* PI als Unbestimmte zurück. Dies kann durch Verwendung von `Type::ConstantIdents` korrigiert werden:

```
>> indets(e) minus Type::ConstantIdents
```

$$\{g, h, u, v, x, y, z\}$$

Das Ergebnis von `indets` ist vollkommen anders, wenn eine der beiden Optionen angegeben wird:

```
>> indets(e, RatExpr)
```

$$\{z, \text{PI}, \sin(y), g^h, x[u], v^{1/2}\}$$

Tatsächlich ist `e` ein rationaler Ausdruck in den „Unbestimmten“ `z`, `PI`, `sin(y)`, `gh`, `x[u]`, `v(1/2)`, denn `e` wird aus diesen Ausdrücken und dem konstanten Ausdruck `f(1/3)` nur durch die Anwendung rationaler Operationen `+`, `-`, `*`, `/` und `^` mit ganzzahligen Exponenten gebildet. Gleichmaßen kann `e` aus `z`, `PI`, `sin(y)`, `gh`, `x[u]`, `v(1/2)` und dem konstanten Ausdruck `f(1/3)` mit den polynomialen Operationen `+`, `-`, `*` und `^` mit nichtnegativen ganzen Exponenten erzeugt werden:

```
>> indets(e, PolyExpr)
```

$$\left\{ \begin{array}{l} \text{PI}, \sin(y), \frac{1}{3}, \frac{1}{h}, \frac{1}{v^{1/2}} \\ z, g^h + x[u] \end{array} \right\}$$

Beispiel 2. `indets` funktioniert auch für verschiedenen anderen Datentypen. Polynome und Funktionen haben keine Unbestimmten:

```
>> delete x, y:
    indets(poly(x*y, [x, y])), indets(sin), indets(x -> x^2+1)

    {}, {}, {}
```

Für Liste und Mengen gibt `indets` die Vereinigung der Unbestimmten aller enthaltenen Objekte zurück:

```
>> indets([x, exp(y)]), indets([x, exp(y)], PolyExpr)

    {x, y}, {x, exp(y)}
```

Bei Tabellen werden nur die Unbestimmten der Einträge zurückgegeben, nicht die Unbestimmten von Indizes:

```
>> indets(table(x = 1 + sin(y), 2 = PI))

    {y, PI}
```

Hintergründe:

- ⌘ Wenn `object` ein Element eines Bibliothek-Domains `T` ist, das die Methode `"indets"` enthält, wird die Methode `T::indets` mit dem Argument `object` aufgerufen. Damit kann die Funktionalität von `indets` für benutzerdefinierte Domains erweitert werden. Wenn eine solche Methode nicht existiert, wird die leere Menge zurückgegeben.

Änderungen:

- ⌘ Keine Änderungen.
-

`indexval` – indizierter Zugriff auf Felder und Tabellen ohne Evaluierung

`indexval(x, i)` bzw. `x, i1, i2, ...` liefert den dem Index `i` bzw. `i1, i2, ...` entsprechenden Eintrag von `x` ohne Evaluierung.

Aufruf(e):

- ⌘ `indexval(x, i)`
- ⌘ `indexval(x, i1, i2, ...)`

Parameter:


- `x` — hauptsächlich eine Tabelle oder ein Array, aber ebenso erlaubt: eine Liste, eine endliche Menge, eine Folge oder eine Zeichenkette
- `i, i1, i2, ...` — Indizes. Für die meisten „Behälter“ `x` sind nur ganze Zahlen als Indizes zugelassen. Ist `x` eine Tabelle, so können beliebige MuPAD-Objekte als Indizes verwendet werden.

Rückgabewert: der dem Index entsprechende Eintrag von `x`. Wenn `x` eine Tabelle oder ein Feld ist, wird der zurückgegebene Eintrag nicht noch einmal evaluiert.

Überladbar durch: `x`

Verwandte Funktionen: `:=, _assign, _index, array, contains, DOM_ARRAY, DOM_LIST, DOM_SET, DOM_STRING, DOM_TABLE, op, table`

Details:

- ☞ Alle drei Aufrufe `indexval(x, i)`, `_index(x, i)` und `x[i]` geben das Element mit Index `i` des Feldes oder der Tabelle `x` zurück. Im Gegensatz zu `_index` und dem äquivalenten Indexoperator `[]` gibt `indexval` den entsprechenden Eintrag ohne Evaluierung zurück. Manchmal ist der Verzicht auf eine Evaluierung wünschenswert, weil eine Evaluierung u. U. zeitaufwendig sein kann.
 - ☞ Die Argumente `i` oder `i1, i2, ...` müssen zulässige Indizes von `x` sein, ansonsten wird eine Fehlermeldung ausgegeben (siehe Beispiel ??). Werden mehrere Argumente `i1, i2, ...` angegeben, wird das als mehrdimensionaler Index interpretiert (siehe Beispiel ??).
 - ☞ Das erste Argument `x` kann auch eine Liste, eine Menge, eine Zeichenkette oder eine Ausdrucksfolge sein. In diesen Fällen jedoch verhält sich `indexval` genau wie `_index` oder `[]`: Es wird das entsprechende Element evaluiert zurückgegeben. Insbesondere gleicht `indexval` sein erstes Element nicht aus.
 - ☞ Für alle anderen Basis-Domains funktioniert `indexval` genau wie `_index`: entweder tritt ein Fehler auf, oder der symbolische `indexval`-Aufruf wird zurückgegeben (siehe Beispiel ??).
 - ☞ `indexval` arbeitet in der aktuellen version nicht mit Matrizen, aber `_index` gibt Matrix-Elemente unevaluiert zurück. 
 - ☞ `indexval` ist eine Funktion des Systemkerns.
-

Beispiel 1. `indexval` arbeitet mit Tabellen:

```
>> T := table("1" = a, Be = b, '+' = a + b):  
      a := 1: b := 2:  
      indexval(T, Be), indexval(T, "1"), indexval(T, '+')  
  
      b, a, a + b
```

Im Gegensatz dazu evaluiert `_index` zurückgegebene Einträge:

```
>> _index(T, Be), _index(T, "1"), _index(T, '+')  
  
      2, 1, 3
```

Die nächste Eingabezeile hat dieselbe Bedeutung wie die letzte:

```
>> T[Be], T["1"], T['+']  
  
      2, 1, 3
```


`indexval` arbeitet ebenso mit Feldern. Das Verhalten ist dasselbe, nur müssen die Indizes positive ganze Zahlen sein:

```
>> delete a, b:
  A := array(1..2, 1..2, [[a, a + b], [a - b, b]]):
  a := 1: b := 2:
  indexval(A, 2, 2), indexval(A, 1, 1), indexval(A, 1, 2)

      b, a, a + b

>> _index(A, 2, 2), _index(A, 1, 1), _index(A, 1, 2)

      2, 1, 3

>> A[2, 2], A[1, 1], A[1, 2]

      2, 1, 3

>> delete A, T, a, b:
```

Beispiel 2. Mit allen anderen Objekten als erstes Argument arbeitet `indexval` wie `_index`:

```
>> delete a, b:
  L := [a, b, 2]:
  b := 5:
  L[2], _index(L, 2), indexval(L, 2), op(L, 2)

      5, 5, 5, 5
```

Gleichermaßen gibt es keinen Unterschied, wenn das erste Argument eine Ausdrucksreihe ist (die von `indexval` nicht ausgeglichen wird):

```
>> delete a, b: S := a, b, 2:
  b := 5:
  S[2], _index(S, 2), indexval(S, 2), op(S, 2)

      5, 5, 5, 5

>> delete L, S, a, b:
```

Beispiel 3. Wenn das zweite Argument kein gültiger Index ist, wird eine Fehlermeldung ausgegeben:

```
>> A := array(1..2, 1..2, [[a, b], [a, b]]):
  indexval(A, 3)
```

```
Error: Index dimension mismatch [array]
```

```
>> indexval(A, 1, 0)
```

```
Error: Illegal argument [array]
```

```
>> indexval("12345", 5)
```

```
Error: Invalid index [string]
```

Das Ergebnis von `indexval` kann aber auch ein symbolischer `indexval`-Aufruf sein:

```
>> T := table(1 = a, 2 = b):
```

```
indexval(T, 3)
```

```
indexval(T, 3)
```

```
>> delete X, i:
```

```
indexval(X, i)
```

```
indexval(X, i)
```

```
>> delete A, T:
```

Beispiel 4. Bei einem Feld muss die Anzahl der Indizes mit der Dimension des Felder übereinstimmen:

```
>> A := array(1..2, 1..2, [[a, b], [a, b]]):
```

```
a := 1: b := 2:
```

```
indexval(A, 1, 2), indexval(A, 2, 1)
```

```
b, a
```

Sonst wird eine Fehlermeldung ausgegeben:

```
>> indexval(A, 1)
```

```
Error: Index dimension mismatch [array]
```

Tabellen können auch Ausdruckssequenzen als mehrdimensionale Indizes haben:

```
>> delete a, b:
```

```
T := table((1, 1) = a, (2, 2) = b):
```

```
a := 1: b := 2:
```

```
indexval(T, 1, 1), indexval(T, 2, 2)
```

```
a, b
```

```
>> delete A, T, a, b:
```

Änderungen:

⌘ Keine Änderungen.

`infinity` – Unendlich

`infinity` repräsentiert den unendlich fernen Punkt auf der positiven reellen Halbachse.

Verwandte Funktionen: `complexInfinity`, `undefined`

Details:

⌘ `infinity` ist ein Element des Domains `stdlib::Infinity`. Es kann in arithmetischen Operationen verwendet werden. Einige System-Funktionen akzeptieren `infinity` als Parameter oder liefern diesen Wert als Resultat.

Beispiel 1. `infinity` kann in arithmetischen Operationen mit reellen Zahlen benutzt werden:

```
>> 7*infinity + 3, -3.0*infinity, 1/infinity,  
    infinity*infinity, infinity^2, sqrt(infinity)  
  
    infinity, -infinity, 0, infinity, infinity, infinity
```

Arithmetik mit komplexen Zahlen oder symbolischen Objekten ergibt symbolische Ausdrücke:

```
>> I*infinity + b  
  
    b + I infinity
```

Die Arithmetik reagiert auf Eigenschaften:

```
>> assume(a > 0): a*infinity  
  
    infinity  
  
>> assume(a < 0): a*infinity  
  
    -infinity  
  
>> unassume(a): a*infinity  
  
    a infinity
```

Die Auslöschung von unendlichen Werten ergibt undefined:

```
>> infinity - infinity, infinity/infinity  
  
undefined, undefined
```

Einige Systemfunktionen akzeptieren `infinity` als Parameter oder liefern diesen Wert als Resultat:

```
>> exp(infinity), sum(1/n, n = 1..infinity),  
    int(exp(-x^2), x = -infinity..infinity),  
    limit(x, x = infinity)  
  
infinity, infinity,  $\frac{1}{2}$  PI, infinity
```

Änderungen:

☞ Keine Änderungen.

info – Ausgabe von Kurzinformationen

`info(object)` gibt kurze Informationen über `object` aus.

`info()` zeigt eine Liste der verfügbaren MuPAD-Bibliotheken an.

Aufruf(e):

☞ `info(object)`
☞ `info()`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: das leere Objekt `null()` vom Typ `DOM_NULL`.

Seiteneffekte: Die Ausgabeformatierung von `info` ist von der Umgebungsvariablen `TEXTWIDTH` abhängig.

Verwandte Funktionen: `help`, `export`, `print`, `setuserinfo`, `userinfo`

Details:

- ⇒ `info` gibt eine kurze Information über `object` aus, falls vorhanden. Typischerweise ist das für Domains und Funktionsumgebungen der Fall.
 - ⇒ Ist `object` ein Domain, werden zusätzliche Informationen zu den Methoden des Domains ausgegeben.
 - ⇒ Ein Aufruf von `info` ohne Argumente gibt die Namen aller verfügbaren System-Bibliotheken aus.
 - ⇒ Durch Überladen von `info` können eigene Informationen zu Funktionen und Domains hinzugefügt werden. Ist `object` ein benutzerdefiniertes Domain oder eine benutzerdefinierte Funktionsumgebung mit einem "info"-Slot, dessen Wert eine Zeichenkette ist, dann gibt der Aufruf `info(object)` diese Zeichenkette aus. Siehe Beispiel ??.
-

Beispiel 1. Mit `info()` erhält man eine Liste aller Bibliotheken:

```
>> info()

-- Libraries:
Ax,          Cat,          Dom,          Network,   RGB,
Series,      Type,        adt,          combinat,  detools,
fp,          generate,    groebner,  import,    intlib,
linalg,      linopt,      listlib,  matchlib,  module,
numeric,     numlib,      ode,      orthpoly,  output,
plot,        polylib,     prog,     property,  solvelib,
specfunc,    stats,      stdlib,   stringlib, student,
transform
```

Das nächste Beispiel zeigt Informationen zu der Bibliothek `property`:

```
>> info(property)

Library 'property': properties of identifiers

-- Interface:
property::hasprop, property::implies, property::simpex

-- Exported:
assume, getprop, is, unassume
```

`info` gibt Informationen zu Voreinstellungen aus:

```
>> info(Pref::promptString)

A character string to be displayed as a prompt.
```

Zu einigen Objekten kann `info` keine Informationen geben:

```
>> info(a + b)

Sorry, no information available.
```

Beispiel 2. `info` gibt Informationen zu einer Funktionsumgebung aus:

```
>> info(sqrt)

sqrt -- the square root

sqrt ist eine Funktionsumgebung und hat einen Slot namens "info":

>> domtype(sqrt), sqrt::info

DOM_FUNC_ENV, "sqrt -- the square root"
```

Eigene Prozeduren können um eine kurze Information ergänzt werden. Zunächst einmal liefert `info` keine sinnvolle Information:

```
>> f := x -> x^2:
    info(f)

Sorry, no information available.
```

Um dies zu ändern, betten wird die Funktion `f` in eine Funktionsumgebung ein und speichern einen Informationstext in deren `"info"`-Slot:

```
>> f := funcenv(f):
    f::info := "the squaring function":
    info(f)

the squaring function

>> delete f:
```

Hintergründe:

- ⌘ Wenn das Argument `object` von `info` ein Domain ist, wird zunächst dessen Eintrag `"info"` als Information ausgegeben (dieser Eintrag muss eine Zeichenkette sein). Dann wird der Eintrag `"interface"` benutzt, um alle öffentlichen Methoden des Domains anzuzeigen (dieser Eintrag muss eine Menge mit Bezeichnern sein). Schließlich wird der Eintrag `"exported"` benutzt, um alle exportierten Funktionen anzuzeigen (dieser Eintrag ist eine Menge von Bezeichnern, die von der Funktion `export` erzeugt wird).

Änderungen:

☞ Keine Änderungen.

input – interaktive Eingabe von MuPAD-Objekten

input ermöglicht die interaktive Eingabe von MuPAD-Objekten.

Aufruf(e):

```
☞ input(<prompt1>)
☞ input(<prompt1,> x1, <prompt2,> x2, ...)
```

Parameter:

prompt1, prompt2, ...	— Eingabeaufforderungen: Zeichenketten
x1, x2, ...	— Bezeichner

Rückgabewert: die letzte Eingabe

Verwandte Funktionen: `finput`, `fprint`, `fread`, `ftextinput`, `print`, `read`, `text2expr`, `textinput`, `write`

Details:

- ☞ `input()` liefert die Eingabeaufforderung „Please enter expression :“ und wartet auf Eingaben vom Benutzer. Die durch Drücken der Eingabetaste <RETURN> abgeschlossene Eingabe wird eingelesen und *unevaluiert* als Funktionswert zurückgegeben.
- ☞ `input(prompt1)` benutzt die Zeichenkette `prompt1` statt der Standardeingabeaufforderung „Please enter expression :“.
- ☞ `input(<prompt1,> x1)` weist die Eingabe dem Bezeichner `x1` zu. Hierbei wird die Standardeingabeaufforderung verwendet, falls keine Zeichenkette `prompt1` übergeben wird.
- ☞ Mehrere Objekte können mit einem `input`-Befehl eingelesen werden. Jeder Bezeichner in der Argumentenfolge veranlasst `input`, eine Eingabeaufforderung auszugeben und auf eine Eingabe zu warten, die dann dem Bezeichner zugewiesen wird. Eine direkt vor dem Bezeichner übergebene Zeichenkette ersetzt dabei die Standardeingabeaufforderung. Siehe Beispiel ???. Argumente, die weder Zeichenketten noch Bezeichner sind, werden ignoriert.

☞ Die Bezeichner x1 etc. dürfen Werte haben. Diese werden durch input überschrieben.

☞ input ist eine Funktion des Systemkerns.

Beispiel 1. Die Standardeingabeaufforderung wird ausgegeben. Die Eingabe wird unevaluiert zurückgegeben:

```
>> input()  
  
Please enter expression : << 1 + 2 >>  
  
1 + 2
```

Eine Zeichenkette wird als Eingabeaufforderung benutzt:

```
>> input("enter a number: ")  
  
enter a number: << 5 >>  
  
5
```

Die Eingabe kann einem Bezeichner zugewiesen werden:

```
>> input(x)  
  
Please enter expression : << 5 >>  
  
5  
  
>> x  
  
5
```

Eine Eingabeaufforderung wird angegeben, die Eingabe wird einem Bezeichner zugewiesen:

```
>> input("enter a number: ", x)  
  
enter a number: << 6 >>  
  
6  
  
>> x  
  
6
```

```
>> delete x:
```


Beispiel 2. Bei der Eingabe mehrerer Objekte kann für jedes Objekt eine eigene Eingabeaufforderung spezifiziert werden:

```
>> input("enter a matrix: ", A, "enter a vector: ", x)

enter a matrix: << matrix([[a11, a12], [a21, a22]]) >>
enter a vector: << matrix([x1, x2]) >>

matrix([x1, x2])

>> A, x
```

+-	--	+-	--
+-	--	+-	--

```
>> delete A, x:
```

Beispiel 3. Die folgende Prozedur fragt nach einem Ausdruck und einer Variablen. Nach interaktiver Eingabe wird die Ableitung des Ausdrucks nach der Variable berechnet:

```
>> interactiveDiff :=
  proc()
    local f, x;
  begin
    f := input("enter an expression: ");
    x := input("enter an identifier: ");
    print(Unquoted, "The derivative of " . expr2text(f) .
      " with respect to " . expr2text(x) . " is:");
    diff(f, x)
  end_proc;
```

```
>> interactiveDiff()
```

```
enter an expression: << x^2 + x*y^3 >>
enter an identifier: << x >>
```

The derivative of $x^2 + x*y^3$ with respect to x is:

$$2x + y^3$$

Da input seine Eingabe nicht evaluiert, können unerwartete Resultate entstehen:

```
>> f := x^2 + x*y^3:
      z := x:
      interactiveDiff()

enter an expression: << f >>
enter an identifier: << z >>

      The derivative of f with respect to z is:

      0
```

Die folgende Modifikation erzwingt mittels eval die vollständige Evaluation:

```
>> interactiveDiff :=
      proc()
        local f, x;
        begin
          f := eval(input("enter an expression: "));
          x := eval(input("enter an identifier: "));
          print(Unquoted, "The derivative of " . expr2text(f) .
            " with respect to " . expr2text(x) . " is:");
          diff(f, x)
        end_proc;

>> interactiveDiff()

enter an expression: << f >>
enter an identifier: << z >>

      The derivative of x^2 + x*y^3 with respect to x is:

      3
      2 x + y

>> delete interactiveDiff, f, z;
```

Änderungen:

☞ Keine Änderungen.

int – bestimmte und unbestimmte Integration

`int(f, x)` bestimmt die formale Stammfunktion $\int f(x) dx$.

`int(f, x = a..b)` berechnet das bestimmte Integral $\int_a^b f(x) dx$.

Aufruf(e):

```

⌘ int(f, x)
⌘ int(f, x = a..b <, Continuous>)
⌘ int(f, x = a..b <, PrincipalValue>)

```

Parameter:

f — Integrand: ein arithmetischer Ausdruck, der eine Funktion in x darstellt
 x — Integrationsvariable: ein Bezeichner
 a, b — Integrationsgrenzen: arithmetische Ausdrücke

Optionen:

`Continuous` — unterbindet die Untersuchung auf Unstetigkeitsstellen.
`PrincipalValue` — berechnet den cauchyschen Hauptwert des Integrals.

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: f

Seiteneffekte: `int` reagiert auf vom Benutzer definierte Eigenschaften von Bezeichnern mit `assume`; siehe Beispiel ??.

Weitere Dokumentation: Abschnitt 7.2 des Tutoriums.

Verwandte Funktionen: `D`, `diff`, `intlib`, `limit`, `numeric::int`, `sum`

Details:

$\text{⌘ int}(f, x)$ berechnet die Stammfunktion $g = \int f \, dx$, d. h. es wird eine Funktion g bestimmt, für die formal $\partial g / \partial x = f$ gilt.
 Man beachte,

- es tritt keine Integrationskonstante im Ergebnis auf.
- das Ergebnis muss nicht notwendig stetig sein, sogar dann nicht, wenn der Integrand stetig ist. Näheres siehe Abschnitt „Hintergründe“.
- Die Ableitung des Ergebnisses stimmt mit f im allgemeinen nur für irgendein offenes Intervall der reellen Zahlen überein.

Es ist algorithmisch nicht immer möglich zu entscheiden, ob die beiden Ausdrücke $\partial g / \partial x$ und f äquivalent sind. Diese Tatsache beruht auf dem so genannten Null-Vergleichsproblem, welches in seiner Allgemeinheit nicht entscheidbar ist.

- ☞ Bei der unbestimmten Integration wird die Integrationsvariable x implizit als reell betrachtet. Bei der bestimmten Integration wird der Gültigkeitsbereich der Integrationsvariablen x implizit sogar auf den Integrationsbereich beschränkt. Näheres siehe Abschnitt „Hintergründe“. Insbesondere kann daher das Ergebnis von `int` für nicht reelle Werte von x ungültig sein. Beispielsweise gilt die Identität $\ln(\exp(x)) = x$ nur für reelle Werte von x ; daher gilt das gleiche auch für $\int \ln(\exp(x)) dx = x^2/2$.
- ☞ Falls MuPAD für das Integral keine geschlossene Lösung finden kann, gibt es einen symbolischen `int` Aufruf zurück. In diesem Fall hat man die Möglichkeit zur numerischen Integration (vgl. Beispiel ??) oder man kann das Integral mittels Reihenentwicklung bestimmen (siehe Beispiel ??).
- ☞ Bei der bestimmten Integration kann es vorkommen, dass aufgrund von Singularitäten im Integrationsbereich, `int` keine geschlossene Lösung findet. Falls es dem System möglich ist zu erkennen, dass ein Integral nicht existieren kann, liefert es `undefined`. In manchen Fällen kann man durch das Ausnutzen von Annahmen oder mit einer der beiden Optionen `Continuous` und `PrincipalValue` eine geschlossene Lösung bekommen (siehe Beispiel ??).
- ☞ Numerische Approximation eines bestimmten Integrals erhält man mit `numeric::int` oder `float`. Numerische Integration ist nur möglich, wenn sich die beiden Integrationsgrenzen a und b mittels `float` in Gleitpunktzahlen konvertieren lassen. Siehe Beispiel ??.

Option `<Continuous>`:

- ☞ Zur Berechnung eines bestimmten Integrals kann vom System zuerst das unbestimmte Integral g von f bzgl. x mit $\partial g / \partial x = f$ bestimmt werden. Um nun das bestimmte Integral zu berechnen, wird darauf der erste Hauptsatz der Differential- und Integralrechnung $\int_a^b f(x) dx = g(b) - g(a)$ angewandt, falls g im Intervall $[a, b]$ stetig ist. Hierzu wird normalerweise g auf Stetigkeit hin überprüft. Im Zweifelsfall wird ein symbolischer `int` Aufruf ausgegeben. Näheres siehe Abschnitt „Hintergründe“. Die Option `Continuous` ist eine rein *technische* Option, die dem System erlaubt, g als stetig anzunehmen. Mit der Option `Continuous` wird die Untersuchung von g auf Stetigkeit im Integrationsintervall unterbunden und der erste Hauptsatz der Differential- und Integralrechnung angewandt, ohne zu überprüfen ob dies mathematisch korrekt ist. Siehe Beispiel ??.
-

Option *<PrincipalValue>*:

☞ Treten an einzelnen inneren Punkten des Integrationsintervalls Pole im Integranden auf oder sind die Integrationsgrenzen $a = -\infty$ und $b = \infty$, dann existiert das bestimmte Integral nicht im strengen mathematischen Sinne. Wechselt der Integrand das Vorzeichen bei allen Polen im Integrationsintervall, dann existiert jedoch noch ein schwächeres bestimmtes Integral, der so genannte cauchysche Hauptwert, welcher es erlaubt, dass sich die „unendlichen Anteile“ links und rechts eines Pols des Integrals gegenseitig auslöschen. Mit der Option *PrincipalValue* berechnet `int` den cauchyschen Hauptwert des Integrals. Falls das (gewöhnliche) bestimmte Integral existiert, dann stimmt es mit dem cauchyschen Hauptwert überein. Siehe Beispiel ??.

Beispiel 1. Die beiden unbestimmten Integrale $\int \frac{1}{x \ln x} dx$ und $\int \frac{1}{x^2 - 8} dx$ berechnet man mit:

```
>> int(1/x/ln(x), x)
ln(ln(x))

>> int(1/(x^2 - 8), x)
1/2      1/2      1/2      1/2
2      ln(x - 2 2 )      2      ln(x + 2 2 )
-----
8                        8
```

Das bestimmte Integral von $(x \ln(x))^{-1}$ im Intervall $[e, e^2]$ berechnet man mit:

```
>> int(1/x/ln(x), x = E..E^2)
ln(2)
```

Die Intervallgrenzen eines bestimmten Integrals dürfen auch $\pm\infty$ sein:

```
>> int(exp(-x^2), x = 0..infinity)
1/2
PI
-----
2
```

Auch mehrfache Integrale, wie beispielsweise das bestimmte mehrfache Integral $\int_0^a \int_0^{1-x/a} \int_0^{1-x/a-y/b} dz dy dx$ kann man bestimmen:

```
>> int(int(int(1, z = 0..c*(1 - x/a - y/b)),
y = 0..b*(1 - x/a)), x = 0..a)
a b c
-----
6
```

Beispiel 2. Das System kann für das folgende bestimmte Integral keine geschlossene Lösung finden und gibt sich selbst symbolisch zurück. Durch Anwenden von `float` kann man von diesem Ergebnis eine numerische Approximation bekommen:

```
>> int(sin(cos(x)), x = 0..1)

      int(sin(cos(x)), x = 0..1)

>> float(%)

      0.738642998
```

Alternativ kann man hierzu die Funktion `numeric::int` benutzen. Dies wird empfohlen, wenn man sich nur für die numerische Approximation interessiert, denn mit dieser Funktion sind keine symbolischen Auswertungen verbunden, womit sie gewöhnlich schneller als das Anwenden von `float` auf einen symbolischen `int` Aufruf ist.

```
>> numeric::int(sin(cos(x)), x = 0..1)

      0.738642998
```

Beispiel 3. `int` kann für das folgende unbestimmte Integral keine geschlossene Lösung finden und gibt sich selbst symbolisch zurück:

```
>> int((x^2 + 1)/sqrt(x^3 + 1), x)

      /      2      \
      |      x  + 1      |
int | -----, x |
      |      3      1/2      |
      \ (x  + 1)      /
```

Man kann `series` benutzen, um eine Reihenentwicklung des Integrals zu erhalten:

```
>> series(%, x = 0)

      3      4      6
      x      x      x
x + -- - -- - -- + O(x )
  3      8      12
```

Alternativ kann man die Reihenentwicklung des Integranden berechnen und diese hinterher integrieren. Dies wird empfohlen, wenn man sich statt für eine geschlossene Lösung des Integrals nur für eine Reihenentwicklung interessiert, denn dieser Weg ist gewöhnlich schneller als der umgekehrte Weg:

```
>> int(series((x^2 + 1)/sqrt(x^3 + 1), x = 0), x)
```

$$x + \frac{x^3}{3} - \frac{x^4}{8} - \frac{x^6}{12} + O(x^7)$$

Beispiel 4. `int` erkennt richtig, dass das folgende bestimmte Integral nicht definiert ist, weil der Integrand im Innern des Integrationsbereiches eine Polstelle besitzt:

```
>> int(1/(x - 1), x = 0..2)
```

undefined

Nichtsdestotrotz existiert der cauchysche Hauptwert des Integrals:

```
>> int(1/(x - 1), x = 0..2, PrincipalValue)
```

0

Treten im Integranden Parameter auf, kann dies dazu führen, dass `int` nicht entscheiden kann, ob der Integrand Polstellen im Integrationsbereich hat. In diesem Fall erscheint eine Warnung und `int` gibt sich selbst symbolisch zurück:

```
>> int(1/(x - a), x = 0..2)
```

```
Warning: Found potential discontinuities of the antiderivative\
.
Try option 'Continuous' or use properties (?assume). [intlib::\
antiderivative]
```

$$\text{int} \left| \frac{1}{x - a}, x = 0..2 \right|$$

Man kann nun dem Vorschlag der Warnung folgen und dem Parameter `a` eine Eigenschaft zuweisen, welche impliziert, dass der Integrand keine Polstellen im Integrationsbereich besitzt. Damit kann `int` hier eine geschlossene Lösung für das Integral bestimmen:

```
>> assume(a > 2): int(1/(x - a), x = 0..2)
```

$\ln(2 - a) - \ln(-a)$

Eine andere Möglichkeit besteht darin, dem Integrierer durch die Option `Continuous` mitzuteilen, dass es sich bei diesem Integranden tatsächlich um eine stetige Funktion im Integrationsintervall handelt:

```
>> unassume(a): int(1/(x - a), x = 0..2, Continuous)
```

$$\ln(2 - a) - \ln(-a)$$

Ein mit der Option *Continuous* berechnetes Ergebnis kann für spezielle Werte der auftretenden Parameter mathematisch falsch sein. Im obigen Beispiel ist das Ergebnis falsch für $0 < a < 2$. Daher sollte diese Option nur als letzte Maßnahme benutzt werden.

Beispiel 5. In diesem Beispiel sollen die Auswirkungen von Annahmen auf die Integrationsvariable aufgezeigt werden. Näheres siehe Abschnitt „Hintergründe“.

Die Integrationsvariable wird implizit als reell angenommen oder bei gegebenem Integrationsintervall als eingeschränkt gültig auf diesem Intervall. Diese Annahmen haben Auswirkungen u. a. auf die Vereinfachung der Ergebnisse. Beispielsweise wird folgendes Integral intern mit dem so genannten Risch-Algorithmus berechnet und nur aufgrund dieser impliziten Annahme in eine reelle Darstellung vereinfacht.

```
>> int(1/cos(x)^2, x)
```

$$\frac{2 \sin(2 x)}{2 \cos(2 x) + 2}$$

Um zu sehen was ohne diese implizite Annahme passieren würde, kann man die Integrationsvariable explizit als komplex definieren:

```
>> assume(x, Type::Complex): int(1/cos(x)^2, x)
```

$$-\frac{2 I}{\cos(2 x) - I \sin(2 x) + 1}$$

In Bezug auf die Integration unzulässige vom Benutzer definierte Eigenschaften führen nicht zu einem Fehler in der Integration wie sie es im Grunde müssten. Jedoch wird der Benutzer auf diese Inkonsistenz aufmerksam gemacht.

```
>> assume(x, Type::Imaginary): int(1/cos(x)^2, x)
```

```
Warning: Cannot integrate when x has property Type::Imaginary.
While integrating, we will assume x has property Type::Complex\
. [intlib::int]
```

$$-\frac{2 I}{\cos(2 x) - I \sin(2 x) + 1}$$

```
>> assume(x, Type::Integer): int(1/cos(x)^2, x)
```


Warning: Cannot integrate when x has property Type::Integer.
While integrating, we will assume x has property Type::Real. [
intlib::int]

$$\frac{2 \sin(2 x)}{2 \cos(2 x) + 2}$$

Entsprechendes gilt auch für die bestimmte Integration.

```
>> assume(x, Type::Interval(-5, -2)): int(x, x = 0..1)
```

Warning: While integrating, we will assume x has property [0, \ 1] of Type::Real instead of given property]-5, -2[of Type::R\ eal. [intlib::defInt]

$$1/2$$

Hintergründe:

- ⌘ Durch die in der Computeralgebra verwendeten Integrationstechniken, wie Integrationstabellen oder Risch-Algorithmus für die unbestimmte Integration, können zu den möglichen Unstetigkeitsstellen des Integranden weitere Unstetigkeitsstellen während des Integrationsprozesses hinzukommen. Dies beruht auf der Tatsache, dass algebraische Zahlen komplex sein können und kann bei numerischer Berechnung Verzweigungsprobleme verursachen, weil z. B. das Argument von Logarithmen komplexe Nullstellen enthalten kann, während der Integrand keine Pole im Integrationsbereich besitzt. Durch den klassischen Algorithmus zum Ersetzen der komplexen Logarithmen durch reelle Arcustangens-Funktionen

$$\sqrt{-1} \frac{d}{dx} \ln \left(\frac{u + \sqrt{-1}}{u - \sqrt{-1}} \right) = 2 \frac{d}{dx} \arctan(u)$$

wobei u ein Element aus $K(x)$ ist, so dass $u^2 \neq -1$ und K ein Unterkörper der reellen Zahlen ist, wird dieses Problem nicht gelöst. Jedoch kann dieser Algorithmus in Integrationstabellen verwendet worden sein. Werden nun die so erzielte Ergebnisse für die bestimmte Integration benutzt, ist es notwendig das berechnete unbestimmte Integral im Integrationsbereich auf Unstetigkeitsstellen hin zu untersuchen.

- ⌘ Die Integrationsvariable wird implizit als reell angenommen (Type::Real). Bei der bestimmten Integration ist der Gültigkeitsbereich sogar auf den Integrationsbereich beschränkt (Type::Interval[a, b]).

Sollte es dabei zu Konflikten mit den vom Benutzer durch `assume` festgelegten Eigenschaften von Bezeichnern kommen, wird eine entsprechende Warnung ausgegeben. Die Warnungen lassen sich mit den beiden Aufrufen `intlib::printWarnings(TRUE)` und `intlib::printWarnings(FALSE)` ein- bzw. ausschalten.

Bei der unbestimmten Integration wird, falls der Konflikt aufgelöst werden kann, mit den vom Benutzer definierten Eigenschaften gerechnet. Kann der Konflikt nicht aufgelöst werden und die gegebene Eigenschaft beschreibt eine Untermenge der reellen Zahlen, wird die Integrationsvariable als reell angenommen. Andernfalls wird die Integrationsvariable als komplex angenommen.

Enthält bei der bestimmten Integration die definierte Eigenschaft das angegebene Integrationsintervall wird diese zur weiteren Berechnung verwendet, sonst wird von obiger Annahme ausgegangen.

Siehe Beispiel ??.

☞ Detaillierte Beschreibungen der benutzten Algorithmen und Vereinfachungsstrategien findet man in:

- M. Bronstein. A Unification of Liouvillian Extension. AAEECC Applicable Algebra in Engineering, Communication and Computing. 1: 5–24, 1990.
- M. Bronstein. The Transcendental Risch Differential Equation. Journal of Symbolic Computation. 9: 49–60, 1990.
- M. Bronstein. Symbolic Integration I: Transcendental Functions. Springer. 1997.
- H. I. Epstein and B. F. Caviness. A Structure Theorem for the Elementary Functions and its Application to the Identity Problem. International Journal of Computer and Information Science. 8: 9–37, 1979.
- W. Fakler. Vereinfachen von komplexen Integralen reeller Funktionen. mathPAD 9 No. 1: 5-9, 1999.
- K. O. Geddes, S. R. Czapor and G. Labahn. Algorithms for Computer Algebra. 1992.

Änderungen:

- ☞ Mehr Klassen von Funktionen können nun integriert werden.
- ☞ Die Option `Continuous` wurde hinzugenommen.

`int2text` – Konvertierung einer ganzen Zahl in eine Zeichenkette

`int2text(n, b)` konvertiert die ganze Zahl `n` in eine Zeichenkette, die der `b`-adischen Darstellung von `n` entspricht.

Aufruf(e):

```
⌘ int2text(n <, b>)
```

Parameter:

- n — eine ganze Zahl
- b — die Basis: eine ganze Zahl zwischen 2 und 36. Die Standardbasis ist 10.

Rückgabewert: eine Zeichenkette.

Verwandte Funktionen: `coerce`, `expr2text`, `genpoly`,
`numlib::g_adic`, `tbl2text`, `text2expr`, `text2int`, `text2list`,
`text2tbl`

Details:

- ⌘ Die von `int2text` gelieferte Zeichenkette besteht aus den ersten `b` Zeichen der Folge $0, 1, \dots, 9, A, B, \dots, Z$. Für Basen größer als 10 stellen die Buchstaben dabei die `b`-adischen Ziffern größer als 9 dar: $A = 10, B = 11, \dots, Z = 35$.
 - ⌘ Für die Basen 2, 8 bzw. 16 liefert `int2text` die Konvertierung von der Dezimaldarstellung in die Binär-, Oktal- bzw. Hexadezimaldarstellung.
 - ⌘ `int2text` ist die Inverse der Funktion `text2int`.
 - ⌘ Da alle numerischen Datentypen MuPADs in der Ausgabe die Dezimaldarstellung verwenden, benutzt `int2text` Zeichenketten zur Darstellung `b`-adischer Zahlen. Die Funktion `numlib::g_adic` bietet eine alternative Darstellung mittels Listen.
 - ⌘ `int2text` ist eine Funktion des Systemkerns.
-

Beispiel 1. Bezüglich der Standardbasis 10 führt `int2text` eine reine Konvertierung des Datentyps von `DOM_INT` nach `DOM_STRING` aus:

```
>> int2text(123), int2text(-45678)
      "123", "-45678"
```

Beispiel 2. Die Dezimalzahl 32 in Binärdarstellung:

```
>> int2text(32, 2)

"100000"
```

Die Dezimalzahl 10^9 in Hexadezimaldarstellung:

```
>> int2text(10^9, 16)

"3B9ACA00"
```

Beispiel 3. Auch negative Werte können konvertiert werden:

```
>> int2text(-15, 8)

"-17"
```

Änderungen:

☞ int2text ist eine neue Funktion.

irreducible – Testen eines Polynoms auf Irreduzibilität

irreducible(p) testet, ob das Polynom p irreduzibel ist.

Aufruf(e):

☞ irreducible(p)

Parameter:

p — ein Polynom vom Typ DOM_POLY oder ein polynomialer Ausdruck

Rückgabewert: TRUE oder FALSE.

Überladbar durch: p

Verwandte Funktionen: content, factor, gcd, icontent, ifactor, igcd, ilcm, isprime, lcm, poly, polylib::divisors, polylib::primpart, polylib::sqrfree

Details:

- ⌘ Ein Polynom $p \in k[x_1, \dots, x_n]$ ist irreduzibel über dem Körper k , falls p weder konstant noch ein Produkt aus zwei nicht konstanten Polynomen aus $k[x_1, \dots, x_n]$ ist.
 - ⌘ `irreducible` gibt `TRUE` zurück, falls das Polynom irreduzibel ist über dem Körper, welcher durch die Koeffizienten impliziert wird, anderenfalls `FALSE`. Siehe die Funktion `factor` für Details der impliziten Annahme des Koeffizientenbereichs.
 - ⌘ Das Polynom kann entweder ein (multivariates) Polynom über den rationalen Zahlen, ein (multivariates) Polynom über einem Körper (z. B., dem Restklassenring `IntMod(n)` mit einer Primzahl n) oder ein univariates Polynom über einem algebraischen Erweiterungskörper (siehe `Dom::AlgebraicExtension`) sein.
 - ⌘ Ein polynomialer Ausdruck wird intern erst in ein Polynom vom Typ `DOM_POLY` konvertiert, bevor Irreduzibilität getestet wird.
-

Beispiel 1. Mit dem folgenden Aufruf wird getestet, ob der polynomialer Ausdruck $x^2 - 2$ irreduzibel ist. Hierbei wird der Koeffizientenkörper implizit als die rationalen Zahlen angenommen:

```
>> irreducible(x^2 - 2)

TRUE

>> factor(x^2 - 2)

      2
     x  - 2
```

Der folgende Irreduzibilitätstest ist negativ, weil $x^2 - 2$ über einer Körpererweiterung der rationalen Zahlen faktorisiert, welche das Radikal $\sqrt{2}$ enthält:

```
>> irreducible(sqrt(2)*(x^2 - 2))

FALSE

>> factor(sqrt(2)*(x^2 - 2))

      1/2      1/2      1/2
      2      (x + 2  ) (x - 2  )
```

Die folgenden Aufrufe benutzen Polynome vom Typ `DOM_POLY`. Der Koeffizientenkörper wird durch die Polynome explizit vorgegeben:

```
>> irreducible(poly(6*x^3 + 4*x^2 + 2*x - 4, IntMod(13)))
```

```

TRUE

>> factor(poly(6*x^3 + 4*x^2 + 2*x - 4, IntMod(13)))

      3      2
      6 poly(x  + 5 x  - 4 x - 5, [x], IntMod(13))

>> irreducible(poly(3*x^2 + 5*x + 2, IntMod(13)))

FALSE

>> factor(poly(3*x^2 + 5*x + 2, IntMod(13)))

      3 poly(x + 5, [x], IntMod(13)) poly(x + 1, [x], IntMod(13))

```

Änderungen:

⌘ Keine Änderungen.

is – Test auf eine mathematische Eigenschaft eines Ausdrucks

`is(x, prop)` testet, ob der Ausdruck `x` die mathematische Eigenschaft `prop` besitzt.

`is(y rel z)` testet, ob die Relation `rel` zwischen `y` und `z` gültig ist.

`is(x in set)` testet, ob `x` in der Menge `set` enthalten ist.

Aufruf(e):

⌘ `is(x, prop)`
 ⌘ `is(y rel z)`
 ⌘ `is(x in set)`

Parameter:

`x, y, z` — arithmetische Ausdrücke
`prop` — eine Eigenschaft
`rel` — `=`, `<`, `>`, `<=`, `>=` oder `<>`
`set` — eine Eigenschaft, die eine Menge von Zahlen repräsentiert (wie z. B. `Type::PosInt`) oder eine Menge, die von `solve` zurückgegeben wird; solch eine Menge kann ein Element von `Dom::Interval`, `Dom::ImageSet`, `piecewise` sein oder `C_`, `R_`, `Q_`, `Z_`.

Rückgabewert: `TRUE`, `FALSE` oder `UNKNOWN`.

Verwandte Funktionen: `assume, bool, getprop, property::implies, unassume`

Details:

☞ Der „Property-Mechanismus“ dient zur Vereinfachung von Ausdrücken mit Unbestimmten, die „mathematische Eigenschaften“ haben. Die Funktion `assume` erlaubt es, Bezeichnern einige elementare Eigenschaften zuzuordnen wie z. B. „ x ist eine reelle Zahl“ oder „ x ist eine ungerade ganze Zahl“. Ausdrücke, die x enthalten, können solche Eigenschaften erben, z. B. impliziert „ x ist reell“ die Eigenschaft „ $1 + x^2$ ist positiv“. Die Funktion `is` ist das wesentliche Hilfsmittel, um mathematische Eigenschaften eines Ausdrucks abzufragen.

Eine Auflistung aller verfügbaren Eigenschaften ist mittels `?property` erhältlich.

☞ `is` erfragt die Eigenschaften des gegebenen Ausdrucks mit `getprop`. Dann wird getestet, ob die Eigenschaft `prop` oder die Relation `y rel z` von den Eigenschaften von x oder y und z abgeleitet werden kann. Ist dies möglich, so gibt `is` den Wert `TRUE` zurück. Kann `is` die logische Negation von `prop` oder der Relation `y rel z` feststellen, so wird `FALSE` zurückgegeben. Ansonsten liefert `is` den Wert `UNKNOWN`.

☞ `is` gibt auch dann `UNKNOWN` zurück, wenn das System aus rein technischen Gründen die Eigenschaft `prop` nicht ableiten kann. Siehe Beispiel ??.

☞ In MuPAD gibt es neben `is` die Funktion `bool`, um Relationen `y rel z` zu überprüfen. Es gibt jedoch zwei wesentliche Unterschiede zwischen `bool` und `is`:

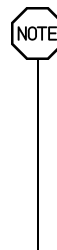
1. `bool` liefert einen Fehler, wenn nicht entschieden werden kann, ob die Relation gültig ist oder nicht. In einer solchen Situation liefert `is(y rel z)` den Wert `UNKNOWN`.
2. `bool` berücksichtigt keine Eigenschaften von Bezeichnern.

Siehe Beispiel ??.

☞ Wenn `bool(y rel z)` einen der Werte `TRUE` oder `FALSE` liefert, so ergibt auch `is(y rel z)` den entsprechenden Wert. Im allgemeinen ist aber `is` mächtiger als `bool`, auch wenn keine Eigenschaften definiert sind. Siehe Beispiel ??.

Andererseits ist `is` in den meisten Fällen wesentlich langsamer als `bool`.

☞ Wird `is` in der Bedingung einer `if`-Anweisung oder einer `for`-, `while`- oder `repeat`-Schleife verwendet, so ist zu beachten, dass diese Konstrukte nicht mit dem Wert `UNKNOWN` umgehen können, der eventuell von `is` geliefert werden kann. In solchen Fällen muss die Bedingung entweder in der Form `is(...) = TRUE` implementiert werden, oder es sollte ein `case`-Konstrukt benutzt werden. Siehe Beispiel ??.



☞ Wenn `is` testen soll, ob ein konstanter symbolischer Ausdruck Null ist, kann in einigen Fällen ein auf Gleitpunktapproximation basierender Nulltest angewendet werden. Trotz interner numerischer Stabilisierung kann dieser Test in sehr seltenen pathologischen Fällen eine falsche Antwort liefern, worauf auch `is` ein falsches Ergebnis zurückgibt.

Beispiel 1. Der Bezeichner `x` soll eine ganze Zahl darstellen:

```
>> assume(x, Type::Integer):  
    is(x, Type::Integer), is(x > 0), is(x^2 >= 0)  
  
TRUE, UNKNOWN, TRUE
```

Der Bezeichner `x` soll eine positive reelle Zahl darstellen:

```
>> assume(x > 0): is(x > 1), is(x >= 0), is(x < 0)  
  
UNKNOWN, TRUE, FALSE
```

```
>> unassume(x):
```

Beispiel 2. `is` kann gewisse Aussagen auch dann überprüfen, wenn keine Eigenschaften explizit definiert wurden:

```
>> is(x > x + 1), is(abs(x) >= 0)  
  
FALSE, TRUE
```

```
>> is(Re(exp(x)), Type::Real)  
  
TRUE
```


Beispiel 3. Für Relationen zwischen Zahlen liefert `is` dieselbe Antwort wie `bool`:

```
>> bool(1 > 0), is(1 > 0)

TRUE, TRUE
```

Bei konstanten symbolischen Ausdrücken kann `is` jedoch mehr entscheiden als `bool`:

```
>> is(sin(5) > 1/2), is(PI^3 + 2 < 33), is(exp(1) > exp(0.9))

FALSE, FALSE, TRUE
```

```
>> bool(sin(5) > 1/2)

Error: Can't evaluate to boolean [_less]
```

```
>> is(sqrt(2) > 1.4), is(PI > 3.1415)

TRUE, TRUE
```

```
>> bool(sqrt(2) > 1.4)

Error: Can't evaluate to boolean [_less]
```

```
>> is(E, Type::Real), is(PI, Type::PosInt)

TRUE, FALSE
```

Beispiel 4. Es folgen Beispiele, wo die angefragte Eigenschaft zwar mathematisch entschieden werden kann, die derzeitige Implementation von `is` aber noch nicht stark genug ist, die Gültigkeit der Eigenschaft abzuleiten:

```
>> assume(x, Type::Real): is(abs(x) >= x)

UNKNOWN

>> assume(x, Type::Interval(0, PI)): is(sin(x) >= 0)

UNKNOWN

>> unassume(x):
```

Beispiel 5. Vorsicht ist angesagt, wenn `is` in `if`-Anweisungen benutzt wird (dasselbe gilt für `for`, `repeat`, `while`):

```
>> myabs := proc(x)
      begin
        if is(x >= 0) then
          x
        elif is(x < 0) then
          -x
        else
          procname(x)
        end_if
      end_proc:

>> assume(x < 0): myabs(1), myabs(-2), myabs(x)

      1, 2, -x
```

Wenn der Aufruf von `is` den Wert `UNKNOWN` zurückgibt, so wird ein Fehler ausgelöst, da `if` entweder `TRUE` oder `FALSE` erwartet:

```
>> unassume(x): myabs(x)

Error: Can't evaluate to boolean [if];
during evaluation of 'myabs'
```

Der einfachste Weg, die gewünschte Funktionalität zu erreichen, ist ein Vergleich des Ergebnisses von `is` mit `TRUE`:

```
>> myabs := proc(x)
      begin
        if is(x >= 0) = TRUE then
          x
        elif is(x < 0) = TRUE then
          -x
        else
          procname(x)
        end_if
      end_proc:

>> myabs(x)

      myabs(x)

>> delete myabs:
```

Beispiel 6. `is` kann mit einigen Mengen umgehen, die von `solve` zurückgegeben werden. Speziell können dies Intervalle vom Typ `Dom::Interval` oder die durch das Objekt `R_ = solvelib::BasicSet(Dom::Real)` repräsentierten reellen Zahlen sein:

```
>> assume(x >= 0): assume(x <= 1, _and):
    is(x in Dom::Interval([0, 1])), is(x in R_)

TRUE, TRUE
```

Der folgende Aufruf von `solve` erzeugt eine unendliche parametrisierte Menge vom Typ `Dom::ImageSet`:

```
>> unassume(x): solutionset := solve(sin(x) = 0, x)

{ X3*PI | X3 in Z_ }

>> domtype(solutionset)

Dom::ImageSet
```

Mit `is` kann überprüft werden, ob ein Ausdruck im mathematischen Sinne in dieser Menge enthalten ist:

```
>> is(20*PI in solutionset), is(PI/2 in solutionset)

TRUE, FALSE

>> delete solutionset:
```

Änderungen:

☞ Die Fähigkeiten des „Property-Mechanismus“ wurden verbessert.

`isprime` – Primzahl-Test

`isprime(n)` testet, ob `n` eine Primzahl ist.

Aufruf(e):

☞ `isprime(n)`

Parameter:

`n` — ein arithmetischer Ausdruck, der eine ganze Zahl darstellt

Rückgabewert: `TRUE` oder `FALSE`, oder ein symbolischer `isprime`-Aufruf.

Verwandte Funktionen: factor, ifactor, igcd, ilcm, irreducible, ithprime, nextprime, numlib::primedivisors, numlib::prevprime, numlib::proveprime

Details:

- ⌘ isprime ist ein schneller stochastischer Primzahltest. Die Funktion gibt TRUE zurück, wenn die ganze Zahl n eine Primzahl oder eine starke Pseudo-Primzahl für zehn zufällig gewählte Basen ist, sonst wird FALSE zurückgegeben.
 - ⌘ Wenn n positiv ist und isprime FALSE zurückgibt, dann ist n mit Sicherheit keine Primzahl. Wenn n positiv ist und isprime TRUE zurückgibt, dann ist n mit großer Wahrscheinlichkeit eine Primzahl.
Die Funktion numlib::proveprime stellt einen Primzahltest zur Verfügung, der stets eine korrekte Antwort liefert, aber im Allgemeinen viel langsamer ist als isprime.
 - ⌘ isprime(0) und isprime(1) ergeben FALSE. isprime gibt immer FALSE zurück wenn n eine negative Zahl ist.
 - ⌘ isprime gibt einen Fehler zurück, wenn das Argument eine Zahl aber keine ganze Zahl ist. Wenn n keine Zahl ist, wird ein symbolischer isprime-Aufruf zurückgegeben.
 - ⌘ isprime ist eine Funktion des Systemkerns.
-

Beispiel 1. Ist die Zahl 989999 prim?

```
>> isprime(989999)

TRUE

>> ifactor(989999)

989999
```

Im Gegensatz zu ifactor ist isprime auch für große Zahlen schnell:

```
>> isprime(2^(2^11) + 1)

FALSE
```

isprime(0) und isprime(1) ergeben FALSE:

```
>> isprime(0), isprime(1)

FALSE, FALSE
```

Negative Zahlen ergeben FALSE:

```
>> isprime(-13)
```

FALSE

Für Argumente, die nicht numerisch sind, wird ein symbolischer `isprime`-Aufruf zurückgegeben:

```
>> delete n: isprime(n)
```

`isprime(n)`

Hintergründe:

- ☞ Verweis: Michael O. Rabin, Probabilistic algorithms, in J. F. Traub, ed., *Algorithms and Complexity*, Academic Press, New York, 1976, S. 21–39.

Änderungen:

- ☞ Keine Änderungen.
-

`isqrt` – ganzzahlige Quadratwurzel

`isqrt(n)` berechnet die ganzzahlige Näherung der Quadratwurzel der ganzen Zahl `n`.

Aufruf(e):

- ☞ `isqrt(n)`

Parameter:

`n` — ein arithmetischer Ausdruck, der eine ganze Zahl darstellt

Rückgabewert: eine nichtnegative ganze Zahl, ein ganzzahliges Vielfaches von 1 oder ein symbolischer `isqrt`-Aufruf.

Überladbar durch: `n`

Verwandte Funktionen: `_power`, `icontent`, `ifactor`, `igcd`, `ilcm`, `numlib::ispower`, `numlib::issqr`, `sqrt`, `trunc`

Details:

- ⌘ Wenn n das Quadrat einer ganzen Zahl ist, gibt `isqrt` die nichtnegative ganze Zahl zurück, deren Quadrat n ist. Wenn n eine beliebige ganze Zahl ist, berechnet `isqrt` `trunc(sqrt(n))`. Der Näherungsfehler ist also kleiner als 1.
 - ⌘ Wenn n eine negative ganze Zahl ist, gibt `isqrt` `trunc(sqrt(-n)) * I` zurück.
 - ⌘ `isqrt` gibt eine Fehlermeldung zurück, wenn das Argument eine Zahl aber keine ganze Zahl ist. Wenn das Argument keine Zahl ist, dann wird ein symbolischer `isqrt`-Aufruf zurückgegeben.
 - ⌘ `isqrt` ist eine Funktion des Systemkerns.
-

Beispiel 1. Das Beispiel zeigt einige ganzzahlige Quadratwurzeln:

```
>> isqrt(4), isqrt(5)

2, 2
```

Der Näherungsfehler ist kleiner als 1:

```
>> isqrt(99), float(sqrt(99))

9, 9.949874371
```

Die ganzzahlige Quadratwurzel einer negativen ganzen Zahl ist ein ganzzahliges Vielfaches von I :

```
>> isqrt(-4), isqrt(-5)

2 I, 2 I
```

Wenn das Argument keine Zahl ist, wird ein symbolischer `isqrt`-Aufruf zurückgegeben:

```
>> delete n: isqrt(n)

isqrt(n)

>> type(%)

"isqrt"
```

Änderungen:

☞ Keine Änderungen.

iszero – generischer Nulltest

`iszero(object)` testet, ob `object` das Nullelement des Domains von `object` ist.

Aufruf(e):

☞ `iszero(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: `TRUE` oder `FALSE`

Überladbar durch: `object`

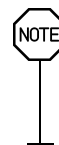
Verwandte Funktionen: `_equal`, `Ax::normalRep`, `bool`, `is`, `normal`, `simplify`, `sign`

Details:

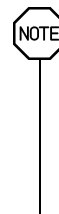
- ☞ Die Bedingung `iszero(object)` sollte anstelle von `object = 0` benutzt werden, um zu entscheiden, ob `object` das Nullelement ist, da `iszero(object)` allgemeiner ist als `object = 0`. Wenn der Aufruf `bool(object = 0)` `TRUE` ergibt, dann gibt auch `iszero(object)` `TRUE` zurück, aber die Umkehrung gilt nicht immer (siehe Beispiel ??).
- ☞ Wenn `object` ein Element eines Basisdomains ist, dann gibt `iszero` genau dann `TRUE` zurück, wenn folgendes gilt: `object` ist die ganze Zahl 0 (des Domains `DOM_INT`), die Gleitkommazahl `0.0` (des Domains `DOM_FLOAT`), oder das Nullpolynom (des Domains `DOM_POLY`). Im Falle eines Polynoms ist das Ergebnis `FALSE` nur dann garantiert richtig, wenn die Koeffizienten des Polynoms in Normalform gegeben sind (d. h. das Nullelement des Koeffizientenrings ist eindeutig darstellbar). Siehe auch `Ax::normalRep`.
- ☞ Wenn `object` ein Domainelement ist, dann wird die Methode "`iszero`" des Domains aufgerufen und das Ergebnis zurückgegeben. Wenn diese Methode nicht definiert ist, gibt `iszero` `FALSE` zurück.

☞ `iszero` führt nur einen syntaktischen Nulltest durch. Wenn `iszero` `TRUE` zurückgibt, dann ist diese Antwort immer richtig. Wenn `iszero` `FALSE` zurückgibt, kann es trotzdem vorkommen, dass `object` mathematisch die Null repräsentiert (siehe Beispiel ??). In solchen Fällen können die MuPAD-Funktionen `normal` oder `simplify` weiterhelfen.

☞ `iszero` beachtet *nicht* die Eigenschaften von Bezeichnern in `object`, die mit `assume` definiert wurden. Um für solche Ausdrücke zu entscheiden, ob sie Null sind, muss `is(object = 0)` verwendet werden (siehe Beispiel ??).



☞ `iszero` sollte nicht als Bedingung innerhalb von `piecewise` verwendet werden. Im Gegensatz zu `object = 0` wird die Anweisung `iszero(object)` sofort ausgewertet, bevor sie an `piecewise` weitergereicht wird. Daher führt die Verwendung von `iszero` in einem `piecewise`-Befehl normalerweise nicht zu dem gewünschten Ergebnis (siehe Beispiel ??).



☞ `iszero` ist eine Funktion des Systemkerns.

Beispiel 1. `iszero` kann die Grund-Typen verarbeiten:

```
>> iszero(0), iszero(1/2), iszero(0.0), iszero(I)
```

```
TRUE, FALSE, TRUE, FALSE
```

`iszero` funktioniert für Polynome:

```
>> p:= poly(x^2 + y, [x]):  
    iszero(p)
```

```
FALSE
```

```
>> iszero(poly(0, [x, y]))
```

```
TRUE
```

`iszero` ist allgemeiner als `=`:

```
>> bool(0 = 0), bool(0.0 = 0), bool(poly(0, [x]) = 0)
```

```
TRUE, FALSE, FALSE
```

```
>> iszero(0), iszero(0.0), iszero(poly(0, [x]))
```

```
TRUE, TRUE, TRUE
```


Beispiel 2. `iszero` beachtet Eigenschaften nicht:

```
>> assume(a = b): is(a - b = 0)
                                     TRUE

>> iszero(a - b)
                                     FALSE
```

Beispiel 3. Obwohl `iszero` im folgenden Beispiel `FALSE` liefert, ist der entsprechende Ausdruck mathematisch identisch Null:

```
>> iszero(sin(x)^2 + cos(x)^2 - 1)
                                     FALSE
```

In diesem Fall kann `simplify` das entscheiden:

```
>> simplify(sin(x)^2 + cos(x)^2 - 1)
                                     0
```

Beispiel 4. `iszero` sollte nicht in einer Bedingung für `piecewise` verwendet werden:

```
>> delete x:
    piecewise([iszero(x), 0], [x <> 0, 1])
                                     piecewise(1 if x <> 0)
```

Der erste Zweig wurde entfernt, weil `iszero(x)` sofort zu `FALSE` ausgewertet wird. Man benutze statt dessen die Bedingung `x = 0`, die unevaluiert an `piecewise` übergeben wird:

```
>> piecewise([x = 0, 0], [x <> 0, 1])
                                     piecewise(0 if x = 0, 1 if x <> 0)
```

Änderungen:

☞ Keine Änderungen.

`ithprime` – die *i*-te Primzahl

`ithprime(i)` gibt die *i*-te Primzahl zurück.

Aufruf(e):

```
# ithprime(i)
```

Parameter:

i — ein arithmetischer Ausdruck

Rückgabewert: eine Primzahl oder ein unevaluierter Aufruf von `ithprime`

Verwandte Funktionen: `ifactor`, `igcd`, `ilcm`, `isprime`, `nextprime`,
`numlib::prevprime`

Details:

- # Ist das Argument *i* eine positive ganze Zahl, so wird die *i*-te Primzahl zurückgeliefert. Ist *i* nicht vom Typ `Type::Numeric`, so wird ein unevaluierter Funktionsaufruf zurückgeliefert. Ist das Argument eine Zahl, aber keine positive ganze Zahl, so wird ein Fehler ausgelöst.
- # Die erste Primzahl `ithprime(1)` ist 2.
- # Falls die *i*-te Primzahl in der internen Primzahltafel des Systems enthalten ist (siehe die Hilfeseite zu `ifactor`), dann wird das Ergebnis durch eine schnelle Kernfunktion ermittelt. Andernfalls nimmt MuPAD einen geeigneten vorberechneten Wert von `ithprime` und ruft wiederholt die Funktion `nextprime` auf. Dies ist für $i \leq 1000000$ immer noch recht schnell. Für größere Werte von *i* steigt die Laufzeit jedoch exponentiell mit der Anzahl der Ziffern von *i*.

Beispiel 1. Die ersten 10 Primzahlen:

```
>> ithprime(i) $ i = 1..10
      2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

Eine größere Primzahl:

```
>> ithprime(123456)
      1632899
```

Symbolische Argumente führen zu unevaluierten Aufrufen:

```
>> ithprime(i)
      ithprime(i)
```

Änderungen:

☞ Keine Änderungen.

`lambertV`, `lambertW` – unterer und oberer reeller Zweig der Lambert-Funktion

Für reelles x stellen $y = \text{lambertV}(x)$ und $y = \text{lambertW}(x)$ die reellen Lösungen der Gleichung $ye^y = x$ dar.

Aufruf(e):

☞ `lambertV(x)`

☞ `lambertW(x)`

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Seiteneffekte: Für Gleitpunktargumente reagieren die Funktionen auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Details:

☞ Für alle reellen $x \geq 0$ hat die Gleichung $ye^y = x$ genau eine reelle Lösung. Sie wird durch $y = \text{lambertW}(x)$ dargestellt.

Für alle reellen x im Bereich $0 > x > -e^{-1}$ gibt es genau zwei reelle Lösungen. Die größere wird durch $y = \text{lambertW}(x)$ dargestellt, die kleinere durch $y = \text{lambertV}(x)$.

Es gibt genau eine reelle Lösung $\text{lambertW}(-e^{-1}) = \text{lambertV}(-e^{-1}) = -1$ für $x = -e^{-1}$.

☞ Dementsprechend ist der obere Zweig `lambertW` für reelle Argumente aus dem Intervall $[-e^{-1}, \infty)$ definiert. Diese Funktion ist monoton steigend mit dem Wertebereich $[-1, \infty)$.

Der untere Zweig `lambertV` ist für reelle Argumente aus dem Intervall $[-e^{-1}, 0)$ definiert. Diese Funktion ist monoton fallend mit dem Wertebereich $(-\infty, -1]$.

☞ Die Werte $\text{lambertV}(0) = -\text{infinity}$ und $\text{lambertW}(0) = 0$ sind implementiert. Weiterhin wird das Ergebnis y für einige Argumente der

Form $x = ye^y$ zurückgeliefert. Für Gleitpunktzahlen aus dem Definitionsbereich werden Gleitpunktwerte berechnet.

Für alle anderen Argumente werden unevaluierte Funktionsaufrufe zurückgeliefert.

☞ Die `float`-Attribute sind Kernfunktionen, d. h., die numerische Auswertung ist schnell.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> lambertV(-4), lambertW(-3), lambertV(-5/2), lambertW(1/2),
    lambertV(I), lambertW(1 + I), lambertV(x + 1)

lambertV(-4), lambertW(-3), lambertV(-5/2), lambertW(1/2),

    lambertV(I), lambertW(1 + I), lambertV(x + 1)
```

Einige exakte Werte werden gefunden:

```
>> lambertV(-exp(-1)), lambertW(-2*exp(-2)),
    lambertV(-3/2*exp(-3/2)), lambertW(exp(1)),
    lambertW(2*exp(2)), lambertW(5/2*exp(5/2))

-1, -2, -3/2, 1, 2, 5/2
```

Für Gleitkommazahlen werden numerische Werte berechnet:

```
>> lambertV(-0.3), lambertW(2000.0)

-1.781337024, 5.836731495
```

Die folgenden Argumente sind nicht aus dem Definitionsbereich und führen zu unevaluierten Funktionsaufrufen:

```
>> lambertV(-1.0), lambertW(-0.4), lambertV(0.1),
    lambertV(exp(1)), lambertV(5*exp(5))

lambertV(-1.0), lambertW(-0.4), lambertV(0.1),

    lambertV(exp(1)), lambertV(5 exp(5))
```

Beispiel 2. Die Funktionen `diff` und `float` verarbeiten die Lambert-Funktion:

```
>> diff(lambertV(x), x), diff(lambertW(x), x)

      lambertV(x)      lambertW(x)
-----, -----
x (lambertV(x) + 1)  x (lambertW(x) + 1)

>> float(ln(3 + lambertW(sqrt(PI))))

1.334475971
```

Hintergründe:

- ⌘ Literatur: R.M. Corless, D.J. Jeffrey and D.E. Knuth: „A sequence of Series for the Lambert W Function“, in: Proceedings of ISSAC'97, Maui, Hawaii. W.W. Kuechlin (ed.). New York: ACM, pp. 197-204, 1997.

Änderungen:

- ⌘ `lambertV` ist eine neue Funktion.
 - ⌘ `lambertW` ist eine neue Funktion.
-

`last` – Zugriff auf ein zuletzt berechnetes Objekt

`%` liefert das Ergebnis des zuletzt ausgeführten Kommandos.

`last(n)` oder `%n` gibt das Ergebnis des n-ten vorherigen Kommandos zurück.

Aufruf(e):

- ⌘ `last(n)`
- ⌘ `%`
- ⌘ `%n`

Parameter:

`n` — eine positive ganze Zahl

Rückgabewert: ein MuPAD-Objekt.

Weitere Dokumentation: Kapitel 12 des Tutoriums.

Verwandte Funktionen: `HISTORY`, `history`

Details:

- ⌘ MuPAD speichert standardmäßig die letzten 20 Kommandos und ihre Ergebnisse in einer internen History-Tabelle. `last(n)` liefert den Ergebniseintrag des n-ten Elementes dieser Tabelle. Die Zählung der Elemente beginnt hierbei am Ende der Tabelle. `last(1)` liefert also das Ergebnis des letzten Kommandos, `last(2)` das Ergebnis des vorletzten Kommandos usw. Anstelle von `last(n)` kann man auch einfacher `%n` schreiben. Anstelle von `last(1)` oder `%1` kann man einfach `%` benutzen.

- ☞ Die Umgebungsvariable `HISTORY` bestimmt die Anzahl der vorherigen Ergebnisse, auf die zugegriffen werden kann, also die Anzahl Einträge in der History-Tabelle. Innerhalb von Prozeduren ist diese Anzahl immer 3, unabhängig vom Wert von `HISTORY`. Zulässige Werte für `n` sind also die natürlichen Zahlen von 1 bis `HISTORY` auf interaktiver Ebene und die Zahlen 1, 2, 3 innerhalb einer Prozedur.

Mit `history` kann man direkt auf Einträge dieser Tabelle zugreifen, insbesondere auch auf das Kommando, welches das entsprechende Resultat geliefert hat.

- ☞ Ein durch `last` oder `%` zurückgegebener Wert wird nicht nochmals evaluiert. Das kann durch die Funktion `eval` nachträglich erzwungen werden. Siehe Beispiel ??.

- ☞ `last` verhält sich auf interaktiver Ebene und innerhalb von Prozeduren unterschiedlich. Auf interaktiver Ebene werden Anweisungsblöcke wie `for`, `repeat`, sowie `while` Schleifen, `if` und `case` Verzweigungen als Ganzes in der History-Tabelle gespeichert. In Prozeduren werden die einzelnen Anweisungen innerhalb eines Anweisungsblocks, anstelle des ganzen Anweisungsblocks, in eine prozedureigene History-Tabelle eingetragen. Siehe Beispiel ??.



- ☞ Kommandos und ihre Ergebnisse werden auch dann in die History-Tabelle eingetragen, wenn die Ausgabe des Ergebnisses durch einen Doppelpunkt unterdrückt wurde. Das Ergebnis von `last (n)` kann sich daher von der `n`-ten sichtbaren Bildschirmausgabe unterscheiden. Siehe Beispiel ??.

- ☞ Durch Doppelpunkt oder Semikolon getrennte Kommandos innerhalb einer Eingabezeile erzeugen mehrere Einträge in der History-Tabelle. Im Gegensatz hierzu wird eine Ausdruckssequenz als ein Kommando betrachtet. Siehe Beispiel ??.

- ☞ Mit `fread` oder `read` aus einer Datei gelesene Kommandos, werden in der History-Tabelle *vor* dem `fread` oder `read` Kommando abgespeichert. Wird die Option `Plain` benutzt, dann wird eine eigene History-Tabelle während des Einlesens der Datei angelegt und die in der Datei ausgeführten Kommandos beeinflussen die History-Tabelle des umgebenden Kontextes nicht. Auf der Hilfeseite von `history` findet man hierzu einige Beispiele.

- ☞ Im allgemeinen ist es schlechter Programmierstil `last` innerhalb von Prozeduren zu benutzen. Man sollte dieses vermeiden. Zukünftige MuPAD Versionen werden die Benutzung von `last` innerhalb von Prozeduren eventuell nicht mehr unterstützen.

- ☞ Wird die abkürzende Schreibweise `%n` benutzt, dann muss `n` eine positive ganze Zahl sein. Ist dies nicht der Fall und `n` evaluiert sich ledig-

lich zu einer positiven ganzen Zahl, so muss die äquivalente funktionale Schreibweise `last(n)` benutzt werden. Siehe Beispiel ??.

☞ `last` ist eine Funktion des Systemkerns.

Beispiel 1. Es folgen einige Beispiele zur Benutzung von `last` auf der interaktiven Ebene. Man beachte, dass `last(n)` sich auf die n -te vorhergehende Berechnung bezieht, unabhängig davon ob diese auf dem Bildschirm angezeigt wurde oder nicht:

```
>> a := 42;
      last(1), %, %1

                        42

                    42, 42, 42

>> a := 34: b := 56: last(2) = %2

                        34 = 34
```

Beispiel 2. Durch Doppelpunkt oder Semikolon getrennte Kommandos innerhalb einer Eingabezeile erzeugen mehrere Einträge in der History-Tabelle:

```
>> "First command"; 11: 22; 33:

                        "First command"

                        22

>> last(1), last(2);

                        33, 22
```

Wird eine Befehlssequenz geklammert, so wird diese als ein Kommando betrachtet:

```
>> "First command"; (11: 22; 33:)

                        "First command"

                        33

>> last(1), last(2);

                        33, "First command"
```

Eine Ausdruckssequenz wird ebenfalls als ein Kommando betrachtet:

```
>> "First command"; 11, 22, 33;

"First command"

11, 22, 33

>> last(1), last(2);

11, 22, 33, "First command"
```

Beispiel 3. Da der MuPAD-Parser eine positive ganze Zahl hinter dem % Zeichen erwartet, besteht ein Unterschied zwischen % und last. last kann mit einem Ausdruck aufgerufen werden, der sich zu einer positiven ganzen Zahl evaluiert:

```
>> n := 2: a := 35: b := 56: last(n)

35
```

Verwendet man statt dessen %, so erhält man einen Fehler:

```
>> n := 2: a := 35: b := 56: %n

Error: Unexpected 'identifier' [line 2, col 0]
```

Beispiel 4. Ein durch last zurückgegebener Wert wird nicht erneut evaluiert:

```
>> delete a, b:
c := a + b + a: a:= b: last(2)

2 a + b
```

Die Funktion eval erzwingt die nachträgliche Auswertung:

```
>> eval(%)

3 b
```


Beispiel 5. Das folgende Beispiel demonstriert den Unterschied bei der Benutzung von `last` auf interaktiver Ebene und innerhalb von Prozeduren:

```
>> 1: for i from 1 to 3 do i: print(last(1)): end_for:
```

1

1

1

`last(1)` bezieht sich auf den aktuellsten Eintrag in der History-Tabelle. Dieser ist hier, der vor der `for`-Schleife ausgeführte Befehl 1. Dies kann man nach Ausführung der Kommandos durch einen Blick in die History-Tabelle überprüfen. Der Befehl `history` liefert eine zweielementige Liste zurück. Der erste Eintrag der Liste das vorher eingegebene MuPAD Kommando und der zweite Eintrag ist das durch mupad berechnete Ergebnis dieses Kommandos. Man stellt fest, dass die History-Tabelle die vollständige `for`-Schleife als eingetragenes Kommando enthält:

```
>> history(history() - 1), history(history())
```

```
[1, 1], [(for i from 1 to 3 do
i;
print(last(1))
end_for), null()]
```

Wird die oben definierte `for`-Schleife innerhalb einer Prozedur ausgeführt, so erhält man ein anderes Ergebnis. In dem folgenden Beispiel verweist `last(1)` auf den zuletzt ausgewerteten Ausdruck `i` innerhalb der `for`-Schleife:

```
>> f := proc()
```

```
begin
```

```
1: for i from 1 to 3 do i: print(last(1)): end_for
end_proc:
```

```
>> f():
```

1

2

3

Der Befehl `history` verweist nur auf die interaktive Eingabe und deren Ergebnisse:

```
>> history(history())
```

```
[f(), null()]
```

Änderungen:

- ⌘ Das Verhalten von `last` innerhalb von Prozeduren und auf der interaktiven Ebene ist nun unterschiedlich.
-

`lasterror` – erneutes Auslösen des letzten Fehlers

`lasterror()` löst den letzten in der momentanen MuPAD-Sitzung aufgetretenen Fehler erneut aus.

Aufruf(e):

- ⌘ `lasterror()`

Verwandte Funktionen: `error`, `traperror`

Details:

- ⌘ Typischerweise wird `lasterror` dazu benutzt, durch `traperror` abgefangene Fehler erneut auszulösen. Siehe Beispiel ??.
 - ⌘ `lasterror` ist eine Funktion des Systemkerns.
-

Beispiel 1. Ein Fehler wird erzeugt:

```
>> x := 0: y := 1/x  
Error: Division by zero
```

Dieser Fehler kann durch `lasterror` erneut ausgelöst werden:

```
>> lasterror()  
Error: Division by zero
```

Ein weiterer Fehler wird erzeugt:

```
>> error("my error")  
Error: my error
```

```
>> lasterror()  
Error: my error
```

```
>> delete x, y:
```

Beispiel 2. Die folgende Prozedur `mysin` berechnet die Sinus-Funktion ihres Arguments. Falls die Systemfunktion `sin` dabei einen Fehler erzeugt, wird als Zusatzinformation das Argument ausgegeben, bevor der Fehler erneut ausgelöst wird:

```
>> mysin := proc(x)
    local result;
    begin
        if traperror((result := sin(x))) = 0 then
            return(result)
        else
            print(Unquoted, "the following error occurred " .
                  "when calling sin(\".expr2text(x).\")");
            lasterror()
        end_if;
    end;
```

In der Tat liefert die Sinus-Funktion des Systems für große Gleitpunktargumente einen Fehler:

```
>> mysin(1.0*10^100)

    the following error occurred when calling sin(10.0e99):
Error: Loss of precision;
during evaluation of 'sin'

>> delete mysin;
```

Änderungen:

⌘ `lasterror` ist eine neue Funktion.

lcm – das kleinste gemeinsame Vielfache von Polynomen

`lcm(p, q, ...)` berechnet das kleinste gemeinsame Vielfache der Polynome p, q, \dots

Aufruf(e):

⌘ `lcm(p, q, ...)`
 ⌘ `lcm(f, g, ...)`

Parameter:

p, q, \dots — Polynome vom Typ `DOM_POLY`
 f, g, \dots — polynomiale Ausdrücke

Rückgabewert: ein Polynom, ein polynomialer Ausdruck oder der Wert FAIL.

Überladbar durch: p, q, f, g

Verwandte Funktionen: `content, factor, gcd, gcdex, icontent, ifactor, igcd, igcdex, ilcm, poly`

Details:

- ⌘ `lcm(p, q, ...)` berechnet den größten gemeinsamen Teiler einer beliebigen Anzahl von Polynomen. Der Koeffizientenring der Polynome kann aus ganzen oder rationalen Zahlen bestehen oder *Expr*, ein Restklassenring *IntMod*(*n*) mit einer Primzahl *n* oder ein Domain sein. Alle Polynome müssen dieselben Unbestimmten und denselben Koeffizientenring haben.
 - ⌘ Polynomiale Ausdrücke werden in Polynome konvertiert. Siehe `poly` zu Details der Konvertierung. `lcm` liefert den Wert FAIL, falls ein Argument nicht konvertiert werden kann,
 - ⌘ Der Rückgabewert ist vom selben Typ wie die Eingabepolynome, d. h., entweder ein Polynom vom Typ DOM_POLY oder ein polynomialer Ausdruck.
 - ⌘ Wenn alle Argumente 1 oder -1 sind oder kein Argument angegeben wird, dann gibt `lcm` 1 zurück. Wenn mindestens ein Argument gleich 0 ist, dann gibt `lcm` 0 zurück.
 - ⌘ Wenn alle Argumente ganzzahlig sind, so wird empfohlen, `ilcm` zu verwenden, da es deutlich schneller ist als `lcm`.
-

Beispiel 1. Das kleinste gemeinsame Vielfache zweier polynomialer Ausdrücke kann wie folgt berechnet werden:

```
>> lcm(x^3 - y^3, x^2 - y^2);
```

$$y^4 - x^4 + x^3 y - x^3 y$$

Als Argumente sind auch Polynome zulässig:

```
>> p := poly(x^2 - y^2, [x, y], IntMod(17));  
    q := poly(x^2 - 2*x*y + y^2, [x, y], IntMod(17));  
    lcm(p, q)
```

$$\text{poly}(x^3 - x^2 y - x^2 y + y^3, [x, y], \text{IntMod}(17))$$

```
>> delete f, g, p, q;
```

Änderungen:

☞ Keine Änderungen.

`lcoeff` – der Leitkoeffizient eines Polynoms

`lcoeff(p)` gibt den Leitkoeffizienten des Polynoms `p` zurück.

Aufruf(e):

☞ `lcoeff(p <, vars> <, order>)`

Parameter:

- `p` — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- `vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- `order` — die Termordnung: entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.

Rückgabewert: ein Element des Koeffizientenrings des Polynoms oder `FAIL`.

Überladbar durch: `p`

Verwandte Funktionen: `coeff`, `collect`, `degree`, `degreevec`, `ground`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- ☞ Das Argument `p` kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `lcoeff` überlädt.
- ☞ Wird eine Liste von Unbestimmten übergeben, so wird `p` als Polynom in diesen Unbestimmten angesehen. Man beachte, dass diese Liste nicht mit den Unbestimmten in `p` übereinzustimmen braucht. Siehe Beispiel ??.
- ☞ Der zurückgelieferte Koeffizient ist „führend“ bezüglich der lexikographischen Ordnung, falls nicht mittels des Arguments `order` eine andere Ordnung angegeben wird. Siehe Beispiel ??.

- ⌘ Das Ergebnis von `lcoeff` wird nicht weiter evaluiert. Vollständige Evaluation kann mit der Funktion `eval` erzwungen werden. Siehe Beispiel ??.
- ⌘ `lcoeff` liefert `FAIL`, wenn das Eingabepolynom nicht in ein Polynom in den angegebenen Unbekannten konvertiert werden kann. Siehe Beispiel ??.
- ⌘ Für die Ordnungen *LexOrder*, *DegreeOrder* und *DegInvLexOrder* ruft `lcoeff` eine schnelle Kernfunktion auf. Andere Ordnungen werden durch langsamere Bibliotheksfunktionen behandelt.

Beispiel 1. Es wird gezeigt, wie die Unbestimmten das Ergebnis beeinflussen:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
      lcoeff(p), lcoeff(p, [x, y]), lcoeff(p, [y, x])

      3, 2, 3
```

Beachten Sie, dass die an `lcoeff` übergebenen Unbestimmten verwendet werden, auch wenn das Polynom sie gar nicht enthält:

```
>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
      lcoeff(p), lcoeff(p, [x, y]), lcoeff(p, [y, x]),
      lcoeff(p, [y]), lcoeff(p, [z])

      2          2
      2, 2, 3, 3 x, 2 x y + 3 x y
```

```
>> delete p:
```

Beispiel 2. Es wird demonstriert, wie verschiedene Ordnungen das Ergebnis beeinflussen:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
      lcoeff(p), lcoeff(p, DegreeOrder), lcoeff(p, DegInvLexOrder)

      5, 4, 3
```

Der folgende Aufruf benutzt die umgekehrte lexikographische Termordnung in 3 Unbestimmten:

```
>> lcoeff(p, Dom::MonomOrdering(RevLex(3)))

      3

>> delete p:
```

Beispiel 3. `lcoeff` evaluiert sein Ergebnis nicht vollständig:

```
>> p := poly(a*x^2 + 27*x, [x]): a := 5:
      lcoeff(p, [x]), eval(lcoeff(p, [x]))

      a, 5

>> delete p, a:
```

Beispiel 4. Es wird ein Polynom über dem Restklassenring modulo 7 definiert:

```
>> p := poly(3*x, [x], Dom::IntegerMod(7)): lcoeff(p)

      3 mod 7
```

Dieses Polynom kann nicht als Polynom in einer anderen Unbestimmten angesehen werden, da der „Koeffizient“ $3*x$ nicht als Element des Koeffizientenrings `Dom::IntegerMod(7)` interpretiert werden kann:

```
>> lcoeff(p, [y])

      FAIL

>> delete p:
```

Änderungen:

- ☞ Selbstdefinierte Termordnungen können nun vorgegeben werden.
 - ☞ Unbestimmte können nun auch für Polynome vom Domain-Typ `DOM_POLY` angegeben werden.
 - ☞ In früheren MuPAD Versionen war `lcoeff` eine Kernfunktion.
-

`ldegree` – der kleinste Grad der Terme eines Polynoms

`ldegree(p)` gibt den kleinsten totalen Grad der Terme des Polynoms `p` zurück.

`ldegree(p, x)` gibt den kleinsten Grad der Terme in `p` bezüglich der Unbestimmten `x` zurück.

Aufruf(e):

```

# ldegree(p)
# ldegree(p, x)
# ldegree(f <, vars>)
# ldegree(f <, vars>, x)

```

Parameter:

p — ein Polynom vom Typ `DOM_POLY`
f — ein polynomialer Ausdruck
vars — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
x — eine Unbestimmte

Rückgabewert: eine nicht-negative Zahl. `FAIL` wird zurückgeliefert, wenn die Eingabe nicht als Polynom interpretiert werden kann.

Überladbar durch: `p`, `f`

Verwandte Funktionen: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- # Wenn das erste Argument `f` nicht Element eines Polynom-Domains ist, so konvertiert `ldegree` diesen Ausdruck mittels `poly(f)` in ein Polynom. Ist eine Liste von Unbestimmten angegeben, so wird das Polynom `poly(f, vars)` betrachtet.
- # `ldegree(f, vars, x)` liefert das Ergebnis 0, falls `x` nicht in der Liste `vars` enthalten ist.
- # `ldegree` liefert 0 für das Null-Polynom.
- # `ldegree` ist eine Funktion des Systemkerns.

Beispiel 1. Der niedrigste im folgenden Polynom vorkommende totale Grad wird berechnet:

```
>> ldegree(x^3 + x^2*y^2)
```

3

Im nächsten Aufruf wird der Ausdruck als Polynom in `x` angesehen, dessen Koeffizienten den Parameter `y` enthalten:

```
>> ldegree(x^3 + x^2*y^2, x)
```


2

Der nächste Ausdruck wird als bi-variables Polynom in x und z aufgefasst, dessen Koeffizienten den Parameter y enthalten. Der totale Grad bezüglich x und z wird berechnet:

```
>> ldegree(x^3*z^2 + x^2*y^2*z, [x, z])
```

3

Der niedrigste vorkommende Grad bezüglich der Variable x wird berechnet:

```
>> ldegree(x^3*z^2 + x^2*y^2*z, [x, z], x)
```

2

Ein Polynom in den Unbestimmten x und z wird als konstant bezüglich jeder anderen Variable angesehen, d. h., der entsprechende Grad ist 0:

```
>> ldegree(poly(x^3*z^2 + x^2*y^2*z, [x, z]), y)
```

0

Änderungen:

⌘ Keine Änderungen.

length – die „Länge“ eines MuPAD Objektes (heuristische Komplexität)

`length(object)` liefert eine ganze Zahl, die die Komplexität des Objekts widerspiegelt.

Aufruf(e):

⌘ `length(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: eine nicht-negative ganze Zahl.

Verwandte Funktionen: `nops`, `op`

Details:

☞ Die (heuristische) Komplexität eines Objekts kann in Algorithmen nützlich sein, die die Komplexität und benötigte Zeit zur Manipulation von Objekten abschätzen müssen. Beispielsweise bevorzugt ein symbolischer Gauss-Algorithmus zur Lösung linearer Gleichungssysteme Pivot-Elemente geringer Komplexität.

☞ Die Komplexität wird folgendermaßen bestimmt:

- Objekte der Domain-Typen DOM_BOOL, DOM_DOMAIN, DOM_EXEC, DOM_FAIL, DOM_FLOAT, DOM_FUNC_ENV, DOM_IDENT, DOM_NIL, DOM_VAR und DOM_PROC_ENV werden als „atomar“ angesehen und haben die Länge 1. Insbesondere haben Bezeichner und reelle Gleitpunktzahlen die Länge 1.
- Die Länge einer ganzen Zahl ist die Anzahl der Dezimalziffern.
- Die Länge einer Zeichenkette ist die Anzahl der Zeichen.
- Die Länge zusammengesetzter Objekte wie komplexe Zahlen, rationale Zahlen, arithmetische Ausdrücke, Listen, Mengen, Arrays, Tabellen etc. ist die Summe der Längen der Operanden plus 1.

☞ `length()` liefert 0.

☞ `length` liefert *nicht* die Anzahl der Elemente oder Einträge von Mengen, Listen oder Tabellen. Hierfür ist `nops` zu benutzen!



☞ `length` ist eine Funktion des Systemkerns.

Beispiel 1. Intuitiv ist die Länge eines Objekts ein Maß für dessen Komplexität:

```
>> length(1 + x) < length(x^3 + exp(a - b)/ln(45 - t) - 1234*I)
3 < 25
```

Beispiel 2. Die Länge einiger einfacher Objekte wird bestimmt:

```
>> length(1.2), length(-1234.5), length(123456), length(-123456)
1, 1, 6, 6

>> length(17), length(123), length(17/123)
2, 3, 6

>> length(12), length(123), length(12 + 123*I)
```

```

2, 3, 6

>> length(x), length(x^2), length(x^12345)

1, 3, 7

>> length("123"), length("")

3, 0

>> length(x), length(a_long_name)

1, 1

```

Beispiel 3. Die Länge eines Arrays ist die Summe der Längen seiner Einträge plus 1:

```

>> A := array(1..2, [x, y]): length(A) = length(x) + length(y) + 1

3 = 3

>> A[1] := 12345: length(A) = length(12345) + length(y) + 1

7 = 7

>> delete A:

```

Beispiel 4. Die Operanden einer Tabelle sind die Gleichungen, die den Indizes die entsprechenden Einträge zuordnen. Die Länge jedes Operanden ist die Länge des Index plus der Länge des Eintrags plus 1:

```

>> T[1] := 45: T

table(
  1 = 45
)

>> length(T) = length(1 = 45) + 1

5 = 5

>> delete T:

```

Änderungen:

- ☞ `length` liefert nun auch die Länge von Zeichenketten. Die entsprechende Funktion `strlen` älterer MuPAD-Versionen wurde damit überflüssig.
-

`level` – Auswertung eines Objekts mit spezifizierter Substitutionstiefe

`level(object, n)` wertet `object` mit Substitutionstiefe `n` aus.

Aufruf(e):

- ☞ `level(object)`
- ☞ `level(object, n)`

Parameter:

- `object` — ein beliebiges MuPAD-Objekt
- `n` — a nicht-negative ganze Zahl kleiner als 2^{31}

Rückgabewert: das ausgewertete Objekt.

Weitere Dokumentation: Kapitel 5 des MuPAD Tutoriums.

Verwandte Funktionen: `context`, `eval`, `hold`, `indexval`, `LEVEL`, `MAXLEVEL`, `val`

Details:

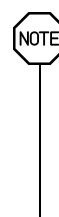
- ☞ Wird ein MuPAD Objekt ausgewertet, so werden alle Bezeichner die in dem Objekt enthalten sind durch ihre Werte ersetzt. Dies passiert rekursiv, d. h., enthalten die Werte selbst wieder Bezeichner, dann werden diese auch ersetzt. `level` dient daher zur Auswertung eines Objektes mit einer vorgegebenen Rekursionstiefe innerhalb dieses Ersetzungsprozesses.
- ☞ Durch `level(object, 0)` wird `object` ausgewertet, ohne in `object` vorkommende Bezeichner durch ihre Werte zu ersetzen. In den meisten Fällen ist dies identisch mit `hold(object)` und `object` wird unevaluiert zurückgegeben. Siehe Beispiel ??.
- ☞ Durch `level(object, 1)` werden alle in `object` vorkommenden Bezeichner durch ihre Werte ersetzt. Dies geschieht hier jedoch nicht rekursiv. Danach werden alle im Ergebnis der Ersetzung vorkommenden Funktionsaufrufe ausgeführt. Innerhalb von Prozeduren ist dies die Voreinstellung bei der Auswertung von MuPAD Objekten.

☞ `level(object)` ist äquivalent zu `level(object, MAXLEVEL)`. Alle in `object` vorkommenden Bezeichner werden bis zu einer Rekursionstiefe von `MAXLEVEL - 1` rekursiv durch Ihre Werte ersetzt. Beim Erreichen der Rekursionstiefe `MAXLEVEL` wird ein Fehler ausgelöst. Im allgemeinen führt dies zu einer vollständigen Evaluierung von `object`. Siehe Beispiel ??.

☞ `level` ohne zweites Argument kann dazu benutzt werden, um Objekte die keine lokalen Variablen oder formale Parameter enthalten innerhalb einer Prozedur vollständig zu evaluieren. Dies kann notwendig werden, da Objekte innerhalb von Prozeduren standardmäßig mit der Substitutionstiefe 1 ausgewertet werden. Siehe Beispiel ??.

Die Benutzung von `level` sollte ansonsten nie notwendig sein.

☞ `level` beeinflusst nicht die Auswertung von lokalen Variablen und formalen Parametern des Typs `DOM_VAR` innerhalb von Prozeduren. Treten solche lokalen Variablen innerhalb von `object` auf, so werden sie immer durch ihren Wert ersetzt. Dies ist unabhängig vom Wert von `n` und der so erhaltene Wert wird nicht weiter rekursiv ersetzt. Siehe Beispiel ??.



☞ `level` setzt zeitweilig den Wert der Variablen `LEVEL` auf `n`, oder auf $2^{31} - 1$ wenn `n` nicht angegeben wurde. In beiden Fällen bleibt der Wert von `MAXLEVEL` jedoch unverändert. Wird während einer Evaluierung die Substitutionstiefe `MAXLEVEL` erreicht, so wird eine Fehlermeldung ausgegeben. Weitere Informationen zu den Umgebungsvariablen `LEVEL` und `MAXLEVEL` findet man auf den entsprechenden Hilfeseiten.

☞ Im Gegensatz zu den meisten anderen Funktionen wird das erste Argument von `level` nicht ausgeglichen, wenn es sich um eine Ausdrucksfolge handelt. Siehe Beispiel ??.

☞ `level` arbeitet auf Arrays, Tabellen, Matrizen oder Polynomen nicht rekursiv. Zum Evaluieren der Einträge eines Arrays, einer Tabelle oder einer Matrix muss der Aufruf `map(object, eval)` benutzt werden. Zum Evaluieren der Koeffizienten eines Polynoms ist der Aufruf `mapcoeffs(object, eval)` zu benutzen. Siehe Beispiel ?? und Beispiel ??.

Weitere Informationen bezüglich der Auswertung von Arrays, Tabellen, Matrizen oder Polynomen findet man auf der `eval` Hilfeseite.

☞ Die maximale Substitutionstiefe von `level` hängt von der Umgebungsvariablen `MAXLEVEL` ab. Die maximale Substitutionstiefe der Funktion `eval` hängt stattdessen von der Umgebungsvariablen `LEVEL` ab. Siehe Beispiel ??.

☞ Da `eval` sein Ergebnis erneut evaluiert existiert ein Unterschied zwischen der Auswertung von Ausdrücken mit Substitutionstiefe `n` durch `level` im Vergleich zu der Auswertung mittels `eval`. Siehe Beispiel ??.

- ☞ Wie bereits erwähnt beeinflusst `level` die Auswertung von lokalen Variablen und formalen Parameters des Typs `DOM_VAR` innerhalb von Prozeduren nicht. `eval` verhält sich hier unterschiedlich. Weitere Information findet man im Beispiel ?? und auf der `eval` Hilfeseite.
- ☞ Das Ergebnis von `level(hold(x))` ist immer `x`, da `x` das Ergebnis der vollständigen Auswertung von `hold(x)` ist. Dies gilt nicht für `eval(hold(x))`, denn `eval` wertet erst sein Argument und anschließend das Ergebnis erneut aus.
- ☞ Die Auswertung von Elementen eines benutzerdefinierten Domains hängt von der Implementation des Domains ab. Normalerweise werden die Domain-Elemente durch `level` nicht weiter evaluiert. Wenn das Domain einen Slot "evaluate" besitzt, wird die entsprechende Slot-Routine mit dem Domain-Element als Argument bei jeder Evaluierung aufgerufen, also einmal bei einer Ausführung von `level`. Siehe Beispiel ??.
- ☞ `level` ist eine Funktion des Systemkerns.

Beispiel 1. Die Wirkung von `level` wird für einige unterschiedliche Werte des zweiten Parameters demonstriert:

```
>> delete a0, a1, a2, a3, a4, b: b := b + 1:
      a0 := a1: a1 := a2 + 2: a2 := a3 + a4: a3 := a4^2: a4 := 5:

>> hold(a0), hold(a0 + a2), hold(b);
      level(a0, 0), level(a0 + a2, 0), level(b, 0);
      level(a0, 1), level(a0 + a2, 1), level(b, 1);
      level(a0, 2), level(a0 + a2, 2), level(b, 2);
      level(a0, 3), level(a0 + a2, 3), level(b, 3);
      level(a0, 4), level(a0 + a2, 4), level(b, 4);
      level(a0, 5), level(a0 + a2, 5), level(b, 5);
      level(a0, 6), level(a0 + a2, 6), level(b, 6);

      a0, a0 + a2, b

      a0, a0 + a2, b

      a1, a1 + a3 + a4, b + 1

      a2 + 2, a2 + a42 + 7, b + 2

      a3 + a4 + 2, a3 + a4 + 32, b + 3

      a42 + 7, a42 + 37, b + 4
```

32, 62, b + 5

32, 62, b + 6

Die Auswertung von `object` durch die Eingabe von `object` auf der interaktiven Ebene ist äquivalent zu `level(object, LEVEL)`:

```
>> LEVEL := 2: MAXLEVEL := 4: a0, a2, b;  
    level(a0, LEVEL), level(a2, LEVEL), level(b, LEVEL)
```

2
a2 + 2, a4 + 5, b + 2

2
a2 + 2, a4 + 5, b + 2

Fehlt das zweite Argument beim Aufruf von `level` so entspricht dies einer vollständigen Evaluation mit Substitutionstiefe `MAXLEVEL - 1`:

```
>> level(a0)  
  
Error: Recursive definition [See ?MAXLEVEL]
```

```
>> level(a2)  
  
30
```

```
>> level(b)  
  
Error: Recursive definition [See ?MAXLEVEL]
```

```
>> delete LEVEL, MAXLEVEL:
```

Beispiel 2. Es wird das Verhalten von `level` innerhalb von Prozeduren gezeigt:

```
>> delete a, b, c: a := b: b := c: c := 42:  
    p := proc()  
        local x;  
        begin  
            x := a:  
            print(level(x, 0), x, level(x, 2), level(x)):  
            print(level(a, 0), a, level(a, 2), level(a)):  
        end_proc:  
    p()
```

b, b, b, b

a, b, c, 42

Da a mit der voreingestellten Substitutionstiefe 1 evaluiert wird, setzt die Zuweisung $x := a$ den Wert der lokalen Variablen x auf den nicht evaluierten Bezeichner b . Man sieht das jede Auswertung von x , sei es mit oder ohne Benutzung der Funktion `level`, x durch seinen aktuellen Wert b ersetzt. Hierbei findet keine weitere rekursive Auswertung von x statt. Im Gegensatz hierzu, wird die Auswertung des Bezeichners a mit der voreingestellten Substitutionstiefe 1 ausgeführt, jedoch wertet `level(a, 2)` den Bezeichner mit der Substitutionstiefe 2 aus.

`level` ohne Benutzung eines zweiten Argumentes kann daher zur vollständigen Auswertung eines Objektes, dass keine lokalen Variablen oder formalen Parameter enthält, benutzt werden.

Beispiel 3. In einigen seltenen Fällen verhalten sich `level(object, 0)` und `hold(object)` unterschiedlich. Dies ist der Fall, wenn `object` kein Bezeichner, also z.B. eine namenslose Funktion ist, denn `level` beeinflusst lediglich die Auswertung von Bezeichnern:

```
>> level((x -> x^2)(2), 0), hold((x -> x^2)(2))
4, (x -> x^2)(2)
```

Aus dem gleichen Grunde verhalten sich `level(object, 0)` und `hold(object)` unterschiedlich, wenn `object` eine lokale Variable oder Prozedur ist:

```
>> f:=proc() local x; begin
    x := 42;
    hold(x), level(x, 0);
end_proc;
f();
delete f;

DOM_VAR(0, 2), 42
```

Beispiel 4. Im Gegensatz zu Listen und Mengen werden bei der Auswertung eines Arrays dessen Elemente nicht evaluiert. `level` hat daher auch keine Auswirkung auf die Evaluierung von Arrays. Das gleiche gilt für Tabellen und Matrizen. Um die Einträge eines Arrays zu evaluieren muss die Funktion `map` benutzt werden. Auf der `eval` Hilfeseite findet man hierzu weitere Beispiele:

```
>> delete a, b:
L := [a, b]: A := array(1..2, L): a := 1: b := 2:
L, A, level(A), map(A, level), map(A, eval)

      +-      -+  +-      -+  +-      -+  +-      -+
[1, 2], | a, b |, | a, b |, | a, b |, | 1, 2 |
      +-      -+  +-      -+  +-      -+  +-      -+
```


Beispiel 5. Das erste Argument von `level` darf eine Ausdruckssequenz sein. Diese wird nicht ausgeglichen. Die Ausdruckssequenz muss jedoch geklammert sein:

```
>> delete a, b: a := b: b := 3:
      level((a, b), 1);
      level(a, b, 1)

                        b, 3
Error: Wrong number of arguments [level]
```

Beispiel 6. Polynome werden bei Evaluierung nicht verändert, und auch `level` bleibt ohne Wirkung:

```
>> delete a, x: p := poly(a*x, [x]): a := 2: x := 3:
      p, level(p)

      poly(a x, [x]), poly(a x, [x])
```

Man benutze die Funktionen `mapcoeffs` und `eval`, um alle Koeffizienten eines Polynoms zu evaluieren:

```
>> mapcoeffs(p, eval)

      poly(2 x, [x])
```

Zur Ersetzung einer Unbestimmten `x` durch einen Wert dient die Funktion `evalp`:

```
>> delete x: evalp(p, x = 3)

      3 a
```

Das Ergebnis der Funktion `evalp` kann unevaluierte Bezeichner enthalten, die durch anschließende Anwendung der Funktion `eval` ausgewertet werden können. Es ist hier notwendig `eval` anstelle der Funktion `level` zu verwenden, da `level` sein Ergebnis nicht evaluiert:

```
>> eval(evalp(p, x = 3))

      6
```

Beispiel 7. Es wird nun der feine Unterschied zwischen den Funktionen `level` und `eval` gezeigt. Die Auswertungstiefe von `eval` wird durch die Umgebungsvariable `LEVEL` begrenzt. `level` beachtet `LEVEL` nicht. Stattdessen evaluiert `level` sein erstes Argument so oft wie es das zweite Argument angibt oder bis dieses vollständig evaluiert ist:

```
>> delete a0, a1, a2, a3:
      a0 := a1 + a2: a1 := a2 + a3: a2 := a3^2 - 1: a3 := 5:
      LEVEL := 1:
      eval(a0), level(a0);
```

$$a2 + a3 + a3^2 - 1, 53$$

Überschreitet die Auswertungstiefe den Wert von `MAXLEVEL`, so wird in beiden Fällen ein Fehler ausgelöst:

```
>> delete LEVEL:
      MAXLEVEL := 3:
      level(a0);

Error: Recursive definition [See ?MAXLEVEL]
```

```
>> delete LEVEL:
      MAXLEVEL := 3:
      eval(a0);
      delete MAXLEVEL:

Error: Recursive definition [See ?MAXLEVEL]
```

Es ist nicht identisch wenn ein Ausdruck `ex` durch `eval` bei gegebener Auswertungstiefe `n`, oder durch `level((ex, n))` evaluiert wird. Dies liegt daran, dass `eval` sein Ergebnis noch einmal auswertet:

```
>> LEVEL := 2: eval(a0), level(a0, 2);
      delete LEVEL:

53, a2 + a3 + a3^2 - 1
```

`level` beeinflusst die Auswertung von lokalen Variablen des Typs `DOM_VAR` nicht. `eval` wertet sie hingegen mit der Auswertungstiefe `LEVEL` aus. Innerhalb einer Prozedur ist diese eins:

```
>> p := proc()
      local x;
      begin
        x := a0:
        print(eval(x), level(x)):
      end_proc:
      p()
```

$$a_2 + a_3 + a_3^2 - 1, a_1 + a_2$$

Beispiel 8. Die Evaluierung eines Elements eines benutzerdefinierten Domains hängt von der Implementation des Domains ab. Normalerweise wird so ein Objekt nicht weiter evaluiert:

```
>> delete a: T := newDomain("T"):
      e := new(T, a): a := 1:
      e, level(e), map(e, level), val(e)

      new(T, a), new(T, a), new(T, a), new(T, a)
```

Wenn der Slot "evaluate" existiert, so wird die entsprechende Slot-Routine jedesmal aufgerufen, wenn ein Domain-Element ausgewertet wird. Wir implementieren die Routine `T::evaluate`, die einfach alle internen Operanden ihres Arguments auswertet, für das Domain `T`. Auf das unevaluierte Domain-Element kann immer noch mit `val` zugegriffen werden:

```
>> T::evaluate := x -> new(T, eval(extop(x))):
      e, level(e), map(e, level), val(e);

      new(T, 1), new(T, 1), new(T, 1), new(T, a)

>> delete e, T:
```

Änderungen:

- ☞ `level` beeinflusst nicht mehr die Auswertung von lokalen Variablen und formalen Parametern, für welche der neue Datentyp `DOM_VAR` eingeführt wurde. Weitergehende Informationen findet man in den Abschnitten „Das LEVEL-Problem“ und „Symbole und Variablen“ des Dokumentes „Von MuPAD 1.4 zu MuPAD 2.0“.

lhs, rhs – die linke bzw. rechte Seite von Gleichungen, Ungleichungen, Relationen und Bereichen

`lhs(f)` liefert die linke Seite (engl.: left hand side) von `f`.

`rhs(f)` liefert die rechte Seite (engl.: right hand side) von `f`.

Aufruf(e):

- ☞ `lhs(f)`
- ☞ `rhs(f)`

Parameter:

f — eine Gleichung $x = y$, eine Ungleichung $x <> y$, eine Relation $x < y$, eine Relation $x \leq y$ oder ein Bereich $x..y$

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: f

Verwandte Funktionen: `op`

Details:

⌘ Die Aufrufe `lhs(f)` bzw. `rhs(f)` sind äquivalent zu den direkten Aufrufen `op(f, 1)` bzw. `op(f, 2)` der Operandenfunktion `op`.

Beispiel 1. Die linke bzw. rechte Seite einiger Objekte wird extrahiert:

```
>> lhs(x = sin(2)), lhs(3.14 <> PI), lhs(x + 3 < 2*y),
    rhs(a <= b), rhs(m-1..n+1)

x, 3.14, x + 3, b, n + 1
```

Die Operanden eines Ausdrucks hängen von der internen Darstellung ab. So werden beispielsweise „größer“-Relationen immer in entsprechende „kleiner“-Relationen umgewandelt:

```
>> y > -infinity; lhs(y > -infinity)

-infinity < y

-infinity

>> y >= 4; rhs(y >= 4)

4 <= y

y
```

Beispiel 2. Die linken und rechten Seiten der Lösung des folgenden Systems werden extrahiert:

```
>> s := solve({x + y = 1, 2*x - 3*y = 2})

{[x = 1, y = 0]}

>> map(op(s), lhs) = map(op(s), rhs)
```

$$[x, y] = [1, 0]$$

Aufrufe der Funktionen `lhs` und `rhs` sind leichter lesbar als die äquivalenten Aufrufe der Operandenfunktion `op`:

```
>> map(op(s), op, 1) = map(op(s), op, 2)
```

$$[x, y] = [1, 0]$$

Innerhalb von Prozeduren sind jedoch direkte `op`-Aufrufe aus Laufzeitgründen vorzuziehen.

```
>> delete s:
```

Änderungen:

⌘ `lhs` und `rhs` sind neue Funktionen.

limit – Grenzwertbestimmung

`limit(f, x = x0)` berechnet den (beidseitigen) Grenzwert $\lim_{\substack{x \rightarrow x_0 \\ x - x_0 \in \mathbb{R} \setminus \{0\}}} f(x)$.

`limit(f, x = x0, Left)` berechnet den linksseitigen Grenzwert $\lim_{\substack{x \rightarrow x_0 \\ x < x_0}} f(x)$.

`limit(f, x = x0, Right)` berechnet den rechtsseitigen Grenzwert $\lim_{\substack{x \rightarrow x_0 \\ x > x_0}} f(x)$.

Aufruf(e):

⌘ `limit(f, x <= x0> <, dir>)`

Parameter:

- `f` — ein arithmetischer Ausdruck, der eine Funktion in `x` repräsentiert
- `x` — ein Bezeichner
- `x0` — Grenzpunkt: ein arithmetischer Ausdruck (ggf. `infinity` oder `-infinity`)

Optionen:

`dir` — entweder `Left` oder `Right`. Hierdurch wird die Richtung der Grenzwertbildung festgelegt.

Rückgabewert: ein arithmetischer Ausdruck, ein Intervall vom Typ `Dom : Interval`, ein Ausdruck vom Typ `"limit"` oder der Wert `FAIL`.

Seiteneffekte: Die Ergebnisse der Funktion hängen vom Wert der Umgebungsvariablen `ORDER` ab, die den Grad der Reihenberechnungen angibt (siehe `series` und das Beispiel ?? unten).

Mittels `assume` gesetzte Eigenschaften von Bezeichnern werden berücksichtigt.

Überladbar durch: `f`

Verwandte Funktionen: `asympt`, `diff`, `discont`, `int`, `O`, `series`, `taylor`

Details:

- ⌘ `limit(f, x = x0)` berechnet den zweiseitigen Grenzwert von `f` für `x` gegen `x0` entlang der reellen Achse. Ist `x0` nicht angegeben, so wird der Grenzwert von `f` für `x` gegen 0 bestimmt.
Ist der Grenzpunkt `x0` gleich ∞ oder $-\infty$, so wird der Grenzwert von links gegen ∞ bzw. von rechts gegen $-\infty$ bestimmt.
Stimmen linksseitiger und rechtsseitiger Grenzwert nicht überein, so wird `undefined` zurückgeliefert.
 - ⌘ `limit(f, x = x0, Left)` bestimmt den linksseitigen Grenzwert von `f` für `x` gegen `x0`. `limit(f, x = x0, Right)` bestimmt den rechtsseitigen Grenzwert. Siehe Beispiel ??.
 - ⌘ Wenn der Grenzwert nicht existiert, `limit` jedoch feststellen kann, dass die Funktion `f` beschränkt ist wenn `x` gegen `x0` strebt, so wird als Ergebnis ein Intervall vom Typ `Dom :: Interval` zurückgeliefert. Das kann beispielsweise der Fall sein, wenn `f` in `x0` eine (beschränkte) Oszillationsstelle besitzt (siehe Beispiel ??). Die Grenzen des Intervalls sind der Limes Inferior und Limes Superior von `f` für $x \rightarrow x_0$.
 - ⌘ Kann `limit` den Grenzwert nicht bestimmen, so wird ein symbolischer `limit`-Aufruf zurückgeliefert (siehe Beispiel ??).
 - ⌘ Enthält `f` Parameter, so werden ggf. Eigenschaften (positiv, negativ etc.) der Parameter bei der Grenzwertbestimmung berücksichtigt, die den entsprechenden Bezeichnern mit der Funktion `assume` zugewiesen werden können (siehe Beispiel ??). Kann der Grenzwert ohne zusätzliche Annahmen über die Parameter nicht bestimmt werden, so liefert `limit` entsprechende Hinweise in Form von Warnungen.
 - ⌘ Intern verwendet `limit` die Funktion `series` zur Grenzwertbestimmung und liefert `FAIL` zurück, falls die Anzahl an Termen, die zur Reihenberechnung herangezogen werden, nicht groß genug ist. In diesem Fall muss der Wert der Umgebungsvariablen `ORDER` entsprechend vergrößert werden, um den Grenzwert bestimmen zu können (siehe Beispiel ??).
-

Beispiel 1. Folgende Eingabe bestimmt den Grenzwert $\lim_{x \rightarrow 0} \frac{1 - \cos x}{x^2}$:

```
>> limit((1 - cos(x))/x^2, x)

1/2
```

Eine mögliche Definition von e ist über den Grenzwert der Folge $(1 + \frac{1}{n})^n$ für $n \rightarrow \infty$ gegeben:

```
>> limit((1 + 1/n)^n, n = infinity)

exp(1)
```

Ein etwas komplexeres Beispiel:

```
>> limit(
    (exp(x*exp(-x)/(exp(-x) + exp(-2*x^2/(x+1)))) - exp(x))/x,
    x = infinity
)

-exp(2)
```

Beispiel 2. Der beidseitige Grenzwert von $f(x) = 1/x$ für $x \rightarrow 0$ existiert nicht:

```
>> limit(1/x, x = 0)

undefined
```

Der links- und rechtsseitige Grenzwert kann durch Angabe der Optionen *Left* und *Right* bestimmt werden:

```
>> limit(1/x, x = 0, Left),
    limit(1/x, x = 0, Right)

-infinity, infinity
```

Beispiel 3. Ist `limit` nicht in der Lage, den Grenzwert zu bestimmen, so wird ein symbolischer `limit`-Aufruf zurückgeliefert:

```
>> delete f: limit(f(x), x = infinity)

limit(f(x), x = infinity)
```

Beispiel 4. $\sin(x)$ oszilliert für wachsende x , die Funktionswerte liegen alle-
samt im Intervall $[-1, 1]$. Der Limes Inferior und Limes Superior ist -1 bzw.
 1 :

```
>> limit(sin(x), x = infinity)

[-1, 1]
```

Liefert `limit` ein Intervall als Ergebnis, so läßt sich daraus nicht notwendi-
gerweise folgern, dass die Grenzfunktion jeden Wert des Bereiches unendlich
oft annimmt, auch wenn das in diesem Beispiel der Fall ist!

Beispiel 5. `limit` kann den Grenzwert von x^n ohne zusätzliche Annahmen
an den Parameter n nicht bestimmen:

```
>> delete n: limit(x^n, x = infinity)

Warning: cannot determine sign of n [stdlib::limit::limitMRV]

      n
  limit(x , x = infinity)
```

Nehmen wir $n > 0$ an, so wissen wir, dass $\lim_{n \rightarrow \infty} x^n = \infty$ gilt. Wir verwenden
`assume`, um die entsprechende Annahme zu treffen:

```
>> assume(n > 0): limit(x^n, x = infinity)

infinity
```

Ist $n < 0$, so ist $\lim_{n \rightarrow \infty} x^n = 0$:

```
>> assume(n < 0): limit(x^n, x = infinity)

0
```

Beispiel 6. Zur Grenzwertbestimmung kann eine Erhöhung der Umgebungs-
variablen `ORDER` notwendig sein, wie das folgende Beispiel zeigt:

```
>> limit((sin(tan(x)) - tan(sin(x)))/x^7, x = 0)

Warning: ORDER seems to be not big enough for series \
computation [stdlib::limit::lterm]

FAIL

>> ORDER := 8: limit((sin(tan(x)) - tan(sin(x)))/x^7, x)

-1/30
```


Hintergründe:

⌘ In einigen Fällen, wie unter Details beschrieben und in den Beispielen ?? und ?? gezeigt, liefert `limit` Informationen in Form von Warnungen, falls eine Grenzwertberechnung nicht durchgeführt werden konnte. Wenn man `limit` von einer eigenen Prozedur aus aufruft, dann möchte man solche Warnungen manchmal unterdrücken. Dies kann mit Hilfe der Prozedur `stdlib::limit::printWarnings` gesteuert werden.

Die Aufrufe `stdlib::limit::printWarnings(TRUE)` bzw. `stdlib::limit::printWarnings(FALSE)` schalten die von `limit` ausgegebenen Warnungen an bzw. aus und liefern den zuvor geltenden Zustand des Schalters zurück. Der Befehl `stdlib::limit::printWarnings()` liefert den aktuellen Zustand zurück. Die Standardeinstellung ist `TRUE`.

⌘ `limit` versucht zuerst mittels reiner Reihenberechnung, den Grenzwert zu bestimmen. Wenn das fehlschlägt, wird ein Algorithmus verwendet, der auf der Dissertation von Dominik Gruntz basiert: „On Computing Limits in a Symbolic Manipulation System“, ETH Zürich, 1995.

Änderungen:

⌘ `limit` kann als Ergebnis ein Intervall vom Typ `Dom::Interval` zurückliefern.

`linsolve` – Lösen von linearen Gleichungssystemen

`linsolve(eqs, vars)` löst ein System von linearen Gleichungen nach den Unbekannten `vars`.

Aufruf(e):

```
⌘ linsolve(eqs)
⌘ linsolve(eqs, vars)
⌘ linsolve(eqs, vars, ShowAssumptions)
⌘ linsolve(eqs, vars, Domain = R)
```

Parameter:

`eqs` — eine Liste oder Menge von linearen Gleichungen oder arithmetischen Ausdrücken
`vars` — eine Liste oder Menge von Unbekannten nach denen aufgelöst wird: Gewöhnlich sind dies Bezeichner oder indizierte Bezeichner

Optionen:

- ShowAssumptions* — liefert zusätzliche Informationen über intern gemachte Annahmen von symbolischen Parametern in eqs
- Domain= R* — löst das System über dem Körper R , d. h. einem Domain der Kategorie `Cat::Field`.

Rückgabewert: Ohne die Option *ShowAssumptions* wird eine Liste von vereinfachten Gleichungen zurückgegeben, welche die allgemeine Lösung des Systems eqs repräsentiert. Falls das System nicht lösbar ist, wird FAIL zurückgegeben.


Mit der Option *ShowAssumptions* bekommt man eine Liste [Lösung, Nebenbedingungen, Pivotelemente]. Lösung ist eine Liste von vereinfachten Gleichungen, welche die allgemeine Lösung repräsentieren. Die Listen Nebenbedingungen und Pivotelemente enthalten Gleichungen und Ungleichungen mit symbolischen Parametern aus eqs. Intern wird während der Systemumformungen angenommen, dass diese alle erfüllt sind.

Verwandte Funktionen: `linalg::matlinsolve`, `numeric::linsolve`, `solve`

Details:

- ⌘ `linsolve(eqs, <, vars <, ShowAssumptions>>)` löst das System eqs linearer Gleichungen nach den Unbekannten vars. Sind keine Unbekannten spezifiziert, dann wird nach allen Unbekannten in eqs gelöst. Dabei werden die Unbekannten intern durch `indets(eqs, PolyExpr)` bestimmt.
- ⌘ `linsolve(eqs, vars, Domain = R)` löst das System über dem Domain R , welches ein Körper sein muss, d.h. ein Domain der Kategorie `Cat::Field`.
- ⌘ Jedes Element von eqs muss entweder eine Gleichung oder ein arithmetischer Ausdruck f sein; f wird als Gleichung $f = 0$ betrachtet.
- ⌘ Die Unbekannten in vars müssen nicht notwendig Bezeichner oder indizierte Bezeichner sein. Es sind auch Ausdrücke wie $\sin(x)$, $f(x)$ oder $y^{(1/3)}$ möglich. Genauer, jeder Ausdruck, welcher von `poly` als Unbestimmte akzeptiert wird, ist eine gültige Unbekannte.
- ⌘ Falls das System lösbar ist und die Option *ShowAssumptions* nicht gesetzt wurde, ist der Rückgabewert eine Liste von Gleichungen der Form `var=Wert`, wobei var eine Unbekannte aus vars ist und Wert ein arithmetischer Ausdruck, welcher keine Unbekannte enthält, die auf den linken Seiten der Rückgabegleichungen auftritt. Beachten Sie: Falls die Mannigfaltigkeit der Lösung eine Dimension größer als Null hat, dann treten einige der Unbekannten aus vars auf den rechten Seiten der

Rückgabegleichungen auf und zeigen die Freiheitsgrade des Systems an. Siehe Beispiel ??.

- ⌘ Ist `vars` eine Liste von Unbekannten, wird damit implizit eine Ordnung der Unbekannten festgelegt. In der allgemeinen Lösung wird dann diese Ordnung beibehalten.
- ⌘ Die Funktion `linsolve` löst ausschließlich Systeme linearer Gleichungen. Für den allgemeinen Fall bzw. für die Lösung nicht-linearer Systeme steht die Funktion `solve` zur Verfügung.
- ⌘ `linsolve` ist eine Schnittstellenfunktion zu den beiden Prozeduren `numeric::linsolve` und `linalg::matlinsolve`. Näheres findet man auf den Hilfeseiten zu `numeric::linsolve` und `linalg::matlinsolve` und im Abschnitt ‚Hintergründe‘ dieser Hilfeseite.
- ⌘ Das System `eqs` wird auf Linearität getestet. Da ein solcher Test aufwändig sein kann, wird in den Fällen in denen die Linearität bekannt ist, empfohlen, die Funktionen `numeric::linsolve` und `linalg::matlinsolve` direkt zu benutzen.
- ⌘ Diese Funktion reagiert *nicht* auf mittels `assume` gesetzte Eigenschaften. 

Option `<ShowAssumptions>`:

- ⌘ Mit dieser Option wird eine Liste [Lösung, Nebenbedingungen, Pivotelemente] zurückgegeben. Dabei ist Lösung eine Liste von vereinfachten Gleichungen, welche, wie weiter oben beschrieben, die allgemeine Lösung von `eqs` repräsentiert. Die Listen Nebenbedingungen und Pivotelemente enthalten Gleichungen und Ungleichungen mit den in `eqs` vorkommenden Parametern. Intern wird angenommen, dass diese Gleichungen und Ungleichungen erfüllt sind. Falls das System nicht lösbar ist, wird `[FAIL, [], []]` zurückgegeben. Näheres findet man auf der Hilfeseite zu `numeric::linsolve`.

Beispiel 1. Sowohl Gleichungen als auch Variablen können als Mengen oder Listen eingegeben werden:

```
>> linsolve({x + y = 1, 2*x + y = 3}, {x, y}),  
    linsolve({x + y = 1, 2*x + y = 3}, [x, y]),  
    linsolve([x + y = 1, 2*x + y = 3], {x, y}),  
    linsolve([x + y = 1, 2*x + y = 3], [x, y])  
  
[y = -1, x = 2], [x = 2, y = -1], [y = -1, x = 2],  
  
[x = 2, y = -1]
```

Es sind auch Ausdrücke als Variablen möglich:

```
>> linsolve({cos(x) + sin(x) = 1, cos(x) - sin(x) = 0},
             {cos(x), sin(x)})

[cos(x) = 1/2, sin(x) = 1/2]
```

Weiter kann man indizierte Bezeichner verwenden:

```
>> linsolve({2*a[1] + 3*a[2] = 5, 7*a[2] + 11*a[3] = 13,
             17*a[3] + 19*a[1] = 23}, {a[1], a[2], a[3]})

[a[1] = 691/865, a[2] = 981/865, a[3] = 398/865]
```

Nun wird die Benutzung der Option *Domain* demonstriert und mit ihr ein System über dem Körper \mathbb{Z}_{23} gelöst:

```
>> linsolve([2*x + y = 1, -x - y = 0],
             Domain=Dom::IntegerMod(23))

[y = 22 mod 23, x = 1 mod 23]
```

Das folgende System besitzt keine Lösung:

```
>> linsolve({x+y=1, 2*x+2*y=3}, {x,y})

FAIL
```

Beispiel 2. Im Folgenden wird die Abhängigkeit der Lösung eines Systems von auftretenden Parametern gezeigt:

```
>> eqs := [x + a*y = b, x + A*y = b]:

>> linsolve(eqs, [x, y])

[x = b, y = 0]
```

Beachten Sie, dass für $a = A$ dies nicht die allgemeine Lösung ist. Durch Benutzen der Option *ShowAssumptions* zeigt sich, dass obiges Ergebnis die allgemeine Lösung unter der Bedingung $a \neq A$ ist:

```
>> linsolve(eqs, [x, y], ShowAssumptions)

[[x = b, y = 0], [], [A - a <> 0]]

>> delete eqs:
```

Beispiel 3. Falls die Lösung des linearen Gleichungssystems nicht eindeutig ist, werden einige der Unbekannten als „freie Parameter“ zum Aufspannen des Lösungsraumes benutzt. Im folgenden Beispiel ist die Unbekannte z solche ein Parameter. Er tritt nicht auf den linken Seiten der gelösten Gleichungen auf:

```
>> eqs := [x + y = z, x + 2*y = 0, 2*x - z = -3*y, y + z = 0]:
>> vars := [w, x, y, z]:
>> linsolve(eqs, vars)

[x = 2 z, y = -z]
```

Hintergründe:

- ☞ Wird die Option *Domain* nicht angewandt, so wird das System durch einen Aufruf von `numeric::linsolve` mit Option *Symbolic* gelöst.
- ☞ Wird die Option *Domain* = \mathbb{R} benutzt und ist \mathbb{R} eines der beiden Domains `Dom::ExpressionField()` oder `Dom::Float`, so wird `numeric::linsolve` benutzt um das System zu lösen. Diese Funktion nutzt eine dünn besetzte Darstellung der Gleichungen.

Andernfalls wird `eqs` erst in eine Matrix konvertiert und anschließend durch `linalg::matlinsolve` gelöst. Eine möglicherweise dünn besetzte Struktur des gegebenen Systems wird hierbei nicht ausgenutzt.

Änderungen:

- ☞ Das System `eqs` wird nun auf Linearität bzgl. `vars` überprüft.
- ☞ Neue Option *ShowAssumptions*.
- ☞ Neue Schreibweise: *Domain* = \mathbb{R} .

lllint – LLL-Gitterbasisreduktion

`lllint(A)` wendet den LLL-Algorithmus auf die Spalten der (nicht notwendigerweise quadratischen) Matrix A mit ganzzahligen Einträgen an.

Aufruf(e):

- ☞ `lllint(A, All)`
- ☞ `lllint(A)`

Parameter:

A — eine Matrix, gegeben als eine Liste von Zeilenvektoren, wobei jede Zeile eine Liste ganzer Zahlen ist

Rückgabewert:

- ☞ Ist die Option `All` angegeben, dann gibt `lllint` eine Liste `[T, B]` zurück, so dass $B = A \cdot T$ gilt und die Spalten von `B` eine LLL-reduzierte Basis des ganzzahligen Gitters bilden, das von den Spalten von `A` erzeugt wird. `B` und `T` werden als Listen von Zeilenvektoren zurückgeliefert.
- ☞ Ohne die Option `All` gibt `lllint` lediglich die Transformationsmatrix `T` in Form einer Liste von Zeilenvektoren zurück.

Verwandte Funktionen: `linalg::basis`, `linalg::factorLU`, `linalg::factorQR`, `linalg::gaussElim`, `linalg::hermiteForm`, `linalg::orthog`

Details:

- ☞ `lllint` wendet den LLL-Algorithmus auf die Spalten der Matrix `A` an. Mathematisch kann die Eingabematrix eine beliebige Matrix mit ganzzahligen Einträgen sein. Sie muss weder quadratisch sein noch vollen Spaltenrang haben.
Die Matrix ist an `lllint` in Form einer Liste von Zeilenvektoren zu übergeben, wobei jeder Zeilenvektor wiederum eine Liste ganzer Zahlen ist. Die Anzahl der Einträge muss in allen Zeilen dieselbe sein. Die von `lllint` zurückgelieferten Matrizen haben ebenfalls diese Form.
Die Berechnungen werden ausschließlich mit ganzen Zahlen durchgeführt und sind sowohl akkurat als auch effizient.
- ☞ Man kann `matrix` benutzen, um eine schönere Bildschirmausgabe für die vorkommenden Matrizen zu erhalten. Siehe Beispiel ??.
- ☞ `lllint` ist eine Funktion des Systemkerns.

Beispiel 1. Wir wenden den LLL-Algorithmus auf eine Matrix mit zwei Zeilen und drei Spalten an:

```
>> A := [[1, 2, 3], [4, 5, 6]]:
      [T, B] := lllint(A, All)

      [[[-1, 4], [1, -3], [0, 0]], [[1, -2], [1, 1]]]
```

Wir verwenden `matrix`, um `A`, `T` und `B` in lesbarer Form auszugeben, und überprüfen die Gültigkeit der Gleichung $B = A \cdot T$:

```
>> matrix(A), matrix(T), matrix(B)
```

$$\begin{array}{ccc|ccc|ccc}
& & & & +- & & & -+ & & & \\
+- & & & -+ & | & -1, & 4 & | & +- & & -+ \\
| & 1, & 2, & 3 & | & & & | & | & 1, & -2 & | \\
| & & & & | & 1, & -3 & | & | & & & | \\
| & 4, & 5, & 6 & | & & & | & | & 1, & 1 & | \\
+- & & & -+ & | & 0, & 0 & | & +- & & -+ \\
& & & & +- & & & -+ & & & &
\end{array}$$

```
>> matrix(B) = matrix(A)*matrix(T)
```

$$\begin{array}{ccc|ccc|ccc}
& & & & +- & & -+ & & +- & & -+ \\
| & 1, & -2 & | & | & 1, & -2 & | & | & & & | \\
| & & & | & = & | & & | & | & & & | \\
| & 1, & 1 & | & | & 1, & 1 & | & | & & & | \\
+- & & & -+ & +- & & -+ & -+ & & & &
\end{array}$$

Das Ergebnis ist wie folgt zu interpretieren: die beiden Spaltenvektoren

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \text{ und } \begin{pmatrix} -2 \\ 1 \end{pmatrix}$$

bilden eine LLL-reduzierte Basis des ganzzahligen Gitters, das von den drei Spaltenvektoren

$$\begin{pmatrix} 1 \\ 4 \end{pmatrix}, \quad \begin{pmatrix} 2 \\ 5 \end{pmatrix}, \text{ und } \begin{pmatrix} 3 \\ 6 \end{pmatrix}$$

erzeugt wird.

Ohne die Option *All* wird nur die Transformationsmatrix *T* zurückgeliefert:

```
>> matrix(lllint([[1, 2, 3], [4, 5, 6]]))
```

$$\begin{array}{ccc|ccc}
& & & & +- & & -+ \\
| & -1, & 4 & | & & & \\
| & & & | & & & \\
| & 1, & -3 & | & & & \\
| & & & | & & & \\
| & 0, & 0 & | & & & \\
+- & & & -+ & & &
\end{array}$$

Hintergründe:

⌘ Literatur:

A. K. Lenstra, H. W. Lenstra Jr. und L. Lovasz, Factoring polynomials with rational coefficients. Math. Ann. 261, 1982, pp. 515–534.

Joachim von zur Gathen und Jürgen Gerhard, Modern Computer Algebra. Cambridge University Press, 1999, Kapitel 16.

George L. Nemhauser und Laurence A. Wolsey, Integer and Combinatorial Optimization. New York, Wiley, 1988.

A. Schrijver, Theory of Linear and Integer Programming. New York, Wiley, 1986.

Änderungen:

⌘ Keine Änderungen.

`lmonomial` – das Leitmonom eines Polynoms

`lmonomial(p)` gibt das Leitmonom des Polynoms `p` zurück.

Aufruf(e):

⌘ `lmonomial(p <, vars> <, order> <, Rem>)`

Parameter:

- `p` — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- `vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- `order` — die Termordnung: entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.

Optionen:

`Rem` — veranlasst `lmonomial`, eine Liste mit dem Leitmonom und dem „Reduktum“ zu liefern

Rückgabewert: ein Polynom vom selben Typ wie `p`. Ist `p` ein Ausdruck, so wird auch das Ergebnis als Ausdruck geliefert. `FAIL` wird zurückgeliefert, wenn die Eingabe nicht als Polynom interpretiert werden kann. Mit `Rem` wird eine Liste mit zwei Polynomen zurückgegeben.

Überladbar durch: `p`

Verwandte Funktionen: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- ⌘ Das Argument `p` kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `lmonomial` überlädt.
- ⌘ Wird eine Liste von Unbestimmten übergeben, so wird `p` als Polynom in diesen Unbestimmten angesehen. Man beachte, dass diese Liste nicht mit den Unbestimmten in `p` übereinzustimmen braucht. Siehe Beispiel ??.
- ⌘ Das zurückgelieferte Monom ist „führend“ bezüglich der lexikographischen Ordnung, falls nicht mittels des Arguments `order` eine andere Ordnung angegeben wird. Siehe Beispiel ??.
- ⌘ `lmonomial` liefert `FAIL`, wenn das Eingabepolynom nicht in ein Polynom in den angegebenen Unbekannten konvertiert werden kann. Siehe Beispiel ??.
- ⌘ Das von `lmonomial` berechnete Ergebnis wird nicht weiter evaluiert. Wird dies gewünscht, so kann man dies mit der Funktionen `mapcoeffs` und `eval` erreichen. Siehe Beispiel ??.
- ⌘ Das Leitmonom des Null-Polynoms ist das Null-Polynom.
- ⌘ Für die Ordnungen *LexOrder*, *DegreeOrder* und *DegInvLexOrder* ruft `lmonomial` eine schnelle Kernfunktion auf. Andere Ordnungen werden durch langsamere Bibliotheksfunktionen behandelt.

Option <Rem>:

- ⌘ Mit dieser Option wird eine Liste mit zwei Polynomen zurückgeliefert: das Leitmonom und das Reduktum. Das Reduktum eines Polynoms `p` ist `p - lmonomial(p)`.

Beispiel 1. Es wird gezeigt, wie die Unbestimmten das Ergebnis beeinflussen:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
      lmonomial(p), lmonomial(p, [x, y]), lmonomial(p, [y, x])

                2      2      2
            3 x y , 2 x  y, 3 x y

>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
      lmonomial(p), lmonomial(p, [x, y]), lmonomial(p, [y, x]),
      lmonomial(p, [y]), lmonomial(p, [z])
```

```

      2      2
poly(2 x  y, [x, y]), poly(2 x  y, [x, y]),

      2      2
poly(3 y  x, [y, x]), poly((3 x) y , [y]),

      2      2
poly(2 x  y + 3 x y , [z])

>> delete p:

```

Beispiel 2. Es wird demonstriert, wie verschiedene Ordnungen das Ergebnis beeinflussen:

```

>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
      lmonomial(p), lmonomial(p, DegreeOrder),
      lmonomial(p, DegInvLexOrder)

      4      3      2
poly(5 x , [x, y, z]), poly(4 x  y z , [x, y, z]),

      2      3
poly(3 x  y  z, [x, y, z])

```

Der folgende Aufruf benutzt die umgekehrte lexikographische Termordnung in 3 Unbestimmten:

```

>> lmonomial(p, Dom::MonomOrdering(RevLex(3)))

      2      3
poly(3 x  y  z, [x, y, z])

>> delete p:

```

Beispiel 3. Das Reduktum eines Polynoms wird berechnet:

```

>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
      q := lmonomial(p, Rem)

      2      2
[poly(2 x  y, [x, y]), poly(3 x y  + 6, [x, y])]

```

Das Leitmonom und das Reduktum addieren sich zum Polynom p auf:

```

>> p = q[1] + q[2]

```

```

      2      2
poly(2 x y + 3 x y + 6, [x, y]) =
      2      2
poly(2 x y + 3 x y + 6, [x, y])
>> delete p, q:

```

Beispiel 4. Ein Polynom über dem Restklassenring modulo 7 wird definiert:

```

>> p := poly(3*x + 4, [x], Dom::IntegerMod(7)): lmonomial(p)

      poly(3 x, [x], Dom::IntegerMod(7))

```

Dieses Polynom kann nicht als Polynom in einer anderen Unbestimmten angesehen werden, da der „Koeffizient“ $3*x$ nicht als Element des Koeffizientenrings $\text{Dom}::\text{IntegerMod}(7)$ interpretiert werden kann:

```

>> lmonomial(p, [y])

      FAIL

>> delete p:

```

Beispiel 5. Das folgende Beispiel demonstriert die Evaluationsstrategie von `lmonomial`:

```

>> p := poly(6*x^6*y^2 + x^2 + 2, [x]): y := 4: lmonomial(p)

      2      6
poly((6 y ) x , [x])

```

Die Evaluierung wird durch `eval` erzwungen:

```

>> mapcoeffs(%, eval)

      6
poly(96 x , [x])

>> delete p, y:

```

Änderungen:

- ☞ Selbstdefinierte Termordnungen können nun vorgegeben werden.
 - ☞ Unbestimmte können nun auch für Polynome vom Typ `DOM_POLY` angegeben werden.
 - ☞ In früheren MuPAD Versionen war `lmonomial` eine Kernfunktion.
-

`ln` – der natürliche Logarithmus

`ln(x)` stellt den natürlichen Logarithmus am Punkt x dar.

Aufruf(e):

- ☞ `ln(x)`

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `dilog`, `log`, `polylog`

Details:

- ☞ Der Logarithmus ist für alle komplexen Argumente $x \neq 0$ definiert.
- ☞ Für die meisten exakten Argumente werden unevaluierte Funktionsaufrufe zurückgegeben. Einige Vereinfachungen werden durchgeführt:
 - Argumente der Form $x = \exp(y)$ mit y vom Typ `Type::Numeric` liefern $\ln(\exp(y)) = y + ki2\pi$. Hierbei ist k eine geeignete ganze Zahl, sodass der Imaginärteil des Ergebnisses im Intervall $(-\pi, \pi]$ liegt.
 - Negative ganzzahlige und rationale Argumente x werden gemäß der Regel $\ln(x) = i\pi + \ln(-x)$ umgeschrieben. Rationale Argumente der Form $x = 1/n$ mit einer ganzen Zahl n werden mittels $\ln(1/n) = -\ln(n)$ umgeschrieben.

- Folgende speziellen Werte sind implementiert: $\ln(1) = 0$, $\ln(-1) = i\pi$, $\ln(\pm i) = \pm i\pi/2$, $\ln(\text{infinity}) = \text{infinity}$, $\ln(-\text{infinity}) = i\pi + \text{infinity}$.

☞ Für Gleitpunktargumente werden Gleitpunktwerte berechnet. Die Imaginärteile der Logarithmuswerte liegen im Intervall $(-\pi, \pi]$. Die negative reelle Halbachse ist ein Verzweigungsschnitt, die Imaginärteile springen beim Überschreiten des Schnitts. Auf der negativen reellen Halbachse hat der Logarithmus gemäß der Regel $\ln(x) = i\pi + \ln(-x)$, $x < 0$, den konstanten Imaginärteil π . Siehe Beispiel ??.

☞ Man beachte, dass im Reellen gültige Rechenregeln wie z. B. $\ln(xy) = \ln(x) + \ln(y)$ nicht in der gesamten komplexen Ebene gelten. Man kann mittels `assume` Bezeichner als reell annehmen und gezielt Funktionen wie `expand`, `combine` oder `simplify` einsetzen, um Ausdrücke mit Logarithmen umzuformen. Siehe Beispiel ??.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> ln(2), ln(-3), ln(1/4), ln(1 + I), ln(x^2)
ln(2), I PI + ln(3), -ln(4), ln(1 + I), ln(x^2)
```

Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> ln(123.4), ln(5.6 + 7.8*I), ln(1.0/10^20)
4.815431112, 2.261980065 + 0.948125538 I, -46.05170186
```

Einige spezielle symbolische Vereinfachungen sind implementiert:

```
>> ln(1), ln(-1), ln(exp(-5)), ln(exp(5 + 27/4*I))
0, I PI, -5, (5 + 27/4 I) - 2 I PI
```

Beispiel 2. Die negative reelle Halbachse ist ein Verzweigungsschnitt. Die Imaginärteile der von `ln` gelieferten Werte springen beim Überschreiten des Schnitts:

```
>> ln(-2.0), ln(-2.0 + I/10^1000), ln(-2.0 - I/10^1000)
0.6931471806 + 3.141592654 I, 0.6931471806 + 3.141592654 I,
0.6931471806 - 3.141592654 I
```

Beispiel 3. Die Funktionen `diff`, `float`, `limit`, `series` etc. verarbeiten `ln`:

```
>> diff(ln(x^2), x), float(ln(PI + I))
      2
      -, 1.192985153 + 0.3081690711 I
      x

>> limit(ln(x)/x, x = infinity), series(x*ln(sin(x)), x = 0, 10)
      3      5      7      9      10
      x      x      x      x
0, x ln(x) - -- - --- - ---- - ----- + O(x )
      6      180   2835  37800
```

Beispiel 4. Die Funktionen `expand`, `combine` und `simplify` reagieren auf mittels `assume` gesetzte Eigenschaften von Bezeichnern. Der folgende Aufruf erzeugt kein expandiertes Ergebnis, da die Regel $\ln(xy) = \ln(x) + \ln(y)$ nicht für beliebiges komplexes x, y gilt:

```
>> expand(ln(x*y))
      ln(x y)

Die Regel ist jedoch gültig, falls einer der Faktoren reell und positiv ist:

>> assume(x > 0): expand(ln(x*y))
      ln(x) + ln(y)

>> combine(%, ln)
      ln(x y)

>> simplify(ln(x^3*y) - ln(x) - ln(y))
      2 ln(x)

>> unassume(x):
```

Änderungen:

⌘ Geringfügige Änderungen der Vereinfachungsregeln.

loadlib – Laden eines Bibliothekspakets

`loadlib(libname)` lädt das Bibliothekspaket `libname` ein.

Aufruf(e):

```
# loadlib(libname)
```

Parameter:

`libname` — der Name des Pakets: eine Zeichenkette

Rückgabewert: `TRUE`, falls das Paket erfolgreich geladen wurde, und `FALSE`, falls das Paket vorher schon geladen war.

Verwandte Funktionen: `export`, `LIBPATH`, `loadmod`, `loadproc`, `package`, `Pref::verboseRead`

Details:

- # **loadlib ist obsolet. Bitte verwenden Sie stattdessen package.**
- # `loadlib` lädt die Bibliothek mit dem Namen `libname`. Die Bibliothekspakete der MuPAD-Distribution, wie z. B. `fp`, werden automatisch beim Start von MuPAD geladen. Daher ist `loadlib` nur relevant, um benutzerdefinierte Pakete zu laden.
- # `loadlib` sucht nach der Initialisierungsdatei für das angegebene Paket. Dies kann entweder eine MuPAD-Binärdatei namens `libname.mb` oder eine MuPAD-Textdatei namens `libname.mu` sein, wobei `libname` der Paketname ist. Nach der Initialisierungsdatei wird im Unterverzeichnis `LIBFILES` relativ zu allen durch `LIBPATH` gegebenen Verzeichnissen gesucht. Zuerst durchsucht `loadlib` alle entsprechenden Verzeichnisse nach der Binärdatei `libname.mb` und lädt die erste, die gefunden wird. Falls keine Binärdatei existiert, dann wird nach der Textdatei `libname.mu` gesucht.
Die Datei `fp.mu` im Unterverzeichnis `LIBFILES` desjenigen Verzeichnisses, in dem die MuPAD-Systembibliotheken installiert sind, kann als Vorlage für eine Initialisierungsdatei herangezogen werden.
- # Man kann auch benutzerdefinierte Bibliothekspakete automatisch beim Systemstart laden lassen, indem man das lokale Verzeichnis, in welchem sich die Pakete befinden, zu `LIBPATH` hinzufügt. Die Initialisierungsdateien für die Pakete müssen im Unterverzeichnis `LIBFILES` des lokalen Verzeichnisses stehen.
- # Die Funktion liefert den Wert `TRUE`, wenn das Laden des Paketes erfolgreich durchgeführt werden konnte, und den Wert `FALSE`, falls das Paket gefunden wurde, aber bereits geladen ist. Konnte das angegebene Paket nicht gefunden werden, so wird eine Fehlermeldung geliefert.
- # Eine Bibliothek wird nur beim ersten Aufruf von `loadlib` geladen. Weitere Aufrufe laden dieselbe Bibliothek nicht erneut.

Änderungen:

⌘ `loadlib` wird in zukünftigen MuPAD-Versionen entfernt.

`loadmod` – Einladen eines Moduls

`loadmod("modulename")` lädt das dynamische Modul namens `modulename`.

`loadmod()` prüft, ob der MuPAD-Kern die Verwendung dynamischer Module unterstützt.

Aufruf(e):

⌘ `loadmod("modulename")`

⌘ `loadmod()`

Parameter:

"modulename" — der Name eines Moduls: eine Zeichenkette

Rückgabewert: `loadmod()` liefert `TRUE` oder `FALSE`; `loadmod("modulename")` liefert ein Modul-Domain vom Typ `DOM_DOMAIN`.

Seiteneffekte: `loadmod("modulename")` weist dem Bezeichner `modulename` einen Wert zu. Beispielsweise hat nach `loadmod("stdmod")` der Bezeichner `stdmod` das geladene Modul als Wert.

Weitere Dokumentation: Dynamic Modules - User's Manual and Programming Guide for MuPAD 1.4, Andreas Sorgatz, Okt 1998, Springer Verlag, Heidelberg, mit CD-ROM, ISBN 3-540-65043-1.

Verwandte Funktionen: `external`, `export`, `module::new`, `package`, `unloadmod`

Details:

⌘ `loadmod()` liefert `TRUE`, wenn die Nutzung dynamischer Module in dieser MuPAD-Version unterstützt wird. Anderenfalls wird `FALSE` zurückgegeben.

⌘ `loadmod("modulename")` lädt das dynamische Modul namens `modulename`. Dabei erzeugt es ein entsprechendes Modul-Domain, weist es dem Bezeichner `modulename` zu und liefert es als Rückgabewert von `loadmod` an die MuPAD-Sitzung. Ein etwaiger Wert des Bezeichners `modulename` wird dabei überschrieben.

- ⌘ Existiert das Modul-Domain bereits, so wird es überschrieben und die Warnung `Warning: Redefinition of domain ...` ausgegeben.
- ⌘ Die Moduldatei „`modulename.mdm`“ wird erst in den durch die Variable `READPATH` spezifizierten Verzeichnissen gesucht, danach im aktuellem Arbeitsverzeichnis und dann im MuPAD-Modulverzeichnis.
- ⌘ Die aktuelle Berechnung wird mit einer Fehlermeldung abgebrochen, falls das Modul nicht geladen werden kann.
- ⌘ Falls die Datei „`modulename.mdg`“ existiert, so enthält sie MuPAD-Objekte die ebenfalls geladen und in das Modul-Domain eingebunden werden. Tritt hierbei ein Fehler auf, so wird eine Warnung ausgegeben, und MuPAD versucht bei jedem Aufruf einer hiervon betroffenen Modulfunktion die Objekte erneut zu laden.
- ⌘ Zur Moduldatei „`modulename.mdm`“ existiert gegebenenfalls eine zusätzliche Textdatei „`modulename.mdh`“, die eine Kurzbeschreibung des Moduls enthält. Diese Online-Dokumentation kann mittels der Modulfunktion `modulename::doc()` bzw. `modulename::doc("MethodenName")` angezeigt werden.
- ⌘ `loadmod` ist eine Funktion des Systemkerns.

Beispiel 1. Der folgende Aufruf lädt das dynamische Modul `stdmod`:

```
>> loadmod("stdmod")

stdmod

>> type(stdmod);

DOM_DOMAIN
```

Da Module als Domains repräsentiert werden, können Sie genau wie MuPAD-Bibliotheken oder andere Domains verwendet werden. Beispielsweise ist eine Modulfunktion mit dem Präfix `modulename` aufzurufen:

```
>> stdmod::which("stdmod")

"/usr/local/mupad/linux/modules/stdmod.mdm"
```

Wie bei Bibliotheken liefert `info` Informationen zu einem geladenen Modul:

```
>> info(stdmod)

Module: 'stdmod' created on 28.Sep.00 by mmg R-2.0.0
Module: Extended Module Management

-- Interface:
stdmod::age, stdmod::doc, stdmod::help, stdmod::max,
stdmod::stat, stdmod::which
```

Die Funktion `export` exportiert alle öffentlichen Funktionen des Moduls. Danach kann z. B. die Methode `"which"` ohne den Domain-Präfix `stdmod` aufgerufen werden:

```
>> export(stdmod): which("stdmod")

"/usr/local/mupad/linux/modules/stdmod.mdm"
```

Beispiel 2. Dokumentation zu einem dynamischen Modul namens `modulename` kann über eine einfache Textdatei „`modulename.mdh`“ bereitgestellt werden, die dann im selben Verzeichnis wie die Moduldatei „`modulename.mdm`“ installiert sein muss. Auf diese Dokumentation kann wie folgt zugegriffen werden. Siehe hierzu auch `module::help`.

```
>> stdmod::doc()

MODULE
  stdmod - Extended Module Management

INTRODUCTION
  This module provides functions for an extended module ...

INTERFACE
  age, doc, help, max, stat, which
```

Während oben die Übersichtsseite der Moduldokumentation angezeigt wurde, bewirkt unten die Angabe des Argumentes `"doc"` die Anzeige der Hilfe-seite zur Modulfunktion `stdmod::doc`:

```
>> stdmod::doc("doc")

NAME
  stdmod::doc - Display online documentation

SYNOPSIS
  ...

SEE ALSO
  info, module::help
```

Hintergründe:

- ☞ Die Kernfunktionen `external`, `loadmod` und `unloadmod` sind Basisfunktionen zum Zugriff auf Module. Weitere Funktionen stehen in der Bibliothek `module` zur Verfügung.

- ⌘ Es kann zu jedem Zeitpunkt jeweils nur eine Instanz eines dynamischen Moduls im Speicher geladen sein. Jeder weitere Aufruf von `loadmod` lädt den Maschinencode nur, falls er zuvor ausgeladen oder verdrängt wurde. Im Gegensatz dazu wird das Modul-Domain stets neu erzeugt und überschrieben.
- ⌘ Der Maschinencode von dynamischen Modulen kann während einer MuPAD-Sitzung mit der Funktion `unloadmod` ausgeladen werden.
- ⌘ MuPAD bietet ein Modul-Ressourcen-Management, das gegebenenfalls den Maschinencode von Modulen aus dem Arbeitsspeicher verdrängt, falls diese gerade nicht benötigt werden oder der Arbeitsspeicher knapp wird.
- ⌘ Neben den *dynamischen* Modulen unterstützt MuPAD auch so genannte *statische* Module, die während der Laufzeit nicht automatisch ausgeladen oder verdrängt werden können. Siehe hierzu auch `unloadmod`.

Änderungen:

- ⌘ Keine Änderungen.

`loadproc` – Laden eines MuPAD-Objekts bei Bedarf

`loadproc` lädt ein MuPAD-Objekt aus einer Datei, sobald auf dieses Objekt zum ersten Mal zugegriffen wird.

Aufruf(e):

⌘ `object := loadproc(object, path, file)`

Parameter:

- `object` — ein MuPAD-Objekt, das auf der linken Seite einer Zuweisung stehen darf
- `path` — ein relativer Pfadname mit abschließendem Pfadseparator: eine Zeichenkette
- `file` — ein Dateiname ohne Suffix: eine Zeichenkette

Rückgabewert: ein Element des Domains `stdlib::LoadProc` (siehe Abschnitt „Hintergründe“ weiter unten).

Verwandte Funktionen: `export`, `finput`, `fread`, `LIBPATH`, `loadmod`, `package`, `pathname`, `Pref::verboseRead`, `read`

Details:

☞ Die MuPAD-Bibliothek ist ziemlich groß. Typischerweise verwendet man davon jedoch nur einen kleinen Teil. Es wäre sehr zeit- und speicheraufwendig, wenn die ganze Bibliothek beim Start von MuPAD geladen würde. `loadproc` bietet ein Konzept, um den Ladeprozess für ein vordefiniertes Objekt, etwa einen Bibliotheksdomain oder eine Bibliotheksprozedur, bis zu dem Zeitpunkt aufzuschieben, an dem das Objekt erstmals benötigt wird.

☞ `loadproc` liefert ein Element eines speziellen Datentyps zurück. Dieses Element speichert lediglich die Information über die Datei, in der `object` definiert ist, liest aber die betreffende Datei noch nicht ein. Das passiert erst dann, wenn `object` zum ersten Mal benutzt wird.

Pfad und Name der Datei werden durch die beiden Zeichenketten `path` bzw. `file` angegeben. Die Funktion `pathname` ist dabei nützlich, um Pfadnamen in plattformunabhängiger Weise zu spezifizieren.

☞ Wenn `object` erstmalig evaluiert wird, dann versucht das System zunächst, die Datei

```
path . "." . file . ".mb"
```

im MuPAD-Binärformat einzulesen, wobei `.` der Konkatenationsoperator ist. MuPAD sucht nach dieser Datei relativ zu allen Verzeichnissen, die durch `LIBPATH` spezifiziert sind. Die erste Datei, die gefunden wird, wird eingelesen. Ist die Suche nicht erfolgreich, so wird nach der Textdatei

```
path . "." . file . ".mu"
```

gesucht.

Die entsprechende Datei muss die tatsächliche Definition von `object` enthalten, typischerweise eine Anweisung der Form `object := value`. Andernfalls kann es zu Endlosrekursionen kommen.



Nach Einlesen der Datei wird schließlich `value` als Wert von `object` zurückgegeben. Der gesamte Ladeprozess ist für den Anwender transparent.

☞ `loadproc` wertet das erste Argument `object` nicht aus, aber die anderen Argumente werden, wie üblich, ausgewertet.

☞ Um Seiteneffekte zu vermeiden, werden Alias-Definitionen während des Einlesens der Datei nicht berücksichtigt, ausgenommen die Aliase, die in der Datei selbst definiert sind. Letztere sind nur lokal innerhalb der Datei gültig und werden gelöscht, wenn der Ladevorgang beendet ist.

Beispiel 1. Beim Systemstart wird der Wert des Bezeichners `int` wie folgt initialisiert:

```
>> int := loadproc(int, pathname("STDLIB"), "int");
```

Damit wird dem System mitgeteilt, dass die eigentliche Definition der Integrationsfunktion `int` in der Datei `"STDLIB/int.mu"` steht. Diese Datei ist relativ zu dem durch `LIBPATH` spezifizierten Bibliotheksverzeichnis zu finden, welches standardmäßig ein Unterverzeichnis des MuPAD-Installationsverzeichnisses ist.

Wenn `int` zum ersten Mal verwendet wird, z. B. nach Eingabe des Aufrufs `int(t^2,t)`, dann lädt MuPAD automatisch die Datei `"STDLIB/int.mu"`. Diese enthält die folgenden Zeilen, welche die eigentliche Definition der Funktionsumgebung `int` darstellen:

```
int := funcenv(  
  proc(f, x = null())  
  begin  
    if args(0) = 0 then error("No argument given") end_if;  
    ...  
  end_proc);
```

Nach Einlesen der Datei wird die Funktionsumgebung als Wert von `int` zurückgeliefert, und alles weiter verläuft wie üblich: Der Aufruf `int(t^2,t)` wird ausgeführt und dessen Resultat $t^3/3$ zurückgeliefert.

Hintergründe:

- ⌘ Der Rückgabewert von `loadproc` ist ein Objekt des Domains `stdlib::LoadProc`. Dieser Datentyp dient nur internen Zwecken; es sollte niemals nötig sein, auf Objekte dieses Typs zuzugreifen. Daher bleibt er undokumentiert.
- ⌘ Oft enthält eine Bibliotheks-Quelldatei Definitionen für mehr als ein mittels `loadproc` zu ladendes Objekt. In so einem Fall kann es passieren, dass ein Objekt geladen wird, bevor zum ersten Mal darauf zugegriffen wird, nämlich dann, wenn auf ein anderes Objekt zugegriffen wird, das in derselben Datei definiert ist.

Änderungen:

- ⌘ Keine Änderungen.

`log` – der Logarithmus zu einer beliebigen Basis

`log(b, x)` stellt den Logarithmus von x zur Basis b dar.

Aufruf(e):

⌘ $\log(b, x)$

Parameter:

- b — ein Bezeichner vom Domain-Typ `DOM_IDENT` oder ein reeller numerischer Wert vom Typ `Type::Positive`.
- x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `dilog`, `ln`, `polylog`

Details:

- ⌘ Mathematisch ist $\log(b, x)$ äquivalent zu $\ln(x)/\ln(b)$. Siehe Beispiel ??
Der Logarithmus ist für alle komplexen Argumente $x \neq 0$ definiert.
- ⌘ Die Basis b muss reell, positiv und ungleich 1 sein. Interne Vereinfachung basieren auf diesen Annahmen.
- ⌘ Für die meisten exakten Argumente werden unevaluierte Funktionsaufrufe zurückgegeben. Einige Vereinfachungen werden durchgeführt:
 - Für $b = \exp(1)$ wird $\log(b, x) = \ln(x)$ zurückgeliefert.
 - Mathematisch gilt $\log(b, b^y) = y$ für reelles y . Diese Vereinfachungsregel ist folgendermaßen implementiert: bei symbolischem b wird sie für y vom Typ `Type::Real` durchgeführt, bei numerischem b für ganzzahliges oder rationales y .
 - Negative ganzzahlige und rationale Argumente x werden gemäß der Regel $\log(b, x) = i\pi/\ln(b) + \log(b, -x)$ umgeschrieben. Rationale Argumente der Form $x = 1/n$ mit einer ganzen Zahl n werden gemäß $\log(b, 1/n) = -\log(b, n)$ umgeschrieben.
 - Folgende speziellen Werte sind implementiert:

$$\log(b, 1) = 0, \log(b, -1) = \frac{i\pi}{\ln(b)}, \log(b, \pm i) = \pm \frac{i\pi}{2\ln(b)}.$$

- ⌘ Gleitpunktwerte werden berechnet, wenn beide Argumente numerisch sind und mindestens ein Argument eine Gleitpunktzahl ist. Die Imaginärteile der Logarithmuswerte liegen im Intervall $(-\pi/\ln(b), \pi/\ln(b)]$ für $b > 1$ (im Intervall $[\pi/\ln(b), -\pi/\ln(b))$ für $b < 1$). Die negative reelle

Halbachse ist ein Verzweigungsschnitt, die Imaginärteile springen beim Überschreiten des Schnitts. Auf der negativen reellen Halbachse hat der Logarithmus gemäß der Regel $\log(b, x) = i\pi/\ln(b) + \log(b, -x)$, $x < 0$, den konstanten Imaginärteil $\pi/\ln(b)$. Siehe Beispiel ??.

⚠ Man beachte, dass im Reellen gültige Rechenregeln wie z. B. $\log(b, xy) = \log(b, x) + \log(b, y)$ nicht in der gesamten komplexen Ebene gelten. Man kann mittels `assume` Bezeichner als reell annehmen und gezielt Funktionen wie `expand` oder `simplify` einsetzen, um Ausdrücke mit Logarithmen umzuformen. Siehe Beispiel ??.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> log(b, 2), log(2, 3), log(10, 10^2), log(10, 2*10^2),
    log(2, I), log(b, x^2)
```

$$\log(b, 2), \log(2, 3), 2, \log(10, 200), \frac{1/2 \, i \, \pi}{\ln(2)}, \log(b, x^2)$$

Man beachte, dass als Basis zwar symbolische Bezeichner, aber keine arithmetischen Ausdrücke zulässig sind:

```
>> log(b + 1, 2)

Error: base must be an identifier or of Type::Positive [log]

>> log(PI^2, 2)

Error: base must be an identifier or of Type::Positive [log]
```

Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> log(2, 123.4), log(2.0, 5.6 + 7.8*I), log(10.0, 2/10^20)

6.947198584, 3.263347423 + 1.367856012 I, -19.69897
```

Einige spezielle symbolische Vereinfachungen sind implementiert:

```
>> log(b, 1), log(b, -1), log(2/3, (4/9)^10), log(b, b^(-5))

0, -----, 20, -5
    I PI
    ln(b)
```

Beispiel 2. Die negative reelle Halbachse ist ein Verzweigungsschnitt. Die Imaginärteile der von `log` gelieferten Werte springen beim Überschreiten des Schnitts:

```
>> log(10, -2.0),
      log(10, -2.0 + I/10^1000),
      log(10, -2.0 - I/10^1000)

0.3010299957 + 1.364376354 I, 0.3010299957 + 1.364376354 I,
      0.3010299957 - 1.364376354 I
```

Beispiel 3. Mittels `rewrite` kann `log` durch `ln` umgeschrieben werden:

```
>> rewrite(log(b, x), ln), rewrite(log(10, 200), ln)

      ln(x)  ln(200)
      ----, -----
      ln(b)  ln(10)
```

Beispiel 4. Die Funktionen `diff`, `float`, `limit`, `series` etc. verarbeiten `log`:

```
>> diff(log(b, x^2), x), float(log(10, PI + I))

      2
      ----, 0.5181068691 + 0.133836127 I
      x ln(b)

>> limit(log(10, x)/x, x = infinity),
      series(x*log(x, sin(x)), x = 0)

      3          5          6
      x          x          x
      0, x - ---- - ---- + O(x )
             6 ln(x)  180 ln(x)
```

Beispiel 5. Die Funktionen `expand` und `simplify` reagieren auf mittels `assume` gesetzte Eigenschaften von Bezeichnern. Der folgende Aufruf erzeugt kein expandiertes Ergebnis, da die Regel $\log(b, xy) = \log(b, x) + \ln(y)$ nicht für beliebiges komplexes x, y gilt. Man beachte, dass `expand` den allgemeinen Logarithmus durch `ln` umschreibt:

```
>> expand(log(10, x*y))
```


$$\frac{\ln(x \cdot y)}{\ln(10)}$$

Die Regel ist jedoch gültig, falls einer der Faktoren reell und positiv ist:

```
>> assume(x > 0): expand(log(b, x*y))
```

$$\frac{\ln(x)}{\ln(b)} + \frac{\ln(y)}{\ln(b)}$$

```
>> simplify(log(b, x^3*y) - log(b, x) - log(b, y))
```

$$2 \log(b, x)$$

```
>> unassume(x):
```

Änderungen:

⌘ log ist eine neue Funktion.

lterm – der Leitterm eines Polynoms

lterm(p) gibt den Leitterm des Polynoms p zurück.

Aufruf(e):

⌘ lterm(p <, vars> <, order>)

Parameter:

- p — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- vars — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- order — die Termordnung: entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.

Rückgabewert: ein Polynom vom selben Typ wie p. Ist p ein Ausdruck, so wird auch das Ergebnis als Ausdruck geliefert. `FAIL` wird zurückgeliefert, wenn die Eingabe nicht als Polynom interpretiert werden kann.

Überladbar durch: p

Verwandte Funktionen: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tccoeff`

Details:

- ⌘ Das Argument `p` kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `lterm` überlädt.
 - ⌘ Es gilt die Identität `lterm(p) lcoeff(p) = lmonomial(p)`.
 - ⌘ Wird eine Liste von Unbestimmten übergeben, so wird `p` als Polynom in diesen Unbestimmten angesehen. Man beachte, dass diese Liste nicht mit den Unbestimmten in `p` übereinzustimmen braucht. Siehe Beispiel ??.
 - ⌘ Der zurückgelieferte Term ist „führend“ bezüglich der lexikographischen Ordnung, falls nicht mittels des Arguments `order` eine andere Ordnung angegeben wird. Siehe Beispiel ??.
 - ⌘ `lterm` liefert `FAIL`, wenn das Eingabepolynom nicht in ein Polynom in den angegebenen Unbekannten konvertiert werden kann. Siehe Beispiel ??.
 - ⌘ Der Leitterm des Null-Polynoms ist das Null-Polynom.
 - ⌘ Für die Ordnungen *LexOrder*, *DegreeOrder* und *DegInvLexOrder* ruft `lterm` eine schnelle Kernfunktion auf. Andere Ordnungen werden durch langsamere Bibliotheksfunktionen behandelt.
-

Beispiel 1. Es wird gezeigt, wie die Unbestimmten das Ergebnis beeinflussen:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
      lterm(p), lterm(p, [x, y]), lterm(p, [y, x])

                2    2          2
              x y , x  y, x y

>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
      lterm(p), lterm(p, [x, y]), lterm(p, [y, x]),
      lterm(p, [y]), lterm(p, [z])

                2                2                2
      poly(x  y, [x, y]), poly(x  y, [x, y]), poly(y  x, [y, x]),

                2
      poly(y , [y]), poly(1, [z])

>> delete p:
```

Beispiel 2. Es wird demonstriert, wie verschiedene Ordnungen das Ergebnis beeinflussen:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
      lterm(p), lterm(p, DegreeOrder), lterm(p, DegInvLexOrder)

      4          3      2
      poly(x , [x, y, z]), poly(x y z , [x, y, z]),

      2 3
      poly(x y z, [x, y, z])
```

Der folgende Aufruf benutzt die umgekehrte lexikographische Termordnung in 3 Unbestimmten:

```
>> lterm(p, Dom::MonomOrdering(RevLex(3)))

      2 3
      poly(x y z, [x, y, z])

>> delete p:
```

Beispiel 3. Ein Polynom über dem Restklassenring modulo 7 wird definiert:

```
>> p := poly(3*x + 4, [x], Dom::IntegerMod(7)): lterm(p)

      poly(x, [x], Dom::IntegerMod(7))
```

Dieses Polynom kann nicht als Polynom in einer anderen Unbestimmten angesehen werden, da der „Koeffizient“ x nicht als Element des Koeffizientenrings $\text{Dom}::\text{IntegerMod}(7)$ interpretiert werden kann:

```
>> lterm(p, [y])

      FAIL

>> delete p:
```

Beispiel 4. Das Leitmonom ist das Produkt aus dem Leitkoeffizienten und dem Leitterm:

```
>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
      mapcoeffs(lterm(p), lcoeff(p)) = lmonomial(p)

      2          2
      poly(2 x y, [x, y]) = poly(2 x y, [x, y])

>> delete p:
```

Änderungen:

- ⌘ Selbstdefinierte Termordnungen können nun vorgegeben werden.
 - ⌘ Unbestimmte können nun auch für Polynome vom Typ `DOM_POLY` angegeben werden.
 - ⌘ In früheren MuPAD Versionen war `lterm` eine Kernfunktion.
-

`match` – Musterkennung („pattern matching“)

`match(expression, pattern)` überprüft, ob die syntaktische Struktur von `expression` von der durch `pattern` vorgeschriebenen Form ist und liefert gegebenenfalls eine Menge von Ersetzungsgleichungen zurück, die `pattern` in `expression` überführen.

Aufruf(e):

⌘ `match(expression, pattern <, option1, option2, ...>)`

Parameter:

<code>expression</code>	— ein MuPAD-Ausdruck
<code>pattern</code>	— das Muster: ein MuPAD-Ausdruck
<code>option1, option2, ...</code>	— optionale Argumente (siehe unten)

Optionen:

$Ass = \{f1, f2, \dots\}$	— es wird angenommen, dass die Bezeichner $f1, f2, \dots$ assoziative Operatoren darstellen
$Comm = \{g1, g2, \dots\}$	— es wird angenommen, dass die Bezeichner $g1, g2, \dots$ kommutative Operatoren darstellen
$Cond = \{p1, p2, \dots\}$	— es werden nur solche Übereinstimmungen zwischen <code>expression</code> und <code>pattern</code> berücksichtigt, die die durch die Prozeduren $p1, p2, \dots$ spezifizierten Bedingungen erfüllen
$Const = \{c1, c2, \dots\}$	— es wird angenommen, dass die Bezeichner $c1, c2, \dots$ Konstanten darstellen
$Null = \{h1 = e1, h2 = e2, \dots\}$	— es wird angenommen, dass die Bezeichner $e1, e2, \dots$ die neutralen Elemente bezüglich der Operatoren $h1, h2, \dots$ darstellen

Rückgabewert: eine Menge von Ersetzungsgleichungen oder FAIL.

Verwandte Funktionen: `matchlib::analyze`, `simplify`, `subs`, `subsex`, `subsop`

Details:

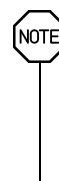
- ☞ `match` berechnet eine Menge von Ersetzungsgleichungen S für die in `pattern` vorkommenden Bezeichner, so dass `subs(pattern, S)` und `expression` bis auf Assoziativität, Kommutativität und neutrale Elemente übereinstimmen.
- ☞ Ein Großteil der Funktionalität von `match` wird durch zusätzliche Optionen verfügbar. `match` befindet sich jedoch noch in einem experimentellen Stadium; ein Teil der Funktionalität kann noch fehlerbehaftet sein.
- ☞ Ohne zusätzliche Optionen arbeitet `match` strikt syntaktisch; Assoziativität, Kommutativität und neutrale Elemente werden nicht berücksichtigt. In diesem Fall gilt `subs(pattern, S) = expression` für die



von `match` zurückgelieferte Menge `S` von Ersetzungsgleichungen. Siehe Beispiel ??.

Man kann diese Eigenschaften durch Angabe der Optionen `Ass`, `Comm` bzw. `Null` für Operatoren deklarieren (siehe unten). Dann sind `subs(pattern, S)` und `expression` nicht notwendigerweise gleiche MuPAD-Ausdrücke, aber sie können durch Anwendung der mit den Optionen implizierten Regeln ineinander überführt werden.

- ☞ Sowohl `expression` als auch `pattern` dürfen beliebige MuPAD-Ausdrücke sein, und zwar sowohl atomare Ausdrücke wie Zahlen, boolsche Konstanten, und Bezeichner, als auch zusammengesetzte Ausdrücke.
- ☞ Jeder in `pattern` vorkommende Bezeichner ohne Wert, einschließlich der 0-ten Operanden, wird als *Mustervariable* angesehen. Das bedeutet, dass er durch einen Ausdruck ersetzt werden darf, um `pattern` in `expression` zu überführen. Man kann die Option `Const` (siehe unten) angeben, um Bezeichner als nicht ersetzbar zu deklarieren.
- ☞ Abgesehen von einigen automatischen Vereinfachungen durch den MuPAD-Kern wird Distributivität *nicht* berücksichtigt. Siehe Beispiel ??.
- ☞ `match` evaluiert seine Argumente. Diese Evaluierung beinhaltet üblicherweise ein gewisses Maß an automatischer Vereinfachung, die die syntaktische Struktur sowohl von `expression` als auch von `pattern` in unerwarteter Weise verändern kann. Siehe Beispiel ??.
- ☞ Selbst wenn es mehrere Möglichkeiten gibt, `pattern` in `expression` zu transformieren, liefert `match` nur eine bestimmte zurück. Siehe Beispiel ??.
- ☞ Wenn die Struktur von `expression` nicht von der durch `pattern` gegebenen Form ist, wird `FAIL` zurückgegeben.
- ☞ Wenn `expression` und `pattern` gleich sind, wird die leere Menge zurückgegeben.
- ☞ Sind `expression` und `pattern` verschieden, aber `match` findet eine strukturelle Übereinstimmung, so wird eine Menge `S` von Ersetzungsgleichungen zurückgeliefert. Für jede in `pattern` vorkommende Mustervariable, die nicht mittels der Option `Const` als konstant deklariert wurde, enthält `S` genau eine Ersetzungsgleichung der Form $x = y$, wobei y der x ersetzende Ausdruck ist.



Option `<Ass = {f1, f2, ...}>`:

- ☞ Es wird angenommen, dass die Operatoren `f1`, `f2`, ... assoziativ sind und beliebig viele Arguments haben dürfen, d.h. Ausdrücke wie `f1(f1(a, b), c)`, `f1(a, f1(b, c))` und `f1(a, b, c)` werden als gleich betrachtet.

- ⌘ Für assoziative Operatoren mit weniger als zwei Argumenten werden keine speziellen Regeln berücksichtigt. Insbesondere werden $f1(a)$ und a als verschieden angesehen.

Option *<Comm = {g1, g2, ...}>*:

- ⌘ Die Operatoren $g1, g2, \dots$ werden als kommutativ angenommen, d.h. Ausdrücke wie $g1(a, b)$ und $g1(b, a)$ werden als gleich betrachtet.

Option *<Cond = {p1, p2, ...}>*:

- ⌘ Es werden nur solche Ersetzungen zugelassen, für die die durch die Prozeduren $p1, p2, \dots$ gegebenen Bedingungen erfüllt sind. Jede Prozedur muss genau ein Argument haben und stellt eine Bedingung an genau eine Mustervariable. Der Name des einzigen formalen Arguments muss gleich dem Namen einer in *pattern* auftretenden Mustervariable sein, die nicht mit der Option *Const* als konstant erklärt wird. Jede Bedingungsprozedur muss einen Ausdruck zurückliefern, den die Funktion *bool* entweder zu *TRUE* oder zu *FALSE* auswerten kann.

Einfache Bedingungen können durch namenlose Prozeduren ausgedrückt werden, die mittels \rightarrow erzeugt werden. Siehe Beispiel ??.

- ⌘ Wenn *match* eine strukturelle Übereinstimmung findet, gegeben durch eine Menge von Ersetzungsgleichungen S , dann wird folgendermaßen überprüft, ob alle angegebenen Bedingungen erfüllt sind. *match* ruft $bool(p1(y1) \text{ and } p2(y2) \text{ and } \dots)$ auf, wobei $y1$ der die Mustervariable $x1$ ersetzende Ausdruck und $x1$ gleichzeitig das formale Argument der Prozedur $p1$ ist, usw. Liefert dieser Aufruf *TRUE* zurück, dann ist S der Rückgabewert von *match*. Andernfalls wird nach der nächsten möglichen Übereinstimmung gesucht.

Ist beispielsweise $p1$ eine Prozedur mit formalem Argument $x1$, wobei $x1$ eine in *pattern* vorkommende Mustervariable ist, dann wird eine Menge $S = \{\dots, x1 = y1, \dots\}$ von Ersetzungsgleichungen nur dann als zulässig betrachtet, wenn $bool(p1(y1))$ den Wert *TRUE* ergibt.

- ⌘ Für jede Mustervariable darf höchstens eine Bedingungsprozedur angegeben werden. Gegebenenfalls sind die logischen Operatoren *and* und *or* sowie die Kontrollstrukturen *if* und *case* zu verwenden, um mehrere Bedingungen für die selbe Mustervariable in einer Bedingungsprozedur zu vereinen. Siehe Beispiel ??.

Option *<Const = {c1, c2, ...}>*:

- ⌘ Die Bezeichner *c1*, *c2*, ... werden als Konstante betrachtet, die nicht ersetzt werden dürfen, um *pattern* in *expression* zu transformieren.

Option *<Null = {h1 = e1, h2 = e2, ...}>*:

- ⌘ Es wird angenommen, dass *e1*, *e2*, ... die neutralen Elemente bezüglich der assoziativen Operationen *h1*, *h2*, ... sind, d.h. Ausdrücke wie *h1(a, e1)*, *h1(e1, a)* und *h1(a)* werden als gleich angesehen.
- ⌘ Diese Deklaration betrifft lediglich solche Operatoren, die auch mit der Option *Ass* als assoziativ erklärt werden. Weiterhin wird nicht implizit von den neutralen Elementen angenommen, dass sie konstant sind.

Beispiel 1. Alle Bezeichner des folgenden Musterausdrucks sind Mustervariablen:

```
>> match(f(a, b), f(X, Y))  
      {X = a, Y = b, f = f}
```

Die Funktion *f* wird als konstant angenommen:

```
>> match(f(a, b), f(X, Y), Const = {f})  
      {X = a, Y = b}
```

Beispiel 2. Der folgende Aufruf enthält eine Bedingung für die Mustervariable *X*:

```
>> match(f(a, b), f(X, Y), Const = {f}, Cond = {X -> not has(X, a)})  
      FAIL
```

Mit der Option *Comm* kann die Funktion *f* als kommutativ erklärt werden. In diesem Fall passt der gegebene Ausdruck auf das Muster (im Gegensatz zum vorigen Beispiel):

```
>> match(f(a, b), f(X, Y), Const = {f}, Comm = {f},  
      Cond = {X -> not has(X, a)})  
      {X = b, Y = a}
```


Beispiel 3. Im folgenden Beispiel kann keine Übereinstimmung gefunden werden, da die Anzahl Argumente des Ausdrucks und des Musters verschieden sind:

```
>> match(f(a, b, c), f(X, Y), Const = {f})
FAIL
```

Wenn wir die Funktion f mit der Option *Ass* als assoziativ erklären, dann passt der gegebene Ausdruck auf das Muster:

```
>> match(f(a, b, c), f(X, Y), Const = {f}, Ass = {f})
{X = a, Y = f(b, c)}
```

Beispiel 4. Hat der Funktionsaufruf im Musterausdruck allerdings mehr Arguments als der entsprechende Funktionsaufruf in *expression*, dann wird keine Übereinstimmung gefunden:

```
>> match(f(a, b), f(X, Y, Z), Const = {f}, Ass = {f})
FAIL
```

Ist das neutrale Element bezüglich des Operators f bekannt, dann können zusätzliche Übereinstimmungen gefunden werden, indem einige der Mustervariablen durch das neutrale Element ersetzt werden:

```
>> match(f(a, b), f(X, Y, Z), Const = {f}, Ass = {f}, Null = {f = 0})
{X = a, Z = b, Y = 0}
```

Beispiel 5. Im allgemeinen wird Distributivität nicht berücksichtigt:

```
>> match(a*x + a*y, a*(X + Y), Const = {a})
FAIL
```

Der folgende Aufruf findet eine passende Ersetzung, jedoch nicht die erwartete:

```
>> match(a*(x + y), X + Y)
{Y = a (x + y), X = 0}
```

Auch die folgenden Deklarationen und Bedingungen führen nicht zu der gewünschten Ersetzung:

```
>> match(a*(x + y), a*X + a*Y, Const = {a},
      Cond = {X -> X <> 0, Y -> Y <> 0})
FAIL
```

Beispiel 6. Automatische Vereinfachungen können die Struktur des gegebenen Ausdrucks oder Musters „zerstören“:

```
>> match(sin(-2), sin(X))
```

FAIL

Das Ergebnis ist FAIL, denn das erste Argument `sin(-2)` wird evaluiert:

```
>> sin(-2)
```

-sin(2)

Durch Verwendung von `hold` kann dieses Problem umgangen werden:

```
>> match(hold(sin(-2)), sin(X))
```

{X = -2}

Beispiel 7. `match` gibt nur eine mögliche Ersetzung zurück:

```
>> match(a + b + c + 1, X + Y)
```

{X = a, Y = b + c + 1}

Um weitere Lösungen zu bekommen, kann man bereits gefundene Lösungen durch Bedingungen ausschließen:

```
>> match(a + b + c + 1, X + Y, Cond = {X -> X <> a})
```

{X = b, Y = a + c + 1}

```
>> match(a + b + c + 1, X + Y,  
        Cond = {X -> X <> a and X <> b})
```

{X = c, Y = a + b + 1}

```
>> match(a + b + c + 1, X + Y,  
        Cond = {X -> not X in {a, b, c}})
```

{Y = a + b + c, X = 1}

Beispiel 8. Jede Mustervariable darf höchstens eine Bedingungs-Prozedur haben. Einfache Bedingungen können durch anonyme Prozeduren (\rightarrow) angegeben werden:

```
>> match(a + b, X + Y, Cond = {X -> X <> a, Y -> Y <> b})
      {X = b, Y = a}
```

Verschiedene Bedingungen für eine Mustervariable können in einer Prozedur kombiniert werden:

```
>> Xcond := proc(X) begin
      if domtype(X) = DOM_IDENT then
        X <> a and X <> b
      else
        X <> 0
      end_if
    end_proc:

>> match(sin(a*b), sin(X*Y), Cond = {Xcond})
      {Y = a b, X = 1}

>> match(sin(a*c), sin(X*Y), Cond = {Xcond})
      {Y = a, X = c}

>> delete Xcond:
```

Änderungen:

☞ match ist eine neue Funktion.

matrix – Erzeugung von Matrizen und Vektoren

`matrix(m, n, [[a11, a12, ...], [a21, a22, ...], ...])` liefert die $m \times n$ -Matrix

$$\begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \vdots \end{pmatrix}.$$

`matrix(n, 1, [a1, a2, ...])` liefert den $n \times 1$ -Spaltenvektor

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \end{pmatrix}.$$

`matrix(1, n, [a1, a2, ...])` liefert den $1 \times n$ -Zeilenvektor

$$\begin{pmatrix} a_1 & a_2 & \cdots \end{pmatrix}.$$

Aufruf(e):

```
⌘ matrix(ListOfRows)
⌘ matrix(List)
⌘ matrix(Array)
⌘ matrix(Matrix)
⌘ matrix(m, n)
⌘ matrix(m, n, ListOfRows)
⌘ matrix(1, n, List)
⌘ matrix(m, 1, List)
⌘ matrix(m, n, List, Diagonal)
⌘ matrix(m, n, List, Banded)
⌘ matrix(m, n, f)
⌘ matrix(m, n, g, Diagonal)
```

Parameter:

<code>ListOfRows</code>	— eine geschachtelte Liste von höchstens m Zeilen; jede Zeile ist eine Liste von höchstens n Matrixeinträgen
<code>Array</code>	— ein ein- oder zweidimensionales Feld
<code>Matrix</code>	— eine Matrix, d. h. ein Objekt eines Datentyps der Kategorie <code>Cat::Matrix</code>
<code>m</code>	— die Zeilenanzahl: eine positive ganze Zahl
<code>n</code>	— die Spaltenanzahl: eine positive ganze Zahl
<code>List</code>	— eine Liste
<code>f</code>	— eine Funktion oder ein funktionaler Ausdruck mit zwei Argumenten
<code>g</code>	— eine Funktion oder ein funktionaler Ausdruck mit einem Argument

Optionen:

<code>Diagonal</code>	— erzeuge eine Diagonalmatrix
<code>Banded</code>	— erzeuge eine Band-Toeplitz-Matrix

Rückgabewert: eine Matrix vom Domain-Typ `Dom::Matrix()`.

Verwandte Funktionen: `array`, `DOM_ARRAY`, `Dom::Matrix`

Details:

- ☞ Über `matrix` lassen sich Matrizen und Vektoren erzeugen. Ein Vektor ist dabei entweder eine $m \times 1$ -Matrix (ein Spaltenvektor) oder eine $1 \times n$ -Matrix (ein Zeilenvektor).

Als Matrix- und Vektorkomponenten für mit `matrix` erzeugte Matrizen sind ausschließlich arithmetische Ausdrücke erlaubt. Für spezielle Komponentenbereiche wird auf die Hilfeseite des Domains `Dom::Matrix` verwiesen.
- ☞ Mit Matrizen wird über die gewöhnlichen arithmetischen Operatoren von MuPAD gerechnet.

Sind beispielsweise `A` und `B` zwei mit `matrix` erzeugte Matrizen, so bestimmt `A + B` die Matrixsumme und `A * B` das Matrixprodukt (passende Zeilen- und Spaltenzahlen vorausgesetzt).

Die Eingabe `A^(-1)` oder `1/A` liefert die Inverse von `A`, falls sie existiert (sonst wird `FAIL` als Ergebnis dieser Berechnung zurückgeliefert).

Siehe Beispiel ??.
- ☞ Eine Reihe an Systemfunktionen können auf Matrizen angewendet werden, so beispielsweise die Funktionen `map`, `subs`, `has`, `zip`, `conjugate` für die komplex-konjugierte Matrix, `norm` zur Bestimmung diverser Normen einer Matrix, und auch `exp` zur Berechnung der Exponentialmatrix. Siehe Beispiel ??.
- ☞ Die meisten Funktionen der Bibliothek `linalg` zur Linearen Algebra erwarten als Eingabe solche Matrizen. Der Aufruf `linalg::det(A)` liefert beispielsweise die Determinante einer mit `matrix` erzeugten quadratischen Matrix `A`. Mittels `linalg::gaussJordan(A)` wird der Gauß-Jordan-Algorithmus auf `A` angewendet, um `A` in eine reduzierte Zeilen-Stufen-Form zu transformieren. Siehe Beispiel ??.

Siehe die Hilfe zu `linalg`, um eine Übersicht über alle zur Verfügung stehenden Funktionen dieses Paketes zu erhalten.
- ☞ `matrix` ist eine abkürzende Schreibweise für das Domain `Dom::Matrix()`. Es wird auf die entsprechende Hilfeseite verwiesen, um weitere Details über diesen Datentyp zu erfahren.
- ☞ Auf Matrixeinträge wird über den gewöhnlichen Indexoperator `[]` zugegriffen, der auch auf Listen, Felder und Tabellen arbeitet. Der Aufruf `A[i, j]` extrahiert den Matrixeintrag der i -ten Zeile und j -ten Spalte.

Entsprechend können Zuweisungen an Matrixeinträge ausgeführt werden, d. h. der Aufruf `A[i, j] := c` ersetzt den Matrixeintrag der i -ten Zeile und j -ten Spalte durch den Wert von `c`.

Liegt einer der Indizes außerhalb des gültigen Bereiches, so reagiert MuPAD mit einer Fehlermeldung.

Mit dem Indexoperator können auch Teilmatrizen extrahiert werden. Der Aufruf `A[r1..r2, c1..c2]` liefert die Teilmatrix von A , die aus den Zeilen mit den Indizes $r_1, r_1 + 1, \dots, r_2$ und den Spalten mit den Indizes $c_1, c_1 + 1, \dots, c_2$ von A besteht.

Siehe Beispiele ?? und ??.

☞ `matrix(ListOfRows)` erzeugt eine $m \times n$ -Matrix mit Einträgen aus der verschachtelten Liste `ListOfRows`, wobei m die Anzahl der Teillisten ist. Jede Teilliste steht für eine Zeile der Matrix, und n ist die Länge der Liste mit den meisten Einträgen. Sowohl m als auch n müssen positiv sein.

Besteht eine Teilliste aus weniger als n Einträgen, so werden die restlichen Einträge der entsprechenden Matrixzeile mit Nullen initialisiert. Siehe Beispiel ??.

☞ `matrix(List)` erzeugt einen $m \times 1$ -Zeilenvektor mit Einträgen aus der angegebenen nichtleeren Liste. Dabei ist m die Anzahl an Einträgen von `List`. Siehe Beispiel ??.

☞ `matrix(Array)` und `matrix(Matrix)` erzeugen eine neue Matrix mit der Zeilen- und Spaltenzahl sowie den Einträgen von `Array` bzw. `Matrix`. Das Feld darf keine uninitialisierten Einträge enthalten. Ist `Array` ein eindimensionales Feld, so ist das Ergebnis ein Spaltenvektor. Siehe Beispiel ??.

☞ Der Aufruf `matrix(m, n)` liefert die $m \times n$ -Nullmatrix.

☞ `matrix(m, n, ListOfRows)` erzeugt eine $m \times n$ -Matrix mit Einträgen aus der Liste `ListOfRows`.

Sind $m \geq 2$ und $n \geq 2$, so darf die Liste `ListOfRows` aus (höchstens) m Teillisten bestehen, die jeweils nicht mehr als n Elemente enthalten. Jede Teilliste repräsentiert eine Zeile der Matrix.

Besteht eine Teilliste aus weniger als n Elementen, so werden die fehlenden Einträge der entsprechenden Matrixzeile mit Nullen vorinitialisiert. Sind weniger als m Teillisten angegeben, so werden für die fehlenden Matrixzeilen Nullzeilen verwendet.

☞ `matrix(1, n, List)` erzeugt den $1 \times n$ -Spaltenvektor mit den Elementen von `List` als Komponenten. Die Liste `List` darf aus höchstens n Einträgen bestehen. Fehlende Einträge werden durch Nullen ersetzt. Siehe Beispiel ??.

☞ `matrix(m, 1, List)` erzeugt den $m \times 1$ -Zeilenvektor mit den Elementen von `List` als Komponenten. Die Liste `List` darf aus höchstens m Einträgen bestehen. Fehlende Einträge werden durch Nullen ersetzt. Siehe Beispiel ??.

- ☞ `matrix(m, n, f)` erzeugt die Matrix, deren (i, j) -te Komponente der Funktionswert $f(i, j)$ ist. Dabei variieren der Zeilenindex i von 1 to m und der Spaltenindex j von 1 bis n . Siehe Beispiel ??.
-

Option *<Diagonal>*:

- ☞ Mit der Option *Diagonal* können Diagonalmatrizen erzeugt werden, deren Diagonalelemente aus einer Liste stammen oder über eine Funktion oder einen funktionaler Ausdruck berechnet werden.
- ☞ `matrix(m, n, List, Diagonal)` erzeugt die $m \times n$ -Diagonalmatrix, deren Hauptdiagonale aus den Elementen der Liste `List` besteht (siehe Beispiel ??).
- `List` darf nicht mehr als $\min(m, n)$ Elemente enthalten. Weniger Elemente werden durch Nullen ersetzt.
- ☞ `matrix(m, n, g, Diagonal)` erzeugt die Diagonalmatrix, deren i -tes Hauptdiagonalelement der Funktionswert $g(i)$ ist. Der Index i variiert dabei von 1 bis $\min(m, n)$. Siehe Beispiel ??.
-

Option *<Banded>*:

- ☞ Mit der Option *Banded* können Bandmatrizen erzeugt werden.
- Eine *Bandmatrix* besteht aus Nulleinträgen außerhalb der Hauptdiagonalen und einiger angrenzender Nebendiagonalen.
- ☞ `matrix(m, n, List, Banded)` erzeugt eine $m \times n$ -Band-Toeplitz-Matrix mit den Elementen aus `List` als Einträgen. Die Anzahl Elemente von `List` muss ungerade, etwa $2h + 1$, und nicht größer als n sein. Die Bandbreite der erzeugten Matrix ist höchstens h .
- Alle Elemente der Hauptdiagonalen der erzeugten Matrix werden mit dem mittleren Element von `List` initialisiert. Alle Elemente der i -ten Subdiagonale werden mit dem $(h + 1 - i)$ -ten Element von `List` initialisiert. Alle Elemente der i -ten Superdiagonale werden mit dem $(h + 1 + i)$ -ten Element von `List` initialisiert. Alle Einträge in den restlichen Sub- und Superdiagonalen werden auf Null gesetzt.
- Siehe Beispiel ??.
-

Beispiel 1. Wir erzeugen die 2×2 -Matrix

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

über eine Liste von Zeilen, wobei jede Zeile eine Liste von zwei Elementen ist, wie folgt:

```
>> A := matrix([[1, 5], [2, 3]])
```

$$\begin{array}{c} \begin{array}{cc} + - & - + \\ | & 1, 5 \\ | & \\ | & 2, 3 \\ | & \\ + - & - + \end{array} \end{array}$$

In gleicher Weise erzeugen wir eine zweite Matrix mit Zeilenzahl 2 und Spaltenzahl 3:

```
>> B := matrix([[-1, 5/2, 3], [1/3, 0, 2/5]])
```

$$\begin{array}{c} \begin{array}{ccc} + - & & - + \\ | & -1, 5/2, 3 & | \\ | & & | \\ | & 1/3, 0, 2/5 & | \\ + - & & - + \end{array} \end{array}$$

Über die gewöhnlichen arithmetischen Operatoren von MuPAD lassen sich Matrixberechnungen durchführen, so beispielsweise das Matrixprodukt $A \cdot B$, die 4. Potenz von A und die skalare Multiplikation von A mit $\frac{1}{3}$:

```
>> A * B, A^4, 1/3 * A
```

$$\begin{array}{c} \begin{array}{ccc} + - & - + & + - \\ | & 2/3, 5/2, 5 & | \\ | & & | \\ | & -1, 5, 36/5 & | \\ + - & - + & + - \end{array}, \begin{array}{ccc} + - & - + & + - \\ | & 281, 600 & | \\ | & & | \\ | & 240, 521 & | \\ + - & - + & + - \end{array}, \begin{array}{ccc} + - & - + & + - \\ | & 1/3, 5/3 & | \\ | & & | \\ | & 2/3, 1 & | \\ + - & - + & + - \end{array} \end{array}$$

Offensichtlich ist die Summe der beiden Matrizen A und B nicht definiert, was MuPAD mit einer Fehlermeldung quittiert:

```
>> A + B
```

```
Error: dimensions don't match [(Dom::Matrix(Dom::ExpressionFile\ld()))::_plus]
```

Die Inverse einer invertierbaren Matrix wird wie folgt berechnet:

```
>> 1/A
```

$$\begin{array}{c} \begin{array}{cc} + - & - + \\ | & -3/7, 5/7 \\ | & \\ | & 2/7, -1/7 \\ | & \\ + - & - + \end{array} \end{array}$$

Ist eine Matrix nicht invertierbar, so ist FAIL das Ergebnis dieser Operation:


```
>> C := matrix([[2, 0], [0, 0]])
```

```

+-      +-
|  2, 0  |
|        |
|  0, 0  |
+-      +-

```

```
>> C^(-1)
```

```
FAIL
```

Beispiel 2. Neben der Matrixarithmetik bietet die Bibliothek `linalg` eine Reihe an Funktionen zum Arbeiten mit Matrizen. Zum Beispiel bestimmt `linalg::rank` den Rang einer Matrix:

```
>> A := matrix([[1, 5], [2, 3]])
```

```

+-      +-
|  1, 5  |
|        |
|  2, 3  |
+-      +-

```

```
>> linalg::rank(A)
```

```
2
```

Die Funktion `linalg::eigenvectors` berechnet die Eigenwerte und Eigenvektoren der Matrix `A`:

```
>> linalg::eigenvectors(A)
```

```

-- --      -- +-      +-      -- -- --
|  |  |      |  |  |      1/2      |  |  |  |
|  |  |      |  |  |      11      |  |  |  |
|  |  |      |  |  |      ----- - 1/2
|  |  |      |  |  |      2
|  |  |      |  |  |      1
-- -- --      -- +-      +-      +- -- -- --
|  |  |      |  |  |      1/2      |  |  |  |
|  |  |      |  |  |      11      |  |  |  |
|  |  |      |  |  |      ----- - 1/2
|  |  |      |  |  |      2
|  |  |      |  |  |      1
-- -- --      -- +-      +-      +- -- -- --

```

Die Zeilen- und Spaltenzahl einer Matrix lässt sich über die Funktion `linalg::matdim` bestimmen:

```
>> linalg::matdim(A)
```

```
[2, 2]
```

Das Ergebnis ist eine Liste zweier positiver ganzer Zahlen, die Zeilen- und Spaltenzahl der Matrix.

Mittels `info(linalg)` erhält man eine Liste der zur Verfügung stehenden Funktionen und über `?linalg` Details über die Bibliothek `linalg`.

Beispiel 3. Auf Matrixeinträge wird über den gewöhnlichen und beispielsweise von Listen her bekannten Indexoperator `[]` zugegriffen:

```
>> A := matrix([[1, 2, 3, 4], [2, 0, 4, 1], [-1, 0, 5, 2]])
```

```

+-              +-
|      1, 2, 3, 4 |
|      2, 0, 4, 1 |
|     -1, 0, 5, 2 |
+-              +-

```

```
>> A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]
```

```
7
```

Entsprechend können Zuweisungen an Matrixeinträge ausgeführt werden:

```
>> A[1, 2] := a^2: A
```

```

+-              +-
|      2          |
|      1, a , 3, 4 |
|      2,  0, 4, 1 |
|     -1,  0, 5, 2 |
+-              +-

```

Neben der gewöhnlichen Indizierung von Matrixkomponenten können über den Indexoperator ganze Teilmatrizen extrahiert werden. Der folgende Aufruf extrahiert die Teilmatrix von A , die aus den Zeilen 2 bis 3 und den Spalten 1 bis 3 der Matrix A besteht:

```
>> A[2..3, 1..3]
```

$$\begin{array}{c} + - \qquad \qquad - + \\ | \quad 2, 0, 4 \quad | \\ | \quad \quad \quad | \\ | \quad -1, 0, 5 \quad | \\ + - \qquad \qquad - + \end{array}$$

Der Indexoperator erlaubt es nicht, Teilmatrizen an eine Matrix zuzuweisen. Das kann aber mit der Funktion `linalg::substitute` erreicht werden.

Beispiel 4. Bestimmte Systemfunktionen können auf Matrizen angewendet werden. Sollen beispielsweise Matrixkomponenten ausmultipliziert werden, so kann dafür die Funktion `expand` verwendet werden:

```
>> delete a, b:
A := matrix([
  [(a - b)^2, a^2 + b^2],
  [a^2 + b^2, (a - b)*(a + b)]
])
```

$$\begin{array}{c} + - \qquad \qquad \qquad - + \\ | \qquad \qquad \qquad 2 \qquad \qquad 2 \qquad 2 \qquad | \\ | \quad (a - b) \, , \qquad \quad a \, + \, b \qquad | \\ | \qquad \qquad \qquad | \\ | \qquad \qquad \qquad 2 \qquad 2 \qquad \qquad | \\ | \quad a \, + \, b \, , \, (a + b) \, (a - b) \quad | \\ + - \qquad \qquad \qquad - + \end{array}$$

```
>> expand(A)
```

$$\begin{array}{c} + - \qquad \qquad \qquad - + \\ | \qquad \qquad \qquad 2 \qquad 2 \qquad 2 \qquad 2 \qquad | \\ | \quad - 2 \, a \, b + a \, + \, b \, , \, a \, + \, b \qquad | \\ | \qquad \qquad \qquad | \\ | \qquad \qquad \qquad 2 \qquad 2 \qquad \qquad 2 \qquad 2 \qquad | \\ | \qquad \qquad \qquad a \, + \, b \, , \qquad \quad a \, - \, b \qquad | \\ + - \qquad \qquad \qquad - + \end{array}$$

Sollen die Matrixeinträge nach einer bestimmten Variablen differenziert werden, so kann das wie folgt geschehen:

```
>> diff(A, a)
```

$$\begin{array}{c} + - \qquad \qquad \qquad - + \\ | \quad 2 \, a - 2 \, b, \, 2 \, a \quad | \\ | \qquad \qquad \qquad | \\ | \qquad \qquad \qquad 2 \, a, \qquad 2 \, a \quad | \\ + - \qquad \qquad \qquad - + \end{array}$$

Man kann alle Matrixeinträge an einem gegebenen Punkt auswerten:

```
>> subs(A, a = 1, b = -1)
```

$$\begin{array}{cc} + - & - + \\ | & 4, 2 \\ | & \\ | & 2, 0 \\ | & \\ + - & - + \end{array}$$

Es ist zu beachten, dass die Funktion `subs` das Ergebnis der Substitution nicht evaluiert! Wir geben ein Beispiel anhand der folgenden Matrix:

```
>> A := matrix([[sin(x), x], [x, cos(x)]])
```

$$\begin{array}{cc} + - & - + \\ | & \sin(x), \quad x \\ | & \\ | & x, \quad \cos(x) \\ | & \\ + - & - + \end{array}$$

Dann substituieren wir in jeder Matrixkomponente x durch den Wert 0:

```
>> B := subs(A, x = 0)
```

$$\begin{array}{cc} + - & - + \\ | & \sin(0), \quad 0 \\ | & \\ | & 0, \quad \cos(0) \\ | & \\ + - & - + \end{array}$$

Man erkennt, dass die Matrixkomponenten nicht vollständig evaluiert worden sind: der Ausdruck `sin(0)` zum Beispiel wird bei direkter Eingabe sofort zu Null evaluiert.

Die Funktion `eval` kann verwendet werden, um das Ergebnis von `subs` nachträglich auszuwerten. Jedoch arbeitet diese Funktion nicht direkt auf Matrizen, und man muss die Funktion `map` verwenden, um `eval` auf jede Matrixkomponente anzuwenden:

```
>> map(B, eval)
```

$$\begin{array}{cc} + - & - + \\ | & 0, 0 \\ | & \\ | & 0, 1 \\ | & \\ + - & - + \end{array}$$

Die Funktion `zip` kann auch auf Matrizen angewendet werden. Der folgende Aufruf kombiniert zwei Matrizen A und B , indem jeder Eintrag von A durch den entsprechenden Eintrag der Matrix B dividiert wird:

```
>> A := matrix([[4, 2], [9, 3]]): B := matrix([[2, 1], [3, -
1]]):
      zip(A, B, '/')
```

```

+-      +-
|  2,  2 |
|      |
|  3, -3 |
+-      +-

```

Beispiel 5. Ein Vektor ist entweder eine $m \times 1$ -Matrix (ein Spaltenvektor) oder eine $1 \times n$ -Matrix (ein Zeilenvektor). Ein Vektor kann über `matrix` mit der Angabe seiner Dimension (Zeilen- und Spaltenzahl) und einer Liste von Vektorkomponenten erzeugt werden:

```
>> row_vector      := matrix(1, 3, [1, 2, 3]);
      column_vector := matrix(3, 1, [1, 2, 3])
```

```

+-      +-
| 1, 2, 3 |
+-      +-

```

```

+-      +-
|  1 |
|   |
|  2 |
|   |
|  3 |
+-      +-

```

Wenn das einzige Argument von `matrix` eine flache Liste oder ein eindimensionales Feld ist, dann wird ein Spaltenvektor zurückgegeben:

```
>> matrix([1, 2, 3])
```

```

+-      +-
|  1 |
|   |
|  2 |
|   |
|  3 |
+-      +-

```

Für einen Zeilenvektor `r` liefern die beiden Aufrufe `r[1, i]` und `r[i]` die i -te Vektorkomponente von `r`. Entsprechend liefern die Aufrufe `c[i, 1]` und `c[i]` die i -te Vektorkomponente eines Spaltenvektors `c`.

Um beispielsweise auf den zweiten Eintrag der Vektoren `row_vector` und `column_vector` zuzugreifen, genügt die Eingabe:

```
>> row_vector[2], column_vector[2]

2, 2
```

Über die Funktion `linalg::vecdim` lässt sich die Anzahl an Komponenten eines Vektors bestimmen:

```
>> linalg::vecdim(row_vector), linalg::vecdim(column_vector)

3, 3
```

Die Anzahl der Komponenten eines Vektors kann auch direkt über den Aufruf `nops(vector)` ermittelt werden.

Die Dimension des Vektors, d. h. seine Zeilen- und Spaltenzahl, lässt sich wie im Fall von Matrizen oben beschrieben ermitteln:

```
>> linalg::matdim(row_vector),
    linalg::matdim(column_vector)

[1, 3], [3, 1]
```

Wir verweisen auf die Bibliothek `linalg` für Funktionen, die mit Vektoren arbeiten. Mit der Systemfunktion `norm` werden Vektornormen berechnet.

Beispiel 6. In den folgenden Beispielen zeigen wir verschiedene Aufrufe von `matrix`, wie sie oben beschrieben sind. Wir starten mit einer Matrix, die über eine Liste von Zeilen beschrieben wird, wobei jede Zeile eine Liste von Zeilen-einträgen ist:

```
>> matrix([[1, 2], [2]])

+-      +-
|  1, 2  |
|        |
|  2, 0  |
+-      +-

```

Die Zeilenzahl ist die Anzahl an Teillisten, d. h. in diesem Beispiel ist $m = 2$. Die Spaltenzahl ist die Länge der Teilliste mit den meisten Einträgen, hier also die erste Teilliste mit zwei Einträgen. Die zweite Teilliste hat nur ein Element, weswegen der zweite Eintrag in der zweiten Matrixzeile auf Null gesetzt worden ist.

Im folgenden Aufruf verwenden wir dieselbe geschachtelte Liste, übergeben aber zusätzlich zwei Dimensionsparameter, um eine 4×4 -Matrix zu erzeugen:

```
>> matrix(4, 4, [[1, 2], [2]])
```

```

+-          +-
|  1, 2, 0, 0  |
|  2, 0, 0, 0  |
|  0, 0, 0, 0  |
|  0, 0, 0, 0  |
+-          +-

```

In diesem Fall ist die Dimension der Matrix durch die Dimensionsparameter gegeben. Wie vorher werden fehlende Einträge in einer Teilliste durch Nullen ersetzt. Fehlende Zeilen werden als Nullzeilen angesehen.

Beispiel 7. Ein ein- oder zweidimensionales Feld aus arithmetischen Ausdrücken, wie z. B.:

```

>> a := array(1..3, 2..4,
  [[1, 1/3, 0], [-2, 3/5, 1/2], [-3/2, 0, -1]]
)

```

```

+-          +-
|  1, 1/3, 0  |
| -2, 3/5, 1/2 |
| -3/2, 0, -1 |
+-          +-

```

kann wie folgt in eine Matrix konvertiert werden:

```

>> A := matrix(a)

```

```

+-          +-
|  1, 1/3, 0  |
| -2, 3/5, 1/2 |
| -3/2, 0, -1 |
+-          +-

```

Felder dienen beispielsweise als ein effizienter strukturierter Datentyp zur Programmierung. Sie besitzen jedoch keinerlei algebraische Bedeutung. Mathematische Operationen sind für Felder nicht definiert. Liegt das Feld aber in Form einer Matrix vor, so steht die gesamte Funktionalität wie oben beschrieben zur Verfügung. Beispielsweise läßt sich dann die Matrix $2A - A^2$ oder die Frobenius-Norm von A ausrechnen:

```

>> 2*A - A^2, norm(A, Frobenius)

```

$$\begin{array}{c} \begin{array}{cc} + - & - + \\ \left| \begin{array}{ccc} 5/3, & 2/15, & -1/6 \\ -1/20, & 113/75, & 6/5 \\ -3, & 1/2, & -3 \end{array} \right| & \begin{array}{cc} 1/2 & 1/2 \\ 450 & 4037 \\ \hline 450 \end{array} \\ + - & - + \end{array} \end{array}, \quad \begin{array}{c} \begin{array}{cc} + - & - + \\ \left| \begin{array}{ccc} 5/3, & 2/15, & -1/6 \\ -1/20, & 113/75, & 6/5 \\ -3, & 1/2, & -3 \end{array} \right| & \begin{array}{cc} 1/2 & 1/2 \\ 450 & 4037 \\ \hline 450 \end{array} \\ + - & - + \end{array} \end{array}$$

Felder können uninitialisierte Einträge enthalten:

```
>> b := array(1..4): b[1] := 2: b[4] := 0: b
```

$$\begin{array}{c} \begin{array}{cc} + - & - + \\ \left| \begin{array}{ccc} 2, & ?[2], & ?[3], & 0 \end{array} \right| \\ + - & - + \end{array} \end{array}$$

Die Funktion `matrix` jedoch erlaubt keine Matrizen (bzw. Vektoren) mit uninitialisierten Komponenten und reagiert daher in solchen Fällen mit einer Fehlermeldung:

```
>> matrix(b)
```

```
Error: unable to define matrix over Dom::ExpressionField() [(D\
om::Matrix(Dom::ExpressionField()))::new]
```

Wir initialisieren die fehlenden Einträge des Feldes und konvertieren das Resultat in eine Matrix, genauer gesagt, in einen Spaltenvektor:

```
>> b[2] := 0: b[3] := -1: matrix(b)
```

$$\begin{array}{c} \begin{array}{cc} + - & - + \\ \left| \begin{array}{ccc} 2 \\ 0 \\ -1 \\ 0 \end{array} \right| \\ + - & - + \end{array} \end{array}$$

Beispiel 8. Dieses Beispiel zeigt die Erzeugung einer Matrix, deren Matrixkomponenten durch die Werte einer Indexfunktion definiert sind. Der Eintrag in der i -ten Zeile und der j -ten Spalte einer Hilbert-Matrix (siehe `linalg::hilbert`) ist $1/(i+j-1)$. Der folgende Befehl erzeugt also eine 2×2 -Hilbert-Matrix:

```
>> matrix(2, 2, (i, j) -> 1/(i + j - 1))
```


$$\begin{array}{c} + - \qquad \qquad - + \\ | \quad 1, \quad 1/2 \quad | \\ | \qquad \qquad \qquad | \\ | \quad 1/2, \quad 1/3 \quad | \\ + - \qquad \qquad - + \end{array}$$

Die folgenden zwei Aufrufe produzieren unterschiedliche Ergebnisse. Im ersten Aufruf wird x als unbekannte Funktion aufgefasst, während es im zweiten Aufruf eine Konstante ist:

```
>> delete x:
      matrix(2, 2, x), matrix(2, 2, (i, j) -> x)
```

$$\begin{array}{c} + - \qquad \qquad - + \quad + - \qquad \qquad - + \\ | \quad x(1, 1), \quad x(1, 2) \quad | \quad | \quad x, \quad x \quad | \\ | \qquad \qquad \qquad | \quad , \quad | \qquad \qquad \qquad | \\ | \quad x(2, 1), \quad x(2, 2) \quad | \quad | \quad x, \quad x \quad | \\ + - \qquad \qquad - + \quad + - \qquad \qquad - + \end{array}$$

Beispiel 9. Diagonalmatrizen können mit der Option *Diagonal* und einer Liste von Diagonaleinträgen erzeugt werden:

```
>> matrix(3, 4, [1, 2, 3], Diagonal)
```

$$\begin{array}{c} + - \qquad \qquad - + \\ | \quad 1, \quad 0, \quad 0, \quad 0 \quad | \\ | \qquad \qquad \qquad | \\ | \quad 0, \quad 2, \quad 0, \quad 0 \quad | \\ | \qquad \qquad \qquad | \\ | \quad 0, \quad 0, \quad 3, \quad 0 \quad | \\ + - \qquad \qquad - + \end{array}$$

Die 3×3 -Einheitsmatrix lässt sich mit der folgenden Eingabe erzeugen:

```
>> matrix(3, 3, [1 $ 3], Diagonal)
```

$$\begin{array}{c} + - \qquad \qquad - + \\ | \quad 1, \quad 0, \quad 0 \quad | \\ | \qquad \qquad \qquad | \\ | \quad 0, \quad 1, \quad 0 \quad | \\ | \qquad \qquad \qquad | \\ | \quad 0, \quad 0, \quad 1 \quad | \\ + - \qquad \qquad - + \end{array}$$

Auch der folgende Aufruf mit einer einstelligen Funktion liefert das Gewünschte:

```
>> matrix(3, 3, i -> 1, Diagonal)
```

```

+-      +-
|  1, 0, 0  |
|  0, 1, 0  |
|  0, 0, 1  |
+-      +-

```

Da die Zahl 1 auch als konstante Funktion aufgefaßt werden kann, erzeugt der folgende kürzere Aufruf ebenfalls dieselbe Matrix:

```
>> matrix(3, 3, 1, Diagonal)
```

```

+-      +-
|  1, 0, 0  |
|  0, 1, 0  |
|  0, 0, 1  |
+-      +-

```

Beispiel 10. Band-Toeplitz-Matrizen (siehe oben) können über die Angabe der Option *Banded* erzeugt werden. Hier wird eine Dreibandmatrix mit dem Eintrag 2 auf der Hauptdiagonalen und dem Eintrag -1 auf den beiden ersten Nebendiagonalen erzeugt:

```
>> matrix(4, 4, [-1, 2, -1], Banded)
```

```

+-      +-
|  2, -1,  0,  0  |
| -1,  2, -1,  0  |
|  0, -1,  2, -1  |
|  0,  0, -1,  2  |
+-      +-

```

Änderungen:

⌘ `matrix` ist eine neue Funktion.

`map` – Anwenden einer Funktion auf alle Operanden eines Objekts

`map(object, f)` wendet die Funktion `f` auf alle Operanden von `object` an.

Aufruf(e):

⌘ `map(object, f <, p1, p2, ...>)`

Parameter:

<code>object</code>	— ein beliebiges MuPAD Objekt
<code>f</code>	— eine Funktion
<code>p1, p2, ...</code>	— beliebige MuPAD-Objekte, die als zusätzliche Argumente für <code>f</code> gültig sind


Rückgabewert: eine Kopie von `object`, in der `f` auf alle Operanden angewandt wurde.

Überladbar durch: `object`

Verwandte Funktionen: `eval`, `mapcoeffs`, `misc::maprec`, `op`, `select`, `split`, `subs`, `subsex`, `subsop`, `zip`

Details:

- ⌘ `map(object, f)` gibt eine Kopie von `object` zurück, in der jeder Operand `x` von `object` durch den `f(x)` ersetzt wurde. Das Argument `object` wird dabei nicht verändert (siehe Beispiel ??).
- ⌘ Das zweite Argument `f` muss eine Prozedur erzeugt durch `->` oder `proc` (z. B. `x -> x^2 + 1`), eine Funktionsumgebung (z. B. `sin`) oder ein funktionaler Ausdruck sein (z. B. `sin@exp + 2*id`).
- ⌘ Wenn zusätzliche optionale Argumente angegeben werden, wird jeder Operand `x` von `object` durch den Aufruf `f(x, p1, p2, ...)` ersetzt (siehe Beispiel ??).
- ⌘ Es ist möglich, einen Operator wie `+` oder `*` auf die Operanden von `object` anzuwenden. Dazu muss die dem Operator entsprechende Funktion wie `_plus` oder `_mult` benutzt werden (siehe Beispiel ??).
- ⌘ Im Gegensatz zu `op` zerlegt `map` keine gebrochenen oder komplexen Zahlen. Wenn das Argument eine gebrochene oder komplexe Zahl ist, wird `f` auf die Zahl selbst und nicht den Zähler und Nenner oder den reellen und imaginären Teil der Zahl angewandt (siehe Beispiel ??).
- ⌘ Wenn `object` eine Zeichenkette ist, wird `f` auf die Zeichenkette angewandt, nicht auf die einzelnen Zeichen (siehe Beispiel ??).
- ⌘ Wenn `object` ein Ausdruck ist, wird `f` auf alle Operanden angewandt, die von `op` zurückgegeben werden (siehe Beispiel ??).

- ⌘ Eine Ausdruckssequenz als erstes Argument wird nicht ausgeglichen (siehe Beispiel ??).
- ⌘ Wenn `object` ein Polynom ist, wird `f` auf das Polynom selbst angewendet, nicht auf die Koeffizienten (siehe Beispiel ??). Dazu kann `mapcoeffs` benutzt werden.
- ⌘ Wenn `object` eine Liste, eine Menge oder ein Feld ist, wird `f` auf alle Elemente von `object` angewendet.
- ⌘ Ist `object` eine Tabelle, wird die Funktion `f` auf alle *Einträge* der Tabelle angewendet, nicht auf die Indizes (die rechte Seite der Operanden einer Tabelle sind die Einträge). 
- ⌘ Wenn `object` ein Element eines Domains ist, wird die Methode "map" dieses Domains mit denselben Argumenten aufgerufen und deren Ergebnis zurückgegeben. Damit kann die Funktionalität von `map` für Benutzerdefinierte Domains erweitert werden. Wenn diese Methode nicht definiert ist, wird `f` auf das Objekt selbst angewendet (siehe Beispiel ??).
- ⌘ Nach der Ersetzung der Operanden evaluiert `map` das Ergebnis nicht noch einmal. Dazu kann `eval` verwendet werden. Interne Vereinfachungen werden aber durchgeführt (siehe Beispiel ??).
- ⌘ `map` wird nicht rekursiv auf Teiloperanden der Operanden von `object` angewandt, nur auf die durch `op(object)` direkt zurückgegebenen Operanden. Die Funktion `misc::maprec` ist eine rekursive Version von `map` (siehe Beispiel ??).
- ⌘ `map` ist eine Funktion des Systemkerns.

Beispiel 1. `map` arbeitet mit Ausdrücken:

```
>> map(a + b + 3, sin)

      sin(a) + sin(b) + sin(3)
```

Die optionalen Argumente werden der Funktion als zusätzliche Argumente übergeben:

```
>> map(a + b + 3, f, x, y)

      f(a, x, y) + f(b, x, y) + f(3, x, y)
```

Im folgenden Beispiel wird zu jedem Element der Liste 10 addiert:

```
>> map([1, x, 2, y, 3, z], _plus, 10)

      [11, x + 10, 12, y + 10, 13, z + 10]
```

Beispiel 2. Wie die meisten anderen MuPAD-Funktionen verändert `map` sein erstes Argument nicht, sondern liefert eine geänderte Kopie:

```
>> a := [0, PI/2, PI, 3*PI/2]:
      map(a, sin)

      [0, 1, 0, -1]
```

Die Liste `a` hat den ursprünglichen Wert:

```
>> a

      --      PI      3 PI --
      |  0,  --,  PI,  ----  |
      --      2      2    --
```

Beispiel 3. `map` zerlegt rationale und komplexe Zahlen nicht:

```
>> map(3/4, _plus, 1), map(3 + 4*I, _plus, 1)

      7/4, 4 + 4 I
```

`map` zerlegt Zeichenketten nicht:

```
>> map("MuPAD", text2expr)

      MuPAD
```

`map` zerlegt keine Polynome:

```
>> map(poly(x^2 + x + 1), _plus, 1)

      2
      poly(x  + x + 1, [x]) + 1

>> mapcoeffs(poly(x^2 + x + 1), _plus, 1)

      2
      poly(2 x  + 2 x + 2, [x])
```

Beispiel 4. Das erste Argument wird nicht ausgeglichen:

```
>> map((1, 2, 3), _plus, 2)

      3, 4, 5
```

Beispiel 5. Einige Funktionen geben mitunter sehr große Ausdrücke zurück, die mathematische Konstanten etc. enthalten. Das Anwenden der Funktion `float` kann solche Ausdrücke teilweise drastisch verkleinern, um eine Veranschaulichung des Ergebnisses zu vermitteln:

```
>> solve(x^4 + x^2 + PI, x)
```

$$\left\{ -\frac{\sqrt[1/2]{2} \sqrt[1/2]{(1 - 4\pi) - 1} \sqrt[1/2]{2} \sqrt[1/2]{(1 - 4\pi) - 1}}{2}, \frac{\sqrt[1/2]{2} \sqrt[1/2]{(1 - 4\pi) - 1} \sqrt[1/2]{2} \sqrt[1/2]{(1 - 4\pi) - 1}}{2}, \right.$$

$$\left. -\frac{\sqrt[1/2]{2} \sqrt[1/2]{(- (1 - 4\pi) - 1)} \sqrt[1/2]{2} \sqrt[1/2]{(- (1 - 4\pi) - 1)}}{2}, \frac{\sqrt[1/2]{2} \sqrt[1/2]{(- (1 - 4\pi) - 1)} \sqrt[1/2]{2} \sqrt[1/2]{(- (1 - 4\pi) - 1)}}{2} \right\}$$

```
>> map(%, float)
```

```
{- 0.7976383425 - 1.065939457 I,
```

```
  - 0.7976383425 + 1.065939457 I,
```

```
  0.7976383425 - 1.065939457 I, 0.7976383425 + 1.065939457 I}
```

Beispiel 6. Im nächsten Beispiel werden alle in der aktuellen MuPAD-Sitzung vom Benutzer verwendeten Bezeichner gelöscht. Der Befehl `anames(All, User)` gibt eine Menge mit allen Benutzer-definierten Bezeichnern zurück. Die Systemfunktion `_delete` wird auf die Operanden der zurückgegebenen Liste angewandt und löscht alle Variablen, die vom Benutzer einen Wert zugewiesen bekommen haben. Der Funktionsaufruf gibt die leere Menge zurück, da `_delete` den Wert `null()` zurückgibt:

```
>> x := 3: y := 5: x + y
```

8

```
>> map(anames(All, User), _delete)
```

{}

```
>> x + y
```

```
x + y
```

Beispiel 7. Es ist möglich, verschiedene Aktionen mit allen Elementen einer Datenstruktur mit einem einzigen `map`-Aufruf durchzuführen. Dazu kann als zweites Argument `f` eine Benutzer-definierte Prozedur angegeben werden. Im folgenden Beispiel wird der Sachverhalt „eine ganze Zahl $n \geq 2$ ist prim genau dann, wenn $\varphi(n) = n - 1$ gilt“ für alle Zahlen zwischen 1 und 10 überprüft. Dabei bezeichnet φ die eulersche Phi-Funktion. Dazu wird das Ergebnis von `isprime(n)` mit der Gültigkeit der Gleichung $\varphi(n) = n - 1$ für alle ganzen Zahlen 2 bis 9 verglichen:

```
>> map([2, 3, 4, 5, 6, 7, 8, 9],
      n -> bool(isprime(n) = bool(numlib::phi(n) = n - 1)))
      [TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE]
```

Beispiel 8. Das Ergebnis von `map` wird nicht weiter evaluiert. Falls dies gewünscht wird, muss `eval` aufgerufen werden:

```
>> map(sin(5), float);
      eval(%)
      sin(5.0)
      -0.9589242747
```

Interne Vereinfachungen des MuPAD-Kerns wie z. B. die Berechnung von arithmetischen Ausdrücken mit numerischen Operanden werden immer durchgeführt. Der folgende Aufruf ersetzt `sin(5)` und `cos(5)` durch die entsprechenden Ausdrücke mit Gleitpunktzahlen wie im letzten Beispiel, das System vereinfacht daraufhin automatisch die Summe:

```
>> map(sin(5) + cos(5), float)
      -0.6752620892
```

Beispiel 9. `map` angewendet auf eine Tabelle ändert nur die rechte Seite (die Einträge) der Operanden der Tabelle. Die Einträge stellen Nettopreise dar, die Steuer (in diesem Fall 16 Prozent) soll addiert werden:

```
>> T := table(1 = 65, 2 = 28, 3 = 42):
      map(T, _mult, 1.16)
```

```

table(
  3 = 48.72,
  2 = 32.48,
  1 = 75.4
)

```

Beispiel 10. `map` kann für Elemente von Domains überladen werden, indem der Slot "map" definiert wird. In diesem Beispiel ist `d` ein Domain, dessen Elemente zwei ganze Zahlen enthalten: einen Index und einen Eintrag (wie bei einer Tabelle). Für vereinfachte Eingabe und lesbare Ausgabe werden die Slots "new" und "print" definiert:

```

>> d := newDomain("d"):
    d::new := () -> new(d, args()):
    d::print := object -> _equal(extop(object)):
    d(1, 65), d(2, 28), d(3, 42)

1 = 65, 2 = 28, 3 = 42

```

Ohne einen Slot "map" wird die Funktion `f` auf das Domainelement selbst angewendet. Da `d` keine `_mult`-Methode hat, ist das Ergebnis der symbolische `_mult`-Aufruf:

```

>> map(d(1, 65), _mult, 1.16),
    type(map(d(1, 65), _mult, 1.16))

1.16 (1 = 65), "_mult"

```

Die Methode `map` des Domains soll die gegebene Funktion nur auf den zweiten Operanden eines Domainelementes anwenden. Das Domain `d` bekommt einen Slot "map", dann arbeitet `map` mit Elementen des Domains `d` wie gewünscht:

```

>> d::map := proc(obj, f)
    begin
        if args(0) > 2 then
            d(extop(obj, 1), f(extop(obj, 2), args(3..args(0))))
        else
            d(extop(obj, 1), f(extop(obj, 2)))
        end_if
    end_proc:
    map(d(1, 65), _mult, 1.16),
    map(d(2, 28), _mult, 1.16),
    map(d(3, 42), _mult, 1.16)

1 = 75.4, 2 = 32.48, 3 = 48.72

```


Beispiel 11. `map` arbeitet nicht rekursiv. Im nächsten Beispiel soll eine verschachtelte Liste „entschachtelt“ werden. `map` wendet die Funktion `op` auf jeden Operanden der gegebenen Liste `l` an. Dadurch wird eine Liste durch die Sequenz ihrer Elemente ersetzt. Damit können aber nicht Listen innerhalb von Listen „entschachtelt“ werden:

```
>> l := [1, [2, [3]], [4, [5]]]:
      map(l, op)

      [1, 2, [3], 4, [5]]
```

In diesem Fall kann die Funktion `misc::maprec` verwendet werden:

```
>> [misc::maprec(l, {DOM_LIST} = op)]

      [1, 2, 3, 4, 5]
```

Änderungen:

☞ Keine Änderungen.

`mapcoeffs` – Anwenden einer Funktion auf die Koeffizienten eines Polynoms

`mapcoeffs(p, F, a1, a2, ...)` wendet die Funktion `F` auf das Polynom `p` an, indem jeder Koeffizient `c` in `p` durch `F(c, a1, a2, ...)` ersetzt wird.

Aufruf(e):

☞ `mapcoeffs(p, F <, a1, a2, ...>)`
 ☞ `mapcoeffs(f, <vars, > F <, a1, a2, ...>)`

Parameter:

<code>p</code>	— ein Polynom vom Typ <code>DOM_POLY</code>
<code>F</code>	— eine Prozedur
<code>a1, a2, ...</code>	— weitere Parameter für die Funktion <code>F</code>
<code>f</code>	— ein polynomialer Ausdruck
<code>vars</code>	— eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck oder `FAIL`.

Überladbar durch: `p`, `f`

Verwandte Funktionen: `coeff`, `degree`, `degreevec`, `lcoeff`, `ldegree`, `lterm`, `map`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `tcoeff`

Details:

- ⌘ Für ein mittels `poly` erzeugtes Polynom `p` vom Typ `DOM_POLY` muss die Funktion `F` Argumente aus dem Koeffizientenring von `p` akzeptieren und entsprechende Werte zurückliefern.
 - ⌘ Ein polynomialer Ausdruck `f` wird zunächst mittels `poly` in ein Polynom in den Unbestimmten `vars` konvertiert. Werden keine Unbestimmten angegeben, so werden diese in `f` gesucht. Siehe `poly` zu Details bezüglich der Konvertierung. Falls `f` nicht in ein Polynom konvertiert werden kann, so liefert `mapcoeffs` den Wert `FAIL`. Nach Anwendung der Funktion `F` wird das Ergebnis wieder in einen Ausdruck konvertiert.
 - ⌘ `mapcoeffs` evaluiert seine Argumente. Man beachte jedoch, dass Polynome vom Typ `DOM_POLY` aus Effizienzgründen ihre Koeffizienten nicht evaluieren. Siehe Beispiel ??.
 - ⌘ `mapcoeffs` ist eine Funktion des Systemkerns.
-

Beispiel 1. Die Sinus-Funktion wird auf die Koeffizienten eines polynomialen Ausdrucks in den Unbestimmten `x` und `y` angewendet:

```
>> mapcoeffs(3*x^3 + x^2*y^2 + 2, sin)

          3          2  2
sin(2) + x sin(3) + x y sin(1)
```

Der folgende Aufruf veranlasst `mapcoeffs`, diesen Ausdruck als Polynom in `x` aufzufassen. Damit wird `y` zu einem Parameter, der in die Koeffizienten eingeht:

```
>> mapcoeffs(3*x^3 + x^2*y^2 + 2, [x], sin)

          3          2      2
sin(2) + x sin(3) + x sin(y )
```

Die Systemfunktion `_plus` addiert ihre Argumente. Im folgenden Aufruf wird sie verwendet, um den Wert 2 zu allen Koeffizienten zu addieren. Hierzu wird dieser Wert als zusätzliches Argument übergeben:

```
>> mapcoeffs(c1*x^3 + c2*x^2*y^2 + c3, [x, y], _plus, 2)

          3          2  2
c3 + x (c1 + 2) + x y (c2 + 2) + 2
```

Beispiel 2. Die Sinus-Funktion wird auf die Koeffizienten eines Polynoms in den Unbestimmten x und y angewendet:

```
>> mapcoeffs(poly(3*x^3 + x^2*y^2 + 2, [x, y]), sin)
```

$$\text{poly}(\sin(3) x^3 + \sin(1) x^2 y^2 + \sin(2), [x, y])$$

Im folgenden Aufruf hat das Polynom die Unbestimmte x . Dabei wird y als Parameter aufgefasst, der in die Koeffizienten eingeht:

```
>> mapcoeffs(poly(3*x^3 + x^2*y^2 + 2, [x]), sin)
```

$$\text{poly}(\sin(3) x^3 + \sin(y) x^2 + \sin(2), [x])$$

Eine benutzerdefinierte Funktion wird auf ein Polynom angewendet:

```
>> F := (c, a1, a2) -> exp(c + a1 + a2):
      mapcoeffs(poly(x^3 + c*x, [x]), F, a1, a2)
```

$$\text{poly}(\exp(a1 + a2 + 1) x^3 + \exp(c + a1 + a2) x, [x])$$

```
>> delete F:
```

Beispiel 3. Es wird ein Polynom über dem Restklassenring modulo 7 betrachtet:

```
>> p := poly(x^3 + 2*x*y, [x, y], Dom::IntegerMod(7)):
```

Die anzuwendende Funktion muss Werte im Koeffizientenring des Polynoms erzeugen:

```
>> mapcoeffs(p, c -> c^2)
```

$$\text{poly}(x^3 + 4 x^2 y, [x, y], \text{Dom}::\text{IntegerMod}(7))$$

Der folgende Aufruf benutzt eine Funktion, die ihr Argument in einen anderen Restklassenring konvertiert. Die Rückgabewerte sind keine zulässigen Koeffizienten von p :

```
>> mapcoeffs(p, c -> Dom::IntegerMod(3)(expr(c)))
```

FAIL

```
>> delete p:
```

Beispiel 4. Man beachte, dass Polynome vom Typ `DOM_POLY` ihre Koeffizienten nicht evaluieren:

```
>> delete a, x: p := poly(a*x, [x]): a := PI: p
      poly(a x, [x])
```

Die Auswertung kann mittels der Funktion `eval` erzwungen werden:

```
>> mapcoeffs(p, eval)
      poly(PI x, [x])
```

Nun wird die Sinus-Funktion auf die Koeffizienten von `p` angewendet. Das Polynom wertet seinen Koeffizienten `sin(a)` nicht zu 0 aus:

```
>> mapcoeffs(p, sin)
      poly(sin(a) x, [x])
```

Die Komposition der Funktionen `sin` und `eval` wird auf die Koeffizienten des Polynoms angewendet:

```
>> mapcoeffs(p, eval@sin)
      poly(0, [x])
```

```
>> delete p, a:
```

Änderungen:

☞ Keine Änderungen.

`maprat` – Anwendung einer Funktion auf einen „rationalisierten“ Ausdruck

`maprat(object, f)` wendet die Funktion `f` auf das „rationalisierte“ Objekt `object` an.

Aufruf(e):

☞ `maprat(object, f <, inspect <, stop>>)`

Parameter:

<code>object</code>	— ein arithmetischer Ausdruck oder eine Folge oder eine Menge oder eine Liste solcher Ausdrücke
<code>f</code>	— eine Prozedur oder ein funktionaler Ausdruck
<code>inspect, stop</code>	— Mengen von Typen oder Prozeduren

Rückgabewert: ein durch die Funktion f bestimmtes Objekt.

Verwandte Funktionen: `map`, `rationalize`

Details:

- ☞ `maprat(object, f, inspect, stop)` berechnet zunächst den „rationalisierten Ausdruck“ `rationalize(object, inspect, stop)`. Dies ist ein rationaler Ausdruck in „temporären Variablen“ (siehe `rationalize`). Die Funktion f wird auf diesen Ausdruck angewendet, zuletzt werden die „temporären Variablen“ im von f erzeugten Objekt wieder durch die ursprünglichen Teilausdrücke ersetzt.
 - ☞ Die Bedeutung und die Standardwerte der Parameter `inspect` und `stop` sind der Hilfeseite für `rationalize` zu entnehmen.
-

Beispiel 1. Die Funktion `partfrac` berechnet eine Partialbruchzerlegung eines rationalen Ausdrucks. Sie kann nicht auf nicht-rationale Ausdrücke angewendet werden:

```
>> object := cos(x)/(cos(x)^2 - sin(x)^2): partfrac(object, x)

Error: not a rational function [partfrac]
```

Man kann den Ausdruck „rationalisieren“, um `partfrac` anwenden zu können:

```
>> rat := rationalize(object)

          D1
-----, {D1 = cos(x), D2 = sin(x)}
      2      2
D1  - D2
```

Wir berechnen die Partialbruchzerlegung des rationalisierten Ausdrucks und ersetzen zuletzt die „temporären Variablen“ $D1$, $D2$ durch die Originalausdrücke:

```
>> part := partfrac(op(rat, 1), D1)

          1          1
----- - -----
      2 (D1 + D2)    2 (D2 - D1)

>> subs(part, op(rat, 2))

          1          1
----- - -----
      2 (cos(x) + sin(x))    2 (sin(x) - cos(x))
```

maprat dient zur Abkürzung dieser Befehlsfolge. Wir definieren eine Funktion f, welche die Partialbruchzerlegung ihres Arguments bezüglich der ersten von indets gefundenen Unbestimmten berechnet:

```
>> f := object -> partfrac(object, indets(object)[1]):
```

maprat wendet diese Funktion nach interner Rationalisierung an:

```
>> maprat(object, f)
```

$$\frac{1}{2 (\cos(x) + \sin(x))} - \frac{1}{2 (\sin(x) - \cos(x))}$$

```
>> delete object, rat, part, f:
```

Beispiel 2. Wir wenden die Funktion gcd auf zwei rationalisierte Ausdrücke an. Das erste Argument für maprat ist eine Folge zweier Ausdrücke p, q, welche gcd als Argumente übernimmt. Man beachte die Klammerung der Folge p, q:

```
>> p := (x - sqrt(2))*(x^2 + sqrt(3)*x - 1):
    q := (x - sqrt(2))*(x - sqrt(3)):
    maprat((p, q), gcd)
```

$$\frac{1}{2} - x$$

```
>> delete p, q:
```

Änderungen:

☞ Keine Änderungen.

max – das Maximum von Zahlen

max(x1, x2, ...) ergibt das Maximum der Zahlen x_1, x_2, \dots

Aufruf(e):

☞ max(x1, x2, ...)

Parameter:

x1, x2, ... — beliebige MuPAD-Objekte

Rückgabewert: eines der Argumente oder ein symbolischer `max`-Aufruf.

Überladbar durch: `x1`, `x2`, ...

Verwandte Funktionen: `_leequal`, `_less`, `min`, `sysorder`

Details:

- ⌘ Wenn alle Argumente von `max` ganze, rationale oder Gleitkommazahlen sind, gibt `max` das numerische Maximum dieser Argumente zurück.
 - ⌘ Der Aufruf `max()` ist nicht erlaubt und ergibt eine Fehlermeldung. Wird `max` mit nur einem Argument `x1` aufgerufen, wird `x1` evaluiert zurückgegeben (siehe Beispiel ??).
 - ⌘ Wenn ein Argument `infinity` ist, wird `infinity` zurückgegeben. Ist ein Argument `-infinity`, wird dieses Argument aus der Liste der Argumente entfernt (siehe Beispiel ??).
 - ⌘ `max` gibt einen Fehler zurück, wenn eines der Argumente eine komplexe Zahl ist (siehe Beispiel ??).
 - ⌘ Wenn ein Argument keine Zahl ist, dann wird ein symbolischer `max`-Aufruf mit dem Maximum aller numerischen Argumente und allen verbleibenden evaluierten Argumenten zurückgegeben (siehe Beispiel ??).
Geschachtelte `max`-Aufrufe mit symbolischen Argumenten werden in einen einzigen `max`-Aufruf umgeschrieben (sie werden ausgeglichen).
Siehe Beispiel ??.
 - ⌘ Eigenschaften von Bezeichnern werden von `max` *nicht* berücksichtigt. Mit der Funktion `simplify` können solche Ausdrücke weiter vereinfacht werden, die Bezeichner mit Eigenschaften enthalten (siehe Beispiel ??).
 - ⌘ `max` ist eine Funktion des Systemkerns.
-

Beispiel 1. `max` berechnet das Maximum von ganzen, rationalen und Gleitkommazahlen:

```
>> max(-3/2, 7, 1.4)
```

7

Wenn ein symbolischer Ausdruck unter den Argumenten ist, wird ein symbolischer `max`-Aufruf zurückgeliefert:

```
>> delete b: max(-4, b + 2, 1, 3)
```

```
max(b + 2, 3)
```

```
>> max(sqrt(2), 1)
```

$$\max\left(2^{\frac{1}{2}}, 1\right)$$

Man kann `simplify` benutzen, um `max`-Ausdrücke mit konstanten symbolischen Argumenten zu vereinfachen:

```
>> simplify(%)
```

$$\frac{1}{2}$$

Beispiel 2. Ein `max`-Aufruf mit einem Argument gibt das Argument evaluiert zurück:

```
>> delete a: max(a), max(sin(2*PI)), max(2)
```

$$a, 0, 2$$

Komplexe Zahlen führen zu einer Fehlermeldung:

```
>> max(0, 1, I)
```

```
Error: Illegal argument [max]
```

Beispiel 3. `infinity` ist immer das Maximum beliebiger Argumente:

```
>> delete x: max(1000000000000, infinity, x)
```

$$\text{infinity}$$

`-infinity` wird aus der Argumentliste entfernt:

```
>> max(1000000000000, -infinity, x)
```

$$\max(x, 1000000000000)$$

Beispiel 4. `max` berücksichtigt keine durch `assume` definierten Eigenschaften von Bezeichnern:

```
>> delete a, b, c:
  assume(a > 0): assume(b > a, _and): assume(c > b, _and):
  max(a, max(b, c), 0)
```


`max(a, b, c, 0)`

Erst eine Anwendung von `simplify` liefert das gewünschte Ergebnis:

```
>> simplify(%)
```

`c`

Änderungen:

⌘ Keine Änderungen.

`min` – das Minimum von Zahlen

`min(x1, x2, ...)` ergibt das Minimum der Zahlen x_1, x_2, \dots .

Aufruf(e):

⌘ `min(x1, x2, ...)`

Parameter:

`x1, x2, ...` — beliebige MuPAD-Objekte

Rückgabewert: eines der Argumente oder ein symbolischer `min`-Aufruf.

Überladbar durch: `x1, x2, ...`

Verwandte Funktionen: `_leequal, _less, min, sysorder`

Details:

- ⌘ Wenn die Argumente von `min` ganze, rationale oder Gleitkommazahlen sind, gibt `min` das numerische Minimum dieser Argumente zurück.
- ⌘ Der Aufruf `min()` ist nicht erlaubt und ergibt eine Fehlermeldung. Wird `min` mit nur einem Argument `x1` aufgerufen, wird `x1` evaluiert zurückgegeben (siehe Beispiel ??).
- ⌘ Wenn ein Argument `-infinity` ist, wird `-infinity` zurückgegeben. Ist ein Argument `infinity`, wird dieses Argument aus der Liste der Argumente entfernt (siehe Beispiel ??).
- ⌘ `min` gibt einen Fehler zurück, wenn eines der Argumente eine komplexe Zahl ist (siehe Beispiel ??).

- ☞ Wenn ein Argument keine Zahl ist, dann wird ein symbolischer `min`-Aufruf mit dem Minimum aller numerischen Argumente und allen verbleibenden evaluierten Argumenten zurückgegeben (siehe Beispiel ??).
Geschachtelte `min`-Aufrufe mit symbolischen Argumenten werden in einen einzigen `min`-Aufruf umgeschrieben (sie werden ausgeglichen).
Siehe Beispiel ??.
 - ☞ Eigenschaften von Bezeichnern werden von `min` *nicht* berücksichtigt. Mit der Funktion `simplify` können solche Ausdrücke weiter vereinfacht werden, die Bezeichner mit Eigenschaften enthalten (siehe Beispiel ??).
 - ☞ `min` ist eine Funktion des Systemkerns.
-

Beispiel 1. `min` berechnet das Minimum von ganzen, rationalen und Gleitkommazahlen:

```
>> min(-3/2, 7, 1.4)
-3/2
```

Wenn ein symbolischer Ausdruck unter den Argumenten ist, wird ein symbolischer `min`-Aufruf zurückgeliefert:

```
>> delete b: min(-4, b + 2, 1, 3)
min(b + 2, -4)

>> min(sqrt(2), 1)
1/2
min(2, 1)
```

Man kann `simplify` benutzen, um `min`-Ausdrücke mit konstanten symbolischen Argumenten zu vereinfachen:

```
>> simplify(%)
1
```

Beispiel 2. Ein `min`-Aufruf mit einem Argument gibt das Argument evaluiert zurück:

```
>> delete a: min(a), min(sin(2*PI)), min(2)
a, 0, 2
```

Komplexe Zahlen führen zu einer Fehlermeldung:

```
>> min(0, 1, I)
Error: Illegal argument [min]
```

Beispiel 3. `-infinity` ist immer das Minimum beliebiger Argumente:

```
>> delete x: min(-1000000000000, -infinity, x)
                        -infinity
```

`infinity` wird aus der Argumentliste entfernt:

```
>> min(-1000000000000, infinity, x)
                        min(x, -1000000000000)
```

Beispiel 4. `min` berücksichtigt keine durch `assume` definierten Eigenschaften von Bezeichnern:

```
>> delete a, b, c:
    assume(a > 0): assume(b > a, _and): assume(c > b, _and):
    min(a, min(b, c), 0)
                        min(a, b, c, 0)
```

Erst eine Anwendung von `simplify` liefert das gewünschte Ergebnis:

```
>> simplify(%)
                        0
```

Änderungen:

⌘ Keine Änderungen.

`mod`, `modp`, `mods` – **die Modulo-Funktionen**

`modp(x, m)` bestimmt den eindeutigen nichtnegativen Rest bei Division der ganzen Zahl x durch die ganze Zahl m .

`mods(x, m)` bestimmt die betragsmäßig kleinste ganze Zahl r , für die die ganze Zahl $x - r$ durch die ganze Zahl m teilbar ist.

$x \bmod m$ und `_mod(x, m)` sind standardmäßig äquivalent zu `modp(x, m)`.

Aufruf(e):

⌘ $x \bmod m$
⌘ `_mod(x, m)`
⌘ `modp(x, m)`
⌘ `mods(x, m)`

Parameter:

x, m — arithmetische Ausdrücke

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x, m

Seiteneffekte: Der Operator `mod` und die Funktion `_mod` sind standardmäßig äquivalent zu `modp`. Dies kann geändert werden, indem man dem Bezeichner `_mod` einen anderen Wert zuweist. Siehe hierzu auch Beispiel ??.

Verwandte Funktionen: `/`, `div`, `divide`, `Dom::IntegerMod`, `frac`, `gcd`, `gcdex`, `igcd`, `igcdex`, `IntMod`, `powermod`

Details:

- ☞ Ist m eine von 0 verschiedene ganze Zahl und x eine ganze Zahl, so berechnen `modp` und `mods` eine ganze Zahl r , die die Gleichung $x = qm + r$ für ein $q \in \mathbb{Z}$ erfüllt. Im Fall von `modp` gilt $0 \leq r < |m|$, und im Fall von `mods` gilt $-|m|/2 < r \leq |m|/2$. Siehe Beispiel ??. In beiden Fällen ist r eindeutig bestimmt. Im Fall von `modp` gilt weiterhin $q = x \text{ div } m$.
- ☞ Ist m eine von 0 verschiedene ganze Zahl und x eine rationale Zahl, etwa $x = u/v$ mit teilerfremden ganzen Zahlen u und v , so berechnen `modp` und `mods` eine ganzzahlige Lösung der Kongruenz $vr \equiv u \pmod{m}$. Dies geschieht, indem zuerst das Inverse w von v modulo m berechnet wird, so dass m ein Teiler von $vw - 1$ ist. So ein w existiert genau dann, wenn v und m teilerfremd sind, der größte gemeinsame Teiler der beiden Zahlen also 1 ist. Dann wird `modp(u*w, m)` bzw. `mods(u*w, m)` als Lösung r der oben erwähnten Kongruenz zurückgegeben. Sind v und m nicht teilerfremd, so wird eine Fehlermeldung ausgegeben. Siehe Beispiel ??.
Die Zahl $x - \text{modp}(x, m)$ ist in diesem Fall kein ganzzahliges Vielfaches von m .
- ☞ Ist m eine rationale Zahl ungleich 0 und x eine ganze oder rationale Zahl, dann liefern `modp` und `mods` als Ergebnis eine ganze oder rationale Zahl r , für die gilt: $x = qm + r$ für ein $q \in \mathbb{Z}$. Im Fall von `modp` gilt weiterhin $0 \leq r < |m|$ und im Fall von `mods` gilt $-|m|/2 < r \leq |m|/2$. Siehe Beispiel ??
- ☞ Falls das zweite Argument m den Wert 0 hat, dann wird eine Fehlermeldung zurückgegeben.
- ☞ `_mod(x, m)` ist das funktionale Äquivalent der Operatorschreibweise $x \bmod m$. Siehe Beispiel ??
- ☞ Standardmäßig sind `_mod` und `modp` äquivalent.

- ⌘ Die Funktionen `modp` und `mods` können dazu benutzt werden, den Modulo-Operator umzudefinieren. Nach der Zuweisung `_mod:=mods` berechnen der Operator `mod` und die äquivalente Funktion `_mod` beispielsweise den betragsmäßig kleinsten Rest. Siehe Beispiel ??.
- ⌘ Die Funktionen geben einen Fehler zurück, wenn eines der Argumente eine Gleitpunktzahl, eine komplexe Zahl oder kein arithmetischer Ausdruck ist.
- ⌘ Sofern eines der Argumente keine Zahl ist, wird ein symbolischer Funktionsaufruf zurückgegeben. Siehe Beispiel ??.
- ⌘ `_mod`, `modp` und `mods` sind Funktionen des Systemkerns.

Beispiel 1. Das Beispiel demonstriert den Bezug zwischen der Funktion `_mod` und dem Operator `mod`:

```
>> hold(_mod(23,5))

23 mod 5

>> 23 mod 5 = _mod(23,5)

3 = 3
```

Beispiel 2. Es folgen einige Beispiele, wo das zweite Argument eine ganze Zahl ist. Man erkennt hierbei auch, dass als Voreinstellung die Funktion `mod` der Funktion `modp` entspricht:

```
>> 27 mod 3, 27 mod 4, modp(27, 4), mods(27, 4)

0, 3, 3, -1

>> 27 = (27 div 4)*4 + modp(27, 4)

27 = 27
```

Nun wird $22/3$ modulo 5 berechnet. Der größte gemeinsame Teiler von 3 und 5 ist 1, und 2 ist das Inverse von 3 modulo 5. Damit folgt $22/3$ modulo 5 = $22 \cdot 2$ modulo 5:

```
>> modp(22/3, 5) = modp(22*2, 5),
    mods(22/3, 5) = mods(22*2, 5)

4 = 4, -1 = -1
```

Der größte gemeinsame Teiler von -15 und 27 ist 3. Daher besitzt 15 kein Inverses modulo 27, und man erhält eine Fehlermeldung:

```
>> modp(-22/15, 27)
```

```
Error: Modular inverse does not exist
```

Die Zahlen 15 und 26 sind teilerfremd. Daher kann $-22/15$ modulo 26 berechnet werden:

```
>> -22/15 mod 26
```

2

Beispiel 3. Es folgen einige Beispiele, bei denen das zweite Argument ein Bruch ist. Es gilt $23/3 = 9 \cdot 4/5 + 7/15 = 10 \cdot 4/5 - 1/3$ und $23 = 28 \cdot 4/5 + 3/5 = 29 \cdot 4/5 - 1/5$. Wir erhalten also:

```
>> modp(23/3, 4/5), mods(23/3, 4/5),
    modp(23, 4/5), mods(23, 4/5)
7/15, -1/3, 3/5, -1/5
```

Beispiel 4. Sofern eines der Argumente keine Zahl ist, wird ein symbolischer Funktionsaufruf zurückgegeben:

```
>> delete x, m:
    x mod m, x mod 2, 2 mod m
x mod m, x mod 2, 2 mod m
```

Werden `modp` und `mods` nicht mit numerischen Argumenten aufgerufen, dann wird zur Ausgabe des Ergebnisses die Operatorschreibweise benutzt:

```
>> modp(x, m), mods(x, m)
x mod m, x mod m
```

Beispiel 5. Der binäre Operator `mod` und die Funktion `_mod` sind standardmäßig äquivalent zu `modp`. Dies lässt sich durch Undefinieren von `_mod` ändern:

```
>> 11 mod 7, modp(11,7), mods(11,7)
4, 4, -3
```

```
>> _mod := mods: 11 mod 7;
    _mod := modp:
```

-3

Änderungen:

⌘ Keine Änderungen.

`multcoeffs` – Multiplikation der Koeffizienten eines Polynoms mit einem Faktor

`multcoeffs(p, c)` multipliziert alle Koeffizienten des Polynoms `p` mit dem Faktor `c`.

Aufruf(e):

⌘ `multcoeffs(p, c)`
⌘ `multcoeffs(f, <vars,> c)`

Parameter:

`p` — ein Polynom vom Typ `DOM_POLY`
`c` — ein arithmetischer Ausdruck oder ein Element des Koeffizientenrings von `p`
`f` — ein polynomialer Ausdruck
`vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck oder `FAIL`.

Überladbar durch: `p`, `f`

Verwandte Funktionen: `coeff`, `degree`, `degreevec`, `lcoeff`, `ldegree`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `tcoeff`

Details:

- ⌘ Ein polynomialer Ausdruck `f` wird zunächst mittels `poly` in ein Polynom in den Unbestimmten `vars` konvertiert. Werden keine Unbestimmten angegeben, so werden diese in `f` gesucht. Siehe `poly` zu Details bezüglich der Konvertierung. Falls `f` nicht in ein Polynom konvertiert werden kann, so liefert `multcoeffs` den Wert `FAIL`. Nach Multiplikation mit `c` wird das Ergebnis wieder in einen Ausdruck konvertiert.
 - ⌘ Für einen polynomialen Ausdruck `f` kann `c` ein beliebiger arithmetischer Ausdruck sein. Für ein Polynom `p` vom Typ `DOM_POLY` muss der Faktor `c` in ein Element des Koeffizientenrings von `p` konvertierbar sein.
 - ⌘ `multcoeffs` ist eine Funktion des Systemkerns.
-

Beispiel 1. Einige einfache Beispiele:

```
>> multcoeffs(3*x^3 + x^2*y^2 + 2, 5)
          3      2  2
        15 x  + 5 x  y  + 10

>> multcoeffs(3*x^3 + x^2*y^2 + 2, c)
          3      2  2
        2 c + 3 c x  + c x  y

>> multcoeffs(poly(x^3 + 2, [x]), sin(y))
          3
      poly(sin(y) x  + 2 sin(y), [x])
```

Beispiel 2. Mathematisch stimmt `multcoeffs(f, c)` mit $f \cdot c$ überein. Allerdings liefert `multcoeffs` eine expandierte Form des Produkts, die von den Unbestimmten abhängt:

```
>> f := 3*x^3 + x^2*y^2 + 2:
      multcoeffs(f, [x], c), multcoeffs(f, [y], c),
      multcoeffs(f, [z], c)

          3      2  2      2  2      3
        2 c + 3 c x  + c x  y , c x  y  + c (3 x  + 2),

          3      2  2
        c (3 x  + x  y  + 2)

>> delete f:
```

Änderungen:

⌘ Keine Änderungen.

new – Erzeugung eines Domain-Elementes

`new(T, object1, object2, ...)` erstellt ein neues Element des Domains `T` mit der internen Darstellung `object1, object2,`

Aufruf(e):

⌘ `new(T, object1, object2, ...)`

Parameter:

T — ein MuPAD-Domain
 $\text{object1}, \text{object2}, \dots$ — beliebige MuPAD-Objekte

Rückgabewert: ein Element des Domains T .

Verwandte Funktionen: `DOM_DOMAIN`, `domain`, `extop`, `extnops`, `extsubsop`, `newDomain`, `op`

Details:

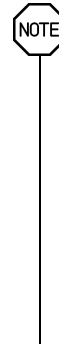
☞ `new` ist eine Basisfunktion, um Elemente von Bibliotheks-Domains zu erzeugen.

Die interne Darstellung eines Domain-Elements besteht aus einem Verweis auf das entsprechende Domain und einer beliebigen Anzahl von MuPAD-Objekten, den internen Operanden des Domain-Elements.

☞ `new(T , object1 , object2 , \dots)` erzeugt ein neues Element des Domains T , dessen interne Darstellung die Folge der Operanden object1 , object2 , \dots ist, und liefert dieses Element zurück.

`new(T)` erstellt ein neues Element des Domains T , dessen interne Darstellung eine leere Folge von Operanden ist.

☞ `new` ist nur für die Programmierung neuer Domains in MuPAD gedacht. Domainelemente sollten nicht direkt mit `new` erzeugt werden. Für die Erzeugung von Elementen eines MuPAD-Domains T sollte der Konstruktor-Aufruf `$T(\dots)$` verwendet werden. Die interne Repräsentation eines MuPAD-Domains kann sich ändern, ohne dass sich der Konstruktor `$T(\dots)$` ändert. Außerdem werden, im Gegensatz zu `new`, bei einem Aufruf der Form `$T(\dots)$` die Argumente überprüft. Die direkte Anwendung von `new` kann zu einer ungültigen internen Darstellung eines MuPAD-Objekts führen.



☞ Neue Domains werden mit der Funktion `newDomain` erzeugt.

☞ Auf die interne Repräsentation der Operanden eines Domainelementes kann mit der Funktion `extop` zugegriffen werden. Im Gegensatz dazu wird die Funktion `op` manchmal von einem Domain überladen, um die interne technische Darstellung der Operanden zu verbergen und eine benutzerfreundliche intuitive Schnittstelle zu bieten.

☞ Entsprechend sind die Funktionen `extnops` und `extsubsop` zu verwenden, um die Anzahl der internen Operanden eines Domainelementes zu erfragen bzw. bestimmte Argumente zu ersetzen. Diese beiden Funktionen können, im Gegensatz zu `nops` und `subsop`, nicht überladen werden.

- ☞ Man kann einen Konstruktor für ein benutzerdefiniertes Domain `T` zur Verfügung stellen, indem man eine `"new"`-Methode implementiert. Diese wird immer dann verwendet, wenn ein Aufruf der Form `T(arg1, arg2, ...)` eingegeben wird. Ein Konstruktor ist empfehlenswert, da er eine elegantere und intuitivere Schnittstelle bietet als `new`. Üblicherweise prüft die `"new"`-Methode ihre Argumente `arg1, arg2, ...` auf Zulässigkeit und konvertiert sie in die interne Darstellung des Domains, unter Benutzung von `new`. Siehe Beispiel ??.
- ☞ `new` ist eine Funktion des Systemkerns.

Beispiel 1. Im folgenden Beispiel wird ein neues Domain `Time` zur Repräsentation einer Uhrzeit erzeugt. Die interne Repräsentation eines Domainelements von `Time` hat zwei Operanden: die Stunden und die Minuten. Dann wird ein neues Domainelement für die Zeit 12 : 45 erzeugt:

```
>> Time := newDomain("Time"):
    a := new(Time, 12, 45)

                                new(Time, 12, 45)
```

Der Domaintyp von `a` ist `Time`, die Anzahl der Operanden ist 2, und die Operanden sind 12 und 45:

```
>> domtype(a), extnops(a)

                                Time, 2

>> extop(a)

                                12, 45
```

Nun wird die Methode `"new"` für das Domain `Time` definiert, die verschiedene Eingabeformate zulässt. Zwei ganze Zahlen werden als Stunden und Minuten interpretiert, eine ganze Zahl als Minuten, von einer Gleitkommazahl als Eingabe wird der Teil vor dem Komma als Stunden, der Teil nach dem Komma als Minuten interpretiert, und kein Argument wird als Mitternacht interpretiert. Zusätzlich wird überprüft, ob der Typ und die Anzahl der Argumente korrekt sind:

```
>> Time::new := proc(HR = 0, MN = 0)
    local m;
    begin
        if args(0) = 2 and domtype(HR) = DOM_INT
            and domtype(MN) = DOM_INT then
            m := HR*60 + MN
        elif args(0) = 1 and domtype(HR) = DOM_INT then
            m := HR
```

```

    elif args(0) = 1 and domtype(HR) = DOM_RAT then
        m := trunc(float(HR))*60 + frac(float(HR))*60
    elif args(0) = 1 and domtype(HR) = DOM_FLOAT then
        m := trunc(HR)*60 + frac(HR)*60
    elif args(0) = 0 then
        m := 0
    else
        error("wrong number or type of arguments")
    end_if;
    new(Time, trunc(m/60), trunc(m) mod 60)
end_proc:

```

Nun können mithilfe dieser Methode neue Objekte des Domains Time erzeugt werden, entweder direkt mit new oder durch den kürzeren Aufruf Time(...):

```

>> Time::new(12, 45), Time(12, 45), Time(12 + 3/4)

    new(Time, 12, 45), new(Time, 12, 45), new(Time, 12, 45)
>> Time(), Time(8.25), Time(1/2)

    new(Time, 0, 0), new(Time, 8, 15), new(Time, 0, 30)

```

Für eine schönere Ausgabe kann ausserdem die Methode "print" definiert werden (siehe print):

```

>> Time::print := proc(TM)
    begin
        expr2text(extop(TM, 1)) . ":" .
        stringlib::format(expr2text(extop(TM, 2)), 2, Right, "0")
    end_proc:

>> Time::new(12, 45), Time(12, 45), Time(12 + 3/4)

    12:45, 12:45, 12:45

>> Time(), Time(8.25), Time(1/2)

    0:00, 8:15, 0:30

```

Änderungen:

☞ Keine Änderungen.

newDomain – Erzeugung eines neuen Datentyps (Domain)

`newDomain(k)` erzeugt ein neues Domain mit Schlüssel `k`.

`newDomain(k, T)` liefert eine Kopie des Domains `T` mit neuem Schlüssel `k`.

`newDomain(k, t)` erzeugt ein neues Domain mit Schlüssel `k` und Slots aus der Tabelle `t`.

Aufruf(e):

```
newDomain(k)
newDomain(k, T)
newDomain(k, t)
```

Parameter:

`k` — ein beliebiges Objekt; typischerweise eine Zeichenkette
`T` — ein Domain
`t` — die Slots des Domains: eine Tabelle

Rückgabewert: ein Objekt vom Typ `DOM_DOMAIN`.

Weitere Dokumentation: Das Dokument „Axioms, Categories and Domains“ ist eine detaillierte technische Referenz über Domains.

Verwandte Funktionen: `DOM_DOMAIN`, `domain`, `domtype`, `new`, `slot`

Details:

☞ In MuPAD werden Datentypen als *Domains* bezeichnet. `newDomain` ist eine Basisfunktion zur Definition neuer Datentypen. Für Links zur Dokumentation über Domains und komfortablere Möglichkeiten, ein Domain zu definieren, konsultiere man den entsprechenden Eintrag des Glossars. Die Hilfeseite zu `DOM_DOMAIN` enthält ein tutorielles Beispiel zur Definition eines Domains mittels `newDomain`.

☞ Technisch gesehen ist ein Domain nichts anderes als eine Tabelle. Die Einträge dieser Tabelle werden *Slots* oder *Methoden* genannt. Sie dienen dazu, die Funktionalität von MuPAD-Standardfunktionen wie den arithmetischen Operationen `+` und `*`, den speziellen mathematischen Funktionen `exp` und `sin` oder den Funktionen `simplify` und `normal` zur symbolischen Manipulation auf die Objekte eines Domains auszudehnen. Dies geschieht in einer modularen, objektorientierten Weise, ohne dass der Quellcode der Standardfunktion verändert werden muss, und wird als *Überladung* bezeichnet.

Die Funktion `slot` und der äquivalente Operator `::` ermöglichen die Definition eines bestimmten Slots eines Domains und den Zugriff darauf. Die Funktion `op` liefert alle Slots eines Domains zurück.

- ⌘ Jedes Domain hat einen ausgezeichneten Slot "key" (Schlüssel), der zur eindeutigen Identifikation dient. Es kann keine zwei verschiedenen Domains mit demselben Schlüssel geben. Typischerweise, aber nicht notwendigerweise, ist der Schlüssel eine Zeichenkette. Der Schlüssel dient jedoch hauptsächlich internen Zwecken und zur Ausgabe. Üblicherweise wird ein Domain unmittelbar nach seiner Erzeugung einem Bezeichner zugewiesen, über den man auf den Domain zugreift.
- ⌘ Falls ein Domain mit dem angegebenen Schlüssel bereits existiert, so liefert `newDomain(k)` dieses Domain zurück; die beiden anderen Formen des Aufrufs von `newDomain` sind in diesem Fall unzulässig und liefern einen Fehler.
- ⌘ `newDomain` ist eine Funktion des Systemkerns.

Beispiel 1. Der folgende Aufruf erzeugt ein Domain mit Schlüssel "my-domain". Dieser Schlüssel wird auch zur Ausgabe benutzt, allerdings ohne Anführungszeichen:

```
>> T := newDomain("my-domain")

my-domain
```

Mit der Funktion `new` werden Elemente dieses Domains erzeugt:

```
>> e := new(T, 42);
domtype(e)

new(my-domain, 42)

my-domain
```

Mit dem Slot-Operator `::` kann man einen neuen Slot definieren oder auf einen bestehenden zugreifen:

```
>> op(T)

"key" = "my-domain"

>> T::key, T::myslot

"my-domain", FAIL

>> T::myslot := 42: op(T)

"myslot" = 42, "key" = "my-domain"

>> T::myslot^2

1764
```

Falls ein Domain mit dem Schlüssel k bereits existiert, dann erzeugt `newDomain(k)` kein neues Domain, sondern liefert statt dessen das existierende Domain zurück:

```
>> T1 := newDomain("my-domain"):
      op(T1)

      "myslot" = 42, "key" = "my-domain"
```

Man beachte, dass man ein Domain nicht löschen kann. Der Befehl `delete T` löscht nur den Wert des Bezeichners T , aber er beseitigt das Domain mit dem Schlüssel "my-domain" nicht:

```
>> delete T, T1:
      T2 := newDomain("my-domain"):
      op(T2);
      delete T2:

      "myslot" = 42, "key" = "my-domain"
```

Beispiel 2. Es können zu einem gegebenen Zeitpunkt nicht zwei verschiedene Domains mit demselben Schlüssel existieren. Wird für ein Domain ein neuer Eintrag definiert, so ändern sich dadurch implizit die Werte aller Bezeichner, deren Wert dieses Domain ist:

```
>> T := newDomain("1st"): T1 := T:
      op(T);
      op(T1);

      "key" = "1st"

      "key" = "1st"

>> T1::mySlot := 42:
      op(T);
      op(T1);

      "mySlot" = 42, "key" = "1st"

      "mySlot" = 42, "key" = "1st"
```

Um dies zu umgehen, kann ein Domain kopiert werden; für die Kopie muss ein neuer, unbenutzter Schlüssel vergeben werden:

```
>> T2 := newDomain("2nd", T):
      T2::anotherSlot := infinity:
      op(T);
      op(T2);
```

```

"mySlot" = 42, "key" = "1st"

"anotherSlot" = infinity, "mySlot" = 42, "key" = "2nd"

>> delete T, T1, T2:

```

Beispiel 3. Man kann einem Domain bereits bei der Erzeugung Slots mitgeben:

```

>> T := newDomain("3rd",
  table("myslot" = 42, "anotherSlot" = infinity)):
op(T);
T::myslot, T::anotherSlot

"key" = "3rd", "anotherSlot" = infinity, "myslot" = 42

42, infinity

>> delete T:

```

Änderungen:

⌘ newDomain hieß früher domain.

next – Überspringen eines Schleifenschritts

next unterbricht den aktuellen Iterationsschritt in for-, repeat- und while-Schleifen. Die Ausführung geht mit dem Beginn des nächsten Schleifenschritts weiter.

Aufruf(e):

⌘ next
 ⌘ _next()

Verwandte Funktionen: break, case, for, quit, repeat, return, while

Details:

⌘ Die next-Anweisung ist äquivalent zum Funktionsaufruf _next(). Der Rückgabewert ist das leere Objekt vom Typ DOM_NULL.

- ⌘ Innerhalb von `for`-, `repeat`- und `while`-Schleifen führt die `next`-Anweisung zum sofortigen Abbruch des momentanen Schleifenschritts. In `for`-Schleifen wird die Schleifenvariable herauf- bzw. heruntergesetzt und die Ausführung wird am Beginn der Schleife fortgesetzt. Analog werden die Kontrollbedingungen am Anfang einer `while`-Schleife bzw. im `until`-Teil einer `repeat`-Schleife überprüft, bevor die Ausführung am Beginn der Schleife fortgesetzt wird.
- ⌘ Außerhalb von `for`-, `repeat`- und `while`-Schleifen hat `next` keinerlei Effekt.
- ⌘ `_next` ist eine Funktion des Systemkerns.

Beispiel 1. In der folgenden `for`-Schleife wird jeder Schritt übersprungen, in dem `i` gerade ist:

```
>> for i from 1 to 5 do
      if testtype(i, Type::Even) then next end_if;
      print(i)
    end_for:
```

1

3

5

In der folgenden `repeat`-Schleife werden alle Schritte mit ungeradem `i` ausgelassen:

```
>> i := 0:
    repeat
      i := i + 1;
      if testtype(i, Type::Odd) then next end_if;
      print(i)
    until i >= 5 end_repeat:
```

2

4

```
>> delete i:
```

Änderungen:

- ⌘ Keine Änderungen.

nextprime – die nächste Primzahl

`nextprime(m)` liefert die kleinste Primzahl, die größer oder gleich `m` ist.

Aufruf(e):

☞ `nextprime(m)`

Parameter:

`m` — ein arithmetischer Ausdruck

Rückgabewert: eine Primzahl oder ein symbolischer Aufruf von `nextprime`.

Verwandte Funktionen: `ifactor`, `igcd`, `ilcm`, `isprime`, `ithprime`, `numlib::prevprime`

Details:

☞ Ist das Argument `m` eine ganze Zahl, so wird die kleinste Primzahl, die größer oder gleich `m` ist zurückgeliefert. Ist `m` nicht vom Typ `Type::Numeric`, so wird ein symbolischer Funktionsaufruf vom Typ "nextprime" zurückgeliefert. Ist das Argument eine Zahl, aber keine ganze Zahl, so wird ein Fehler ausgelöst.

☞ Die erste Primzahl ist 2.

☞ `nextprime` ist eine Funktion des Systemkerns.

Beispiel 1. Die erste Primzahl wird berechnet:

```
>> nextprime(-13)
```

2

Ist das Argument von `nextprime` eine Primzahl, so wird diese Zahl als Ergebnis geliefert:

```
>> nextprime(11)
```

11

Eine größere Primzahl wird berechnet:

```
>> nextprime(56475767478567)
```

56475767478601

Symbolische Argumente führen zu einem symbolischen Aufruf:

```
>> nextprime(x)
```

```
nextprime(x)
```

Hintergründe:

- ⇒ `nextprime` benutzt einen schnellen probabilistischen Primzahltest (Miller-Rabin Test) um zu entscheiden, ob das berechnete Ergebnis eine Primzahl ist. Das von `nextprime` gelieferte Ergebnis ist entweder eine Primzahl oder eine starke Pseudo-Primzahl für 10 zufällig gewählte Basen.
- ⇒ Verweis: Michael O. Rabin, Probabilistic algorithms, in J. F. Traub, ed., *Algorithms and Complexity*, Academic Press, New York, 1976, S. 21-39.

Änderungen:

- ⇒ Keine Änderungen.
-

`nops` – die Anzahl der Operanden

`nops(object)` gibt die Anzahl der Operanden des Objekts `object` zurück.

Aufruf(e):

- ⇒ `nops(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: eine nichtnegative ganze Zahl.

Überladbar durch: `object`

Verwandte Funktionen: `extnops`, `extop`, `extsubsop`, `length`, `op`, `subsop`

Details:

- ☞ Auf der Hilfeseite für `op` sind weitere Informationen zu MuPADs Konzept von „Operanden“ zu finden.
- ☞ Für Mengen, Listen und Tabellen liefert `nops` die Anzahl der Elemente bzw. Einträge. Man beachte, dass Ausdrücke vom Typ `DOM_EXPR` sowie Arrays einen 0-ten Operanden besitzen, der von `nops` *nicht mitgezählt* wird. Nicht-initialisierte Array-Elemente werden von `nops` gezählt.
- ☞ Das leere Objekt `null()`, die leere Liste `[]`, die leere Menge `{}` und die leere Tabelle `table()` haben keine Operanden: `nops` liefert 0 zurück. Cf. example ??.
- ☞ Ganze Zahlen vom Typ `DOM_INT`, Gleitpunktzahlen vom Typ `DOM_FLOAT`, boolesche Konstanten vom Typ `DOM_BOOL`, Bezeichner vom Typ `DOM_IDENT` und Zeichenketten vom Typ `DOM_STRING` sind „atomare“ Objekte, die nur einen Operanden haben: das Objekt selbst. Rationale Zahlen vom Typ `DOM_RAT` und komplexe Zahlen vom Typ `DOM_COMPLEX` haben 2 Operanden: Zähler und Nenner bzw. Real- und Imaginärteil. Siehe Beispiel ??.
- ☞ An `nops` übergebene Ausdrucksfolgen werden nicht ausgeglichen. Siehe Beispiel ??.
- ☞ `nops` ist eine Funktion des Systemkerns.

Beispiel 1. Der folgende Ausdruck hat den Typ `"_plus"` und die drei Operanden `a*b`, `3*c` und `d`:

```
>> nops(a*b + 3*c + d)
```

3

Für Mengen und Listen liefert `nops` die Anzahl der Elemente. In den folgenden Beispielen werden die Liste `[1, 2, 3]` und die Menge `{1, 2}` als jeweils ein Operand gezählt:

```
>> nops({a, 1, [1, 2, 3], {1, 2}})
```

4

```
>> nops([[1, 2, 3], 4, 5, {1, 2}])
```

4

Leere Objekte haben keine Operanden:

```
>> nops(null()), nops([]), nops({}), nops(table())
```

0, 0, 0, 0

Die Anzahl der Operanden von symbolischen Funktionsaufrufen ist die Anzahl der Argumente:

```
>> nops(f(3*x, 4, y + 2)), nops(f())  
3, 0
```

Beispiel 2. Ganze Zahlen und reelle Gleitpunktzahlen haben nur einen Operanden:

```
>> nops(12), nops(1.41)  
1, 1
```

Dasselbe gilt für Zeichenketten. Die Anzahl der Zeichen kann mit der Funktion `length` erfragt werden:

```
>> nops("MuPAD"), length("MuPAD")  
1, 5
```

Rationale und komplexe Zahlen haben zwei Operanden, auch wenn der Realteil verschwindet:

```
>> nops(-3/2), nops(1 + I), nops(2*I)  
2, 2, 2
```

Eine Funktionsumgebung hat 3, eine Prozedur hat 13 Operanden:

```
>> nops(sin), nops(op(sin, 1))  
3, 13
```

Beispiel 3. Ausdrucksfolgen werden von `nops` nicht ausgeglichen:

```
>> nops((1, 2, 3))  
3
```

Im Gegensatz zum obigen Aufruf wird im Folgenden `nops` mit drei Argumenten aufgerufen:

```
>> nops(1, 2, 3)  
Error: Wrong number of arguments [nops]
```

Änderungen:

☞ Keine Änderungen.

norm – die Norm einer Matrix, eines Vektors oder eines Polynoms

`norm(M, kM)` berechnet die Norm mit Index `kM` der Matrix `M`.

`norm(v, kv)` berechnet die Norm mit Index `kv` des Vektors `v`.

`norm(p, kp)` berechnet die Norm mit Index `kp` des Polynoms `p`.

Aufruf(e):

☞ `norm(M <, kM>)`

☞ `norm(v <, kv>)`

☞ `norm(p <, kp>)`

☞ `norm(f <, vars> <, kp>)`

Parameter:

- `M` — eine Matrix des Domaintyps `Dom::Matrix(...)`
- `kM` — der Index der Matrixnorm: entweder 1 oder *Frobenius* oder *Infinity*. Die Voreinstellung ist *Infinity*.
- `v` — ein Vektor (eine 1-dimensionale Matrix)
- `kv` — der Index der Vektornorm: entweder eine positive ganze Zahl oder *Frobenius* oder *Infinity*. Die Voreinstellung ist *Infinity*.
- `p` — ein mittels `poly` erzeugtes Polynom
- `f` — ein polynomialer Ausdruck
- `vars` — eine Liste von Bezeichnern oder indizierten Bezeichnern, die als Unbestimmte von `f` aufgefasst werden
- `kp` — der Index der Polynomnorm: eine reelle Zahl ≥ 1 . Wird kein Index angegeben, so wird die Maximumsnorm (mit dem Index ,unendlich') berechnet.

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `p`, `f`

Verwandte Funktionen: `coeff`, `float`, `matrix`, `poly`

Details:

☞ In MuPAD gibt es keinen Unterschied zwischen Matrizen und Vektoren: ein Vektor ist eine Matrix der Dimension $1 \times n$ bzw. $n \times 1$.

- ☞ Für eine $m \times n$ Matrix $M = (M_{ij})$ mit $\min(m, n) > 1$ kann nur die 1-Norm (Spaltensummennorm)

$$\text{norm}(M, 1) = \max_{j=1, \dots, n} \sum_{i=1}^m |M_{ij}|,$$

die Frobenius-Norm

$$\text{norm}(M, \text{Frobenius}) = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |M_{ij}|^2}$$

und die ∞ -norm (Zeilensummennorm)

$$\text{norm}(M) = \text{norm}(M, \text{Infinity}) = \max_{i=1, \dots, m} \sum_{j=1}^n |M_{ij}|$$

berechnet werden. Die 1-Norm und die *Infinity*-Norm sind Operatornormen bezüglich der entsprechenden Norm des Vektorraumes auf dem die Matrix operiert.

Für numerische Matrizen entspricht die Spektralnrm (die Operatornorm bezüglich der euklidischen Norm (Index 2)) dem größten Singulärwert. Dieser kann mit `numeric::singularvalues` berechnet werden.

- ☞ Für Vektoren $v = (v_i)$, die durch $1 \times n$ oder $n \times 1$ Matrizen repräsentiert werden, kann die k -Norm für jede positive ganze Zahl sowie die *Infinity*-Norm berechnet werden. Für ganze Zahlen $k > 1$ ist die Vektornorm sowohl für Spalten- als auch für Zeilenvektoren durch

$$\text{norm}(v, k) = \left(\sum_{i=1}^n |v_i|^k \right)^{1/k}$$

gegeben.

Für die Indizes 1, *Infinity* und *Frobenius* ist die Vektornorm durch die entsprechende Matrixnorm gegeben. Für Spaltenvektoren ist die 1-Norm die Summennorm

$$\text{norm}(v, 1) = \sum_{i=1}^n |v_i|,$$

und die *Infinity*-Norm ist die Maximumsnorm

$$\text{norm}(v) = \text{norm}(v, \text{Infinity}) = \max(|v_1|, \dots, |v_n|)$$

(dies ist der Grenzwert der k -Normen, wenn k gegen unendlich strebt).

Für Zeilenvektoren ist die 1-Norm die Maximumsnorm, während die *Infinity*-Norm die Summennorm ist.



Die Frobenius-Norm entspricht für Spalten- und Zeilenvektoren der 2-Norm. Siehe Beispiel ??.

- ☞ Matrizen und Vektoren können symbolische Einträge enthalten; es wird keine interne Gleitpunktkonvertierung durchgeführt.
- ☞ Man beachte auch die Hilfeseite des Matrizen-Domains `Dom::Matrix` (man beachte, dass die Funktion `matrix` Matrizen vom Typ `Dom::Matrix()` erzeugt).
- ☞ Für Polynome `p` mit den Koeffizienten c_i ist die Norm durch

$$\text{norm}(p) = \max |c_i|, \quad \text{norm}(p, k) = \left(\sum_i |c_i|^k \right)^{1/k}$$

gegeben. `norm` kann auch auf multivariate Polynome angewandt werden. Hierbei werden die Koeffizienten bezüglich aller Unbestimmten berücksichtigt.

- ☞ Für Polynome können nur numerische Normen berechnet werden. Die Koeffizienten dürfen keine symbolischen Parameter enthalten, die nicht in Gleitpunktzahlen konvertiert werden können. Koeffizienten werden intern in Gleitpunktzahlen konvertiert, wenn sie symbolische numerische Ausdrücke wie z. B. `PI+1`, `sqrt(2)` etc. enthalten. Siehe Beispiel ??.
- ☞ Für Indizes $k > 1$ liefert `norm` immer eine Gleitpunktzahl. Die 1-Norm liefert eine exakte Zahl, wenn alle Koeffizienten ganze oder rationale Zahlen sind. Die ∞ -Norm `norm(p)` liefert eine exakte Zahl, wenn der betragsgrößte Koeffizient eine ganze oder rationale Zahl ist. In allen anderen Fällen liefern auch die 1- und die ∞ -Norm Gleitpunktzahlen. Siehe Beispiel ??.
- ☞ Für Polynome über dem Koeffizientenring `IntMod(m)` liefert `norm` einen Fehler.
- ☞ Ist der Koeffizientenring des Polynoms ein Domain, muss dieses die Methode "`norm`" enthalten. Diese Methode muss die Norm der Koeffizienten als Zahlen oder numerische Ausdrücke zurückgeben, die mittels `float` zu einer Gleitpunktzahl konvertiert werden können. Mit der Koeffizientennorm $\|c_i\|$ berechnet `norm(p)` die Maximumsnorm $\max_i \|c_i\|$; `norm(p, n)` berechnet $(\sum_i \|c_i\|^k)^{1/k}$.
- ☞ Ein polynomialer Ausdruck `f` wird von `norm` intern mittels `poly(f)` in ein Polynom konvertiert. Ist eine Liste von Unbestimmten angegeben, so wird die Norm des Polynoms `poly(f, vars)` berechnet.
- ☞ Für Polynome und polynomialen Ausdrücke werden Normen durch eine Funktion des Systemkerns berechnet.

Beispiel 1. Für eine 2×3 Matrix werden einige Normen berechnet:

```
>> M := matrix([[2, 5, 8], [-2, 3, 5]]):
      norm(M) = norm(M, Infinity), norm(M, 1), norm(M, Frobenius)
```

$$\|15\|_2 = \sqrt{15^2 + 13^2 + 13^2}$$

Für Matrizen liefert norm exakte symbolische Ergebnisse:

```
>> M := matrix([[2/3, 63, PI],[x, y, z]]): norm(M)
max(PI + 191/3, abs(x) + abs(y) + abs(z))
>> norm(M, 1)
max(abs(x) + 2/3, abs(y) + 63, PI + abs(z))
>> norm(M, Frobenius)
sqrt(PI^2 + abs(x)^2 + abs(y)^2 + abs(z)^2 + 35725/9)
>> delete M:
```

Beispiel 2. Ein Spaltenvektor col und ein Zeilenvektor row werden betrachtet:

```
>> col := matrix([x1, PI]): row := matrix([[x1, PI]]): col, row
```

$$\begin{array}{cc|cc} + & - & - & + \\ | & x1 & | & + & - \\ | & & | & x1 & , & PI \\ | & PI & | & + & - \\ + & - & - & + \end{array}$$

```
>> norm(col, 2) = norm(row, 2)
sqrt(x1 conjugate(x1) + PI^2) = sqrt(x1 conjugate(x1) + PI^2)
>> norm(col, 3) = norm(row, 3)
(PI^3 + abs(x1)^3)^(1/3) = (PI^3 + abs(x1)^3)^(1/3)
```

Man beachte, dass die Normen vom Index 1 und *Infinity* vertauschte Bedeutungen bei Spalten- und Zeilenvektoren haben:

```
>> norm(col, 1) = norm(row, Infinity)
PI + abs(x1) = PI + abs(x1)
>> norm(col, Infinity) = norm(row, 1)
max(abs(x1), PI) = max(abs(x1), PI)
>> delete col, row:
```


Beispiel 3. Es werden Normen von Polynomen berechnet:

```
>> p := poly(3*x^3 + 4*x, [x]): norm(p), norm(p, 1)
4, 7
```

Sind die Koeffizienten keine ganzen oder rationalen Zahlen, so wird automatisch in Gleitpunktzahlen konvertiert:

```
>> p := poly(3*x^3 + sqrt(2)*x + PI, [x]): norm(p), norm(p, 1)
3.141592654, 7.555806216
```

Für Indizes > 1 werden immer Gleitpunktzahlen geliefert:

```
>> p := poly(3*x^3 + 4*x + 1, [x]):
norm(p, 1), norm(p, 2), norm(p, 5), norm(p, 10), norm(p)
8, 5.099019514, 4.174686339, 4.021974513, 4
>> delete p:
```

Beispiel 4. Es werden Normen von polynomialen Ausdrücken berechnet:

```
>> norm(x^3 + 1, 1), norm(x^3 + 1, 2), norm(x^3 + PI)
2, 1.414213562, 1
```

Der folgende Aufruf führt zu einem Fehler, da der Ausdruck als ein Polynom in x mit den unzulässigen symbolischen Koeffizienten $6y$ und $9y^2$ aufgefasst wird:

```
>> f := 6*x*y + 9*y^2 + 2: norm(f, [x])
Error: Illegal argument [norm]
```

Als bi-variables Polynom in den Unbestimmten x, y sind die Koeffizienten $6, 9$ und 2 . Nun können Normen berechnet werden:

```
>> norm(f, [x, y], 1), norm(f, [x, y], 2), norm(f, [x, y])
17, 11.0, 9
>> delete f:
```

Änderungen:

⌘ Keine Änderungen.

normal – Normalform eines Ausdrucks

`normal(f)` liefert die Normalform eines rationalen Ausdrucks `f`, d. h., einen gekürzten rationalen Ausdruck mit ausmultipliziertem Zähler und Nenner.

`normal(object)` ersetzt die Operanden von `object` durch ihre Normalform.

Aufruf(e):

⌘ `normal(f)`
⌘ `normal(object)`

Parameter:

`f` — ein arithmetischer Ausdruck
`object` — ein Polynom vom Typ `DOM_POLY`, eine Liste, eine Menge, eine Tabelle, ein Array, eine Gleichung, eine Ungleichung oder ein Bereich

Rückgabewert: ein Objekt desselben Typs wie das Eingabeobjekt.

Überladbar durch: `object`

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `collect`, `combine`, `denom`, `expand`, `factor`, `gcd`, `indets`, `numer`, `partfrac`, `rationalize`, `rectform`, `rewrite`, `simplify`

Details:

⌘ Enthält das Argument `f` nicht-rationale Teilausdrücke wie z. B. `sin(x)` oder `x^(-1/3)`, so werden diese vor der Normalisierung durch Hilfsvariablen ersetzt, die nach der Normalisierung wieder durch die originalen Teilausdrücke ersetzt werden. Algebraische Abhängigkeiten zwischen solchen Teilausdrücken werden nicht berücksichtigt. Die Operanden nicht-rationaler Teilausdrücke werden rekursiv normalisiert.

☞ Für spezielle Objekte wird `normal` automatisch auf die Operanden angewendet. Speziell werden die Koeffizienten von Polynomen vom Typ `DOM_POLY` normalisiert. Ist `object` eine Menge, eine Liste, eine Tabelle oder ein Array, so wird `normal` automatisch auf alle Einträge angewendet. Weiterhin werden die linken und rechten Seiten von Gleichungen (vom Typ `"_equal"`), Ungleichungen (vom Typ `"_unequal"`) und Vergleichsrelationen (vom Typ `"_less"` oder `"_leequal"`) normalisiert. Auch die Operanden von Bereichen (vom Typ `"_range"`) werden automatisch normalisiert.

Beispiel 1. Es werden die Normalformen einiger rationaler Ausdrücke berechnet:

```
>> normal(x^2 - (x + 1)*(x - 1))
1

>> normal((x^2 - 1)/(x + 1))
x - 1

>> normal(1/(x + 1) + 1/(y - 1))
x + y
-----
y - x + x y - 1
```

Der folgende Ausdruck ist als rationaler Ausdruck in y und $\sin(x)$ anzusehen:

```
>> normal(1/sin(x)^2 + y/sin(x))
y sin(x) + 1
-----
2
sin(x)
```

Beispiel 2. Im Folgenden werden Beispiele zu nicht-rationalen Ausdrücken als Argument gegeben. Zuerst werden die Einträge einer Liste normalisiert:

```
>> [(x^2 - 1)/(x + 1), x^2 - (x + 1)*(x - 1)]
-- 2 --
| x  - 1  2 |
| -----, x  - (x - 1) (x + 1) |
-- x + 1 --
```

```
>> normal(%)
```

```
[x - 1, 1]
```

Bei einem Polynom als Argument werden dessen Koeffizienten normalisiert:

```
>> poly((x^2-1)/(x+1)*Y^2 + (x^2-(x+1)*(x-1))*Y - 1, [Y])
```

```

      / / 2      \
      | | x  - 1 | 2      2
poly| | ----- | Y  + (x  - (x - 1) (x + 1)) Y - 1, [Y] |
      \ \ x + 1  /

```

```
>> normal(%)
```

```

      2
poly((x - 1) Y  + Y - 1, [Y])

```

Änderungen:

☞ Keine Änderungen.

nterms – die Anzahl der Terme eines Polynoms

nterms(p) liefert die Anzahl der Terme des Polynoms p.

Aufruf(e):

☞ nterms(p)

☞ nterms(f <, vars>)

Parameter:

p — ein Polynom vom Typ DOM_POLY

f — ein polynomialer Ausdruck

vars — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: eine nicht-negative Zahl. FAIL wird zurückgeliefert, wenn die Eingabe nicht als Polynom interpretiert werden kann.

Überladbar durch: p

Verwandte Funktionen: coeff, degree, degreevec, ground, lcoeff, ldegree, lmonomial, lterm, nthcoeff, nthmonomial, nthterm, poly, poly2list, tcoeff

Details:

- ⌘ Wenn das erste Argument `f` nicht Element eines Polynom-Domains ist, so konvertiert `nterms` diesen Ausdruck mittels `poly(f)` in ein Polynom. Ist eine Liste von Unbestimmten angegeben, so wird das Polynom `poly(f, vars)` betrachtet.
 - ⌘ Ein Null-Polynom hat keine Terme: der Rückgabewert ist 0.
 - ⌘ `nterms` ist eine Funktion des Systemkerns.
-

Beispiel 1. Einige einfache, selbsterklärende Beispiele:

```
>> nterms(x^2*y^2 + x^2 + y + 2, [x, y])  
  
4  
  
>> nterms(poly(x^2*y^2 + x^2 + y + 2))  
  
4  
  
>> nterms(poly(0, [x]))  
  
0
```

Beispiel 2. Der folgende polynomiale Ausdruck kann in unterschiedlicher Weise als Polynom angesehen werden:

```
>> f := x^2*y^2 + x^2 + y + 2:  
      nterms(f, [x]), nterms(f, [y]), nterms(f, [x, y]),  
      nterms(f, [z])  
  
2, 3, 4, 1  
  
>> delete f:
```

Änderungen:

- ⌘ Keine Änderungen.
-

`nthcoeff` – der `n`-te nicht-triviale Koeffizient eines Polynoms

`nthcoeff(p, n)` gibt den `n`-ten nicht-trivialen Koeffizienten des Polynoms `p` zurück.

Aufruf(e):

```
# nthcoeff(p, <vars,> n <, order>)
```

Parameter:

- p — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- vars — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- n — eine positive ganze Zahl
- order — die Termordnung: entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.

Rückgabewert: ein Element des Koeffizientenrings des Polynoms. Ist p ein Ausdruck, so wird auch der Koeffizient als arithmetischer Ausdruck geliefert. Ist n größer als die tatsächliche Anzahl von Termen, so wird `FAIL` zurückgeliefert.

Überladbar durch: p

Verwandte Funktionen: `coeff`, `collect`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- # Das Argument p kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `nthcoeff` überlädt.
- # Wird eine Liste von Unbestimmten übergeben, so wird das erste Argument als Polynom in diesen Unbestimmten angesehen. Man beachte, dass diese Liste nicht mit den Unbestimmten in p übereinzueinstimmen braucht.
- # Der „erste“ Koeffizient ist der führende Koeffizient, wie er auch von `lcoeff` geliefert wird. Der „letzte“ Koeffizient stimmt mit dem von `tcoeff` gelieferten Resultat überein.
- # `nthcoeff` liefert den n-ten nicht-trivialen Koeffizient bezüglich der lexikographischen Ordnung, falls nicht mittels des Arguments `order` eine andere Ordnung angegeben wird. Siehe Beispiel ??.
- # Das Ergebnis von `nthcoeff` wird nicht weiter evaluiert. Vollständige Evaluation kann mit der Funktion `eval` erzwungen werden. Siehe Beispiel ??.

- ☞ Ein Null-Polynom hat keine Terme, es wird FAIL geliefert.
 - ☞ nthcoeff ist eine Bibliotheksfunktion. Wird keine Termordnung spezifiziert, so werden die Argumente an eine schnelle Kernroutine übergeben.
-

Beispiel 1. Einige einfache, selbsterklärende Beispiele:

```
>> p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
      nthcoeff(p, 1), nthcoeff(p, 2), nthcoeff(p, 3)

      100, 49, 7
```

```
>> nthcoeff(p, 4)

      FAIL
```

```
>> nthcoeff(poly(0, [x]), 1)

      FAIL
```

```
>> delete p:
```

Beispiel 2. Hier wird gezeigt, wie die Angabe einer Liste von Unbestimmten das Ergebnis beeinflusst:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
      nthcoeff(p, [x, y], 2), nthcoeff(p, [y, x], 2)

      3, 2
```

```
>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
      nthcoeff(p, 1), nthcoeff(p, [y, x], 1)

      2, 3
```

```
>> delete p:
```

Beispiel 3. Wir demonstrieren den Effekt der Termordnung:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z])

      4      3      2      2      3
      poly(5 x  + 4 x  y z  + 3 x  y  z + 2, [x, y, z])

>> nthcoeff(p, 1), nthcoeff(p, 1, DegreeOrder),
      nthcoeff(p, 1, DegInvLexOrder)
```

5, 4, 3

Der folgende Aufruf benutzt die umgekehrte lexikographische Termordnung in 3 Unbestimmten:

```
>> nthcoeff(p, 1, Dom::MonomOrdering(RevLex(3)))
```

3

```
>> delete p:
```

Beispiel 4. Das folgende Beispiel demonstriert, dass das Ergebnis von `nthcoeff` nicht immer vollständig evaluiert ist:

```
>> p := poly(3*x^3 + 6*x^2*y^2 + 2, [x]): y := 4:
    nthcoeff(p, 2)
```

$6 y^2$

Die Evaluierung wird durch `eval` erzwungen:

```
>> eval(%)
```

96

```
>> delete p, y:
```

Änderungen:

- ⌘ Selbstdefinierte Termordnungen können nun vorgegeben werden.
 - ⌘ Unbestimmte können nun auch für Polynome vom Typ `DOM_POLY` angegeben werden.
 - ⌘ In früheren MuPAD Versionen war `nthcoeff` eine Kernfunktion.
-

`nthmonomial` – das n-te Monom eines Polynoms

`nthmonomial(p, n)` gibt das n-te nicht-verschwindende Monom des des Polynoms `p` zurück.

Aufruf(e):

⌘ `nthmonomial(p, <vars,> n <, order>)`

Parameter:

- `p` — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- `vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- `n` — eine positive ganze Zahl
- `order` — die Termordnung: entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.

Rückgabewert: ein Polynom vom selben Typ wie `p`. Ist `p` ein Ausdruck, so wird auch das Ergebnis als Ausdruck geliefert. Ist `n` größer als die tatsächliche Anzahl von Termen des Polynoms, so wird `FAIL` zurückgeliefert.

Überladbar durch: `p`

Verwandte Funktionen: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- ☞ Das Argument `p` kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `nthmonomial` überlädt.
- ☞ Wird eine Liste von Unbestimmten übergeben, so wird das erste Argument als Polynom in diesen Unbestimmten angesehen. Das zurückgelieferte Ergebnis ist ebenfalls ein Polynom in diesen Unbestimmten. Man beachte, dass diese Liste nicht mit den Unbestimmten im Eingabepolynom übereinstimmen braucht.
- ☞ Das „erste“ Monom ist das führende Monom, wie es auch von `lmonomial` geliefert wird.
- ☞ `nthmonomial` liefert das `n`-te nicht-verschwindende Monom bezüglich der lexikographischen Ordnung, falls nicht mittels des Arguments `order` eine andere Ordnung angegeben wird. Siehe Beispiel ??.
- ☞ Das von `nthmonomial` berechnete Ergebnis wird nicht weiter evaluiert. Wird dies gewünscht, so kann man dies mit der Funktionen `mapcoeffs` und `eval` erreichen. Siehe Beispiel ??.
- ☞ Ein Null-Polynom hat keine Terme, es wird `FAIL` geliefert.
- ☞ `nthmonomial` ist eine Bibliotheksfunktion. Wird keine Termordnung spezifiziert, so werden die Argumente an eine schnelle Kernroutine übergeben.

Beispiel 1. Einige einfache, selbsterklärende Beispiele:

```
>> p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
      nthmonomial(p, 1), nthmonomial(p, 2), nthmonomial(p, 3)

              100              49              7
      poly(100 x  , [x]), poly(49 x  , [x]), poly(7 x  , [x])

>> nthmonomial(p, 4)

                        FAIL

>> nthmonomial(poly(0, [x]), 1)

                        FAIL

>> delete p:
```

Beispiel 2. Hier wird gezeigt, wie die Angabe einer Liste von Unbestimmten das Ergebnis beeinflusst:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
      nthmonomial(p, [x, y], 2), nthmonomial(p, [y, x], 2)

              2      2
              3 x y , 2 x  y

>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
      nthmonomial(p, 2), nthmonomial(p, [y, x], 2)

              2              2
      poly(3 x y , [x, y]), poly(2 y x , [y, x])

>> delete p:
```

Beispiel 3. Wir demonstrieren den Effekt der Termordnung:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
      nthmonomial(p, 1), nthmonomial(p, 1, DegreeOrder),
      nthmonomial(p, 1, DegInvLexOrder)

              4              3      2
      poly(5 x  , [x, y, z]), poly(4 x  y z , [x, y, z]),

              2  3
      poly(3 x  y  z, [x, y, z])

>> delete p:
```

Beispiel 4. Im Folgenden wird eine benutzerdefinierte Termordnung verwendet, nämlich die umgekehrt lexikographische Termordnung in 3 Unbestimmten:

```
>> order := Dom::MonomOrdering(RevLex(3)):
    p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
    nthmonomial(p, 2, order)
```

$$\text{poly}(4 x^3 y^2 z^2, [x, y, z])$$

Der folgende Aufruf liefert alle Monome:

```
>> nthmonomial(p, i, order) $ i = 1..nterms(p)

      2 3          3 2
poly(3 x y z, [x, y, z]), poly(4 x y z, [x, y, z]),

      4
poly(5 x, [x, y, z]), poly(2, [x, y, z])

>> delete order, p:
```

Beispiel 5. Das folgende Beispiel demonstriert die Evaluationsstrategie von `nthmonomial`:

```
>> p := poly(3*x^3 + 6*x^2*y^2 + 2, [x]): y := 4:
    nthmonomial(p, 2)
```

$$\text{poly}((6 y^2) x^2, [x])$$

Die Evaluierung wird durch `eval` erzwungen:

```
>> mapcoeffs(%, eval)

      2
poly(96 x, [x])

>> delete p, y:
```

Änderungen:

- ⌘ Selbstdefinierte Termordnungen können nun vorgegeben werden.
- ⌘ Unbestimmte können nun auch für Polynome vom Typ `DOM_POLY` angegeben werden.

☞ In früheren MuPAD Versionen war `nthmonomial` eine Kernfunktion.

`nthterm` – der *n*-te Term eines Polynoms

`nthterm(p, n)` gibt den *n*-ten Term des Polynoms *p* zurück.

Aufruf(e):

☞ `nthterm(p, <vars,> n <,> order)`

Parameter:

- p* — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- vars* — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- n* — eine positive ganze Zahl
- order* — die Termordnung: entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.

Rückgabewert: ein Polynom vom selben Typ wie *p*. Ist *p* ein Ausdruck, so wird auch das Ergebnis als Ausdruck geliefert. Ist *n* größer als die tatsächliche Anzahl von Termen des Polynoms, so wird `FAIL` zurückgeliefert.

Überladbar durch: *p*

Verwandte Funktionen: `coeff, degree, degreevec, ground, lcoeff, ldegree, lmonomial, lterm, nterms, nthcoeff, nthmonomial, poly, poly2list, tcoeff`

Details:

- ☞ Das Argument *p* kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `nthterm` überlädt.
- ☞ Es gilt die Identität `nthterm(p, n) nthcoeff(p, n) = nthmonomial(p, n)`.
- ☞ Wird eine Liste von Unbestimmten übergeben, so wird das erste Argument als Polynom in diesen Unbestimmten angesehen. Das zurückgelieferte Ergebnis ist ebenfalls ein Polynom in diesen Unbestimmten. Man beachte, dass diese Liste nicht mit den Unbestimmten im Eingabepolynom übereinstimmen braucht.

- ⌘ Der „erste“ Term ist der Leitterm, wie er auch von `lmonomial` geliefert wird.
- ⌘ `nthterm` liefert den n -ten nicht-verschwindenden Term bezüglich der lexikographischen Ordnung, falls nicht mittels des Arguments `order` eine andere Ordnung angegeben wird. Siehe Beispiel ??.
- ⌘ Ein Null-Polynom hat keine Terme, es wird `FAIL` geliefert.
- ⌘ `nthterm` ist eine Bibliotheksfunktion. Wird keine Termordnung spezifiziert, so werden die Argumente an eine schnelle Kernroutine übergeben.

Beispiel 1. Einige einfache, selbsterklärende Beispiele:

```
>> p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
      nthterm(p, 1), nthterm(p, 2), nthterm(p, 3)

           100           49           7
      poly(x    , [x]), poly(x    , [x]), poly(x    , [x])

>> nthterm(p, 4)

                        FAIL

>> nthterm(poly(0, [x]), 1)

                        FAIL

>> delete p:
```

Beispiel 2. Hier wird gezeigt, wie die Angabe einer Liste von Unbestimmten das Ergebnis beeinflusst:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
      nthterm(p, [x, y], 2), nthterm(p, [y, x], 2)

           2    2
          x y , x y

>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
      nthterm(p, 2), nthterm(p, [y,x], 2)

           2           2
      poly(x y , [x, y]), poly(y x , [y, x])

>> delete p:
```

Beispiel 3. Wir demonstrieren den Effekt der Termordnung:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x,y,z]):
      nthterm(p, 1), nthterm(p, 1, DegreeOrder),
      nthterm(p, 1, DegInvLexOrder)

      4              3      2
      poly(x , [x, y, z]), poly(x y z , [x, y, z]),

      2  3
      poly(x y z, [x, y, z])

>> delete p:
```

Beispiel 4. Im Folgenden wird eine benutzerdefinierte Termordnung verwendet, nämlich die reverse lexikographische Termordnung in 3 Unbestimmten:

```
>> order := Dom::MonomOrdering(RevLex(3)):
      p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x,y,z]):
      nthterm(p, 2, order)

      3      2
      poly(x y z , [x, y, z])
```

Der folgende Aufruf liefert alle Terme:

```
>> nthterm(p, i, order) $ i = 1..nterms(p)

      2  3              3      2
      poly(x y z, [x, y, z]), poly(x y z , [x, y, z]),

      4
      poly(x , [x, y, z]), poly(1, [x, y, z])

>> delete order, p:
```

Beispiel 5. Das n-te Monom ist das Produkt aus dem n-ten Koeffizienten und dem n-ten Term:

```
>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
      mapcoeffs(nthterm(p, 2), nthcoeff(p, 2)) =
      nthmonomial(p, 2)

      2              2
      poly(3 x y , [x, y]) = poly(3 x y , [x, y])

>> delete p:
```

Änderungen:

- ⌘ Selbstdefinierte Termordnungen können nun vorgegeben werden.
 - ⌘ Unbestimmte können nun auch für Polynome vom Typ `DOM_POLY` angegeben werden.
 - ⌘ In früheren MuPAD Versionen war `nthterm` eine Kernfunktion.
-

`null` – Erzeugen des leeren Objekts vom Typ `DOM_NULL`

`null()` gibt das leere Element des Typs `DOM_NULL` zurück.

Aufruf(e):

- ⌘ `null()`

Rückgabewert: das leere Element des Typs `DOM_NULL`.

Verwandte Funktionen: `_exprseq`, `_stmtseq`, `FAIL`, `NIL`

Details:

- ⌘ `null()` gibt das einzige Element des Domains `DOM_NULL` zurück. Es repräsentiert eine leere Folge von MuPAD-Objekten.
 - ⌘ Das leere Objekt erzeugt keine Bildschirmausgabe.
 - ⌘ Einige Systemfunktionen wie z. B. `print` oder `reset` liefern das leere Objekt zurück.
 - ⌘ Das leere Objekt wird aus Ausdrucksfolgen entfernt („ausgleichen“). Es kann dazu benutzt werden, Elemente aus Listen oder Mengen zu entfernen. Siehe Beispiel ??.
 - ⌘ `null` ist eine Funktion des Systemkerns.
-

Beispiel 1. `null()` gibt das leere Objekt zurück, das keine Bildschirmausgabe erzeugt:

```
>> null()
```

Das resultierende Objekt ist vom Domain-Typ `DOM_NULL`:

```
>> domtype(null())
```

DOM_NULL

Leere Ausdrucks- und Befehlsfolgen werden durch `null()` repräsentiert:

```
>> domtype(_exprseq()), domtype(_stmtseq())
```

DOM_NULL, DOM_NULL

Einige Systemfunktionen wie z. B. `print` liefern das leere Objekt:

```
>> print("Hello world!):
```

"Hello world!"

```
>> domtype(%)
```

DOM_NULL

Beispiel 2. Das leere Element wird automatisch aus Listen, Mengen und Ausdrucksfolgen entfernt:

```
>> [null(), a, b, null(), c], {null(), a, b, null(), c},  
    f(null(), a, b, null(), c)
```

[a, b, c], {a, b, c}, f(a, b, c)

```
>> a + null() + b = _plus(a, null(), b)
```

a + b = a + b

```
>> subsop([a, x, b], 2 = null()), subs({a, x, b}, x = null())
```

[a, b], {a, b}

In Arrays und Tabellen kann `null()` jedoch als gültiger Eintrag vorkommen:

```
>> a := array(1..2): a[1] := 1: a[2] := null(): a
```

+-		-+
	1, null()	
+-		-+

```
>> domtype(a[1]), domtype(a[2])
```

DOM_INT, DOM_NULL

```
>> t := table(null() = "void", 1 = 2.5, b = null())
```



```

table(
  b = null(),
  l = 2.5,
  null() = "void"
)

>> domtype(t[b]), t[]

DOM_NULL, "void"

>> delete a, t:

```

Beispiel 3. Wenn aus einer Ausdrucksfolge alle Elemente gelöscht werden, ist das Ergebnis das leere Objekt:

```

>> a := (1, b): delete a[1]: delete a[1]: domtype(a)

DOM_NULL

```

op angewendet auf Objekte ohne Operanden gibt das leere Objekt null() zurück:

```

>> domtype(op([])), domtype(op({})), domtype(op(f()))

DOM_NULL, DOM_NULL, DOM_NULL

>> delete a:

```

Änderungen:

⌘ Keine Änderungen.

numer – der Zähler eines rationalen Ausdrucks

numer(f) liefert den Zähler des Ausdrucks f.

Aufruf(e):

⌘ numer(f)

Parameter:

f — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `f`

Verwandte Funktionen: `denom`, `factor`, `gcd`, `normal`

Details:

☞ `numer` behandelt die Eingabe als einen rationalen Ausdruck: nicht-rationale Teilausdrücke wie z. B. `sin(x)`, `x^(1/2)` etc. werden intern durch „temporäre Variablen“ ersetzt. Der Zähler dieses rationalisierten Ausdrucks wird berechnet, die temporären Variablen werden zuletzt wieder durch die ursprünglichen Teilausdrücke ersetzt.

☞ Es wird nicht immer gekürzt: der von `numer` berechnete Zähler ist im allgemeinen nicht teilerfremd zu dem mittels `denom` bestimmten Nenner. Mittels `normal` kann erreicht werden, daß Zähler und Nenner gekürzt werden. Siehe Beispiel ??.



Beispiel 1. Die Zähler einiger Ausdrücke werden berechnet:

```
>> numer(-3/4)
```

-3

```
>> numer(x + 1/(2/3*x - 2/x))
```

$$\frac{3}{2x^2 - 3x}$$

```
>> numer((cos(x)^2 - 1)/(cos(x) - 1))
```

$$\frac{\cos(x)^2}{\cos(x) - 1}$$

Beispiel 2. Ein Bruch wird nicht gekürzt, wenn er in der Form „Zähler/Nenner“ gegeben ist:

```
>> r := (x^2 - 1)/(x^3 - x^2 + x - 1): numer(r)
```

$$\frac{x^2}{x^3 - 1}$$

Dieser Zähler ist nicht teilerfremd zum Nenner von `r`. Mit `normal` wird das Kürzen gemeinsamer Faktoren in Zähler und Nenner erzwungen:

```
>> numer(normal(r))
```

$$x + 1$$

Ist dagegen der rationale Ausdruck eine Summe rationaler Ausdrücke, so wird automatisch normalisiert:

```
>> numer(r + x/(x + 1) + 1/(x + 1) - 1)
```

$$x + 1$$

```
>> delete r:
```

Änderungen:

⌘ Keine Änderungen.

ode – das Domain gewöhnlicher Differentialgleichungen

`ode(eq, y(x))` repräsentiert eine gewöhnliche Differentialgleichung (engl.: ordinary differential equation) für die Funktion $y(x)$.

`ode({eq1, eq2, ...}, {y1(x), y2(x), ...})` repräsentiert ein System gewöhnlicher Differentialgleichungen für die Funktionen $y1(x), y2(x)$ etc.

Aufruf(e):

⌘ `ode(eq, y(x))`

⌘ `ode({eq <, inits>}, y(x))`

⌘ `ode({eq1, eq2, ... <, inits>}, {y1(x), y2(x), ...})`

Parameter:

<code>eq, eq1, eq2, ...</code>	— Gleichungen oder arithmetische Ausdrücke in den gesuchten Funktionen und ihren Ableitungen nach x . Ein arithmetischer Ausdruck wird als Gleichung mit verschwindender rechter Seite behandelt.
<code>y, y1, y2, ...</code>	— die gesuchten Funktionen: Bezeichner
<code>x</code>	— die unabhängige Variable: ein Bezeichner
<code>inits</code>	— die Anfangs- oder Randbedingungen: eine Folge von Gleichungen

Rückgabewert: ein Objekt vom Typ `ode`.

Weitere Dokumentation: Abschnitt 8.3 des Tutoriums.

Verwandte Funktionen: `numeric::odesolve`, `numeric::odesolve2`,
`plot::ode`

Details:

- ☞ In den Gleichungen `eq`, `eq1` usw. müssen die unbekannten Funktionen in der Form $y(x)$, $y1(x)$ usw. repräsentiert werden. Die Ableitungen werden entweder mit Hilfe der Funktion `diff` oder dem Differentialoperator `D` dargestellt. Man beachte, dass man das Zeichen `'` zur abkürzenden Schreibweise verwenden kann: $y'(x) = D(y)(x) \equiv \text{diff}(y(x), x)$.
- ☞ Die unbekannten Funktionen müssen univariat in der unabhängigen Variablen x sein. Multivariate Ausdrücke wie z. B. $y(x, t)$ werden nicht akzeptiert.
- ☞ Anfangs- und Randbedingungen werden durch eine Folge von Gleichungen beschrieben, deren linke Seiten die unbekannte Funktionen oder eine ihrer Ableitungen enthalten. Auf der rechten Seite der Gleichungen sind die entsprechenden Werte anzugeben. Um die Werte von Ableitungen an Punkten zu spezifizieren, muss der Differentialoperator `D` (oder das Zeichen `'`) verwendet werden. Beispielsweise ist

$$y(1) = 2, y'(0) = 0, y''(0) = 1$$

eine zulässige Folge von Randbedingungen für `inits`.

Es sind Randbedingungen der ersten und zweiten Art erlaubt. Allgemeinere Randbedingungen werden nicht akzeptiert.

Anfangs-/Randpunkte und die entsprechenden Anfangs-/Randwerte können symbolische Ausdrücke sein.

- ☞ Die Angabe der Bedingungen von skalaren Anfangswert- und Randwertproblemen erfolgt in der Form `ode({eq, inits}, y(x))`.
- ☞ Ein System von Differentialgleichungen muss so viele Gleichungen wie unbekannte Funktionen enthalten.
- ☞ Das Domains `ode` dient im Wesentlichen dazu, eine Umgebung bereitzustellen, in der die Funktion `solve` zum Lösen gewöhnlicher Differentialgleichungen überladen werden kann.
Im Fall einer (skalaren) Differentialgleichung (auch mit Anfangs- oder Randbedingungen) liefert `solve` eine Menge expliziter Lösungen oder eine implizite Lösung. Jedes Element der Lösungsmenge stellt einen Lösungsweig dar.
Bei Systemen von Differentialgleichungen wird eine Menge von Listen von Gleichungen für die gesuchten Funktionen ausgegeben. Jede Liste entspricht einem Lösungsweig.

Ein symbolischer solve-Aufruf wird zurückgegeben, falls keine Lösung gefunden wird.

☞ Nach `setuserinfo(ode, 10)` liefert ein `solve`-Aufruf Informationen zu den verwendeten Lösungsstrategien.

Beispiel 1. Im Folgenden wird gezeigt, wie man gewöhnliche skalare Differentialgleichungen erzeugen und lösen kann. Zunächst soll die Differentialgleichung $x^2 y'(x) + 3x y(x) = \sin(x)/x$ definiert werden. Zur Eingabe der Ableitung wird hier das Zeichen `'` verwendet:

```
>> eq := ode(x^2*y'(x) + 3*x*y(x) = sin(x)/x, y(x))
```

$$\text{ode} \left| \begin{array}{c} \frac{\sin(x)}{x} \\ 3x y(x) - \frac{\sin(x)}{x} + x^2 \text{diff}(y(x), x), y(x) \end{array} \right|$$

Man erhält ein Element des Domains `ode`, welches man mit `solve` lösen kann:

```
>> solve(eq)
```

$$\left\{ \begin{array}{l} C1 + \cos(x) \\ - \frac{\sin(x)}{x^3} \end{array} \right\}$$

```
>> delete eq:
```

Beispiel 2. Ein Anfangswertproblem wird definiert als eine Menge, die aus der Differentialgleichung und den Anfangsbedingungen besteht:

```
>> ivp := ode({f''(t) + 4*f(t) = sin(2*t),
               f(0) = a, f'(0) = b}, f(t))
```

$$\text{ode}(\{f(0) = a, D(f)(0) = b, 4 f(t) + \text{diff}(f(t), t, t) - \sin(2 t)\}, f(t))$$

```
>> solve(ivp)
```

$$\left\{ \begin{array}{l} \frac{\sin(2 t)}{8} + a \cos(2 t) + \frac{b \sin(2 t)}{2} - \frac{t \cos(2 t)}{4} \end{array} \right\}$$

```
>> delete ivp:
```

```
>> sys := {x'(t) - x(t) + y(t) = 0, y'(t) - x(t) - y(t) = 0}
          {y(t) - x(t) + D(x)(t) = 0, D(y)(t) - y(t) - x(t) = 0}
```

```
>> solution := solve(ode(sys, {x(t), y(t)}))
{[y(t) = C7 exp((1 + I) t) + C8 exp((1 - I) t),
  x(t) = I C7 exp((1 + I) t) - I C8 exp((1 - I) t)]}
```

```
>> eval(subs(rewrite(sys, diff), op(solution)))
```

$$\{0 = 0\}$$

```
>> solve(ode({y'(x) + x*y(x) = cos(x), y(0) = 3}, y(x)))
```

$$\left\{ \left\{ \frac{1}{3 \exp\left(-\frac{x^2}{2}\right)} + \frac{\int \cos(t^2) \exp(t^2) dt, t^2 = 0..x}{\exp(x^2/2)} \right\} \right\}$$

```
>> sys := {x'(t) = -3*y(t)*z(t),
           y'(t) = 3*x(t)*z(t),
           z'(t) = -x(t)*y(t)}:
solution := solve(ode(sys, {x(t), y(t), z(t)}))
```

$$\{ [x(t) = (C10 - C11 + 3 z(t))^{2/2}, y(t) = (-C10 - 3 z(t))^{2/2}]$$

$$, [x(t) = (C10 - C11 + 3 z(t))^{2/2},$$

$$y(t) = -(-C10 - 3 z(t))^{2/2}],$$

$$[x(t) = -(C10 - C11 + 3 z(t))^{2/2},$$

$$y(t) = (-C10 - 3 z(t))^{2/2}], [$$

$$x(t) = -(C10 - C11 + 3 z(t))^{2/2},$$

$$y(t) = -(-C10 - 3 z(t))^{2/2}] \}$$

In diesen vier Lösungszweigen wird keine Lösung für die gesuchte Funktion $z(t)$ angegeben. Allerdings sind die obigen Lösungen nur unter einer zusätzlichen Bedingung an $z(t)$ gültig. Die erste der vier Lösungen wird wieder in das Gleichungssystem `sys` eingesetzt:

```
>> eval(subs(rewrite(sys, diff), solution[1]))
```

$$\{$$

$$\{ \text{diff}(z(t), t) = -(-C10 - 3 z(t))^{2/2}$$

$$\{$$

$$\{$$

$$(C10 - C11 + 3 z(t))^{2/2}, - \frac{3 z(t) \text{diff}(z(t), t)}{(-C10 - 3 z(t))^{2/2}} =$$

$$3 z(t) (C10 - C11 + 3 z(t))^{2/2},$$

$$\frac{3 z(t) \text{diff}(z(t), t)}{(C10 - C11 + 3 z(t))^{2/2}} = -3 z(t) (-C10 - 3 z(t))^{2/2} \}$$

$$\}$$

$$\}$$

Für jeden Lösungszweig verbleibt genau eine Differentialgleichung für $z(t)$.

```
>> delete sys, solution:
```

Beispiel 5. Es kommt vor, dass MuPAD eine gegebene Differentialgleichung nicht lösen kann. In einem solchen Fall wird der symbolische solve-Aufruf zurückgeliefert:

```
>> solve(ode(y'(x) + y(x)^2 = b + a*x, y(x)))
solve(ode(- b - a x + diff(y(x), x) + y(x)^2, y(x)))
```

Beispiel 6. MuPADs Differentialgleichungslöser enthält auch algebraische Algorithmen zur Berechnung liouvillescher Lösungen von gewöhnlichen linearen Differentialgleichungen über den rationalen Funktionen. Diese Algorithmen basieren auf der Differential-Galois-Theorie. Für Gleichungen zweiter Ordnung ist ein zum Kovacic-Algorithmus ähnlicher Algorithmus implementiert. Jedoch werden bei diesem Algorithmus, anstelle der Berechnung des so genannten Riccati-Polynoms, die Lösungen direkt durch Lösungsformeln berechnet. Nur wenn beide Lösungen algebraisch sind wird es in drei Fällen notwendig das Minimalpolynom einer Lösung zu berechnen, wieder durch Lösungsformeln. Für das berühmte Kovacic-Beispiel

$$y'' + \left(\frac{3}{16x^2} + \frac{2}{9(x-1)^2} - \frac{3}{16x(x-1)} \right) y = 0$$

kann daher hier das Minimalpolynom einer Lösung berechnet werden:

```
>> solve(ode(y''(x) + (3/(16*x^2) + 2/(9*(x - 1)^2)
- 3/(16*x*(x - 1)))*y(x), y(x)))
{C2 RootOf(_Y1^24 + 2985984 x^8 (x - 1)^8 -
2799360 x^4 _Y1^8 (x - 1)^6 - 4320 x^2 _Y1^16 (x - 1)^3 -
165888 x^5 _Y1^4 (x - 2) (I x^3 - I x^3) +
5760 I x^3 _Y1^12 (x - 2) (I x^3 - I x^3) , _Y1)}
```

Auch für Gleichungen höherer Ordnung kann MuPAD liouvillesche Lösungen finden. Dabei ist aber nicht mehr garantiert, dass alle liouvilleschen Lösungen gefunden werden. Folgende Gleichung dritter Ordnung kann vollständig gelöst werden:

$$\frac{C3 (2 x + 2 (x (x - 1)))^{1/2} - 1)^{1/14}}{}$$

$$\frac{\int \frac{(2x^2 + 2x(x-1))^{\frac{1}{2}} - 1}{(2x^2 + 2x(x-1))^{\frac{1}{2}} - 1} \exp(-x) (x(x-1))^{\frac{1}{2}} dx}{(2x^2 + 2x(x-1))^{\frac{1}{2}} - 1}, x \int \frac{\exp(-x) (x(x-1))^{\frac{1}{2}}}{(2x^2 + 2x(x-1))^{\frac{1}{2}} - 1} dx / 28$$

Hintergründe:

☞ Die in `ode` verwendeten Methoden zur Lösung von gewöhnlichen Differentialgleichungen stammen hauptsächlich aus

- Daniel Zwillinger. Handbook of differential equations. San Diego: Academic Press (1992).

Die mehr algebraischen Algorithmen zur Lösung von linearen Differentialgleichungen sind beschrieben in

- Winfried Fakler. Algebraische Algorithmen zur Lösung von linearen Differentialgleichungen. Stuttgart, Leipzig: Teubner, Reihe MuPAD Reports (1999).
- Winfried Fakler. On second order homogeneous linear differential equations with Liouvillian solutions. Theor. Comp. Science **187**, 27-48 (1997).

Änderungen:

☞ Keine Änderungen.

`op` – die Operanden eines Objekts

`op(object)` liefert alle Operanden des Objekts.

`op(object, i)` liefert den i -ten Operanden.

`op(object, i..j)` liefert den i -ten bis j -ten Operanden.

Aufruf(e):

☞ `op(object)`

☞ `op(object, i)`

☞ `op(object, i..j)`

☞ `op(object, [i1, i2, ...])`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

`i, j` — nichtnegative ganze Zahlen

`i1, i2, ...` — nichtnegative ganze Zahlen oder Bereiche
nichtnegativer ganzer Zahlen

Rückgabewert: eine Folge von Operanden oder der angeforderte Operand.
`FAIL` wird geliefert, wenn kein entsprechender Operand existiert.

Überladbar durch: `object`

Verwandte Funktionen: `_index`, `contains`, `extnops`, `extop`,
`extsubsop`, `map`, `new`, `nops`, `select`, `split`, `subsop`, `zip`

Details:

- ☞ MuPAD-Objekte bestehen aus einfachen Bausteinen, den „Operanden“. Die Funktion `op` ist das Hilfsmittel, Objekte in diese Bausteine zu zerlegen bzw. einzelne Operanden zu extrahieren. Die tatsächliche Bedeutung der Operanden hängt dabei vom Typ des Objekts ab. Der Abschnitt ‚Hintergründe‘ dieser Hilfeseite erklärt die Bedeutung für einige der wichtigsten Basistypen.
 - ☞ `op(object)` gibt eine Folge aller Operanden (außer dem 0-ten) zurück. Dieser Aufruf ist äquivalent zu `op(object, 1..nops(object))`. Siehe Beispiel ??.
 - ☞ `op(object, i)` gibt den *i*-ten Operanden zurück. Siehe Beispiel ??.
 - ☞ `op(object, i..j)` gibt eine aus dem *i*-ten bis zum *j*-ten Operanden bestehende Folge zurück; *i* und *j* müssen dabei nichtnegative ganze Zahlen und *i* darf nicht größer als *j* sein. Dieser Aufruf ist äquivalent zu `op(object, k) $ k = i..j`. Siehe Beispiel ??.
 - ☞ `op(object, [i1, i2, ...])` ist eine abkürzende Schreibweise für den geschachtelten Aufruf `op(...op(op(object, i1), i2), ...)`, falls *i1*, *i2*, ... ganze Zahlen sind.
Ein Aufruf der Form `op(object, [i..j, i2])` mit ganzen Zahlen *i* < *j* entspricht `map(op(object, i..j), op, i2)`. Siehe Beispiel ??.
 - ☞ `op` liefert `FAIL`, wenn ein angeforderter Operand nicht existiert. Siehe Beispiel ??.
 - ☞ Ausdrücke vom Domain-Typ `DOM_EXPR` sowie Arrays besitzen einen 0-ten Operanden.
 - Für Ausdrücke ist dies der „Operator“, der die Operanden verbindet. Für symbolische Funktionsaufrufe ist dies der Funktionsname.
 - Für ein Array ist der 0-te Operand eine Folge aus einer ganzen Zahl (die Dimension des Arrays) sowie Bereichsangaben für jeden Array-Index.
- Anderen Kerntypen wie z. B. Listen oder Mengen besitzen keinen 0-ten Operanden. Siehe Beispiel ??.
- ☞ Für Bibliotheksdatentypen (Domains) kann `op` überladen werden. In der "op"-Methode kann mit `extop` auf die interne Repräsentation zugegriffen werden. In einer solchen überladenden Funktion brauchen lediglich die Aufruftypen `op(x)`, `op(x, i)` und `op(x, i..j)` behandelt zu werden; der Fall `op(x, [i1, i2, ...])` wird von der Systemfunktion aufgelöst. Siehe Beispiel ??.

☞ `op` ist für Kerndatentypen nicht überladbar.

☞ `op` ist eine Funktion des Systemkerns.

Beispiel 1. Der Aufruf `op(object)` liefert alle Operanden:

```
>> op([a, b, c, [d, e], x + y])
      a, b, c, [d, e], x + y

>> op(a + b + c^d)
      a, b, c
           d

>> op(f(x1, x2, x3))
      x1, x2, x3
```

Beispiel 2. Der Aufruf `op(object, i)` liefert einen einzelnen Operanden:

```
>> op([a, b, c, [d, e], x + y], 4)
      [d, e]

>> op(a + b + c^d, 3)
      c
           d

>> op(f(x1, x2, x3), 2)
      x2
```

Beispiel 3. Der Aufruf `op(object, i..j)` liefert mehrere Operanden:

```
>> op([a, b, c, [d, e], x + y], 3..5)
      c, [d, e], x + y

>> op(a + b + c^d, 2..3)
      b, c
           d

>> op(f(x1, x2, x3), 2..3)
```

x2, x3

Ein Bereich kann auch auf den 0-ten Operanden zugreifen, sofern er existiert:

```
>> op(a + b + c^d, 0..2)
```

_plus, a, b

```
>> op(f(x1, x2, x3), 0..2)
```

f, x1, x2

Beispiel 4. Der Aufruf `op(object, [i1, i2, ...])` liefert Suboperanden:

```
>> op([a, b, c, [d, e], x + y], [4, 1])
```

d

```
>> op(a + b + c^d, [3, 2])
```

d

```
>> op(f(x1, x2, x3 + 17), [3, 2])
```

17

Auch Bereiche von Suboperanden können angefordert werden:

```
>> op([a, b, c, [d, e], x + y], [4..5, 2])
```

e, y

```
>> op(a + b + c^d, [2..3, 1])
```

b, c

```
>> op(f(x1, x2, x3 + 17), [2..3, 1])
```

x2, x3

Beispiel 5. Nicht existierende Operanden werden als FAIL zurückgeliefert:

```
>> op([a, b, c, [d, e], x + y], 8), op(a + b + c^d, 4),  
    op(f(x1, x2, x3), 4)
```

FAIL, FAIL, FAIL

Beispiel 6. Der 0-te Operand eines Ausdrucks vom Typ `DOM_EXPR` ist „der Operator“, der die anderen Operanden verbindet:

```
>> op(a + b + c, 0), op(a*b*c, 0), op(a^b, 0), op(a[1, 2], 0)
      _plus, _mult, _power, _index
```

Für symbolische Funktionsaufrufe ist dies der Funktionsname:

```
>> op(f(x1, x2, x3), 0), op(sin(x + y), 0), op(besselJ(0, x), 0)
      f, sin, besselJ
```

Der 0-te Operand eines Arrays ist eine Folge aus der Array-Dimension und den Bereichen der Array-Indizes:

```
>> op(array(3..100), 0)
      1, 3..100

>> op(array(1..2, 1..3, 2..4), 0)
      3, 1..2, 1..3, 2..4
```

Für andere Kerndatentypen wie z.B. Listen, Mengen, Tabellen etc. existiert kein 0-ter Operand:

```
>> op([1, 2, 3], 0), op({1, 2, 3}, 0), op(table(1 = y), 0)
      FAIL, FAIL, FAIL
```

Beispiel 7. Für Bibliotheksdatentypen kann `op` überladen werden. Zunächst wird mittels `newDomain` ein neuer Datentyp `d` definiert. Die `"new"`-Methode dient zur Erzeugung von Elementen dieses Typs. Die interne Darstellung des Datentyps ist eine Liste aus den Argumenten der `"new"`-Methode:

```
>> d := newDomain("d"): d::new := () -> new(dom, [args()]):
```

Die `"op"`-Methode dieses Datentyps wird definiert. Sie soll die Elemente einer sortierten Kopie der internen Liste zurückliefern. In der Implementation wird dabei mittels `extop` auf die interne Liste zugegriffen:

```
>> d::op := proc(x, i = null())
      local internalList;
      begin
        internalList := extop(x, 1);
        op(sort(internalList), i)
      end_proc;
```

Durch Überladung wird diese Methode aufgerufen, wenn die Operanden eines Objekts vom Typ `d` mittels `op` angefordert werden:

```
>> e := d(3, 7, 1): op(e); op(e, 2); op(e, 1..2)

1, 3, 7

3

1, 3

>> delete d, e;
```

Beispiel 8. Bezeichner, ganze Zahlen, reelle Gleitpunktzahlen, boolesche Konstanten und Zeichenketten sind „atomare“ Objekte. Ihr einziger Operand ist das Objekt selbst:

```
>> op(x), op(17), op(0.1234), op("Hello World!")

x, 17, 0.1234, "Hello World!"
```

Für rationale Zahlen sind die Operanden der Zähler und der Nenner:

```
>> op(17/3)

17, 3
```

Für komplexe Zahlen sind die Operanden der Real- und der Imaginärteil:

```
>> op(17 - 7/3*I)

17, -7/3
```

Beispiel 9. Für Mengen liefert `op` die Operanden gemäß ihrer *internen* Anordnung. Diese kann sich von der Anordnung der Elemente unterscheiden, mit der Mengen auf dem Bildschirm ausgegeben werden:

```
>> s := {1, 2, 3}

{1, 2, 3}

>> op(s)

3, 2, 1
```

Indizierter Zugriff auf Mengenelemente benutzt die auf dem Bildschirm sichtbare Ausgabeordnung:

```
>> s[1], s[2], s[3]
```

```
1, 2, 3
```

Man beachte, dass der Zugriff auf Mengenelemente über `op` wesentlich schneller ist als über indizierte Aufrufe:

```
>> s := {sqrt(i) $ i = 1..500}:
      time([op(s)]) / time([s[i] $ i = 1..nops(s)]);

1/364
```

```
>> delete s:
```

Beispiel 10. Die Operanden einer Liste sind die Listenelemente:

```
>> op([a, b, c, [d, e]])

a, b, c, [d, e]

>> op([[a11, a12], [a21, a22]], [2, 1])

a21
```

Beispiel 11. Intern bilden die Operanden von Arrays eine „lineare“ Struktur aus allen Einträgen:

```
>> op(array(1..2, 1..2, [[11, 12], [21, 22]]))

11, 12, 21, 22
```

Nicht initialisierte Einträge werden als `NIL` zurückgegeben:

```
>> op(array(1..2, 1..2))

NIL, NIL, NIL, NIL
```

Beispiel 12. Die Operanden einer Tabelle sind die Gleichungen, die einem Index den jeweiligen Tabelleneintrag zuordnen:

```
>> T := table((1, 2) = x + y, "diff(sin)" = cos, a = b)

table(
  a = b,
  "diff(sin)" = cos,
  (1, 2) = x + y
)
```



```
>> op(T)
```

```
      a = b, "diff(sin)" = cos, (1, 2) = x + y
```

```
>> delete T:
```

Beispiel 13. Ausdrucksfolgen werden nicht ausgeglichen:

```
>> op((a, b, c), 2)
```

```
      b
```

Man beachte jedoch, dass die Argumente von `op` ausgewertet werden. Im folgenden Aufruf wird das Objekt `x` durch die Auswertung ausgeglichen:

```
>> x := hold((1, 2), (3, 4)): op(x, 1)
```

```
      1
```

Die Vereinfachung von `x` lässt sich mit `val` verhindern:

```
>> op(val(x), 1)
```

```
      1, 2
```

```
>> delete x:
```

Hintergründe:

☞ Hier wird die Bedeutung der Operanden für einige wichtige Basistypen erklärt:

- Bezeichner, ganze Zahlen, reelle Gleitpunktzahlen, die booleschen Konstanten sowie Zeichenketten sind „atomare“ Objekte. Sie haben nur einen Operanden: das Objekt selbst. Siehe Beispiel ??.
- Eine rationale Zahl vom Typ `DOM_RAT` hat 2 Operanden: den Zähler und den Nenner. Siehe Beispiel ??.
- Eine komplex Zahl vom Typ `DOM_COMPLEX` hat 2 Operanden: den Real- und den Imaginärteil. Siehe Beispiel ??.
- Die Operanden einer Menge sind ihre Elemente.

Man beachte jedoch, dass die Anordnung der Elemente, mit der die Menge auf dem Bildschirm ausgegeben wird, nicht mit der internen Ordnung übereinzustimmen braucht, auf die sich `op` bezieht. Siehe Beispiel ??.



- Die Operanden einer Liste sind ihre Elemente. Siehe Beispiel ??.

- Die Operanden eines Arrays sind seine Einträge. Nicht initialisierte Einträge werden als NIL zurückgeliefert. Siehe die Beispiele ?? sowie ??.
- Die Operanden einer Tabelle sind die Gleichungen, die den Indizes die entsprechenden Einträgen zuordnen. Siehe Beispiel ??.
- Die Operanden einer Ausdrucksfolge sind deren Elemente. Man beachte, dass Folgen nicht durch op ausgeglichen werden. Siehe Beispiel ??.
- Die Operanden eines symbolischen Funktionsaufrufs wie z. B. $f(x, y, \dots)$ sind die Argumente x, y etc. Der Funktionsname f ist der 0-te Operand.
- Die Operanden von Ausdrücken vom Typ DOM_EXPR sind durch ihre interne Darstellung gegeben. Es gibt einen 0-ten Operanden („der Operator“), der den Typ des Ausdrucks bestimmt. Intern entspricht der Operator einer Funktion, der Ausdruck einem Funktionsaufruf. Beispielsweise ist $a + b + c$ als `_plus(a, b, c)` zu interpretieren, ein symbolischer indizierter Aufruf wie beispielsweise $A[i, j]$ entspricht `_index(A, i, j)`. Der Name der Systemfunktion ist der 0-te Operand (z. B., `_plus` und `_index` in den vorherigen Beispielen), die Argumente des Funktionsaufrufs sind die weiteren Operanden.

Änderungen:

⌘ Keine Änderungen.

operator – Definition eines neuen Operatorsymbols

`operator(symb, f, typ, prio)` führt ein neues Operatorsymbol `symb` vom Typ `typ` mit Priorität `prio` ein. Die Funktion `f` dient zur Auswertung von Ausdrücken, in denen der neue Operator verwendet wird.

`operator(symb, Delete)` löscht das Operatorsymbol `symb`.

Aufruf(e):

⌘ `operator(symb, f <, T, prio>)`

⌘ `operator(symb, Delete)`

Parameter:

- `symb` — das Operatorsymbol: eine Zeichenkette.
- `f` — eine Funktion zur Auswertung von Ausdrücken, in denen der Operator verwendet wird.
- `T` — der Typ des Operators: eine der Optionen *Prefix*, *Postfix*, *Binary* oder *Nary*. Die Voreinstellung ist *Nary*.
- `prio` — die Priorität des Operators: eine ganze Zahl zwischen 1 und 1999. Die Voreinstellung ist 1300.

Optionen:

- Prefix* — der Operator ist ein unärer Operator mit Präfix-Notation
- Postfix* — der Operator ist ein unärer Operator mit Postfix-Notation
- Binary* — der Operator ist ein nicht-assoziativer binärer Operator mit Infix-Notation
- Nary* — der Operator ist ein assoziativer binärer Operator mit Infix-Notation
- Delete* — der Operator mit dem Symbol `symb` wird gelöscht

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Seiteneffekte: Das neue Operatorsymbol `symb` wird vom Parser erkannt und kann verwendet werden, um Ausdrücke einzugeben. Das neue Operatorsymbol wird *nicht* verwendet, wenn Dateien mit der Funktion `read` und deren Option *Plain* eingelesen werden.

Details:

- ☞ `operator` wird verwendet, um neue Operatorsymbole zu definieren oder selbstdefinierte zu löschen.
- ☞ Gegeben sei das Operatorsymbol `"++"` mit der Evaluationsfunktion `f`. Dann baut der Parser für den jeweiligen Operatortyp die folgenden Ausdrücke auf:
 - Prefix:** Die Eingabe `++x` resultiert in `f(x)`.
 - Postfix:** Die Eingabe `x++` resultiert in `f(x)`.
 - Binary:** Die Eingabe `x ++ y ++ z` resultiert in `f(f(x, y), z)`.
 - Nary:** Die Eingabe `x ++ y ++ z` resultiert in `f(x, y, z)`.
- ☞ Operatorsymbole können Präfixe von anderen, längeren Operatorsymbolen sein. Der Scanner liest so viele Zeichen wie möglich und wählt dann das längste Operatorsymbol, welches aus den eingelesenen Zeichen besteht. Siehe Beispiel ??.
- ☞ Es ist nicht möglich, zwei Operatoren mit demselben Symbol zu definieren. Somit kann man nicht gleichzeitig ein unäres `++` und ein binäres `++` definieren.

☞ Die Zeichenkette `symp` für das Operatorsymbol muss folgenden Bedingungen genügen:

- Sie darf nicht länger als 32 Zeichen sein.
- Sie darf nicht mit einem Leerzeichen beginnen.
- Sie darf nicht mit dem Zeichen `\` (Backslash) beginnen.

Somit sind die Zeichenketten `" @"` und `"\\/"` keine zulässigen Operatorsymbole. Bitte beachten Sie, dass dies zur Zeit von `operator` nicht überprüft werden kann.

☞ Es ist möglich, die vordefinierten Systemoperatoren umzudefinieren.

☞ Es ist nicht möglich, andere Formen von Operatoren wie z. B. $|x|$ oder dreistellige Operatoren zu definieren.

☞ Das neue Operatorsymbol wird auch verwendet, wenn Dateien eingelesen werden. Es gibt folgende Ausnahme: Wird eine Datei mit der Funktion `read` und deren Option `Plain` gelesen, so werden vom Benutzer außerhalb der Datei definierte Operatoren nicht berücksichtigt. Diese Option wird verwendet, um MuPADs Bibliotheksdateien zu lesen. Dadurch wird vermieden, dass benutzerdefinierte Operatoren die Bedeutung der Bibliotheksdateien unkontrolliert verändern.

☞ Bislang gibt es keine komfortable Möglichkeit, die Ausgabe von Ausdrücken zu konfigurieren, die benutzerdefinierte Operatoren enthalten. (Man kann die Funktion `builtin` verwenden, um die textuelle Ausgabe von Ausdrücken zu definieren. Dies wird jedoch nicht empfohlen.)

☞ Die Prioritäten der vordefinierten Operatoren sind in der MuPAD 2.0 Kurzübersicht aufgelistet.

Option **<Prefix>**:

☞ Der neue Operator wird als unärer Operator mit Präfix-Notation angesehen. Gegeben sei das Operatorsymbol `"++"` und die Evaluierungsfunktion `f`. Dann wird die Eingabe `++x` vom Parser in den Ausdruck `f(x)` umgewandelt.

Option **<Postfix>**:

☞ Der neue Operator wird als unärer Operator mit Postfix-Notation angesehen. Gegeben sei das Operatorsymbol `"++"` und die Evaluierungsfunktion `f`. Dann wird die Eingabe `x++` vom Parser in den Ausdruck `f(x)` umgewandelt.

Option <Binary>:

- ☞ Der neue Operator wird als nicht-assoziativer binärer Operator mit Infix-Notation angesehen. Gegeben sei das Operatorsymbol "++" und die Evaluierungsfunktion f . Dann wird die Eingabe $x ++ y ++ z$ vom Parser in den Ausdruck $f(f(x, y), z)$ umgewandelt. Der Operator bindet also von links nach rechts.

Option <Nary>:

- ☞ Der neue Operator wird als assoziativer n-ärer Operator mit Infix-Notation angesehen. Gegeben sei das Operatorsymbol "++" und die Evaluierungsfunktion f . Dann wird die Eingabe $x ++ y ++ z$ vom Parser in den Ausdruck $f(x, y, z)$ umgewandelt.

Beispiel 1. Im Folgenden wird ein Operator für die logische Äquivalenz definiert:

```
>> equiv := (a, b) -> (a and b) or (not a and not b):  
      operator("<=>", equiv, Binary, 50):
```

Nach diesem Aufruf kann das Symbol `<=>` zur Eingabe von Ausdrücken benutzt werden:

```
>> a <=> FALSE, bool(1 < 0 <=> 1 > 0)  
      not a, FALSE  
  
>> operator("<=>", Delete):
```

Beispiel 2. Bezeichner sind als Operatorsymbole erlaubt:

```
>> operator("x", _vector_product, Binary, 1000):  
  
>> a x b x c  
      _vector_product(_vector_product(a, b), c)  
  
>> operator("x", Delete):
```

Beispiel 3. Im Folgenden wird gezeigt, dass der Scanner immer das längste mögliche Operatorsymbol verwendet:

```
>> operator("~", F, Prefix, 1000):  
      operator("~>", F1, Prefix, 1000):  
      operator("~~>", F2, Prefix, 1000):  
  
>> ~~ x, ~~> x, ~ ~> x, ~~~> x  
  
      F(F(x)), F2(x), F(F1(x)), F(F2(x))  
  
>> operator("~", Delete):  
      operator("~>", Delete):  
      operator("~~>", Delete):
```

Hintergründe:

- ⌘ Wenn der Scanner ein neues Token einliest, verwirft er zuerst Leerzeichen und das Backslash-Zeichen. Danach versucht er, ein benutzerdefiniertes Operatorsymbol zu lesen. Das längste passende Symbol wird schließlich verwendet. Kann kein passendes Symbol gefunden werden, so liest der Scanner das nächste Token gemäß der vordefinierten Syntax.
- ⌘ Der Parser verwendet sowohl „recursive-descend“ als auch „operator precedence“ Techniken. Vordefinierte und benutzerdefinierte Operatoren werden gemäß ihrer Priorität geparkt („operator precedence parsing“).

Änderungen:

- ⌘ `operator` ist eine neue Funktion.
-

package – Einladen eines Pakets neuer Bibliotheksfunktionen

`package(dirname)` lädt eine neue Bibliothek ein.

Aufruf(e):

- ⌘ `package(dirname <, Quiet> <, Forced>)`

Parameter:

`dirname` — ein zulässiger Verzeichnispfad: Eine Zeichenkette

Optionen:

- `Quiet` — unterdrückt die Bildschirmausgabe während des Ladevorgangs
- `Forced` — ermöglicht das Neuladen bereits geladener Bibliotheken

Rückgabewert: der Wert der letzten Anweisung in der Initialisierungsdatei `init.mu` des Pakets.

Seiteneffekte: Der Pfadname `dirname/lib` wird *vorn* in den Suchpfad `LIBPATH` eingefügt. Der Pfad `dirname/modules/OSName` wird *vorn* in den Suchpfad `READPATH` eingefügt (hierbei ist `OSName` der Name des Betriebssystems, siehe `sysname`). Damit werden Funktionen zuerst im Paket gesucht. Im Paket enthaltene Module werden automatisch gefunden. Im Falle eines Namenskonflikts wird die Paketfunktion statt der Systemfunktion gleichen Namens aufgerufen.

Verwandte Funktionen: `export`, `LIBPATH`, `loadmod`, `loadproc`, `newDomain`, `read`, `READPATH`

Details:

- ☞ In MuPAD sind Prozeduren nach mathematischen Anwendungsgebieten geordnet in diversen Bibliotheken installiert. Beispielsweise enthält die Bibliothek `numlib` Algorithmen der Zahlentheorie, die Bibliothek `numeric` enthält numerische Algorithmen. Auch ein Anwender sollte eigene Routinen inhaltlich geordnet als Bibliothekspakete installieren. Bei einer passenden Struktur des Verzeichnisses, in dem die Daten mit den MuPAD-Quelltexten installiert sind, kann die gesamte Bibliothek mit einem Aufruf von `package` eingeladen werden.
 - ☞ Technisch gesehen ist eine Bibliothek ein Domain. Die installierten Funktionen sind die „Slots“ und können mittels des „Slot-Operators“ `::` angesprochen werden, z. B., `numlib::fibonacci`, `numeric::int` etc.
 - ☞ Typischerweise soll ein neues Bibliotheks-Domain erzeugt und Funktionen sollen darin installiert werden, oder neue Funktionen sollen in ein bereits existierendes Bibliotheks-Domain aufgenommen werden. Das detaillierte Beispiel ?? ist dem ersten Fall gewidmet, Beispiel ?? beschreibt den zweiten Fall. Im letzteren Fall sollte der Anwender mit Vorsicht verfahren, um nicht bestehende Funktionalität einer existierenden Bibliothek zu überschreiben.
 - ☞ Der Ordner, nennen wir ihn `mypack`, der das Bibliothekspaket enthält, kann überall im Dateisystem installiert werden. Der Pfadname, der im `package`-Aufruf verwendet wird, kann entweder ein absoluter Pfadname von der Wurzel des Dateisystems zum Ordner `mypack` sein, oder auch ein relativer Pfad vom „aktuellen Arbeitsverzeichnis“ aus.
- Man beachte, dass das aktuelle Arbeitsverzeichnis auf den unterschiedlichen Betriebssystemen unterschiedlich sein kann. So ist es unter Windows dasjenige, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem die aktuelle MuPAD-Sitzung gestartet wurde.

- ☞ Das Paket muss die gleiche hierarchische Struktur haben wie die standardmäßig installierten MuPAD-Bibliotheken. Insbesondere muss das Paketverzeichnis ein Unterverzeichnis `lib` haben, welches die Quelldateien (oder Unterverzeichnisse mit Quelldateien) der neuen Funktionen enthält. Speziell muss im `lib`-Verzeichnis eine Initialisierungsdatei `init.mu` existieren.

Beispielsweise sollte auf einem UNIX- oder Linux-System das Verzeichnis `mypack` folgende Struktur besitzen (bis auf andere Trennzeichen in den Pfadnamen gilt dies auch für andere Betriebssysteme):

```
mypack/lib/init.mu
mypack/lib/LIBFILES/mylib.mu
mypack/lib/MYLIB/Zeug.mu
mypack/lib/MYLIB/...
mypack/lib/MYLIB/SUBDIR/mehrZeug.mu
mypack/lib/MYLIB/SUBDIR/...
```

Gewöhnlich wird die Initialisierungsdatei `init.mu` dazu benutzt, mittels der Funktion `loadproc` die neuen Objekte des Pakets (neue Bibliotheks-Domains und/oder neue Funktionen) zu definieren.

Soll ein neues Bibliotheks-Domain erzeugt werden, so sollte das `lib`-Verzeichnis ein Unterverzeichnis `LIBFILES` mit einer Datei `LIBFILES/mylib.mu` enthalten. Die `loadproc`-Befehle innerhalb von `init.mu` sollten auf die Datei `mylib.mu` verweisen, in der das neue Bibliotheks-Domain mittels `newDomain` zu erzeugen ist. Die Funktionen („Slots“) der neuen Bibliothek sind innerhalb von `mylib.mu` wiederum mittels `loadproc` zu definieren, wobei auf den Ort verwiesen wird, an dem sich die entsprechenden Dateien mit dem MuPAD-Quelltext innerhalb des Paketverzeichnisses befinden. Es wird empfohlen, die Quelldateien in Unterverzeichnissen `lib/MYLIB`, `lib/MYLIB/SUBDIR` etc. zu installieren.

Diese Struktur sowie der Lademechanismus mittels `loadproc` entspricht der Organisation der standardmäßig vorinstallierten MuPAD-Bibliotheken (statt `init.mu` wird die Initialisierungsdatei `MuPAD_ROOT_PATH/lib/sysinit.mu` verwendet).

- ☞ Soll durch das Laden des Pakets ein neues Bibliotheks-Domain, sagen wir `mylib`, erzeugt werden, so hat die Initialisierungsdatei `init.mu` auf die Datei `LIBFILES/mylib.mu` zu verweisen, in der das Bibliotheks-Domain dann wirklich erzeugt wird:

```
// ----- Datei mypack/lib/init.mu -----
// Laden des Bibliotheks-Domains 'mylib'
alias(path = pathname("LIBFILES")):
mylib := loadproc(mylib, path, "mylib"):
unalias(path):
stdlib::LIBRARIES := stdlib::LIBRARIES union {"mylib"}:
// Der Rueckgabewert des package-Aufrufs:
```



```

null():
// ----- Ende der Datei init.mu -----

```

Durch das Hinzufügen der neuen Bibliothek `mylib` zur Menge `std-lib::LIBRARIES` ruft ein `package`-Kommando beim Laden automatisch die `info`-Funktion auf, die Informationen über die Bibliothek ausgibt. Diese Informationen bestehen unter anderem aus dem in der Datei `LIBFILES/mylib.mu` zu definierende Eintrag `mylib::info` des neuen Bibliotheks-Domains.

Der Wert der letzten Anweisung in der Datei `init.mu` bestimmt den Rückgabewert eines `package`-Aufrufs. Typischerweise ist die letzte Anweisung `null()`, um unerwünschte Bildschirmausgaben beim Laden des Pakets zu vermeiden. Alternativ kann eine Information wie beispielsweise die Zeichenkette "Bibliothek 'mylib' erfolgreich geladen" zurückgeliefert werden.

Beispiel ?? liefert weitere Details.

☞ Die Datei `LIBFILES/mylib.mu` hat das neue Bibliotheks-Domain mittels `newDomain` zu erzeugen. Einige Standardeinträge wie z.B. `mylib::Name`, `mylib::info` sowie `mylib::interface` sollten definiert werden. Die Funktionen `mylib::function1` etc. der neuen Bibliothek sollten mittels `loadproc` auf die Dateien mit dem Quelltext verweisen:

```

// ---- Datei mypack/lib/LIBFILES/mylib.mu ----
// mylib -- eine Bibliothek mit meinen Funktionen
mylib := newDomain("mylib"):
mylib::Name := "mylib":
mylib::info := "Library 'mylib': a library with my functions":
mylib::interface := {hold(function1), hold(function2), ...}:
// Definiere die in ../MYLIB/function1.mu etc.
// implementierten Funktionen:
alias(path = pathname("MYLIB")):
mylib::function1 := loadproc(mylib::function1, path, "function1"):
mylib::function2 := loadproc(mylib::function2, path, "function2"):
...
unalias(path):
// Definiere die in ../MYLIB/SUBDIR/more1.mu etc.
// implementierten Funktionen:
alias(path = pathname("MYLIB", $SUBDIR)):
mylib::more1 := loadproc(mylib::more1, path, "more1"):
mylib::more2 := loadproc(mylib::more2, path, "more2"):
...
unalias(path):
null():
// ----- Ende der Datei mylib.mu -----

```

Das Beispiel ?? liefert weitere Informationen.

Option <Quiet>:

- ☞ Diese Option unterdrückt die Ausgabe von Information während des Ladens.
-

Option <Forced>:

- ☞ Ein Paket wird üblicherweise nur einmal eingelesen. Ein Versuch, das Paket erneut zu laden, führt zu einem Fehler. Mit dieser Option kann das Neuladen eines bereits geladenen Pakets erzwungen werden.
-

Beispiel 1. Im Folgenden wird beschrieben, wie ein Paket strukturiert sein sollte, welches ein neues Bibliotheks-Domain mit vom Nutzer definierten Funktionen erzeugt. In Beispiel ?? werden dieselben Funktionen geladen, aber statt in eine neue Bibliothek in eine der bereits existierenden Bibliotheken der Standardinstallation eingefügt.

Angenommen, man hat einige neue Funktionen auf den ganzen Zahlen wie beispielsweise eine Fakultätsfunktion oder eine neue Funktion zum Potenzieren implementiert. Dann bietet es sich an, all diese Funktionen in eine gemeinsame Bibliothek zu stellen. Die Bibliothek soll `numfuncs` heißen (Bibliothek für elementare Funktionen der Zahlentheorie) und sei als Paket im Verzeichnis `demoPack1` installiert. Das Verzeichnis hat die folgende Struktur:

```
demoPack1/lib/init.mu
demoPack1/lib/LIBFILES/numfuncs.mu
demoPack1/lib/NUMFUNCS/factorial.mu
demoPack1/lib/NUMFUNCS/russian.mu
```

Die Initialisierungsdatei `init.mu` kann wie folgt implementiert sein:

```
// ----- file demoPack1/lib/init.mu -----
// loads the library 'numfuncs'
alias(path = pathname("LIBFILES")):
numfuncs := loadproc(numfuncs, path, "numfuncs"):
stdlib::LIBRARIES := stdlib::LIBRARIES union {"numfuncs"}:
unalias(path):
// return value of package:
"library 'numfuncs' successfully loaded":
// ----- end of file init.mu -----
```

Die Funktion `pathname` wird benutzt, um die Pfadnamen so zu erzeugen, wie es für das benutzte Betriebssystem angebracht ist. Der `loadproc`-Aufruf bezieht sich auf die Datei `LIBFILES/numfuncs.mu`, in der das neue Bibliotheks-Domain tatsächlich erzeugt wird:

```
// --- file demoPack1/lib/LIBFILES/numfuncs.mu ---
// numfuncs -- the library for elementary number theory
numfuncs := newDomain("numfuncs"):
numfuncs::Name := "numfuncs":
numfuncs::info := "Library 'numfuncs': the library of ".
                  "functions for elementary number theory":
numfuncs::interface := {hold(factorial), hold(russianPower)}:
// define the functions implemented in ../NUMFUNCS/factorial.mu etc:
alias(path = pathname("NUMFUNCS")):
numfuncs::factorial :=
    loadproc(numfuncs::factorial, path, "factorial"):
numfuncs::odd :=
    loadproc(numfuncs::odd, path, "russian"):
numfuncs::russianPower :=
    loadproc(numfuncs::russianPower, path, "russian"):
unalias(path):
null():
// ----- end of file numfuncs.mu -----
```

Hier wird die Bibliothek mittels `newDomain` erzeugt. Jede Bibliothek sollte die Einträge `Name` und `info` besitzen. Zusätzlich kann man den `interface`-Eintrag definieren, der alle Methoden enthält, die ein Benutzer kennen und aufrufen soll.

Die obige Datei enthält darüberhinaus die Definitionen der Funktionen `factorial`, `odd` und `russianPower`, welche in dem Unterverzeichnis „`demoPack1/lib/NUMFUNCS`“ implementiert sind (die Details dieser Implementierungen werden im Beispiel ?? erläutert; man braucht dort nur `numlib` durch `numfuncs` zu ersetzen).

Der Quelltext der Funktion `numfuncs::factorial` ist in einer einzelnen Datei installiert. Der Quelltext der Funktionen `numfuncs::odd` und `numfuncs::russianPower` befindet sich in einer gemeinsamen Datei `russian.mu`.

`numfuncs::odd` ist eine interne Hilfsfunktion und nicht dafür gedacht, von Benutzern aufgerufen zu werden. Daher wird sie nicht im `interface`-Eintrag der `numfuncs`-Bibliothek aufgenommen.

Wir demonstrieren nun das Einladen der neuen Bibliothek. Angenommen, es existieren mehrere Bibliothekspakete, die in einem Ordner namens `myMuPADFolder` installiert sind:

```
/home/myLoginName/myMuPADFolder/demoPack1
/home/myLoginName/myMuPADFolder/demoPack2
...
```

Die im Verzeichnis `demoPack1` installierte Bibliothek `numfuncs` wird nun durch einen Aufruf von `package` eingeladen:

```
>> package("/home/myLoginName/myMuPADFolder/demoPack1")
```

```
Library 'numfuncs': the library of functions for elementary \
number theory
```

```
-- Interface:
numfuncs::factorial, numfuncs::russianPower
```

```
"library 'numfuncs' successfully loaded"
```

In der Initialisierungsdatei `init.mu` war die neue Bibliothek zu `stdlib::LIBRARIES` hinzugefügt worden. Daher wird beim Laden die obige Information über die Bibliothek ausgegeben. Dies gilt nur für den ersten Ladevorgang. Standardmäßig kann ein Bibliothekspaket nur einmal geladen werden:

```
>> package("/home/myLoginName/myMuPADFolder/demoPack1")

Warning: Package already defined. For redefinition use option \
Forced [package]
```

Dem Text der Warnung folgend kann man die existierende Bibliothek `numfuncs` mit der Option *Forced* überschreiben:

```
>> package("/home/myLoginName/myMuPADFolder/demoPack1",
           Forced)

Warning: Package redefined [package]

"library 'numfuncs' successfully loaded"
```

Die neue Bibliothek `numfuncs` ist nun vollständig in das System integriert. Die `numfuncs`-Funktionen können wie die Funktionen anderer MuPAD-Bibliotheken aufgerufen werden :

```
>> numfuncs::factorial(41)

33452526613163807108170062053440751665152000000000

>> numfuncs::russianPower(123, 12)

11991163848716906297072721
```

Beispiel 2. Hier wird demonstriert, wie ein Paket zu strukturieren ist, das neue Funktionen in ein bereits existierendes Bibliotheks-Domain einfügen soll.

Wir betrachten dieselbe Funktion wie im Beispiel ???. Im Gegensatz zu Beispiel ?? soll hier jedoch kein neues Bibliotheks-Domain eingeführt werden, stattdessen soll die bestehende MuPAD-Bibliothek `numlib` erweitert werden. Speziell sollen die neuen Funktionen `numlib::factorial` und `numlib::russianPower` durch das Paket installiert werden. Vor dem Laden

dieser Funktion sollten wir sicherstellen, dass hierdurch keine existierenden Funktion der numlib-Standardinstallation überschrieben werden. Als einfacher Test, dass die Standardinstallation keine Funktion `numlib::factorial` zur Verfügung stellt, kann einfach versucht werden, diese Funktion aufzurufen:

```
>> numlib::factorial
```

FAIL

Diese Funktion existiert damit in der Tat nicht und soll nun als Erweiterung der numlib-Bibliothek zur Verfügung gestellt werden. Neue Funktionen sind im Ordner `demoPack2` installiert:

```
demoPack2/lib/init.mu
demoPack2/lib/NUMLIB/factorial.mu
demoPack2/lib/NUMLIB/russian.mu
```

Es soll kein neues Bibliotheks-Domain eingeführt werden. Dementsprechend braucht man im Gegensatz zu Beispiel ?? keine Definitionsdatei `demoPack2/lib/LIBFILES/numlib` (Genauer: eine solche Datei mit der Definition des Domains `numlib` würde die bestehende Definition von `numlib` überschreiben und damit bestehende Funktionalität zerstören.) Die neuen Funktionen können stattdessen wie folgt direkt in der Initialisierungsdatei `init.mu` deklariert werden:

```
// ----- file demoPack2/lib/init.mu -----
// loads additional functions for the existing library 'numlib'
numlib::interface := numlib::interface
    union {hold(factorial), hold(russianPower)}:
// define the functions implemented in ../NUMLIB/factorial.mu etc:
alias(path = pathname ("NUMLIB")):
numlib::factorial :=
    loadproc(numlib::factorial, path, "factorial"):
numlib::odd :=
    loadproc(numlib::odd, path, "russian"):
numlib::russianPower :=
    loadproc(numlib::russianPower, path, "russian"):
unalias(path):
// return value of package:
"new numlib functions successfully loaded":
// ----- end of file init.mu -----
```

Analog zu Beispiel ?? wurden die wichtigsten Funktionen zum schon existierenden `interface`-Eintrag der numlib-Bibliothek hinzugefügt.

Nun wird auf die Implementierungsdateien `factorial.mu` und `russian.mu` mit dem Quelltext der Funktionen eingegangen:

```
// ---- file demoPack2/lib/NUMLIB/factorial.mu ----
numlib::factorial :=
  proc(n : Type::NonNegInt) : Type::PosInt
    // factorial(n) computes n!
  begin
    if n = 0 then 1
    else n*numlib::factorial(n - 1)
    end_if
  end_proc:
// ----- end of file factorial.mu -----
```

Die Funktion `numlib::odd` ist eine Hilfsfunktion, die nur von `numlib::russianPower` verwendet wird. Beide Funktionen sind deshalb in derselben Datei implementiert:

```
// ---- file demoPack2/lib/NUMLIB/russian.mu ----
numlib::odd := m -> not(iszero(m mod 2)):

numlib::russianPower :=
  proc(m : DOM_INT, n : Type::NonNegInt) : DOM_INT
    // computes the n-th power of m using the
    // russian peasant method of multiplication
    local d;
  begin
    d := 1;
    while n>0 do
      if numlib::odd(n) then
        d := d*m;
        n := n - 1;
      else
        m := m*m;
        n := n div 2;
      end_if
    end_while;
    d
  end_proc:
// ----- end of file russian.mu -----
```

Wir demonstrieren nun das Einladen der neuen Funktionen. Angenommen, es existieren mehrere Bibliothekspakete, die in einem Ordner namens `myMuPADFolder` installiert sind:

```
/home/myLoginName/myMuPADFolder/demoPack1
/home/myLoginName/myMuPADFolder/demoPack2
...
```

Die in `demoPack2` installierten Funktionen werden durch einen `package-`Aufruf eingeladen:

```
>> package("/home/myLoginName/myMuPADFolder/demoPack2")

      "new numlib functions successfully loaded"
```

Die zum interface-Eintrag von numlib hinzugefügten neuen Funktionen werden durch info aufgelistet:

```
>> info(numlib)

Library 'numlib': the package for elementary number theory

-- Interface:
numlib::Lambda,          numlib::Omega,
...
numlib::factorial,      numlib::fibonacci,
...
numlib::proveprime,     numlib::russianPower,
...
```

Die neuen Funktionen sind nun voll in die Bibliothek integriert und können wie alle anderen Funktionen der MuPAD-Bibliothek aufgerufen werden :

```
>> numlib::factorial(41)

      33452526613163807108170062053440751665152000000000

>> numlib::russianPower(123, 12)

      11991163848716906297072721
```

Änderungen:

⌘ package ist eine neue Funktion.

pade – Pade-Approximation

pade(f, ..) berechnet eine Pade-Approximation des Ausdrucks f.

Aufruf(e):

⌘ pade(f, x <, [m, n]>)
 ⌘ pade(f, x = x0 <, [m, n]>)

Parameter:

- f — ein arithmetischer Ausdruck oder eine mittels `series` erzeugte Reihe vom Domain-Typ `Series::Puisseux`
- x — ein Bezeichner
- x_0 — ein arithmetischer Ausdruck. Falls x_0 nicht spezifiziert wird, so wird $x_0 = 0$ benutzt.

Optionen:

- $[m, n]$ — eine Liste mit nicht-negativen ganzen Zahlen, die die Approximationsordnung festlegt. Die Voreinstellung ist $[3, 3]$.

Rückgabewert: ein arithmetischer Ausdruck oder `FAIL`.

Verwandte Funktionen: `series`

Details:

- ☞ Die Pade-Approximation der Ordnung $[m, n]$ um $x = x_0$ ist ein rationaler Ausdruck

$$(x - x_0)^p \frac{a_0 + a_1(x - x_0) + \cdots + a_m(x - x_0)^m}{1 + b_1(x - x_0) + \cdots + b_n(x - x_0)^n}.$$

Die Parameter p und a_0 sind durch den führenden Term $f = a_0(x - x_0)^p + O((x - x_0)^{p+1})$ der Reihenentwicklung von f um $x = x_0$ bestimmt. Die Parameter a_1, \dots, b_n werden so gewählt, dass die Reihenentwicklung der Pade-Approximation mit der Reihenentwicklung von f bis zur maximal möglichen Ordnung übereinstimmt.

- ☞ `FAIL` wird zurückgeliefert, falls keine Reihenentwicklung von f bestimmt werden kann. Weiterhin muss die von `series` berechnete Entwicklung eine Taylor- oder Laurent-Reihe sein, d. h., es muss eine Entwicklung nach ganzzahligen Potenzen von $x - x_0$ existieren.

Beispiel 1. Die Pade-Approximation ist eine rationale Approximation einer Reihen-Entwicklung:

```
>> f := cos(x)/(1 + x): P := pade(f, x, [2, 2])
```

$$\frac{x^2 - 7x^2 + 12}{14x + x^2 + 12}$$

In den meisten Fällen stimmen bei Ausdrücken mit der führenden Ordnung 0 die Reihenentwicklung der Pade-Approximation und die Reihenentwicklung des Ausdrucks bis zur Ordnung $m + n$ überein:


```
>> S := series(f, x, 6)
```

$$1 - x + \frac{x^2}{2} - \frac{x^3}{2} + \frac{13x^4}{24} - \frac{13x^5}{24} + O(x^6)$$

Dies unterscheidet sich in der Ordnung 5 von der Entwicklung der Pade-Approximation:

```
>> series(P, x, 6)
```

$$1 - x + \frac{x^2}{2} - \frac{x^3}{2} + \frac{13x^4}{24} - \frac{85x^5}{144} + O(x^6)$$

Die Reihenentwicklung kann direkt als Eingabe von pade verwendet werden:

```
>> pade(S, x, [2, 3]), pade(S, x, [3, 2])
```

$$\frac{12 - 5x^2}{12x^2 + x^3 + x^2 + 12}, \frac{12x^2 + 7x^2 - 7x^3 - 12}{13x^2 - 12}$$

Beide Pade-Approximationen nähern f bis einschließlich der Ordnung $m + n = 5$ an:

```
>> map( [%], series, x)
```

```
--          2      3      4      5
|          x      x      13 x      13 x      6
|  1 - x + -- - -- + ----- - ----- + O(x ),
--          2      2      24      24

          2      3      4      5      --
          x      x      13 x      13 x      6      |
1 - x + -- - -- + ----- - ----- + O(x )      |
          2      2      24      24      --
```

```
>> delete f, P, S:
```

Beispiel 2. Der folgende Ausdruck hat keine Laurent-Entwicklung um $x = 0$:

```
>> series(x^(1/3)/(1 - x), x)
```


Rückgabewert: ein arithmetischer Ausdruck.

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `collect`, `denom`, `divide`, `expand`, `factor`, `normal`, `numer`, `rectform`, `rewrite`, `simplify`

Details:

- ☞ Man betrachte den rationalen Ausdruck $f(x) = g(x) + p(x)/q(x)$ mit Polynomen g, p, q , die $\text{degree}(p) < \text{degree}(q)$ erfüllen. Hierbei ist $q = \text{denom}(f)$ der Nenner von f und die durch $(g, p) = \text{divide}(\text{numer}(f), q, [x])$ gegebenen Polynome sind der Quotient bzw. der Rest der Polynomdivision des Zählers von f durch den Nenner q . Sei

$$q(x) = q_1(x)^{e_1} \cdot q_2(x)^{e_2} \cdot \dots$$

eine Faktorisierung des Nenners in nicht konstante und paarweise teilerfremde Polynome q_i mit ganzzahligen Exponenten e_i . Die auf dieser Faktorisierung beruhende Partialbruchzerlegung ist eine Darstellung

$$f(x) = g(x) + \frac{p_{11}(x)}{q_1(x)} + \dots + \frac{p_{1e_1}(x)}{q_1(x)^{e_1}} + \frac{p_{21}(x)}{q_2(x)} + \dots + \frac{p_{2e_2}(x)}{q_2(x)^{e_2}} + \dots$$

mit Polynomen p_{ij} , die $\text{degree}(p_{ij}) < \text{degree}(q_i)$ erfüllen. Insbesondere sind die Polynome p_{ij} konstant, wenn q_i ein lineares Polynom ist.

`partfrac` benutzt die von der Funktion `factor` gefundenen Faktoren q_i von $q = \text{denom}(f)$. Die Faktorisierung wird dabei über dem von den Koeffizienten des Nennerpolynoms implizierten Körper berechnet (siehe `factor` für Details). Siehe Beispiel ??.

- ☞ Das zweite Argument x in einem Aufruf von `partfrac` kann weggelassen werden, wenn f nur eine Unbestimmte enthält.
- ☞ Eine Partialbruchzerlegung kann auch von Ausdrücken berechnet werden, die rational bezüglich eines symbolischen Funktionsaufrufs sind. Dieser Funktionsaufruf muss als Unbestimmte übergeben werden. Siehe Beispiel ??.
-

Beispiel 1. In den folgenden Aufrufen braucht keine Unbestimmte angegeben zu werden, da die rationalen Ausdrücke univariat sind:

```
>> partfrac(x^2/(x^3 - 3*x + 2))
```

$$\frac{5}{9(x-1)} + \frac{1}{3(x-1)^2} + \frac{4}{9(x+2)}$$

```
>> partfrac(23 + (x^4 + x^3)/(x^3 - 3*x + 2))
```

$$x + \frac{19}{9(x-1)} + \frac{2}{3(x-1)^2} + \frac{8}{9(x+2)} + \frac{24}{9(x+2)^2}$$

Der folgende Ausdruck enthält zwei Unbestimmte x und y . Es muss angegeben werden, bezüglich welcher Variable die Partialbruchzerlegung berechnet werden soll:

```
>> f := x^2/(x^2 - y^2): partfrac(f, x), partfrac(f, y)
```

$$1 - \frac{y}{2(y-x)} - \frac{y}{2(x+y)}, \quad \frac{x}{2(x+y)} - \frac{x}{2(y-x)}$$

```
>> delete f:
```

Beispiel 2. Im Folgenden wird auf die Abhängigkeit der Partialbruchzerlegung von den Möglichkeiten der Faktorisierung mittels `factor` eingegangen:

```
>> partfrac(1/(x^2 + 2), x)
```

$$\frac{1}{x^2 + 2}$$

Man beachte, dass der Nenner $x^2 + 2$ über den rationalen Zahlen nicht faktorisiert:

```
>> factor(x^2 + 2)
```

$$x^2 + 2$$

Dieser Nenner faktorisiert jedoch über einer Körpererweiterung, die $\sqrt{-2}$ enthält. Dieser erweiterte Koeffizientenkörper wird in den folgenden Aufrufen implizit von `factor` und dementsprechend auf von `partfrac` angenommen:

```
>> factor(sqrt(-2)*x^2 + 2*sqrt(-2))
```

$$(I\sqrt{2})^{1/2} (x - I\sqrt{2})^{1/2} (x + I\sqrt{2})^{1/2}$$

```
>> partfrac(x/(sqrt(-2)*x^2 + 2*sqrt(-2)), x)
```

$$- \frac{1}{2} \frac{1}{\sqrt{2} (x - I\sqrt{2})^{1/2}} + \frac{1}{2} \frac{1}{\sqrt{2} (x + I\sqrt{2})^{1/2}}$$

Beispiel 3. Rationale Ausdrücke über symbolischen Funktionsaufrufen können ebenfalls in eine Partialbruchdarstellung zerlegt werden:

```
>> partfrac(1/(sin(x)^4 - sin(x)^2 + sin(x) - 1), sin(x))
```

$$\frac{1}{3(\sin(x) - 1)} + \frac{-\frac{2\sin(x)}{3} - \frac{\sin^2(x)}{3} - \frac{2}{3}}{\sin^2(x) + \sin(x) + 1}$$

Änderungen:

☞ Keine Änderungen.

patchlevel – die „Patch“-Nummer der installierten MuPAD-Bibliothek

`patchlevel()` gibt die Patch-Nummer der aktuell benutzten MuPAD-Bibliothek zurück.

Aufruf(e):

☞ `patchlevel()`

Rückgabewert: eine nichtnegative ganze Zahl.

Verwandte Funktionen: `Pref::kernel, version`

Details:

- ☞ `patchlevel` beschreibt zusammen mit der Funktion `version` die aktuell installierte MuPAD-Bibliothek. Zu einem MuPAD-Release werden gelegentlich „Patches“ (Bug-Fixes) von Sciface Software oder der MuPAD-Gruppe zur Verfügung gestellt, die nachträglich in eine bestehende MuPAD-Installation integriert werden können. Jeder neu eingespielte Patch erhöht die von `patchlevel` gelieferte Patch-Nummer um 1. Die aktuell verwendete Bibliothek ist durch ihre Versionsnummer (`version`) zusammen mit ihrer Patch-Nummer eindeutig bestimmt.
- ☞ `patchlevel` bezieht sich auf Patches der Bibliotheken und ist unabhängig von der Versionsnummer des MuPAD-Kerns (`Pref::kernel`).

☞ Jede neue MuPAD-Version erscheint mit der Patch-Nummer 0.

☞ Für Informationen über neue Patches besuchen Sie bitte unsere Website www.mupad.de.

Beispiel 1. Um den Stand der aktuell installierten MuPAD-Bibliothek zu erfragen, muss man nach ihrer Versionsnummer

```
>> version()
```

```
[2, 0, 0]
```

sowie nach ihrer Patch-Nummer fragen:

```
>> patchlevel()
```

```
0
```

Ist die zurückgegebene Patch-Nummer größer als 0, wurde ein Patch zur installierten Bibliothek eingespielt.

Änderungen:

☞ Keine Änderungen.

pathname – Erzeugen eines systemabhängigen Pfadnamens

pathname(dir, subdir, ...) erzeugt einen auf dem benutzten Betriebssystem gültigen relativen Pfadnamen.

Aufruf(e):

☞ pathname(dir, subdir, ..)

☞ pathname(*Root*, dir, subdir, ..)

Parameter:

dir, subdir, .. — Verzeichnisnamen: Zeichenketten

Optionen:

Root — veranlasst pathname, einen absoluten Pfadnamen zu erzeugen

Rückgabewert: eine Zeichenkette.

Verwandte Funktionen: `fclose`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `LIBPATH`, `loadproc`, `package`, `print`, `protocol`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

- ☞ `pathname` wird dazu genutzt, Pfadnamen mittels MuPAD-Zeichenketten zu erzeugen. Pfadnamen sind systemabhängig: Speziell werden auf verschiedenen Betriebssystemen unterschiedliche Trennzeichen zwischen Verzeichnissen und Unterverzeichnissen benutzt. Die in einem `pathname`-Aufruf angegebenen Namen werden automatisch mit dem auf dem aktuell benutzten Betriebssystem zulässigen Trennzeichen verknüpft und zu einem zulässigen Pfadnamen zusammengesetzt. Hiermit kann beispielsweise die Lage von Bibliotheksdateien unabhängig vom Betriebssystem angegeben werden.
- ☞ Neben den vom Betriebssystem abhängigen Konventionen für Verzeichnisnamen muss beachtet werden, dass die Namen keines der Zeichen `„/“`, `„\“` oder `„:“` enthalten. Die Übereinstimmung mit diesen Einschränkungen wird von `pathname` überprüft.
- ☞ Unter Windows ist es nicht möglich, mit `pathname` einen Laufwerksnamen anzugeben. Namen beziehen sich immer auf das gegenwärtige Laufwerk.
- ☞ Beispiele:

Aufruf	Ergebnis	Plattform
<code>pathname("lib", "linalg")</code>	<code>"lib/linalg/"</code> <code>"lib\\linalg\\"</code> <code>"lib:linalg:"</code>	UNIX/Linux Windows MacOS
<code>pathname(Root, "lib", "linalg")</code>	<code>"/lib/linalg/"</code> <code>"\\lib\\linalg\\"</code> <code>"lib:linalg:"</code>	UNIX/Linux Windows MacOS

Beispiel 1. Die folgenden Beispiele sind auf einem UNIX/Linux-System erzeugt:

```
>> pathname("lib", "linalg")
      "lib/linalg/"
>> pathname(Root, "lib", "linalg") . "det.mu"
      "/lib/linalg/det.mu"
```

Änderungen:

☞ Keine Änderungen.

pdivide – Pseudo-Division von Polynomen

`pdivide(p, q)` berechnet die Pseudo-Division der univariaten Polynome `p` und `q`.

Aufruf(e):

☞ `pdivide(p, q <, mode>)`
☞ `pdivide(f, g <, [x]> <, mode>)`

Parameter:

`p, q` — univariate Polynome vom Typ `DOM_POLY`.
`f, g` — arithmetische Ausdrücke
`x` — ein Bezeichner oder ein indizierter Bezeichner. Multivariate Ausdrücke werden als univariate Polynome in der Unbestimmten `x` aufgefasst.

Optionen:

`mode` — entweder *Quo* or *Rem*. Mit *Quo* wird nur der Pseudo-Quotient geliefert, mit *Rem* nur der Pseudo-Rest.

Rückgabewert: ein Polynom, oder ein polynomialer Ausdruck, oder eine Folge aus einem Element des Koeffizientenrings der Eingabepolynome sowie zwei Polynomen/polynomialen Ausdrücken oder der Wert `FAIL`.

Überladbar durch: `p, q, f, g`

Verwandte Funktionen: `content, degree, divide, factor, gcd, gcdex, ground, lcoeff, multcoeffs, poly`

Details:

☞ `pdivide(p, q)` berechnet die Pseudo-Division der univariaten Polynome `p` und `q`. Es wird die Folge `b, s, r` zurückgeliefert. Hierbei ist $b = \text{lcoeff}(q)^{(\text{degree}(p) - \text{degree}(q) + 1)}$ ein Element des Koeffizientenrings. Die Polynome `s` (der Pseudo-Quotient) und `r` (der Pseudo-Rest) erfüllen $bp = sq + r$, $\text{degree}(p) = \text{degree}(s) + \text{degree}(q)$, $\text{degree}(r) < \text{degree}(q)$.

☞ Die ersten beiden Argumente können Polynome oder arithmetische Ausdrücke sein.

Polynome müssen vom gleichen Typ sein, d. h., ihre Unbestimmten und ihre Koeffizientenringe müssen übereinstimmen.

Ausdrücke werden intern in Polynome konvertiert (siehe die Funktion `poly`). Wird keine Unbestimmte `x` angegeben, so werden alle symbolischen Variablen in den Ausdrücken als Unbestimmte angesehen. `FAIL` wird zurückgeliefert, falls dabei mehr als eine Unbestimmte gefunden wird. `FAIL` wird auch zurückgeliefert, wenn die Ausdrücke nicht in Polynome umgewandelt werden können.

Die zurückgelieferten Polynome sind vom gleichen Typ wie die ersten beiden Argumente, d. h., entweder Polynome vom Typ `DOM_POLY` oder polynomiale Ausdrücke.

☞ Im Gegensatz zu `divide` braucht bei `pdivide` der Koeffizientenring der Polynome keine `"_divide"`-Methode zu implementieren: in diesem Algorithmus werden die Koeffizienten nicht dividiert.

☞ `pdivide` ist eine Funktion des Systemkerns.

Beispiel 1. Dieses Beispiel zeigt das Ergebnis der Pseudo-Division zweier Polynome:

```
>> p:= poly(x^3 + x + 1):  q:= poly(3*x^2 + x + 1):  
    [b, s, r] := [pdivide(p, q)]  
  
    [9, poly(3 x - 1, [x]), poly(7 x + 10, [x])]
```

Das Ergebnis erfüllt die folgende Gleichung:

```
>> multcoeffs(p, b) = s*q + r  
  
          3                      3  
poly(9 x  + 9 x + 9, [x]) = poly(9 x  + 9 x + 9, [x])
```

Pseudo-Quotienten und -Reste können auch separat berechnet werden:

```
>> pdivide(p, q, Quo), pdivide(p, q, Rem)  
  
    poly(3 x - 1, [x]), poly(7 x + 10, [x])  
  
>> delete p, q, b, s, r:
```

Beispiel 2. Der Koeffizientenring kann ein beliebiger Ring sein, z. B., der Restklassenring modulo 8:

```
>> pdivide(poly(x^3 + x + 1, IntMod(8)),
            poly(3*x^2 + x + 1, IntMod(8)))

1, poly(3 x - 1, [x], IntMod(8)), poly(- x + 2, [x], IntMod(8))
```

Beispiel 3. Hier besteht die Eingabe aus multivariaten polynomialen Ausdrücken, die als univariate Polynome in x aufgefasst werden:

```
>> pdivide(x^3 + x + y, a*x^2 + x + 1, [x])

      2      2
a , a x - 1, a y + x (a (a - 1) + 1) + 1
```

Beispiel 4. Das erste Argument kann nicht in ein Polynom konvertiert werden. Der Rückgabewert ist FAIL:

```
>> pdivide(1/x, x)

FAIL
```

Änderungen:

☞ Keine Änderungen.

piecewise – das Domain der durch Fallunterscheidung definierten Objekte

`piecewise([condition1, object1], [condition2, object2], ...)`
 ist das per Fallunterscheidung definierte Objekt, das gleich `object1` ist, falls Bedingung `condition1` erfüllt ist, gleich `object2`, falls Bedingung `condition2` erfüllt ist, usw.

Elementkonstruktor(en):

☞ `piecewise([condition1, object1], [condition2, object2], ...)`

Parameter:

<code>condition1, condition2, ...</code>	— boolsche Konstanten oder Ausdrücke, die logische Formeln repräsentieren
<code>object1, object2, ...</code>	— beliebige Objekte

Seiteneffekte: Mittels `assume` gesetzte Eigenschaften von Bezeichnern werden berücksichtigt.

Verwandte Funktionen: `_case`, `_if`, `assume`, `bool`, `is`

Details:

- ☞ `piecewise` unterscheidet sich in zweierlei Hinsicht von Fallunterscheidungen mittels `if` oder `case`. Erstens wird der `property`-Mechanismus benutzt, um den Wahrheitswert der einzelnen Bedingungen zu ermitteln; das Ergebnis hängt daher von den Eigenschaften der Bezeichner ab, die in den Bedingungen vorkommen. Zweitens bewertet `piecewise` den mathematischen Wahrheitsgehalt einer Bedingungen, während `if` und `case` syntaktisch arbeiten. Siehe Beispiel ??.
- ☞ Wir bezeichnen eine zweielementige Liste `[condition, object]` im Folgenden als *Zweig*. Ist die Bedingung `condition` nachweislich falsch, so wird der Zweig weggelassen; ist sie nachweislich wahr, so liefert der Aufruf von `piecewise` das Objekt `object` zurück. Kann keine der Bedingungen als wahr nachgewiesen werden, so wird ein Objekt vom Typ `piecewise` erzeugt, das alle nicht weggelassenen Zweige enthält.
Sind alle Bedingungen beweisbar falsch oder ist kein Zweig angegeben, so liefert `piecewise` das Ergebnis `undefined`. Siehe Beispiel ??.
- ☞ Die Bedingungen brauchen weder alle möglichen Fälle abzudecken, noch müssen sie sich gegenseitig ausschließen. Nach Ersetzung der Parameter durch Konstanten dürfen alle Bedingungen falsch werden, aber es darf auch passieren, dass mehr als eine Bedingung wahr wird.
- ☞ Sind mehrere Bedingungen gleichzeitig wahr, so liefert `piecewise` dasjenige Objekt, das unter der ersten als wahr *erkannten* Bedingung definiert ist. Die Anwenderin muss sicherstellen, dass alle zu den wahren Bedingungen gehörenden Objekte dieselbe mathematische Bedeutung haben. Man kann sich nicht darauf verlassen, dass stets die in der Reihenfolge erste mathematisch wahre Bedingung vom System als wahr erkannt wird.
- ☞ Immer wenn ein Objekt vom Typ `piecewise` ausgewertet wird, so wird der Wahrheitswert der Bedingungen unter Berücksichtigung der gegenwärtigen Werte und Eigenschaften der darin auftretenden Bezeichner

neu ermittelt. Dies kann man benutzen, um das Ergebnis einer Berechnung nacheinander unter verschiedenen Annahmen weiter zu vereinfachen.

☞ Fallunterscheidungen können geschachtelt werden: sowohl Bedingungen als auch Objekte in einer Fallunterscheidung können selbst per Fallunterscheidung definiert sein. `piecewise` „entschachtelt“ solche Fallunterscheidungen automatisch. Beispielsweise wird eine Fallunterscheidung des Typs „falls A, dann (falls B, dann C)“ zu „falls A und B, dann C“. Siehe Beispiel ??.

☞ Für per Fallunterscheidung definierte Objekte sind dieselben arithmetischen und Mengenoperationen definiert wie für die in den Zweigen auftretenden Objekte. Ist f eine solche Operation und sind p_1, p_2, \dots per Fallunterscheidung definierte Objekte, dann ist $f(p_1, p_2, \dots)$ dasjenige per Fallunterscheidung definierte Objekt, das aus allen Zweigen der Form $[condition1 \text{ and } condition2 \text{ and } \dots, f(object1, object2, \dots)]$ besteht, wobei $[condition1, object1]$ ein Zweig von p_1 ist, $[condition2, object2]$ ein Zweig von p_2 , usw. Anders gesagt, die Anwendung von f vertauscht mit jeder Zuweisung an die freien Parameter in den Bedingungen.

Durch Fallunterscheidung definierte Objekte dürfen bei solchen Operationen auch mit anderen Objekten gemischt werden: Ist beispielsweise p_1 nicht durch Fallunterscheidung definiert, dann wird es wie eine triviale Fallunterscheidung mit genau einem Zweig $[TRUE, p_1]$ behandelt.

Siehe Beispiele ?? und ??.

☞ Insbesondere gilt die vorige Anmerkung für unäre Operatoren und Funktionen mit einem Argument: werden sie mit einem per Fallunterscheidung definierten Objekt als Argument aufgerufen, so werden sie auf die Objekte in jedem einzelnen Zweig angewandt. Siehe Beispiel ??.

Missing file stdlib.met

Mathematische Methoden

Methode `_in`: Elementbeziehung mit `piecewise` auf der linken Seite

```
_in(piecewise p, set S)
```

☞ Diese Methode liefert eine Aussageform, die gleichbedeutend damit ist, dass p ein Element von S ist.

☞ Diese Methode überlädt `_in`.

Methode **contains**: Anwenden der Funktion **contains** auf die Objekte in allen Zweigen

`contains(piecewise p, any a)`

- ⌘ Diese Methode wendet die Funktion `contains` mit zweitem Argument `a` auf die Objekte in allen Zweigen von `p` an. Das Ergebnis ist im allgemeinen wieder ein fallweise definiertes Objekt.
- ⌘ Diese Methode überlädt die Funktion `contains`. Die Objekte in allen Zweigen müssen gültige erste Argumente für `contains` sein.

Methode **diff**: (partielle) Ableitung

`diff(piecewise p <, identifizier x, ...>)`

- ⌘ Diese Methode berechnet die partielle Ableitung der Objekte in allen Zweigen von `p` nach den angegebenen Variablen, wobei mit der am weitesten links stehenden Variable begonnen wird.
- ⌘ Sind keine Variablen angegeben, so wird `p` selbst zurückgeliefert.
- ⌘ Diese Methode überlädt `diff`.

Methode **discont**: Unstetigkeitsstellen einer abschnittsweise definierten Funktion

`discont(piecewise p, identifizier x <, domain F>)`

- ⌘ Diese Methode liefert eine Obermenge aller Unstetigkeitsstellen von `p` als Funktion von `x`, nämlich die Vereinigung aller Ergebnisse, die man durch Anwendung von `discont` auf die Objekte in allen Zweigen erhält, und zusätzlich die Randpunkte der Bedingungen von `p` bezüglich `x`.
- ⌘ Die Objekte in allen Zweigen von `p` müssen arithmetische Ausdrücke sein.
- ⌘ Diese Methode überlädt `discont`.
- ⌘ Die Bedeutung des optionalen dritten Parameters ist die gleiche wie bei der Funktion `discont`.

Methode **piecewise::disregardPoints**: Heuristik zur Vereinfachung von Bedingungen

`piecewise::disregardPoints(piecewise p)`

- ⌘ Auf Bedingungen, die weder als wahr noch als falsch erkannt wurden, wird folgende Heuristik angewandt: Gleichungen gelten immer als falsch. Da die Menge aller Nullstellen einer Gleichung meist

vom Lebesgue-Maß 0 ist, ergibt diese Methode in der Regel eine einfachere Fallunterscheidung, die für fast alle Parameterwerte zur ursprünglichen gleichwertig ist. Siehe Beispiel ??.

Methode `expand`: Anwenden der Funktion `expand` auf die Objekte in allen Zweigen

```
expand(piecewise p)
```

☞ Diese Methode überlädt `expand`.

Methode `piecewise::getElement`: liefere irgendein Element einer fallweise definierten Menge

```
piecewise::getElement(piecewise p)
```

☞ Diese Methode liefert ein Element, das in den Objekten aller Zweige von `p` enthalten ist. Alle solchen Objekte müssen Mengen repräsentieren.

☞ Das Ergebnis ist `FAIL`, falls kein solches Element gefunden werden kann.

☞ Diese Methode überlädt die Funktion `solveLib::getElement`.

Methode `has`: Test auf Vorhandensein eines Teilobjekts

```
has(piecewise p, any a)
```

☞ Diese Methode testet, ob `a` syntaktisch in einer Bedingung oder einem Objekt von `p` vorkommt. Sie liefert `TRUE`, wenn dies der Fall ist, und andernfalls `FALSE`.

☞ Diese Methode überlädt `has`.

Methode `int`: Berechnung bestimmter und unbestimmter Integrale einer abschnittsweise definierten Funktion

```
int(piecewise p, identifizier x <, range r>)
```

☞ Wird kein Bereich angegeben, so berechnet diese Methode das unbestimmte Integral von `p`, wobei `p` als abschnittsweise definierte Funktion von `x` betrachtet wird. Sie wendet die Funktion `int` auf die Objekte in allen Zweigen von `p` an.

☞ Wird ein Integrationsbereich `a..b` angegeben, so berechnet diese Methode das bestimmte Integral von `p`, wenn `x` den angegebenen Bereich durchläuft.

☞ Diese Methode überlädt `int`.

Methode `piecewise::isFinite`: Überprüfung einer durch Fallunterscheidung definierten Menge auf Endlichkeit

```
piecewise::isFinite(piecewise p)
```

- ⌘ Diese Methode liefert TRUE, falls die Objekte in allen Zweigen von *p* endliche Mengen sind, FALSE, falls die Objekte in allen Zweigen von *p* unendliche Mengen sind, und andernfalls UNKNOWN.
- ⌘ Diese Methode überlädt `solveLib::isFinite`.

Methode `normal`: Anwenden der Funktion `normal` auf die Objekte in allen Zweigen

```
normal(piecewise p)
```

- ⌘ Diese Methode überlädt `normal`.

Methode `piecewise::restrict`: Einschränken durch eine zusätzliche Bedingung

```
piecewise::restrict(any p, condition C)
```

- ⌘ Ist *p* nicht per Fallunterscheidung definiert, so erzeugt diese Methode eine Fallunterscheidung mit dem einzigen Zweig [*C*, *p*]. Ist *p* per Fallunterscheidung definiert, so wird jede Bedingung *cond* in *p* durch *cond* and *C* ersetzt.

Methode `piecewise::set2expr`: Elementbeziehung mit `piecewise` auf der rechten Seite

```
piecewise::set2expr(piecewise p, identifier x)
```

- ⌘ Diese Methode liefert eine Aussageform mit freier Variable *x*, die gleichbedeutend damit ist, dass *x* ein Element von *p* ist.
- ⌘ Die Objekte in allen Zweigen von *p* müssen Mengen repräsentieren.
- ⌘ Diese Methode überlädt die Systemfunktion `_in`.

Methode `simplify`: Vereinfachung eines per Fallunterscheidung definierten Objekts

```
simplify(piecewise p)
```

- ⌘ Diese Methode führt die folgenden Vereinfachungen durch:
 - Zuerst wird `simplify` auf die Objekte in allen Zweigen angewandt.

- Zweige, die dasselbe Objekt definieren, werden zusammengefasst.
- Impliziert die Bedingung in einem Zweig, dass ein freier Parameter gleich einer Konstanten ist, so wird der Parameter in jenem Zweig durch diese Konstante ersetzt.

Siehe Beispiel ??.

☞ Diese Methode überlädt `simplify`.

Methode `solve`: Lösen einer per Fallunterscheidung definierten Gleichung oder Ungleichung

```
solve(piecewise p, identifier x <, option1, option2, ...>)
```

- ☞ Diese Methode löst `p` nach der Variablen `x` auf. Die Objekte in allen Zweigen von `p` müssen entweder Gleichungen, Ungleichungen oder arithmetische Ausdrücke sein. Jeder arithmetische Ausdruck `e` wird durch eine Gleichung `e = 0` ersetzt.
- ☞ Für jeden Zweig `[condition, object]` von `p`, wobei `object` eine Gleichung oder eine Ungleichung ist, bestimmt die Methode die Menge aller Werte `x`, für die `condition` und `object` simultan erfüllt sind, und liefert die Vereinigung all dieser Lösungsmengen zurück. Das Ergebnis kann eine durch Fallunterscheidung definierte Menge sein.
- ☞ Diese Methode überlädt die Funktion `solve`. In der Hilfeseite zu dieser Funktion sind nähere Angaben zu möglichen Optionen und Rückgabewerten zu finden.

Methode `piecewise::solveConditions`: löse alle Bedingungen nach einem gegebenen Bezeichner auf

```
piecewise::solveConditions(piecewise p, identifier x)
```

- ☞ Diese Methode bringt jede Bedingung von `p`, in der `x` vorkommt, in die Form „`x` in `S`“ für eine passende Menge `S`.

Methode `piecewise::Union`: Vereinigung über ein System von Mengen

```
piecewise::Union(piecewise p, identifier x, set indexset)
```

- ☞ Diese Methode liefert die Menge aller Elemente von Elementen von `p`, wobei `p` als durch `x` parametrisiertes Mengensystem aufgefasst wird und `x` die Menge `indexset` durchläuft.
- ☞ Die Objekte in allen Zweigen von `p` müssen Mengen repräsentieren.

- ⌘ Für jeden Zweig `[condition, object]` von `p` tut diese Methode folgendes. Sie substituiert in `object` all diejenigen Werte aus der Menge `indexset` für `x`, die die Bedingung `condition` erfüllen, und berechnet die Vereinigung über alle so erhaltenen Mengen. Dann gibt sie die Vereinigung dieser Mengen über alle Zweige zurück.
 - ⌘ Diese Methode überlädt die Funktion `solveLib::Union`.
-

Zugriffsmethoden

Methode `_concat`: Zusammenfügen von Fallunterscheidungen

`_concat(piecewise p, ...)`

- ⌘ Diese Methode liefert ein per Fallunterscheidung definiertes Objekt, das aus sämtlichen Zweigen aller Argumente besteht.
- ⌘ Diese Methode überlädt `_concat`.

Methode `piecewise::condition`: die Bedingung eines bestimmten Zweigs

`piecewise::condition(piecewise p, positive integer i)`

- ⌘ Diese Methode liefert die Bedingung des `i`-ten Zweigs von `p` zurück. Siehe Beispiel ??.

Methode `piecewise::expression`: das Objekt in einem bestimmten Zweig

`piecewise::expression(piecewise p, positive integer i)`

- ⌘ Diese Methode liefert das Objekt des `i`-ten Zweigs von `p` zurück. Siehe Beispiel ??.

Methode `piecewise::insert`: Einfügen eines Zweiges an einer gegebenen Stelle

`piecewise::insert(piecewise p, branch b, positive integer i)`

- ⌘ Diese Methode liefert `p` zurück, wobei `b` als zusätzlicher Zweig an der `i`-ten Stelle eingefügt wird.
- ⌘ `b` kann entweder ein mittels `op` erhaltener Zweig eines anderen per Fallunterscheidung definierten Objekts oder eine Liste `[condition, object]` sein.
- ⌘ Die Zahl `i` darf nicht größer sein als die Anzahl der Zweige von `p` plus eins.
- ⌘ Siehe Beispiel ??.

Methode `map`: wende eine Funktion auf die Objekte in allen Zweigen an

```
map(piecewise p, any f <, any a, ...>)
```

- ⌘ Für jeden Zweig [*condition*, *object*] von *p* wird *object* durch *f*(*object* <, *a*, ...>) ersetzt.
- ⌘ Diese Methode überlädt `map`.

Methode `piecewise::mapConditions`: wende eine Funktion auf die Bedingungen in allen Zweigen an

```
piecewise::mapConditions(piecewise p, any f <, any a, ...>)
```

- ⌘ Für jeden Zweig [*condition*, *object*] von *p* wird *condition* durch *f*(*condition* <, *a*, ...>) ersetzt.

Methode `piecewise::mapMap`: wende die Funktion `map` auf die Objekte in allen Zweigen an

```
piecewise::mapMap(any p, any f <, any a, ...>)
```

- ⌘ Für jeden Zweig [*condition*, *object*] von *p* wird *object* durch `map`(*object*, *f* <, *a*, ...>) ersetzt.
- ⌘ Ist *p* kein durch Fallunterscheidung definiertes Objekt, dann wird einfach `map`(*p*, *f* <, *a*, ...>) zurückgegeben.

Methode `piecewise::remove`: Entfernen eines Zweiges

```
piecewise::remove(piecewise p, positive integer i)
```

- ⌘ Diese Methode liefert dasjenige per Fallunterscheidung definierte Objekt zurück, das man aus *p* durch Löschen des *i*-ten Zweigs erhält. Siehe Beispiel ??.

Methode `piecewise::selectConditions`: wähle Zweige in Abhängigkeit von ihrer Bedingung aus

```
piecewise::selectConditions(piecewise p, any f <, any a, ...>)
```

- ⌘ Diese Methode arbeitet wie die Funktion `select`, indem sie das durch *f* gegebene Auswahlkriterium auf die Bedingungen von *p* anwendet. Sie liefert dasjenige fallweise definierte Objekt, das aus *p* entsteht, wenn man alle Zweige [*condition*, *object*] entfernt, für die *f*(*condition* <, *a*, ...>) nicht `TRUE` ergibt.

- ☞ Für jede Bedingung `condition` in `p` muss `f(condition <, a, ...>)` eine Boole'sche Konstante ergeben.
- ☞ Erfüllt keine der Bedingungen das Auswahlkriterium, so wird `undefined` zurückgeliefert.

Methode `piecewise::splitConditions`: teile Zweige in Abhängigkeit von ihrer Bedingung auf

```
piecewise::splitConditions(piecewise p, any f <, any a, ...>)
```

- ☞ Diese Methode arbeitet wie die Funktion `split`, indem sie das durch `f` gegebene Aufteilungskriterium auf die Bedingungen von `p` anwendet. Sie liefert eine Liste bestehend aus drei per Fallunterscheidung definierten Objekten, die diejenigen Zweige von `p` umfassen, für die `f(condition <, a, ...>)` den Wert `TRUE`, `FALSE` bzw. `UNKNOWN` liefert.
Ergibt für einen der drei boolschen Werte kein Zweig diesen Wert, dann enthält die zurückgegebene Liste an der entsprechenden Stelle `undefined` an Stelle eines per Fallunterscheidung definierten Objekts mit null Zweigen.
- ☞ Für jede Bedingung `condition` in `p` muss `f(condition <, a, ...>)` eine Boole'sche Konstante ergeben.
- ☞ Siehe Beispiel ??.

Methode `subs`: Ersetzung

```
subs(piecewise p, substitution s, ...)
```

- ☞ Diese Methode führt die Substitution(en) `s` sowohl in den Bedingungen als auch in den Objekten von `p` durch.
- ☞ Diese Methode überlädt die Funktion `subs` und hat dieselbe Aufrufsyntax. Siehe die entsprechende Hilfeseite für eine Beschreibung der verschiedenen Typen, die für `s` zulässig sind.

Methode `zip`: punktweise Anwendung einer binären Verknüpfung

```
zip(any p1, any p2, any f)
```

- ☞ Sind sowohl `p1` als auch `p2` durch Fallunterscheidung definierte Objekte, dann liefert diese Methode dasjenige durch Fallunterscheidung definierte Objekt zurück, das aus allen Zweigen der Form `[condition1 and condition2, f(object1, object2)]` besteht, wobei `[condition1, object1]` ein Zweig von `p1` und `[condition2, object2]` ein Zweig von `p2` ist.

- ☞ Betrachtet man fallweise definierte Objekte als Funktionen von der Menge A aller möglichen Parameterwerte in eine Menge B von Objekten, so realisiert diese Methode die kanonische Fortsetzung der binären Verknüpfung f auf B zu einer Verknüpfung g auf der Menge B^A aller Funktionen von A nach B , die durch $(g(p1, p2))(a) = f(p1(a), p2(a))$ für alle a in A gegeben ist.
- ☞ Ist nur eines der ersten beiden Argumente, etwa $p1$, vom Typ `piecewise`, so wird jeder Zweig `[condition, object]` darin durch `[condition, f(object, p2)]` ersetzt.
- ☞ Falls weder $p1$ noch $p2$ vom Typ `piecewise` sind, so liefert `piecewise::zip(p1, p2, f)` das Ergebnis $f(p1, p2)$ zurück.
- ☞ Diese Methode überlädt `zip`.

Beispiel 1. Sei f die charakteristische Funktion des Intervalls $[0, 1]$:

```
>> f := x -> piecewise([x < 0 or x > 1, 0], [x >= 0 or x <= 1, 1])
      x -> piecewise([x < 0 or 1 < x, 0], [0 <= x or x <= 1, 1])
```

Keine der Bedingungen kann direkt zu `TRUE` oder `FALSE` ausgewertet werden, wenn nicht mehr über die Variable x bekannt ist. Wertet man f an einer Stelle aus, so werden die Bedingungen neu überprüft:

```
>> f(0), f(2), f(I)
      1, 0, undefined
```

Beispiel 2. `piecewise` führt eine Fallunterscheidung durch und verwendet dabei den property-Mechanismus. Die gegebenen Bedingungen werden daraufhin untersucht, ob sie *mathematisch* wahr oder falsch sind; hierbei kann sich auch herausstellen, dass die vorhandenen Informationen zu einer Entscheidung nicht ausreichen. Im folgenden Beispiel kann nicht entschieden werden, ob $a=0$ ist, solange keine Annahmen über a gemacht wurden:

```
>> delete a:
      p := piecewise([a = 0, 0], [a <> 0, 1/a])
      piecewise| 0 if a = 0, - 1/a if a <> 0 |
```

Im Gegensatz dazu werden `if`-Bedingungen syntaktisch ausgewertet: $a=0$ ist *technisch* falsch, weil der Bezeichner a und die ganze Zahl 0 verschiedene Objekte sind:

```
>> if a = 0 then 0 else 1/a end
```

$$\frac{1}{a}$$

Außerdem berücksichtigt `piecewise` Eigenschaften von Bezeichnern:

```
>> assume(a = 0):
    p;
    delete a, p:
```

$$0$$

Beispiel 3. Fallweise definierte Objekte können auch mittels `rewrite` definiert werden:

```
>> f := rewrite(sign(x), piecewise)
```

$$\text{piecewise} \left| \begin{array}{l} 1 \text{ if } 0 < x, -1 \text{ if } x < 0, 0 \text{ if } x = 0, \\ \hline \frac{x}{(\text{Im}(x)^2 + \text{Re}(x)^2)^{1/2}} \text{ if not } x \text{ in } \mathbb{R}_+ \end{array} \right|$$

Es ist üblich, im Gegensatz zu MuPAD die Funktion `sign` nur auf den reellen Zahlen zu betrachten. Dazu muss der Definitionsbereich von `f` eingeschränkt werden:

```
>> f := piecewise::restrict(f, x in R_)

piecewise(1 if x in ]0, infinity[, -1 if x in ]-infinity, 0[,
          0 if x in {0})
```

Für per Fallunterscheidung definierte arithmetische Ausdrücke sind im Wesentlichen dieselben Operationen definiert wie für arithmetische Ausdrücke. Arithmetische Operationen sind hierbei argumentweise definiert; im Definitionsbereich des Ergebnisses liegen nur Punkte, die im Definitionsbereich jedes Arguments liegen:

```
>> f + piecewise([x < 2, 5])

piecewise(6 if 0 < x and x < 2, 4 if x < 0, 5 if x = 0)
```

Beispiel 4. Es gibt mehrere Methoden zum Zugriff auf einzelne Zweige, Bedingungen und Objekte. Wir betrachten das folgende per Fallunterscheidung definierte Objekt:

```
>> f := piecewise([x > 0, 1], [x < -3, x^2])
```

$$\text{piecewise}(1 \text{ if } 0 < x, x^2 \text{ if } x < -3)$$

Es ist möglich, auf eine einzelne Bedingung oder ein einzelnes Objekt zuzugreifen:

```
>> piecewise::condition(f, 1), piecewise::expression(f, 2)
```

$$0 < x, x^2$$

Die Funktion `op` greift ganze Zweige heraus:

```
>> op(f, 1)
```

$$1 \text{ if } 0 < x$$

Es ist auch möglich, ein anderes per Fallunterscheidung definiertes Objekt zu erzeugen, das nurmehr aus denjenigen Zweigen besteht, deren Bedingungen ein gegebenes Auswahlkriterium erfüllen, oder die Eingabe in zwei per Fallunterscheidung definierte Objekte aufzuspalten, analog zum Verhalten der Systemfunktionen `select` und `split` für Listen:

```
>> piecewise::selectConditions(f, has, 0)
```

$$\text{piecewise}(1 \text{ if } 0 < x)$$

```
>> piecewise::splitConditions(f, has, 0)
```

$$[\text{piecewise}(1 \text{ if } 0 < x), \text{piecewise}(x^2 \text{ if } x < -3), \text{undefined}]$$

Es ist auch möglich, eine Kopie von `f` zu erzeugen, aus der bestimmte Zweige gelöscht oder in die bestimmte Zweige eingefügt sind:

```
>> piecewise::remove(f, 1)
```

$$\text{piecewise}(x^2 \text{ if } x < -3)$$

```
>> piecewise::insert(f, [x > -3 and x < 0, sin(x)], 2)
```

$$\text{piecewise}(1 \text{ if } 0 < x, \sin(x) \text{ if } x < 0 \text{ and } -3 < x, x^2 \text{ if } x < -3)$$

Beispiel 5. Die meisten Funktionen, die nur ein Argument haben, sind für `piecewise` überladen, indem sie auf die Objekte in allen Zweige einzeln angewandt werden. Dies kann auch mittels `map` erreicht werden:

```
>> f := piecewise([x >= 0, arcsin(x)], [x < 0, arccos(x)]):
      sin(f)
```

$$\text{piecewise}(x \text{ if } 0 \leq x, (-x^2 + 1)^{1/2} \text{ if } x < 0)$$

```
>> map(f, sin)
```

$$\text{piecewise}(x \text{ if } 0 \leq x, (-x^2 + 1)^{1/2} \text{ if } x < 0)$$

Dies führt zu folgendem Problem. Falls irgendeine Bedingung z.B. wegen einer Annahme über x wahr wird, so wird f zu einem Objekt ausgewertet, das nicht vom Typ `piecewise` ist. Der Aufruf von `sin` liefert dann weiter das gewünschte Ergebnis, aber `map` wirkt dann auf die Operanden von f . Daher sollte `map` nur mit Vorsicht verwendet werden:

```
>> assume(x < 0):
      sin(f);
      map(f, sin);
```

$$(1 - x^2)^{1/2}$$

$$\arccos(\sin(x))$$

```
>> delete x:
```

Das umgekehrte Problem tritt auf, wenn die Funktion `map` auf die Objekte in allen Zweigen angewandt werden soll. In diesem Fall sollte die Methode `mapMap` verwendet werden.

Beispiel 6. Mengen können ebenfalls fallweise definiert sein. Solche Mengen werden manchmal von `solve` geliefert:

```
>> S := solve(a*x = 0, x)
```

$$\text{piecewise}(C_ \text{ if } a = 0, \{0\} \text{ if } a \neq 0)$$

Für solche Mengen sind die üblichen Mengenoperationen definiert:

```
>> S intersect Dom::Interval(3, 5)
```

$$\text{piecewise}([3, 5[\text{ if } a = 0, \{\} \text{ if } a \neq 0)$$

Manchmal ist es nützlich, „seltene Fälle“ auszuschließen, die nur auf eine kleine Menge von Parameterwerten zutreffen:

```
>> piecewise::disregardPoints(S)
```

$$\{0\}$$

Beispiel 7. Wir betrachten die folgende Fallunterscheidung:

```
>> p1 := piecewise([a > 0, a^2], [a <= 0, -a^2]):
      p2 := piecewise([b > 0, a + b], [b = 0, p1 + b], [b < 0, a + b])

      2
      piecewise(a + b if 0 < b, b + a  if 0 < a and b = 0,

      2
      b - a  if b = 0 and a <= 0, a + b if b < 0)
```

Man sieht, dass die in `p1` durchgeführte Fallunterscheidung automatisch auf die oberste Ebene verschoben wurde. Aber es sind noch weitere Vereinfachungen möglich: die Zweige `b<0` und `b>0` können zusammengefasst werden, und im Fall `b=0` kann der Bezeichner `b` durch `0` ersetzt werden:

```
>> simplify(p2)

      2
      piecewise(a + b if b <> 0, a  if 0 < a and b = 0,

      2
      - a  if b = 0 and a <= 0)
```

Hintergründe:

- ☞ Die Operanden eines fallweise definierten Objekts sind Paare, bestehend aus einer Bedingung und dem unter dieser Bedingung gültigen Objekt. Für solche Paare existiert ein eigener Datentyp `stdlib::branch`.
- ☞ Methoden, die Systemfunktionen überladen, gehen davon aus, dass sie nur per Überladung aufgerufen werden: unter ihren Argumenten muss sich immer ein per Fallunterscheidung definiertes Objekt befinden. Alle anderen Methoden setzen nicht zwingend voraus, dass ihr Argument vom Typ `piecewise` ist. Dies erleichtert die Benutzung von `piecewise`: es ist stets erlaubt, `p:=piecewise(...)` zu definieren und auf `p` eine Methode von `piecewise` anzuwenden, selbst dann, falls `p` nicht vom Typ `piecewise` ist, weil eine Bedingung als wahr oder alle Bedingungen als falsch erkannt wurden.

Änderungen:

- ☞ `piecewise` ist eine neue Funktion.

plot – Anzeige graphischer Objekte auf dem Bildschirm

`plot(scene)` zeigt die graphische Szene `scene` auf dem Bildschirm an.

`plot(object1, object2, ...)` zeigt die graphischen Objekte `object1`, `object2` etc. auf dem Bildschirm an.

Aufruf(e):

```
# plot(scene)
# plot(object1 <, object2, ...> <, option1, option2,
#     ...>)
```

Parameter:

<code>scene</code>	— eine graphische Szene: ein Objekt vom Datentyp <code>plot::Scene</code>
<code>object1, object2, ...</code>	— zwei- oder dreidimensionale graphische Objekte
<code>option1, option2, ...</code>	— Szeneoptionen der Form <code>OptionsName = Wert</code>

Überladbar durch: `object1`

Verwandte Domains: `plot::Scene`

Verwandte Funktionen: `plot2d`, `plot3d`

Details:

- # Graphische Szenen werden durch `plot::Scene` erzeugt. Details sind auf der entsprechenden Hilfeseite zu finden.
- # Die Parameter `object1`, `object2` etc. müssen graphische Objekte sein, die von der Bibliothek `plot` zur Verfügung gestellt werden. Darunter fallen Funktionsgraphen (`plot::Function2d` und `plot::Function3d`), Punkte und Polygone (`plot::Point` bzw. `plot::Polygon`) sowie parametrisierte Flächen (`plot::Surface3d`). Siehe Beispiel ??.
- # Funktionen der `plot`-Bibliothek wie beispielsweise `plot::vectorfield`, `plot::ode` oder `plot::implicit` liefern komplexere graphische Objekte, die ebenfalls mit der Funktion `plot` auf dem Bildschirm angezeigt werden können. Siehe Beispiel ??.
- # Szeneoptionen `option1`, `option2` etc. werden als Gleichungen der Form `OptionsName = Wert` angegeben. Auf der Hilfeseite von `plot::Scene` findet sich eine Tabelle aller zulässigen Szeneoptionen.
- # Die graphischen Objekte `object1`, `object2` etc. müssen dieselbe Dimension besitzen. Die Angabe zwei- und dreidimensionaler Objekte in einer gemeinsamen Szene wird nicht unterstützt!



Beispiel 1. Die folgenden Aufrufe liefern zwei Objekte, die die Graphen der Sinus- und Kosinus-Funktion über dem Intervall $[0, 2\pi]$ repräsentieren:

```
>> 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)
```

Um diese Objekte auf dem Bildschirm zu zeichnen, geben wir ein:

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

Dieser Aufruf verwendet die Standardeinstellungen der Szeneoptionen wie auf der Hilfeseite von `plot::Scene` dokumentiert. Szeneoptionen können als zusätzliche Argumente an `plot` übergeben werden. Um beispielsweise die obige Graphik mit Gitterlinien zu unterlegen, geben wir ein:

```
>> plot(f1, f2, GridLines = Automatic)
```

Details zur `GridLines`-Option sind auf der Hilfeseite `plotOptions2d` zu finden.

```
>> delete f1, f2:
```

Beispiel 2. Die Bibliothek `plot` bietet eine Reihe von Routinen zur Erzeugung komplexerer graphischer Objekte wie beispielsweise Vektorfelder, Lösungskurven gewöhnlicher Differentialgleichungen oder implizit definierte Kurven.

Um z. B. die implizit durch die Gleichung $x^2 + x + 2 = y^2$ definierte Kurve mit x, y aus dem Intervall $[-5, 5]$ zu zeichnen, verwenden wir die Funktion `plot::implicit`:

```
>> plot(  
    plot::implicit(  
        x^3 + x + 2 - y^2, x = -5..5, y = -5..5  
    ),  
    Scaling = Constrained  
)
```

Wir verwendeten hierbei die `Scaling`-Option, um das Skalierungsverhältnis 1:1 zwischen den x - und y -Koordinaten unabhängig von der Größe des graphischen Fensters zu garantieren (siehe `plotOptions2d` für Details).

Hintergründe:

☞ Technisch gesehen ist `plot` keine Funktion, sondern ein Domain, das die Bibliothek `plot` repräsentiert. Der Aufruf `plot(...)` bewirkt intern den Aufruf `plot::new(...)`.

Die Methode "new" arbeitet wie folgt: Wird eine graphische Szene als Übergabeparameter angegeben (siehe Parameter `scene`), so wird die Methode "getPlotdata" des Domains `plot::Scene` aufgerufen. Sie liefert eine graphische Szene in einer `plot2d`-konformen Syntax zurück (bzw. in einer `plot3d`-konformen Syntax, falls die Szene dreidimensional ist). Das Ergebnis wird der Funktion `plot2d` bzw. `plot3d` übergeben, die die graphische Szene zeichnet.

Sind dagegen graphische Objekte `object1`, `object2` etc. als Parameter angegeben, so erzeugt die Methode `plot::new` zuerst eine graphische Szene vom Datentyp `plot::Scene`, die aus diesen Objekten besteht, und fährt dann wie oben beschrieben fort.

Änderungen:

☞ `plot` ist ein neues Domain.

`plot2d` – 2-dimensionale Graphiken

`plot2d(object1, object2, ...)` erzeugt eine 2-dimensionale Graphik von Objekten wie parametrisierten Kurven, Punkten und Polygonen.

Aufruf(e):

☞ `plot2d(<SceneOptions,> object1, object2, ...)`

Parameter:

`object1, object2, ...` — graphische Objekte wie unten beschrieben

Optionen:

`SceneOptions` — eine Folge von Szeneoptionen, die das allgemeine Aussehen der graphischen Szene bestimmen.
Details sind mit `Mup[plotOptions2d]?plotOptions2d` anzufordern.

Rückgabewert: MuPADs Graphik-Werkzeug wird aufgerufen, um das angeforderte Bild anzuzeigen. Das `null()`-Objekt wird an die MuPAD-Sitzung zurückgeliefert.

Verwandte Funktionen: `plot`, `plotfunc2d`, `plot3d`, `plotfunc3d`

Details:

☞ `plot2d` ist eine recht technische Funktion, um 2D-Graphiken aus Primitiven aufzubauen. Für Funktionsgraphen stehen die einfacher zu bedienenden spezialisierten Routinen `plotfunc2d` und `plot::Function2d` zur Verfügung. Bei aus graphischen Primitiven aufzubauenden Szenen ist die Benutzung der `plot`-Bibliothek zu empfehlen, die zahlreiche Primitive und Werkzeuge zur Verfügung stellt. In den meisten Fällen wird der Nutzer die leicht handhabbaren Objekte der `plot`-Bibliothek einem Aufruf von `plot2d` vorziehen.

☞ `plot2d` akzeptiert zwei Typen graphischer Objekte: i) Listen graphischer Primitive (Punkte und Polygone) sowie ii) parametrisierte Kurven.

☞ i) *Listen graphischer Primitive* sind Objekte der Form

`[Mode = List, [Primitiv1, Primitiv2, ...] <, Options>]`

Die zur Verfügung stehenden Primitive sind Punkte, Polygonzüge sowie gefüllte Polygone, die mittels der MuPAD-Funktionen `point` bzw. `polygon` erzeugt werden. Aus solchen Primitiven können komplexere graphische Objekte zusammengesetzt werden.

Optionen werden als Gleichungen `OptionsName = Wert` übergeben. Die folgende Tabelle gibt einen Überblick über die zur Verfügung stehenden Optionen:

OptionsName	mögliche Werte	Standardwert
<i>Color</i>	<code>[Flat]</code> , <code>[Flat, [r,g,b]]</code> , <code>[Height]</code> , <code>[Height, [r,g,b], [R,G,B]]</code> , <code>[Function, f]</code>	<code>[Height]</code>
<i>LineStyle</i>	<code>SolidLines</code> , <code>DashedLines</code>	<code>SolidLines</code>
<i>LineWidth</i>	positive ganze Zahlen	1
<i>PointStyle</i>	<code>Circles</code> , <code>FilledCircles</code> , <code>FilledSquares</code> , <code>Squares</code>	<code>FilledSquares</code>
<i>PointWidth</i>	positive ganze Zahlen	30
<i>Title</i>	Zeichenketten	" "
<i>TitlePosition</i>	<code>[x, y]</code>	

Erläuterungen und Details zu jeder Option sind weiter unten aufgeführt.

☞ ii) *Parametrisierte Kurven* werden durch eine Abbildung $u \mapsto [x(u), y(u)]$ mit Funktionen $x(u)$, $y(u)$ definiert, welche die Koordinaten in Abhängigkeit eines Kurvenparameters u angeben. In `plot2d` ist eine solche Kurve als ein Objekt der folgenden Form anzugeben:

`[Mode = Curve, [x(u), y(u)], u = [umin, umax] <, Options>]`

Die Parametrisierung $x(u), y(u)$ besteht aus arithmetischen Ausdrücken in einer Unbestimmten u (ein Bezeichner). Sie dürfen neben dem Kurvenparameter keine weiteren symbolischen Parameter enthalten, die sich nicht in Gleitpunktzahlen konvertieren lassen. Der Bereich des Kurvenparameters u wird durch die Zahlen oder numerischen Ausdrücke u_{\min} und u_{\max} angegeben.

Liegt eine Parametrisierung durch benutzerdefinierte Funktionen vor, die nur numerische Argumente akzeptieren, so kann die Auswertung verzögert werden, indem $\text{hold}(x)(u)$, $\text{hold}(y)(u)$ mit dem symbolischen Kurvenparameter u übergeben wird.

Optionen werden durch Gleichungen $\text{OptionsName} = \text{Wert}$ angegeben. Alle oben für Listen graphischer Primitive angegebenen Optionen sind zulässig. Für Kurven können darüberhinaus die folgenden weiteren Optionen verwendet werden:

OptionsName	zulässige Werte	Standardwert
<i>Grid</i>	[n]	[100]
<i>Smoothness</i>	[n]	[0]
<i>Style</i>	[Points], [Lines], [LinesPoints], [Impulses]	[Lines]

Details zu jeder Option sind weiter unten aufgeführt.

- ☞ Der Graph einer Funktion $f(x)$ kann als parametrisierte Kurve $[\text{Mode} = \text{Curve}, [x, f(x)], x = [x_{\min}, x_{\max}] <, \text{Options}>]$ erzeugt werden. Es wird jedoch empfohlen, stattdessen die auf Funktionsgraphen spezialisierten Routinen `plotfunc2d` bzw. `plot::Function2d` zu verwenden. Diese können im Gegensatz zu `plot2d` singuläre Funktionen verarbeiten.

- ☞ MuPAD-Graphiken können in einer Vielzahl graphischer Formate gespeichert werden. Für die beiden MuPAD-spezifischen Formate *Ascii* und *Binary* kann dies über die Szeneoption `PlotDevice` direkt im Aufruf von `plot2d` geschehen. Details hierzu liefert `?plotOptions2d`.

Die Speicherung in graphischen Standardformaten wie z. B. *Postscript*, *JPEG*, *TIFF* etc. kann nicht direkt innerhalb einer MuPAD-Sitzung über ein Plot-Kommando geschehen. Man muss stattdessen die Oberfläche des graphischen Werkzeugs VCam interaktiv benutzen: innerhalb eines MuPAD Pro Notebooks wird diese Oberfläche durch einen Doppelklick auf die Graphik aktiviert. Durch Auswahl der Menüpunkte „Bearbeiten/Exportieren...“ wird die Dialogbox „Exportiere Graphik“ geöffnet, in der das gewünschte Format eingestellt werden kann.

Option **<Color = Wert>**:

☞ Diese Option legt die Farbe des Objekts fest. Zulässige Werte sind `[Flat]`, `[Flat, [r,g,b]]`, `[Height]`, `[Height, [r,g,b]]`, `[R,G,B]` und `[Function, f]`. Die Voreinstellung ist `Color = [Height]`.

- Mit `Color = [Flat]` wird das Objekt einheitlich in einer Farbe dargestellt. Diese wird automatisch gewählt.
- Mit `Color = [Flat, [r, g, b]]` wird das Objekt einheitlich in einer Farbe dargestellt. Die Werte `r`, `g`, `b` stellen den Rot-, Grün und Blauanteil gemäß des RGB-Farbmodells dar. Sie müssen reelle numerische Werte zwischen 0 und 1 sein. Vordefinierter Farben werden durch MuPADs RGB-Datenstruktur zur Verfügung gestellt.
- Mit `Color = [Height]` wird ein von der y -Koordinate abhängender Farbverlauf benutzt. Die tatsächlichen Farben werden automatisch gewählt.
- Mit `Color = [Height, [r, g, b], [R, G, B]]` wird ein von der y -Koordinate abhängender Farbverlauf benutzt. Die Teile des Objektes mit kleinen y -Werten werden in der Farbe `[r, g, b]` dargestellt, die Teile mit großen y -Werten in der Farbe `[R, G, B]`. Dazwischen werden interpolierte Farbwerte benutzt.
- Mit `Color = [Function, f]` kann der Nutzer sein eigenes Farbschema implementieren. Der Parameter `f` muss eine MuPAD Prozedur sein, welche Farbwerte als Listen `[r, g, b]` produziert.
 - Innerhalb eines Kurvenobjekts muss die Prozedur `f` drei Parameter akzeptieren:

```
f := proc(x, y, u) begin ...; return([r, g, b]) end;
```

Während der numerischen Auswertung der Graphik wird diese Prozedur mit den Argumenten `(x(u), y(u), u)` aufgerufen, wo `u` der Kurvenparameter ist und `x(u), y(u)` die entsprechenden Koordinatenwerte.
 - Innerhalb einer Liste graphischer Primitive muss die Prozedur `f` zwei Parameter akzeptieren:

```
f := proc(x, y) begin ...; return([r, g, b]) end;
```

Während der numerischen Auswertung der Graphik wird diese Prozedur mit Argumenten `(x, y)` aus dem Sichtbarkeitsbereich des Objekts aufgerufen.

Man beachte, dass Polygone mit jeweils einer einheitlichen Farbe gezeichnet werden.

Wird die Farbfunktion `f` innerhalb einer Prozedur erzeugt, wobei `f` auf lokale Variablen der Prozedur zugreift, so muss die erzeugende Prozedur `option escape` verwenden.



Option `<Grid = [n]>`:

- ☞ Diese Option legt die Anzahl *n* der Stützpunkte von Kurven fest. Zwischen diesen Punkten werden Kurven linear interpoliert. Die ganze Zahl *n* muss mindestens 2 sein, die Voreinstellung ist *Grid = [100]*. Durch eine hohe Anzahl von Stützpunkten werden glatte Kurven erzeugt. Alternativ kann der *Smoothness*-Parameter heraufgesetzt werden.

Option `<LineStyle = Wert>`:

- ☞ Diese Option legt den Stil fest, mit der Linien innerhalb des Objekts dargestellt werden. Zulässige Werte sind *SolidLines* und *DashedLines*: es werden entweder durchgezogene oder gestrichelte Linien gezeichnet. Die Voreinstellung ist *LineStyle = SolidLines*.

Option `<LineWidth = n>`:

- ☞ Diese Option legt die Breite der Linien innerhalb des Objekts fest. Zulässige Werte für *n* sind nicht-negative ganze Zahlen, die Voreinstellung ist *LineWidth = 1*.

Option `<PointStyle = Wert>`:

- ☞ Diese Option legt den Stil fest, mit der Punkte dargestellt werden. Zulässige Werte sind *Circles*, *Squares*, *FilledCircles* und *FilledSquares*. Die Voreinstellung ist *PointStyle = FilledSquares*.

Option `<PointWidth = n>`:

- ☞ Diese Option legt die Größe von Punkten fest. Zulässige Werte für *n* sind positive ganze Zahlen, die Voreinstellung ist *PointWidth = 30*.

Option <Smoothness = [n]>:

- ☞ Diese Option legt die Anzahl von Zwischenpunkten fest, die zwischen jeweils zwei der durch die *Grid*-Option festgelegten Stützpunkte von Kurven eingeschoben werden. Linien werden als verbindende Geradenstücke zwischen jeweils zwei dieser Zwischenpunkte dargestellt. Hohe Werte von *n* liefern damit glatte Kurven. Zulässig sind ganze Zahlen zwischen 0 und 20, die Voreinstellung ist *Smoothness* = [0].

Option <Style = Wert>:

- ☞ Diese Option legt den Stil fest, mit dem Kurven dargestellt werden. Zulässige Werte sind [*Points*], [*Lines*], [*LinesPoints*] und [*Impulses*]. Die Voreinstellung ist *Style* = [*Lines*].

- Mit *Style* = [*Points*] werden nur die Stützpunkte dargestellt.
- Mit *Style* = [*Lines*] wird die Kurve als Menge von Geradensegmenten zwischen den Stützpunkten dargestellt.
- Mit *Style* = [*LinesPoints*] werden sowohl die Stützpunkte als auch die verbindenden Geradenstücke dargestellt.
- Mit *Style* = [*Impulses*] wird die Kurve wie ein „Histogramm“ dargestellt: es werden vertikale Linien von der Unterkante der Szene zu den Stützpunkten gezeichnet.

Option <Title = Titel>:

- ☞ Diese Option haftet eine durch die Zeichenkette *Titel* gegebene Überschrift an das Objekt. Die Voreinstellung ist die leere Zeichenkette *Title* = "", also keine Überschrift.

Option <TitlePosition = [x, y]>:

- ☞ Diese Option legt die Platzierung der Objektüberschrift fest. Zulässige Werte sind Listen [*x*, *y*] mit numerischen Werten *x*, *y* zwischen 0 und 10. Die Position [0, 0] ist die linke obere Ecke der Szene, [10, 10] ist die rechte untere Ecke. Man beachte, dass die Platzierung relativ zur gesamten Szene spezifiziert wird. Die Platzierungen der Überschriften mehrerer Objekte sollte demgemäß unterschiedlich sein, um Überlappungen zu vermeiden.

- ☞ Objektüberschriften können innerhalb der angezeigten Graphik mit der Maus bewegt und so interaktiv an eine geeignete Stelle verschoben werden.

Beispiel 1. Ein Halbkreis mit Radius 1 wird durch den Polarwinkel u parametrisiert. Die Option *Scaling = Constrained* stellt sicher, dass der Kreis nicht zu einer Ellipse deformiert wird:

```
>> plot2d(Scaling = Constrained, Labeling = TRUE,
           [Mode = Curve, [cos(u), sin(u)], u = [0, PI]])
```

Beispiel 2. Es werden zwei Punktprimitive, ein Linienprimitiv und ein Flächenprimitiv erzeugt:

```
>> point1 := point(1, 1, Color = RGB::Red):
   point2 := point(-1, 1, Color = RGB::Green):
   line := polygon(point(1, 0), point(0, 1), point(0, 0),
                   Color = RGB::Blue):
   triangle := polygon(point(0, 0), point(0, 1), point(-1, 0),
                       Closed = TRUE, Filled = TRUE,
                       Color = RGB::Antique):
```

Diese werden zu einem graphischen Objekt zusammengefasst:

```
>> object := [Mode = List, [point1, point2, line, triangle]]:
```

Das Objekt wird dargestellt:

```
>> plot2d(BackGround = RGB::White, PointWidth = 50,
           PointStyle = FilledCircles, object)

>> delete point1, point2, line, triangle, object:
```

Beispiel 3. Der Graph der Sinus-Funktion wird in unterschiedlichen Stilen dargestellt.

```
>> plot2d(BackGround = RGB::White, ForeGround = RGB::Black,
           Labeling = TRUE, PointWidth = 50,
           [Mode = Curve, [x, sin(10*x)], x = [0, 1],
            Color = [Flat, RGB::Red], Grid = [50], Smoothness = [0],
            PointStyle = FilledSquares, Style = [Points]
           ],
           [Mode = Curve, [x, 0.1 + sin(10*x)], x = [0, 1],
            Color = [Flat, RGB::Green],
```

```

Grid = [20], Smoothness = [1],
PointStyle = FilledCircles, Style = [LinesPoints]
],
[Mode = Curve, [x, 0.2 + sin(10*x)], x = [0, 1],
Color = [Flat, RGB::Blue], Grid = [100], Style = [Lines]
])

```

Beispiel 4. Die *ViewingBox*-Option wird demonstriert.

```

>> spiral := [Mode = Curve, [u*cos(u), u*sin(u)], u = [0, 2*PI],
Grid = [50]]:

```

Zunächst wird dieses Objekt ohne Einschränkung des Sichtbarkeitsbereichs gezeichnet:

```

>> plot2d(Axes = Box, Labeling = TRUE, spiral)

```

In der nächsten Graphik wird der Sichtbarkeitsbereich auf $x \in [-4, 1]$, $y \in [-2, 2]$ eingeschränkt:

```

>> plot2d(Axes = Box, Labeling = TRUE,
ViewingBox = [-4..1, -2..2], spiral)

```

```

>> delete spiral:

```

Beispiel 5. Dieses Beispiel demonstriert selbstdefinierte Farbfunktionen. Die folgende Funktion liefert zulässige RGB-Werte zwischen 0 und 1 für Objekte mit Koordinaten im Bereich $x \in [0, 1]$ und $y \in [0, 1]$:

```

>> myColor := (x, y) -> [x, 0.5 + abs(x - y)/(1 + x + y), y]:

```

Das Einheitsquadrat soll durch diese Funktion gefärbt werden. Das Quadrat wird dazu durch $2n^2$ Dreiecke überdeckt, die jeweils einheitlich mit einer durch *myColor* bestimmten Farbe gefüllt sind:

```

>> n := 30:
plot2d([Mode = List,
[polygon(point((i-1)/n, (j-1)/n),
point((i-1)/n, j/n),
point(i/n, j/n),
Filled = TRUE
) $ i = 1..n $ j = 1..n,
polygon(point((i-1)/n, (j-1)/n),
point(i/n, (j-1)/n),
point(i/n, j/n),

```

```

        Filled = TRUE
        ) $ i = 1..n $ j = 1..n
    ],
    Color = [Function, myColor]
  ]):

>> delete myColor, n:

```

Änderungen:

- ⌘ Die neuen Szeneoptionen *Discont*, *GridLines*, *GridLinesColor*, *GridLinesStyle*, *GridLinesWidth*, *RealValuesOnly* und *ViewingBox* wurden eingeführt. Die Funktionalität der Szeneoption *Ticks* wurde erweitert.
 - ⌘ Die Voreinstellungen diverser Optionen wurden geändert.
 - ⌘ Überschriften der Szene sowie Objektüberschriften können nun mit der Maus verschoben werden.
-

plot3d – 3-dimensionale Graphiken

`plot3d(Objekt1, Objekt2, ...)` erzeugt eine 3-dimensionale Graphik von Objekten wie Kurven, Flächen, Punkten und Polygonen.

Aufruf(e):

⌘ `plot3d(<SceneOptions,> object1, object2, ...)`

Parameter:

`object1, object2, ...` — graphische Objekte wie unten beschrieben

Optionen:

`SceneOptions` — eine Folge von Szeneoptionen, die das allgemeine Aussehen der graphischen Szene bestimmen.
Details sind mit `?plotOptions3d` anzufordern.

Rückgabewert: MuPADs Graphik-Werkzeug wird aufgerufen, um das angeforderte Bild anzuzeigen. Das `null()`-Objekt wird an die MuPAD-Sitzung zurückgeliefert.

Verwandte Funktionen: `plot`, `plotfunc2d`, `plot2d`, `plotfunc3d`

Details:

☞ `plot3d` ist eine recht technische Funktion, um 3D-Graphiken aus Primitiven aufzubauen. Für Funktionsgraphen stehen die einfacher zu bedienenden spezialisierten Routinen `plotfunc3d` und `plot::Function3d` zur Verfügung. Bei aus graphischen Primitiven aufzubauenden Szenen ist die Benutzung der `plot`-Bibliothek zu empfehlen, die zahlreiche Primitive und Werkzeuge zur Verfügung stellt. In den meisten Fällen wird der Nutzer die leicht handhabbaren Objekte der `plot`-Bibliothek einem Aufruf von `plot3d` vorziehen.

☞ `plot3d` kann drei Arten graphischer Objekte zeichnen: i) Listen graphischer Primitive (Punkte und Polygone), ii) parametrisierte Kurven sowie iii) parametrisierte Flächen.

☞ i) *Listen graphischer Primitive* sind Objekte der Form

`[Mode = List, [Primitiv1, Primitiv2, ...] <, Options>]`

Die zur Verfügung stehenden Primitive sind Punkte, Polygonzüge sowie gefüllte Polygone, die mittels der MuPAD-Funktionen `point` bzw. `polygon` erzeugt werden. Aus solchen Primitiven können komplexere graphische Objekte zusammengesetzt werden.

Optionen werden als Gleichungen `OptionsName = Wert` übergeben. Die folgende Tabelle gibt einen Überblick über die zur Verfügung stehenden Optionen:

OptionsName	mögliche Werte	Standardwert
<i>Color</i>	<code>[Flat], [Flat, [r,g,b]], [Height], [Height, [r,g,b], [R,G,B]], [Function, f]</code>	<code>[Height]</code>
<i>LineStyle</i>	<code>SolidLines, DashedLines</code>	<code>SolidLines</code>
<i>LineWidth</i>	positive ganze Zahlen	1
<i>PointStyle</i>	<code>Circles, FilledCircles, FilledSquares, Squares</code>	<code>FilledSquares</code>
<i>PointWidth</i>	positive ganze Zahlen	30
<i>Title</i>	Zeichenketten	" "
<i>TitlePosition</i>	<code>[x, y]</code>	

Erläuterungen und Details zu jeder Option sind weiter unten aufgeführt.

☞ ii) *Parametrisierte Kurven* sind als Abbildung $u \mapsto [x(u), y(u), z(u)]$ mit Funktionen $x(u)$, $y(u)$, $z(u)$ definiert, welche die Koordinaten in Abhängigkeit eines Kurvenparameters u angeben. In `plot3d` ist eine solche Kurve als ein Objekt der folgenden Form anzugeben:

`[Mode=Curve, [x(u), y(u), z(u)], u = [umin, umax] <, Options>]`

Die Parametrisierung $x(u), y(u), z(u)$ besteht aus arithmetischen Ausdrücken in einer Unbestimmten u (ein Bezeichner). Sie dürfen neben dem Kurvenparameter keine weiteren symbolischen Parameter enthalten, die sich nicht in Gleitpunktzahlen konvertieren lassen. Der Bereich des Kurvenparameters u wird durch die Zahlen oder numerischen Ausdrücke u_{\min} und u_{\max} angegeben.

Liegt eine Parametrisierung durch benutzerdefinierte Funktionen vor, die nur numerische Argumente akzeptieren, so kann die Auswertung verzögert werden, indem $\text{hold}(x)(u), \text{hold}(y)(u), \text{hold}(z)(u)$ mit dem symbolischen Kurvenparameter u übergeben wird.

Optionen werden durch Gleichungen $\text{OptionsName} = \text{Wert}$ angegeben. Alle oben für Listen graphischer Primitive angegebenen Optionen sind zulässig. Für Kurven können darüberhinaus die folgenden weiteren Optionen verwendet werden:

OptionsName	zulässige Werte	Standardwert
<i>Grid</i>	[ganze Zahl]	[100]
<i>Smoothness</i>	[ganze Zahl]	[0]
<i>Style</i>	[Points], [Lines], [LinesPoints], [Impulses]	[Lines]

Details zu jeder Option sind weiter unten aufgeführt.

iii) *Parametrisierte Flächen* sind durch Abbildungen

$$(u, v) \mapsto [x(u, v), y(u, v), z(u, v)]$$

mit Ausdrücken $x(u, v), y(u, v), z(u, v)$ definiert, welche die Koordinaten als Funktionen zweier Flächenparameter u, v angeben. In `plot3d` ist eine solche Fläche als ein Objekt der folgenden Form anzugeben:

```
[Mode = Surface, [x(u, v), y(u, v), z(u, v)],  
u = [umin, umax], v = [vmin, vmax] <, Options>]
```

Die Parametrisierung $x(u, v), y(u, v), z(u, v)$ besteht aus arithmetischen Ausdrücken in zwei Unbestimmten u, v (Bezeichner). Sie dürfen neben den Flächenparametern keine weiteren symbolischen Parameter enthalten, die sich nicht in Gleitpunktzahlen konvertieren lassen. Der Bereich der Flächenparameter u und v wird durch die reellen Zahlen oder numerische Ausdrücke u_{\min}, u_{\max} , bzw. v_{\min}, v_{\max} angegeben.

Liegt eine Parametrisierung durch benutzerdefinierte Funktionen vor, die nur numerische Argumente akzeptieren, so kann die Auswertung verzögert werden, indem $\text{hold}(x)(u, v), \text{hold}(y)(u, v), \text{hold}(z)(u, v)$ mit den symbolischen Flächenparametern u, v übergeben wird.

Optionen werden durch Gleichungen `OptionsName = Wert` angegeben. Alle oben für Listen graphischer Primitive angegebenen Optionen sind zulässig. Für Flächen können darüberhinaus die folgenden weiteren Optionen verwendet werden:

	OptionsName	zulässige Werte	Standardwert
<i>Grid</i>		[ganze Zahl, ganze Zahl]	[20, 20]
<i>Smoothness</i>		[ganze Zahl, ganze Zahl]	[0, 0]
<i>Style</i>		[<i>Points</i>] [<i>WireFrame</i> , <i>Mesh</i>] [<i>WireFrame</i> , <i>ULine</i>] [<i>WireFrame</i> , <i>VLine</i>] [<i>HiddenLine</i> , <i>Mesh</i>] [<i>HiddenLine</i> , <i>ULine</i>] [<i>HiddenLine</i> , <i>VLine</i>] [<i>ColorPatches</i> , <i>Only</i>] [<i>ColorPatches</i> , <i>AndMesh</i>] [<i>ColorPatches</i> , <i>AndULine</i>] [<i>ColorPatches</i> , <i>AndVLine</i>] [<i>Transparent</i> , <i>Only</i>] [<i>Transparent</i> , <i>AndMesh</i>] [<i>Transparent</i> , <i>AndULine</i>] [<i>Transparent</i> , <i>AndVLine</i>]	[<i>ColorPatches</i> , <i>AndMesh</i>]

Details zu jeder Option sind weiter unten aufgeführt.

☞ Der Graph einer Funktion $f(x, y)$ kann als parametrisierte Fläche

```
[Mode = Surface, [x, y, f(x, y)], x = [xmin, xmax],
y = [ymin, ymax] <, Options>]:
```

erzeugt werden. Es wird jedoch empfohlen, stattdessen die auf Funktionsgraphen spezialisierten Routinen `plotfunc3d` bzw. `plot::Function3d` zu verwenden.

☞ MuPAD-Graphiken können in einer Vielzahl graphischer Formate gespeichert werden. Für die beiden MuPAD-spezifischen Formate *Ascii* und *Binary* kann dies über die Szeneoption *PlotDevice* direkt im Aufruf von `plot3d` geschehen. Details hierzu liefert `?plotOptions3d`.

Die Speicherung in graphischen Standardformaten wie z. B. *Postscript*, *JPEG*, *TIFF* etc. kann nicht direkt innerhalb einer MuPAD-Sitzung über ein Plot-Kommando geschehen. Man muss stattdessen die Oberfläche des graphischen Werkzeugs VCam interaktiv benutzen: innerhalb eines MuPAD Pro Notebooks wird diese Oberfläche durch einen Doppelklick auf die Graphik aktiviert. Durch Auswahl der Menüpunkte „Bearbeiten/Exportieren...“ wird die Dialogbox „Exportiere Graphik“ geöffnet, in der das gewünschte Format eingestellt werden kann.

Option **<Color = Wert>**:

☞ Diese Option legt die Farbe des Objekts fest. Zulässige Werte sind `[Flat]`, `[Flat, [r,g,b]]`, `[Height]`, `[Height, [r,g,b]]`, `[R,G,B]` und `[Function, f]`. Die Voreinstellung ist `Color = [Height]`.

- Mit `Color = [Flat]` wird das Objekt einheitlich in einer Farbe dargestellt. Diese wird automatisch gewählt.
- Mit `Color = [Flat, [r, g, b]]` wird das Objekt einheitlich in einer Farbe dargestellt. Die Werte `r`, `g`, `b` stellen den Rot-, Grün und Blauanteil gemäß des RGB-Farbmodells dar, sie müssen reelle numerische Werte zwischen 0 und 1 sein. Vordefinierter Farben werden durch MuPADs RGB-Datenstruktur zur Verfügung gestellt.
- Mit `Color = [Height]` wird ein von der y -Koordinate abhängender Farbverlauf benutzt. Die tatsächlichen Farben werden automatisch gewählt.
- Mit `Color = [Height, [r, g, b], [R, G, B]]` wird ein von der y -Koordinate abhängender Farbverlauf benutzt. Die Teile des Objektes mit kleinen y -Werten werden in der Farbe `[r, g, b]` dargestellt, die Teile mit großen y -Werten in der Farbe `[R, G, B]`. Dazwischen werden interpolierte Farbwerte benutzt.
- Mit `Color = [Function, f]` kann der Nutzer sein eigenes Farbschema implementieren. Der Parameter `f` muss eine MuPAD Prozedur sein, welche Farbwerte als Listen `[r, g, b]` produziert.
 - Innerhalb einer Liste graphischer Primitive muss die Prozedur `f` drei Parameter akzeptieren:

```
f := proc(x, y, z) begin ..; return([r, g, b]) end:
```

Während der numerischen Auswertung der Graphik wird diese Prozedur mit Argumenten `(x, y, z)` aus dem Bereich des Objekts aufgerufen.
Man beachte, dass Polygone mit jeweils einer einheitlichen Farbe gezeichnet werden.
 - Innerhalb eines Kurvenobjekts muss die Prozedur `f` vier Parameter akzeptieren:

```
f := proc(x, y, z, u) begin ..; return([r, g, b]) end:
```

Während der numerischen Auswertung der Graphik wird diese Prozedur mit den Argumenten `(x(u), y(u), z(u), u)` aufgerufen, wo `u` der Kurvenparameter ist und `x(u)`, `y(u)`, `z(u)` die entsprechenden Koordinatenwerte sind.
 - Innerhalb eines Flächenobjekts muss die Prozedur `f` fünf Parameter akzeptieren:

```
f := proc(x, y, z, u, v) begin ..; return([r, g, b]) end:
```

Während der numerischen Auswertung der Graphik wird diese Prozedur mit den Argumenten $(x(u,v), y(u,v), z(u,v), u, v)$ aufgerufen, wo u, v die Flächenparameter und $x(u,v), y(u,v), z(u,v)$ die entsprechenden Koordinatenwerte sind.

Wird die Farbfunktion f innerhalb einer Prozedur erzeugt, wobei f auf lokale Variablen der Prozedur zugreift, so muss die erzeugende Prozedur `option escape` verwenden.



Option `<Grid = [n] (für Kurven)>`:

- ☞ In Kurven legt diese Option die Anzahl n der Stützpunkte des Kurvenparameters fest. Zwischen diesen Punkten wird linear interpoliert. Die ganze Zahl n muss größer als 1 sein, die Voreinstellung ist `Grid = [100]`. Durch eine hohe Anzahl von Stützpunkten werden glatte Kurven erzeugt. Alternativ kann der *Smoothness*-Parameter heraufgesetzt werden.

Option `<Grid = [nu, nv] (für Flächen)>`:

- ☞ Bei Flächenobjekten legt diese Option die Anzahl nu, nv der Stützpunkte für die Flächenparameter u und v fest. Zwischen diesen Punkten wird linear interpoliert. Die ganzen Zahlen nu, nv müssen größer als 1 sein, die Voreinstellung ist `Grid = [20, 20]`. Durch eine hohe Anzahl von Stützpunkten werden glatte Flächen erzeugt. Alternativ kann der *Smoothness*-Parameter heraufgesetzt werden.

Option `<LineStyle = Wert>`:

- ☞ Diese Option legt den Stil fest, mit der Linien innerhalb des Objekts dargestellt werden. Zulässige Werte sind *SolidLines* und *DashedLines*: es werden entweder durchgezogene oder gestrichelte Linien gezeichnet. Die Voreinstellung ist `LineStyle = SolidLines`.

Option `<LineWidth = n>`:

- ☞ Diese Option legt die Breite der Linien innerhalb des Objekts fest. Zulässige Werte für n sind nicht-negative ganze Zahlen, die Voreinstellung ist `LineWidth = 1`.

Option <PointSize = Wert>:

- ☞ Diese Option legt den Stil fest, mit der Punkte dargestellt werden. Zulässige Werte sind *Circles*, *Squares*, *FilledCircles* und *FilledSquares*. Die Voreinstellung ist *PointSize = FilledSquares*.

Option <PointWidth = n>:

- ☞ Diese Option legt die Größe von Punkten fest. Zulässige Werte für n sind positive ganze Zahlen, die Voreinstellung ist *PointWidth = 30*.

Option <Smoothness = [n] (für Kurven)>:

- ☞ In Kurven legt diese Option die Anzahl von Zwischenpunkten fest, die zwischen jeweils zwei der durch die *Grid*-Option festgelegten Stützpunkte des Kurvenparameters eingeschoben werden. Linien werden als verbindende Geradenstücke zwischen jeweils zwei dieser Zwischenpunkte dargestellt. Hohe Werte von n liefern damit glatte Kurven. Zulässig sind ganze Zahlen zwischen 0 und 20, die Voreinstellung ist *Smoothness = [0]*.

Option <Smoothness = [nu, nv] (für Flächen)>:

- ☞ Bei Flächenobjekten legt diese Option die Anzahl nu, nv von Zwischenpunkten fest, die zwischen jeweils zwei der durch die *Grid*-Option festgelegten Stützpunkte der Flächenparameter u, v eingeschoben werden. Zwischen diesen Punkten wird linear interpoliert. Zulässige Werte für nu, nv sind ganze Zahlen zwischen 0 und 20, die Voreinstellung ist *Smoothness = [0, 0]*. Durch hohe Werte nu, nv werden glatte Flächen erzeugt.

Option <Style = Wert (für Kurven)>:

- ☞ Diese Option legt den Stil fest, in dem Kurven dargestellt werden. Zulässige Werte sind *[Points]*, *[Lines]*, *[LinesPoints]* und *[Impulses]*. Die Voreinstellung ist *Style = [Lines]*.
- Mit *Style = [Points]* werden nur die Stützpunkte dargestellt.
 - Mit *Style = [Lines]* wird die Kurve als Menge von Geradensegmenten zwischen den Stützpunkten dargestellt.

- Mit *Style = [LinesPoints]* werden sowohl die Stützpunkte als auch die verbindenden Geradenstücke dargestellt.
- Mit *Style = [Impulses]* wird die Kurve wie ein „Histogramm“ dargestellt: es werden vertikale Linien von der Unterkante der Szene zu den Stützpunkten gezeichnet.

Option <Style = Wert (für Flächen)>:

☞ Diese Option legt den Stil fest, in dem parametrisierte Flächen dargestellt werden. Die Voreinstellung ist *Style = [ColorPatches, AndMesh]*.

- Mit *Style = [Points]* werden nur die durch die *Grid*-Option festgelegten Punkte gezeichnet.
- Mit *Style = [WireFrame, Mesh]* wird ein „Drahtmodell“ aus den Parameterlinien beider Flächenparameter gezeichnet.
- Mit *Style = [WireFrame, ULine]* wird ein „Drahtmodell“ aus den Parameterlinien des Flächenparameters *u* gezeichnet.
- Mit *Style = [WireFrame, VLine]* wird ein „Drahtmodell“ aus den Parameterlinien des Flächenparameters *v* gezeichnet.
- Mit *Style = [HiddenLine, Mesh]* wird die Fläche als undurchsichtiges Objekt mit den Parameterlinien beider Flächenparameter dargestellt.
- Mit *Style = [HiddenLine, ULine]* wird die Fläche als undurchsichtiges Objekt mit den Parameterlinien des Flächenparameters *u* dargestellt.
- Mit *Style = [HiddenLine, VLine]* wird die Fläche als undurchsichtiges Objekt mit den Parameterlinien des Flächenparameters *v* dargestellt.
- Mit *Style = [ColorPatches, Only]* wird die Fläche als eingefärbtes undurchsichtiges Objekt ohne Parameterlinien dargestellt.
- Mit *Style = [ColorPatches, AndMesh]* wird die Fläche als eingefärbtes undurchsichtiges Objekt mit den Parameterlinien beider Flächenparameter dargestellt.
- Mit *Style = [ColorPatches, AndULine]* wird die Fläche als eingefärbtes undurchsichtiges Objekt mit den Parameterlinien des Flächenparameters *u* dargestellt.
- Mit *Style = [ColorPatches, AndVLine]* wird die Fläche als eingefärbtes undurchsichtiges Objekt mit den Parameterlinien des Flächenparameters *v* dargestellt.
- Mit *Style = [Transparent, Only]* werden die Flächensegmente mit Mustern gefüllt, die Transparenz simulieren. Es werden keine Parameterlinien gezeichnet.

- Mit *Style* = [*Transparent*, *AndMesh*] werden die Flächen-segmente mit Mustern gefüllt, die Transparenz simulieren. Die Parameterlinien beider Flächenparameter werden gezeichnet.
- Mit *Style* = [*Transparent*, *AndULine*] werden die Flächen-segmente mit Mustern gefüllt, die Transparenz simulieren. Die Parameterlinien des Flächenparameter *u* werden gezeichnet.
- Mit *Style* = [*Transparent*, *AndVLine*] werden die Flächen-segmente mit Mustern gefüllt, die Transparenz simulieren. Die Parameterlinien des Flächenparameter *v* werden gezeichnet.

Option **<Title = Titel>**:

- ☞ Diese Option haftet eine durch die Zeichenkette *Titel* gegebene Überschrift an das Objekt. Die Voreinstellung ist die leere Zeichenkette *Title* = "", d. h., keine Überschrift.

Option **<TitlePosition = [x, y]>**:

- ☞ Diese Option legt die Platzierung der Objektüberschrift fest. Zulässige Werte sind Listen [*x*, *y*] mit numerischen Werten *x*, *y* zwischen 0 und 10. Die Position [0, 0] ist die linke obere Ecke der Szene, [10, 10] ist die rechte untere Ecke. Man beachte, dass die Platzierung relativ zur gesamten Szene spezifiziert wird. Die Platzierungen der Überschriften mehrerer Objekte sollte demgemäß unterschiedlich sein, um Überlappungen zu vermeiden.
- ☞ Objektüberschriften können innerhalb des angezeigten Bildes mit der Maus bewegt und so interaktiv an eine geeignete Stelle verschoben werden.

Beispiel 1. Die Darstellung graphischer Primitive wird demonstriert. Zunächst werden mittels *point* und *polygon* drei Punkte, eine Linie und ein gefülltes Dreieck erzeugt:

```
>> p1 := point(0, 0, 0, Color = RGB::Red):
    p2 := point(0, 1, 1/2, Color = RGB::Green):
    p3 := point(-1, 1, 1, Color = RGB::Blue):
    line := polygon(point(0, 0, 0), point(0, 1, 1/2),
                    point(-1, 1, 1), Closed = TRUE,
                    Color = RGB::Black):
    triangle := polygon(point(0, 0, 0), point(-1, 0.2, 0.4),
                       point(-1, 1, 0), Closed = TRUE,
                       Filled = TRUE, Color = RGB::Antique):
```

Diese werden zu einem graphischen Objekt zusammengefasst:

```
>> object := [Mode = List, [p1, p2, p3, line, triangle]]:
```

Das Objekt wird dargestellt:

```
>> plot3d(BackGround = RGB::White, ForeGround = RGB::Black,  
          PointWidth = 70, PointStyle = FilledCircles,  
          Axes = Box, object)
```

```
>> delete p1, p2, p3, line, triangle, object:
```

Beispiel 2. Es werden Raumkurven gezeichnet. Das folgende Bild demonstriert diverse Stile:

```
>> plot3d(Axes = Box, Ticks = 0,  
          BackGround = RGB::White, ForeGround = RGB::Black,  
          [Mode = Curve, [u, -PI, cos(u)], u = [-PI, PI],  
            Grid = [40], Style = [Points], PointWidth = 40  
          ],  
          [Mode = Curve, [u, -PI/3, cos(u)], u = [-PI, PI],  
            Grid = [40], Style = [Lines]  
          ],  
          [Mode = Curve, [u, PI/3, cos(u)], u = [-PI, PI],  
            Grid = [40], Style = [LinesPoints], PointWidth = 30  
          ],  
          [Mode = Curve, [u, PI, cos(u)], u = [-PI, PI],  
            Grid = [40], Style = [Impulses]  
          ]):
```

Der folgende Befehl zeichnet einen „histogrammartigen“ Graphen der Kosinus-Funktion über dem Einheitskreis der x - y -Ebene:

```
>> plot3d(Axes = Box, Ticks = 5, CameraPoint = [20, -10, 30],  
          BackGround = RGB::White, ForeGround = RGB::Black,  
          Labeling = TRUE, Labels = ["x", "y", "z"],  
          Title = "A curve in space",  
          [Mode = Curve, [cos(u), sin(u), sin(3*u)], u = [0, 2*PI],  
            Grid = [200], Style = [Impulses]  
          ])
```

Der folgende Befehl zeichnet eine Spirale auf der Einheitskugel:

```
>> plot3d(Axes = Box, Ticks = 0, Scaling = Constrained,  
          Title = "spiral", TitlePosition = Below,  
          [Mode = Curve,  
            [cos(12*u*PI)*sin(u*PI),
```

```

        sin(12*u*PI)*sin(u*PI),
        cos(u*PI)],
    u = [0, 1], Grid = [50], Smoothness = [5]
])

```

Beispiel 3. Flächen werden demonstriert. Der nächste Befehl erzeugt Kugeln vom Radius 1, die über die üblichen Polarkoordinaten parametrisiert werden. Diverse Flächenstile werden demonstriert:

```

>> plot3d(Axes = Box, Ticks = 0, Scaling = Constrained,
    BackGround = RGB::White, ForeGround = RGB::Black,
    CameraPoint = [6, -21, 8],
    [Mode = Surface,
        [-2.5 + sin(u)*cos(v), sin(u)*sin(v), cos(u)],
        u = [0, PI], v = [0, 2*PI],
        Grid = [20, 20], Smoothness = [0, 0],
        Style = [HiddenLine, Mesh]
    ],
    [Mode = Surface,
        [sin(u)*cos(v), sin(u)*sin(v), cos(u)],
        u = [0, PI], v = [0, 2*PI],
        Grid = [15, 30], Smoothness = [0, 0],
        Style = [ColorPatches, AndULine]
    ],
    [Mode = Surface,
        [2.5 + sin(u)*cos(v), sin(u)*sin(v), cos(u)],
        u = [0, PI], v = [0, 2*PI],
        Grid = [10, 10], Smoothness = [0, 0],
        Style = [Transparent, AndVLine]
    ])

```

Der Effekt der Optionen *Grid* und *Smoothness* wird an in der *x-y*-Ebene liegenden Kreisscheiben demonstriert:

```

>> plot3d(Axes = None, Scaling = Constrained,
    BackGround = RGB::White, ForeGround = RGB::Black,
    CameraPoint = [0, -1, 20],
    [Mode = Surface, [-2.5 + v*sin(u), v*cos(u), 0],
        u = [-PI, PI], v = [0, 1], Style = [WireFrame, Mesh],
        Grid = [ 6,  6], Smoothness = [0, 0]
    ],
    [Mode = Surface, [v*sin(u), v*cos(u), 0],
        u = [-PI, PI], v = [0, 1], Style = [WireFrame, Mesh],
        Grid = [ 6,  6], Smoothness = [3, 2]
    ],
    [Mode = Surface, [2.5 + v*sin(u), v*cos(u), 0],

```

```

    u = [-PI, PI], v = [0, 1], Style = [WireFrame, Mesh],
    Grid = [20, 10], Smoothness = [0, 0]
  )

```

Ein Funktionsgraph wird als parametrisierte Kurve gezeichnet:

```

>> plot3d(Axes = Box, Ticks = 8,
  BackGround = RGB::White, ForeGround = RGB::Black,
  Title = "Plot of  $\sin(u^2 + v^2)$ ", TitlePosition = Below,
  [Mode = Surface, [u, v,  $\sin(u^2 + v^2)$ ],
    u = [0, PI], v = [0, PI],
    Grid = [30, 30], Style = [HiddenLine, Mesh]
  ])

```

Mehrere Objekte unterschiedlichen Typs werden zu einer Szene zusammengefasst:

```

>> plot3d(Axes = None, Scaling = Constrained,
  BackGround = RGB::White, ForeGround = RGB::Black,
  Title = "Three surfaces and a curve",
  TitlePosition = Below,
  CameraPoint = [13, -24, 20],
  [Mode = Surface,
    [(4 + cos(v))*cos(u), (4 + cos(v))*sin(u), sin(v)],
    u = [0, 2*PI], v = [0, 2*PI],
    Grid = [20, 20], Smoothness = [2, 0],
    Style = [HiddenLine, Mesh]
  ],
  [Mode = Surface,
    [2*cos(u)*sin(v), 2*sin(u)*sin(v), 2*cos(v)],
    u = [0, 2*PI], v = [0, PI],
    Grid = [10, 10], Smoothness = [2, 2],
    Style = [ColorPatches, AndMesh]
  ],
  [Mode = Surface, [u, v, -3], u = [-5, 5], v = [-
5, 5],
    Grid = [5, 5], Smoothness = [0, 0],
    Style = [ColorPatches, Only]
  ],
  [Mode = Curve,
    [6*cos(12*u)*sin(u), 6*sin(12*u)*sin(u), 6*cos(u)],
    u = [0, PI], Grid = [50], Smoothness = [5],
    Title = "spiral"
  ])

```

Beispiel 4. Dieses Beispiel demonstriert selbstdefinierte Farbfunktionen. Die folgende Funktion liefert zulässige RGB-Werte zwischen 0 und 1 für Objekte mit Koordinaten $x, y, z \in [-1, 1]$:

```
>> myColor := (x, y, z, u, v) ->
              [(abs(x) + 1)/2, abs(x - y)/(3 + z), abs(y)]:
```

Ein Hyperboloid über dem Einheitsquadrat soll durch diese Funktion gefärbt werden. Dazu wird der Graph der Funktion $(x, y) \mapsto x^2 - y^2$ als parametrisierte Fläche gezeichnet:

```
>> plot3d(Axes = Box,
           BackGround = RGB::White, ForeGround = RGB::Black,
           [Mode = Surface, [x, y, x^2 - y^2],
            x = [-1, 1], y = [-1, 1],
            Grid = [15, 15], Smoothness = [3, 3],
            Style = [ColorPatches, AndMesh],
            Color = [Function, myColor]
           ])

>> delete myColor:
```

Änderungen:

- ⌘ Die Funktionalität der Szeneoption *Ticks* wurde erweitert.
- ⌘ Die Voreinstellungen diverser Optionen wurden geändert.
- ⌘ Überschriften der Szene sowie Objektüberschriften können nun mit der Maus verschoben werden.

plotfunc2d – 2-dimensionale Funktionsgraphen

`plotfunc2d(f1, f2, ...)` erzeugt 2-dimensionale Graphen der univariaten Funktionen f_1, f_2 etc.

Aufruf(e):

- ⌘ `plotfunc2d(<SceneOptions,> f1, f2, ... <, Grid = n>)`
- ⌘ `plotfunc2d(<SceneOptions,> f1, f2, ..., x = xmin..xmax <, Grid = n>)`
- ⌘ `plotfunc2d(<SceneOptions,> f1, f2, ..., x = xmin..xmax, y = ymin..ymax <, Grid = n>)`

Parameter:

- `f1, f1, ...` — die Funktionen: arithmetische Ausdrücke oder `piecewise`-Objekte in einer Unbestimmten `x`
- `x` — die horizontale Koordinate: ein Bezeichner
- `xmin, xmax` — der horizontale Sichtbarkeitsbereich: endliche reelle numerische Ausdrücke
- `y` — ein beliebiger Name für die vertikale Koordinate: ein Bezeichner. Dieser Name wird zur Beschriftung der y -Achse verwendet.
- `ymin, ymax` — der vertikale Sichtbarkeitsbereich: endliche reelle numerische Ausdrücke

Optionen:

- `SceneOptions` — eine Folge von Szeneoptionen, die das allgemeine Aussehen der graphischen Szene bestimmen. Details sind mit `?plotOptions2d` anzufordern.
- `Grid = n` — legt die Anzahl der Stützpunkte der Graphik fest. Die ganze Zahl `n` muss mindestens 2 sein, die Voreinstellung ist `Grid = 100`.

Rückgabewert: MuPADs Graphik-Werkzeug wird aufgerufen, um das angeforderte Bild anzuzeigen. Das `null()`-Objekt wird an die MuPAD-Sitzung zurückgeliefert.

Verwandte Funktionen: `plot`, `plot::Function2d`, `plot2d`, `plot3d`, `plotfunc3d`

Details:

- ☞ Die Funktionen dürfen neben der horizontalen Koordinate keine symbolischen Parameter enthalten, die nicht in Gleitpunktzahlen konvertiert werden können.
- ☞ Wird kein horizontaler Bereich angegeben, so wird die Voreinstellung `x = -5..5` benutzt.
- ☞ Bei Vorgabe eines vertikalen Bereichs `y = ymin..ymax` werden nur Funktionswerte zwischen `ymin` und `ymax` gezeichnet. Der Name `y` der vertikalen Koordinate ist beliebig: jeder Bezeichner kann benutzt werden.
- ☞ Nicht-reelle Funktionswerte werden nicht gezeichnet. Siehe Beispiel ??.
- ☞ Funktionen mit Singularitäten werden verarbeitet. Siehe Beispiel ??.
- ☞ Unstetigkeiten sowie stückweise definierte Funktionen werden verarbeitet. Siehe Beispiele ??, ??.

- ☞ Der Graph einer Funktion $f(x)$ kann auch über `plot2d` als parametrisierte Kurve

```
[Mode = Curve, [x, f(x)], x = [xmin, xmax] <, Options>]
```

gezeichnet werden. Hierdurch können Plotbereiche, Farb- und Stiloptionen für jede Funktion getrennt spezifiziert werden. Siehe `plot2d` für Details.

- ☞ Die Bibliothek `plot` stellt die Routine `plot::Function2d` zur Verfügung, mit der ein Funktionsgraph als graphisches Primitiv erzeugt und mit anderen graphischen Objekten kombiniert werden kann.

- ☞ MuPAD-Graphiken können in einer Vielzahl graphischer Formate gespeichert werden. Für die beiden MuPAD-spezifischen Formate *Ascii* und *Binary* kann dies über die Szeneoption *PlotDevice* direkt im Aufruf von `plot2d` geschehen. Details hierzu liefert `?plotOptions2d`.

Die Speicherung in graphischen Standardformaten wie z. B. *Postscript*, *JPEG*, *TIFF* etc. kann nicht direkt innerhalb einer MuPAD-Sitzung über ein Plot-Kommando geschehen. Man muss stattdessen die Oberfläche des graphischen Werkzeugs VCam interaktiv benutzen: innerhalb eines MuPAD Pro Notebooks wird diese Oberfläche durch einen Doppelklick auf die Graphik aktiviert. Durch Auswahl der Menüpunkte „Bearbeiten/Exportieren...“ wird die Dialogbox „Exportiere Graphik“ geöffnet, in der das gewünschte Format eingestellt werden kann.

Option **<Grid = n>**:

- ☞ Diese Option legt die Anzahl n der Stützpunkte und damit der Funktionsauswertungen fest. Zwischen diesen Punkten wird der Graph linear interpoliert. Die ganze Zahl n muss mindestens 2 sein, die Voreinstellung ist *Grid* = 100. Durch eine hohe Stützpunktzahl werden glatte Graphen erzeugt.

Beispiel 1. Der folgende Aufruf zeichnet die Graphen der Sinus- und der Kosinus-Funktion über dem Intervall $[-\pi, \pi]$:

```
>> plotfunc2d(sin(x), cos(x), x = -PI..PI):
```

Beispiel 2. Nur reelle Funktionswerte werden dargestellt:

```
>> plotfunc2d(sqrt(1 - x), sqrt(x), x = -2..2):
```

Beispiel 3. Die folgenden Funktionen haben in dem angegebenen Intervall Singularitäten:

```
>> plotfunc2d(x/(x^3 - 4*x), x = -5..5):  
>> plotfunc2d(1/sin(x), tan(x), x = 0..2*PI):
```

Beispiel 4. Ein vertikaler Sichtbarkeitsbereich wird spezifiziert:

```
>> plotfunc2d(tan(x), x = -3..3, y = -10..10):
```

Beispiel 5. Die folgende Funktion hat eine Sprungstelle:

```
>> plotfunc2d((x^2 - x)/(2*abs(x - 1)), x = -3..3, y = -3..3)
```

Beispiel 6. Stückweise definierte Funktionen werden verarbeitet:

```
>> f := piecewise([x < 1, -x^2 + 1], [x >= 1, x]):  
    plotfunc2d(BackGround = RGB::White,  
               ForeGround = RGB::Black,  
               GridLines = Automatic,  
               Ticks = [Steps = 1, Steps = 1],  
               f(x), x = -3..3, y = -3..3)  
  
>> f := piecewise([x <= 0, x], [x > 0, 1/x]):  
    plotfunc2d(BackGround = RGB::White,  
               ForeGround = RGB::Black,  
               GridLines = Automatic,  
               Ticks = [Steps = 1, Steps = 1],  
               f(x), x = -3..3, y = -3..3)  
  
>> delete f:
```

Beispiel 7. Mittels der Szeneoption *AxesScaling* wird ein einfach-logarithmischer Plot erzeugt:

```
>> plotfunc2d(AxesScaling = [Lin, Log], x^2, x^3, x = 1/10..10^3):
```

Verschiedene weitere Szeneoptionen werden in einem doppelt-logarithmischen Plot demonstriert:

```
>> plotfunc2d(Axes = Box,
               AxesScaling = [Log, Log],
               Discont = FALSE,
               BackGround = RGB::White,
               ForeGround = RGB::Black,
               GridLines = Automatic,
               GridLinesStyle = SolidLines,
               GridLinesColor = RGB::Gray,
               Ticks = [[10^i $ i = -1..3], [10^i $ i = -3..9]],
               x^2, x^3/(1 + x^(1/2)), x^3, x = 1/10..10^3):
```

Änderungen:

- ⌘ `plotfunc2d` hieß früher `plotfunc`.
 - ⌘ Die neuen Szeneoptionen *Discont*, *GridLines*, *GridLinesColor*, *GridLinesStyle*, *GridLinesWidth*, *RealValuesOnly* und *ViewingBox* wurden eingeführt. Die Funktionalität der Szeneoption *Ticks* wurde erweitert.
 - ⌘ Die Voreinstellungen diverser Optionen wurden geändert.
 - ⌘ Überschriften der Szene sowie Objektüberschriften können nun mit der Maus verschoben werden.
-

`plotfunc3d` – 3-dimensionale Funktionsgraphen

`plotfunc3d(f1, f2, ...)` erzeugt 3-dimensionale Graphen der bivariaten Funktionen *f1*, *f2* etc.

Aufruf(e):

- ⌘ `plotfunc3d(<SceneOptions,> f1, f2, ... <, Grid = [nx, ny]>)`
- ⌘ `plotfunc3d(<SceneOptions,> f1, f2, ..., x = xmin..xmax <, Grid = [nx, ny]>)`
- ⌘ `plotfunc3d(<SceneOptions,> f1, f2, ..., x = xmin..xmax, y = ymin..ymax <, Grid = [nx, ny]>)`

Parameter:

- f_1, f_1, \dots — die Funktionen: arithmetische Ausdrücke oder *piecewise*-Objekte in zwei Unbestimmten x, y
- x, y — die unabhängigen Variablen: Bezeichner
- x_{\min}, x_{\max} — der x -Bereich: endliche reelle numerische Ausdrücke
- y_{\min}, y_{\max} — der y -Bereich: endliche reelle numerische Ausdrücke

Optionen:

- SceneOptions* — eine Folge von Szeneoptionen, die das allgemeine Aussehen der graphischen Szene bestimmen. Details sind mit `?plotOptions3d` anzufordern.
- Grid* = $[n_x, n_y]$ — legt die Anzahl der Stützpunkte der Graphik fest. Die ganzen Zahlen n_x, n_y müssen größer als 1 sein, die Voreinstellung ist *Grid* = $[20, 20]$.

Rückgabewert: MuPADs Graphik-Werkzeug wird aufgerufen, um das angeforderte Bild anzuzeigen. Das `null()` Objekt wird an die MuPAD-Sitzung zurückgeliefert.

Verwandte Funktionen: `plot`, `plot::Function3d`, `plot2d`, `plot3d`, `plotfunc2d`

Details:

- ☞ Die Funktionen dürfen neben den Unbestimmten x und y keine symbolischen Parameter enthalten, die nicht in Gleitpunktzahlen konvertiert werden können.
- ☞ Werden keine Bereiche angegeben, so werden die Voreinstellungen $x = -5..5$ und $y = -5..5$ benutzt.
- ☞ Stückweise definierte Funktionen werden verarbeitet. Siehe Beispiel ??.
- ☞ Der Graph einer Funktion $f(x, y)$ kann auch über `plot3d` als parametrisierte Fläche

```
[Mode = Surface, [x, y, f(x, y)], x = [xmin, xmax],
 y = [ymin, ymax] <, Options>]:
```

gezeichnet werden. Hierdurch können Plotbereiche, Farb- und Stiloptionen etc. für jede Funktion getrennt spezifiziert werden. Siehe `plot3d` für Details.

☞ Die Bibliothek `plot` stellt die Routine `plot::Function3d` zur Verfügung, mit der ein Funktionsgraph als graphisches Primitiv erzeugt und mit anderen graphischen Objekten kombiniert werden kann.

☞ MuPAD-Graphiken können in einer Vielzahl graphischer Formate gespeichert werden. Für die beiden MuPAD-spezifischen Formate *Ascii* und *Binary* kann dies über die *PlotDevice*-Option von `plotfunc3d` geschehen. Details hierzu finden sich auf der Hilfeseite `plotOptions3d`.

Die Speicherung in graphischen Standardformaten wie z. B. *Postscript*, *JPEG*, *TIFF* etc. kann nicht direkt innerhalb einer MuPAD-Sitzung über ein Plot-Kommando geschehen. Man muss stattdessen die Oberfläche des graphischen Werkzeugs VCam interaktiv benutzen: innerhalb eines MuPAD Pro Notebooks wird diese Oberfläche durch einen Doppelklick auf die Graphik aktiviert. Durch Auswahl der Menüpunkte „Bearbeiten/Exportieren...“ wird die Dialogbox „Exportiere Graphik“ geöffnet, in der das gewünschte Format eingestellt werden kann.

Option `<Grid = [nx, ny]>`:

☞ Diese Option legt die Anzahl `nx` bzw. `ny` der Stützpunkte in x bzw. y -Richtung fest. Zwischen diesen Punkten wird der Graph linear interpoliert. Die ganzen Zahlen `nx`, `ny` müssen größer als 1 sein, die Voreinstellung ist `Grid = [20, 20]`. Durch hohe Stützpunktzahlen werden glatte Graphen erzeugt.

Beispiel 1. Der folgende Aufruf zeichnet 2 Funktionen über dem Einheitsquadrat:

```
>> plotfunc3d(BackGround = RGB::White,
               ForeGround = RGB::Black,
               Axes = Box,
               sin(x^2 + y^2), cos(x^2 - y^2),
               x = 0..1, y = 0..1):
```

Beispiel 2. Der Effekt einiger Szeneoptionen wird demonstriert:

```
>> plotfunc3d(Axes = Box, Ticks = 5,
               abs(x + I*y), x = -1..1, y = -1..1)

>> plotfunc3d(Arrows = FALSE, Axes = Corner, Ticks = 8,
               Grid = [40, 40], CameraPoint = [10, -5, 15],
               abs(x + I*y), x = -1..1, y = -1..1)
```

Beispiel 3. Im Gegensatz zu `plotfunc2d` führen nicht-reelle Funktionswerte in 3-dimensionalen Plots zu Fehlern:

```
>> plotfunc3d(sqrt(1 - x^2 - y^2), x = -1..1, y = -1..1):  
  
Error: Plot function(s) must return real numbers.  
      Type of the returned value is DOM_COMPLEX;  
      during evaluation of 'plot3d'
```

Beispiel 4. Stückweise definierte Funktionen werden verarbeitet:

```
>> f := piecewise([x < y, -x^2 + 1], [x >= y, 1 - y^2]):  
      plotfunc3d(BackGround = RGB::White,  
                 ForeGround = RGB::Black,  
                 Ticks = [Steps = 1, Steps = 1, Steps = 1],  
                 f(x, y), x = -3..3, y = -3..3)  
  
>> delete f:
```

Beispiel 5. Mittels der Szeneoption *AxesScaling* wird ein einfach-logarithmischer Plot erzeugt:

```
>> plotfunc3d(AxesScaling = [Lin, Lin, Log],  
              exp(x + y^2), x = 0..10, y = 0..10):
```

Änderungen:

- ⌘ Die Funktionalität der Szeneoption *Ticks* wurde erweitert.
- ⌘ Die Voreinstellungen diverser Optionen wurden geändert.
- ⌘ Überschriften der Szene sowie Objektüberschriften können nun mit der Maus verschoben werden.

plotOptions2d – Szeneoptionen für 2-dimensionale Graphiken

Diese Seite beschreibt die Szeneoptionen, die den Routinen `plot2d`, `plotfunc2d`, `plot::Scene` und `plot` zur Erzeugung 2-dimensionaler Graphiken übergeben werden können. Szeneoptionen sind Attribute, die das allgemeine Aussehen einer graphischen Szene bestimmen, z. B., Hintergrundfarbe, Titel, Stil der Achsen etc.

Aufruf(e):

```

# plot2d(<SceneOpt1, SceneOpt2, ...>, graphical ob-
      jects)
# plotfunc2d(<SceneOpt1, SceneOpt2, ...>, graphical
      objects)
# plot::Scene(graphical objects, <SceneOpt1, Sce-
      neOpt2, ...>)
# plot(graphical objects, <SceneOpt1, SceneOpt2, ...>)

```

Verwandte Funktionen: `plot`, `plot::Scene`, `plot`, `plot2d`,
`plotfunc2d`, `plot3d`, `plotfunc3d`, `plotOptions3d`

Parameter:

`graphical objects` — siehe die Hilfeseiten von `plot2d`,
`plotfunc2d`, `plot::Scene` und `plot`

Optionen:

`SceneOpt1, SceneOpt2, ...` — Szeneoptionen: Gleichungen der
Form `OptionsName = Wert`.

OptionsName	zulässige Werte	Standardwert
<i>Arrows</i>	TRUE, FALSE	FALSE
<i>Axes</i>	<i>Box, Corner, None, Origin</i>	<i>Origin</i>
<i>AxesOrigin</i>	<i>Automatic, [x0, y0]</i>	<i>Automatic</i>
<i>AxesScaling</i>	<i>[Lin/Log, Lin/Log]</i>	<i>[Lin, Lin]</i>
<i>BackGround</i>	<i>[r, g, b]</i>	RGB::White
<i>Discont</i>	TRUE, FALSE	FALSE (plot2d) TRUE (plotfunc2d) FALSE (plot) FALSE (plot::Scene)
<i>FontFamily</i>	"helvetica", "lucida", ..	"helvetica"
<i>FontSize</i>	positive ganze Zahlen	8
<i>FontStyle</i>	"bold", ..	"bold"
<i>ForeGround</i>	<i>[r, g, b]</i>	RGB::Black
<i>GridLines</i>	<i>Automatic, None</i> oder <i>[xWert, yWert]</i> . Zulässige Werte für <i>xWert, yWert</i> sind <i>Automatic</i> , ganze Zahlen, <i>Steps = d</i> oder <i>Steps = [d, n]</i> .	<i>None</i>
<i>GridLinesColor</i>	<i>[r, g, b]</i>	RGB::Gray
<i>GridLinesWidth</i>	positive ganze Zahlen	5
<i>GridLinesStyle</i>	<i>SolidLines, DashedLines</i>	<i>DashedLines</i>
<i>Labeling</i>	TRUE, FALSE	TRUE
<i>Labels</i>	<i>[string, string]</i>	<i>["x", "y"]</i>
<i>LineStyle</i>	<i>SolidLines, DashedLines</i>	<i>SolidLines</i>

OptionsName	zulässige Werte	Standardwert
<i>LineWidth</i>	positive ganze Zahlen	1
<i>PlotDevice</i>	<i>Screen</i> , "Dateiname", ["Dateiname", <i>Ascii</i>], ["Dateiname", <i>Binary</i>]	<i>Screen</i>
<i>PointStyle</i>	<i>Circles</i> , <i>FilledCircles</i> , <i>FilledSquares</i> , <i>Squares</i>	<i>FilledSquares</i>
<i>PointWidth</i>	positive ganze Zahlen	30
<i>RealValuesOnly</i>	TRUE, FALSE	FALSE (plot2d) TRUE (plotfunc2d) FALSE (plot::Scene) FALSE (plot)
<i>Scaling</i>	<i>Constrained</i> , <i>UnConstrained</i>	<i>UnConstrained</i>
<i>Ticks</i>	<i>Automatic</i> , <i>None</i> , eine ganze Zahl oder [xWert, yWert]. Zulässige Werte für xWert, yWert sind <i>Automatic</i> , ganze Zahlen, <i>Steps</i> = d, <i>Steps</i> = [d, n] oder Listen selbstdefinierter Marken.	<i>Automatic</i>
<i>Title</i>	Zeichenketten	" " (plot2d) "f(x)" (plotfunc2d) " " (plot::Scene) " " (plot)
<i>TitlePosition</i>	<i>Above</i> , <i>Below</i> , [x, y]	<i>Above</i>
<i>ViewingBox</i>	<i>Automatic</i> oder [xWert, yWert]. Zulässige Werte für xWert, yWert sind <i>Automatic</i> oder ein Bereich a..b.	<i>Automatic</i>

Option <Arrows = Wert>:

- ☞ Diese Option legt fest, ob die Koordinatenachsen mit oder ohne Pfeilspitzen dargestellt werden. Zulässige Werte sind TRUE oder FALSE, die Voreinstellung ist *Arrows* = FALSE. Diese Option wird bei *Axes* = *None* oder *Axes* = *Box* ignoriert.

Option <Axes = Wert>:

- ☞ Diese Option legt den Typ der Koordinatenachsen fest. Zulässige Werte sind *Box*, *Corner*, *None* und *Origin*, die Voreinstellung ist *Axes* = *Origin*.

- Mit *Axes* = *Box* wird ein Rahmen um die Szene gezeichnet.

- Mit *Axes = Corner* wird die *x*-Achse unterhalb der Szene platziert, die *y*-Achse links. Die Achsen kreuzen sich in der linken unteren Ecke.
- Mit *Axes = None* werden keine Achsen gezeichnet.
- Mit *Axes = Origin* wird ein Koordinatenkreuz gezeichnet, das in dem durch *AxesOrigin* gegebenen Punkt zentriert ist.

Option <AxesOrigin = Wert>:

☞ Diese Option setzt den Punkt, an dem das Koordinatenkreuz zentriert ist. Zulässige Werte sind *Automatic* und $[x_0, y_0]$, die Voreinstellung ist *AxesOrigin = Automatic*.

- Mit *AxesOrigin = Automatic* kreuzen sich die Achsen im mathematischen Ursprung (0,0) der *x-y*-Ebene, falls dieser innerhalb der Szene liegt. Ist dies nicht der Fall, so kreuzen sich die Achsen in derjenigen Ecke des Bildes, die dem Ursprung am nächsten liegt.
- Mit *AxesOrigin = $[x_0, y_0]$* kreuzen sich die Achsen im angegebenen Punkt. Zulässige Werte für die Koordinaten sind reelle numerische Ausdrücke sowie die Bezeichner *XMin*, *XMax*, *YMin*, *YMax*. Diese repräsentieren die extremalen Koordinaten der Szene, welche während der numerischen Auswertung der Graphik bestimmt werden.

Option <AxesScaling = [xWert, yWert]>:

☞ Diese Option legt den Skalierungstyp des Graphik fest. Zulässige Werte für *xWert* und *yWert* sind jeweils *Lin* für eine lineare Skala oder *Log* für eine logarithmische Skala. Die Voreinstellung ist *AxesScaling = [Lin, Lin]*.

☞ Bei *RealValuesOnly = FALSE* darf der Koordinatenbereich bei logarithmischen Skalen nur positive Werte enthalten, sonst wird ein Fehler ausgelöst! Bei *RealValuesOnly = TRUE* werden negative Koordinatenbereiche in logarithmischen Plots nicht angezeigt.

Option <BackGround = [r, g, b]>:

☞ Diese Option legt die Hintergrundfarbe des Bildes fest. Die Werte *r*, *g*, *b* sind die Rot-, Grün- und Blauanteile gemäß des RGB-Farbmodells. Sie müssen reelle Zahlen zwischen 0 und 1 sein. Vordefinierte Farben werden durch MuPADs RGB-Datenstruktur zur Verfügung gestellt. Die Voreinstellung ist *BackGround = [1, 1, 1] = RGB::White*.

Option <Discont = Wert>:

- ☞ Diese Option legt fest, ob die graphischen Objekte auf Unstetigkeiten untersucht werden. Zulässige Werte sind `TRUE` und `FALSE`, die Voreinstellung ist `Discont = FALSE` in `plot2d`, `plot::Scene` und `plot` bzw. `Discont = TRUE` in `plotfunc2d`.
 - `Discont = TRUE` schaltet die symbolische Suche nach Unstetigkeiten an. Werden solche gefunden, so werden unerwünschte graphische Effekte wie z. B. Linien zwischen links- und rechtsseitigen Grenzwerten einer Sprungstelle vermieden.
 - `Discont = FALSE` schaltet die symbolische Suche nach Unstetigkeiten ab.
- ☞ Die symbolische Suche nach Unstetigkeiten kann zeitaufwendig sein. Man sollte `Discont = FALSE` benutzen, wenn bekannt ist, dass die graphischen Objekte stetig sind.
- ☞ Man beachte, dass einige Objekte der `plot`-Bibliothek ein *Objekt*-Attribut `Discont` besitzen, welches den Wert der Szeneoption `Discont` überschreibt. Insbesondere ist für die Objekte `plot::Function2d` und `plot::Curve2d` die Voreinstellung des Objektattributs `Discont = TRUE`, welches die Voreinstellung der Szeneoption `Discont = FALSE` in Aufrufen von `plot::Scene` und `plot` überschreibt.

Option <FontFamily = Fontfamilie>:

- ☞ Diese Option legt die Fontfamilie für den Titel, die Achsenbeschriftung und die Markenbeschriftung fest. Der Wert `Fontfamilie` kann eine der Zeichenketten `"helvetica"`, `"lucida"` etc. sein. Die Voreinstellung ist `FontFamily = "helvetica"`.

Option <FontSize = n>:

- ☞ Diese Option legt die Fontgröße für den Titel, die Achsenbeschriftung und die Markenbeschriftung fest. Die ganze Zahl `n` kann Werte zwischen 7 und 36 annehmen. Die Voreinstellung ist `FontSize = 8`.

Option <FontStyle = Fontstil>:

- ☞ Diese Option legt den Fontstil für den Titel, die Achsenbeschriftung und die Markenbeschriftung fest. Der Wert *Fontstil* kann eine der Zeichenketten "bold", ... sein. Die Voreinstellung ist *FontStyle* = "bold".

Option <Foreground = [r, g, b]>:

- ☞ Diese Option legt die „Vordergrundfarbe“ fest, d. h., die Farbe für die Koordinatenachsen, die Achsenbezeichnungen, die Achsenmarken, deren Beschriftung und den Titel. Auch Punkte und Randlinien gefüllter Polygone werden in dieser Farbe gezeichnet. Die Werte *r*, *g*, *b* legen die Rot-, Grün- und Blauanteile gemäß des RGB-Farbmodells fest. Sie müssen reelle Zahlen zwischen 0 und 1 sein. Vordefinierte Farben werden durch MuPADs RGB-Datenstruktur zur Verfügung gestellt. Die Voreinstellung ist *Foreground* = [0, 0, 0] = RGB::Black.
- ☞ Die Vordergrundfarbe bestimmt nicht die Farbe der graphischen Objekte. Diese wird entweder automatisch gewählt oder kann über die Farbeoption der Objekte festgelegt werden.

Option <GridLines = Wert>:

- ☞ Diese Option legt Gitterlinien fest, mit denen die Graphik unterlegt wird. Zulässige Werte sind *None*, *Automatic* oder eine Liste [*xWert*, *yWert*]. Die Voreinstellung ist *GridLines* = *None*.

- Mit *GridLines* = *None* werden keine Gitterlinien gezeichnet.
- Mit *GridLines* = *Automatic* werden Gitterlinien erzeugt, die an die Marken der Achsen gebunden sind.
- Mit *GridLines* = [*xWert*, *yWert*] können die Gitterlinien in den beiden Koordinatenrichtungen getrennt spezifiziert werden.

Die Werte *xWert* und *yWert* können *Automatic*, eine nicht-negative ganze Zahl, *Steps* = *d* oder *Steps* = [*d*, *n*] sein.

- *Automatic* erzeugt Gitterlinien, die an die Achsenmarken gebunden sind. *GridLines* = [*Automatic*, *Automatic*] entspricht *GridLines* = *Automatic*.
- Ein nicht-negativer ganzzahliger Wert setzt die minimale Anzahl der Gitterlinien. Die tatsächliche Anzahl und die Position der Linien werden heuristisch bestimmt. Wird die Zahl 0 angegeben, so

werden keine Gitterlinien gezeichnet. *GridLines* = *n* ist äquivalent zu *GridLines* = [*n*, *n*].

- *Steps* = *d* erzeugt Gitterlinien an den Stellen *jd*, wobei *j* diejenigen ganzzahligen Wert annimmt, für die die Gitterlinien im Bereich der graphischen Szene liegen. Die Distanz *d* zwischen zwei Gitterlinien muss ein reeller positiver Wert sein.
- *Steps* = [*d*, *n*] ist äquivalent zu *Steps* = *d* / (*n*+1), d. h., es werden jeweils *n* zusätzliche Gitterlinien zwischen die durch *Steps* = *d* erzeugten Linien eingeschoben. Der Parameter *n* muss eine nicht-negative ganze Zahl sein.

Option <*GridLinesColor* = [*r*, *g*, *b*]>:

- ☞ Diese Option legt die Farbe der Gitterlinien fest. Die Werte *r*, *g*, *b* sind die Rot-, Grün- und Blauanteile gemäß des RGB-Farbmodells. Sie müssen reelle Zahlen zwischen 0 und 1 sein. Vordefinierte Farben werden durch MuPADs RGB-Datenstruktur zur Verfügung gestellt. Die Voreinstellung ist *GridLines* = RGB::Gray.

Option <*GridLinesWidth* = *n*>:

- ☞ Diese Option legt die Breite der Gitterlinien fest. Zulässige Werte für *n* sind nicht-negative ganze Zahlen. Die Voreinstellung ist *GridLinesWidth* = 5.

Option <*GridLinesStyle* = *Wert*>:

- ☞ Diese Option legt den Stil der Gitterlinien fest. Zulässige Werte sind *SolidLines* und *DashedLines*: es werden entweder durchgezogene oder gestrichelte Linien gezeichnet. Die Voreinstellung ist *GridLinesStyle* = *DashedLines*.

Option <*Labeling* = *Wert*>:

- ☞ Diese Option legt fest, ob die Achsen und die Achsenmarken beschriftet werden. Zulässige Werte sind TRUE oder FALSE, die Voreinstellung ist *Labeling* = TRUE.
- ☞ Die Standardbeschriftung der Achsen kann mittels *Labels* geändert werden. Die Achsenmarken sowie deren Beschriftung können mittels *Ticks* gesetzt werden.

Option <Labels = [xString, yString]>:

- ☞ Diese Option legt über die Zeichenketten *xString*, *yString* die Beschriftung der Achsen fest. Die Standardbeschriftung ist *Labels* = ["x", "y"].

Option <LineStyle = Wert>:

- ☞ Diese Option legt den Stil fest, mit der alle Linienobjekte der Szene dargestellt werden. Zulässige Werte sind *SolidLines* und *DashedLines*: es werden entweder durchgezogene oder gestrichelte Linien gezeichnet. Die Voreinstellung ist *LineStyle* = *SolidLines*.
- ☞ Linienobjekte sind Funktionsgraphen, mittels [Mode = Curve, ...] erzeugte Kurven sowie mittels [Mode = List, [...Polygone...]] erzeugte Polygone. Man kann die *LineStyle*-Option der graphischen Objekte benutzen, diese Szeneoption zu überschreiben und jedes Linienobjekt in seinem eigenen Linienstil darzustellen.

Option <LineWidth = n>:

- ☞ Diese Option legt die Breite aller Linienobjekte der Szene fest. Zulässige Werte für *n* sind nicht-negative ganze Zahlen, die Voreinstellung ist *LineWidth* = 1.
- ☞ Linienobjekte sind Funktionsgraphen, mittels [Mode = Curve, ...] erzeugte Kurven sowie mittels [Mode = List, [...Polygone...]] erzeugte Polygone. Man kann die *LineWidth*-Option der graphischen Objekte benutzen, diese Szeneoption zu überschreiben und die Linienobjekte mit unterschiedlichen Breiten darzustellen.

Option <PlotDevice = Wert>:

- ☞ Diese Option legt fest, ob die Graphik auf dem Bildschirm erscheint oder in eine Datei geschrieben wird. Zulässige Werte sind *Screen*, eine Zeichenkette "Dateiname", ["Dateiname", *Ascii*] oder ["Dateiname", *Binary*]. Die Voreinstellung ist *PlotDevice* = *Screen*.
 - Mit *PlotDevice* = *Screen* wird die Graphik auf dem Bildschirm angezeigt.

- Mit `PlotDevice = ["Dateiname", Format]` wird die Graphik im spezifizierten Format in der Datei namens `Dateiname` gespeichert. Die zur Verfügung stehenden Formate sind *Ascii* und *Binary*. Dies sind MuPAD-spezifische Formate, die von MuPADs Graphikwerkzeug VCam verstanden werden. Die Datei kann später mit VCam geöffnet und angezeigt werden.
 - `PlotDevice = "Dateiname"` entspricht `PlotDevice = ["Dateiname", Binary]`.
- ☞ MuPAD-Graphiken können auch in einer Vielzahl graphischer Standardformate wie z. B. *Postscript*, *JPEG*, *TIFF* etc. gespeichert werden. Die Konvertierung in diese Formate kann jedoch nicht über einen Plot-Befehl innerhalb einer MuPAD-Sitzung geschehen. Man muss stattdessen interaktiv die Oberfläche des graphischen Werkzeugs VCam benutzen: innerhalb eines MuPAD Pro Notebooks wird diese Oberfläche durch einen Doppelklick auf die Graphik aktiviert. Durch Auswahl der Menüpunkte „Bearbeiten/Exportieren...“ wird die Dialogbox „Exportiere Graphik“ geöffnet, in der das gewünschte Format eingestellt werden kann.

Option **<PointSize = Wert>**:

- ☞ Diese Option legt den Stil fest, mit der alle Punktoobjekte der Szene dargestellt werden. Zulässige Werte sind *Circles*, *Squares*, *FilledCircles* und *FilledSquares*. Die Voreinstellung ist `PointSize = FilledSquares`.
- ☞ Punktoobjekte sind graphische Primitive, die mittels der MuPAD-Funktion `point` erzeugt werden. Sie können mit `plot2d` über Objekte des Typs `[Mode = List, [...Punkte...]]` dargestellt werden. Man kann mittels der Objektoption `PointSize` diese Szeneoption überschreiben und jeden Punkt in seinem eigenen Stil darstellen.

Option **<PointWidth = n>**:

- ☞ Diese Option legt die Größe aller Punktoobjekte der Szene fest. Zulässige Werte für `n` sind positive ganze Zahlen, die Voreinstellung ist `PointWidth = 30`.
- ☞ Punktoobjekte sind graphische Primitive, die mittels der MuPAD-Funktion `point` erzeugt werden. Sie können mit `plot2d` über Objekte des Typs `[Mode = List, [...points...]]` dargestellt werden. Man kann mittels der Objektoption `PointWidth` diese Szeneoption überschreiben und Punkte mit unterschiedlichen Größen darstellen.

Option `<RealValuesOnly = Wert>`:

- ☞ Falls ein graphisches Objekt wie etwa eine Funktion während der numerischen Auswertung der Graphik einen nicht-reellen Wert liefert, so wird ein Fehler ausgelöst.

Mittels `RealValuesOnly = TRUE` können solche Fehler abgefangen werden: lediglich die reellwertigen Teile der Objekte werden dargestellt. Beispielsweise kann mit dieser Option die Funktion `sqrt(x)` über dem Intervall $x \in [-1, 1]$ dargestellt werden, wobei die Graphik nur die reellen Funktionswerte für $x \geq 0$ enthält.

Mit `RealValuesOnly = FALSE` wird kein interner Test zum Abfangen komplexer Werte durchgeführt, und ein Fehler wird ausgelöst, falls solche Werte entstehen.

Die Voreinstellung ist `RealValuesOnly = FALSE` in `plot2d`, `plot::Scene`, und `plot` bzw. `RealValuesOnly = TRUE` in `plotfunc2d`.

- ☞ Die Kurzform `RealsOnly` ist gleichbedeutend mit `RealValuesOnly`.
- ☞ Der Test auf reelle Werte kann zeitaufwendig sein. Man benutze `RealsOnly = FALSE`, wenn die Objekte sicher reellwertig sind.
- ☞ Man beachte, dass einige Objekte der `plot`-Bibliothek ein Objekt-Attribut `RealValuesOnly` besitzen, welches den Wert der Szeneoption `RealValuesOnly` überschreibt. Insbesondere ist für die Objekte `plot::Function2d` und `plot::Curve2d` die Voreinstellung des Objektattributs `RealValuesOnly = TRUE`, welches die Voreinstellung der Szeneoption `RealValuesOnly = FALSE` in Aufrufen von `plot::Scene` und `plot` überschreibt.

Option `<Scaling = Wert>`:

- ☞ Diese Option legt das Skalierungsverhältnis zwischen den x - und y -Koordinaten fest. Zulässige Werte sind `Constrained` und `UnConstrained`, die Voreinstellung ist `Scaling = UnConstrained`.
 - Mit `Scaling = Constrained` wird das Skalierungsverhältnis auf 1 : 1 festgelegt. Damit werden Kreise stets als Kreise gezeichnet. Dieser Modus ist ungeeignet für Szenen, deren x - und y -Durchmesser von unterschiedlicher Größenordnung ist.
 - Mit `Scaling = UnConstrained` wird das Skalierungsverhältnis so gewählt, dass die Szene optimal in das graphische Fenster passt. In diesem Fall können Kreise zu Ellipsen werden.

Option **<Ticks = Wert>**:

☞ Diese Option legt die Marken auf den Koordinatenachsen fest. Zulässige Werte sind *None*, *Automatic*, eine nicht-negative ganze Zahl oder eine Liste `[xWert, yWert]`. Die Voreinstellung ist *Ticks = Automatic*.

- Mit *Ticks = None* werden keine Marken gezeichnet.
- Mit *Ticks = Automatic* werden Marken heuristisch gewählt.
- Mit *Ticks = n* wird eine minimale Anzahl von Marken auf den beiden Achsen durch die nicht-negative ganze Zahl *n* spezifiziert. Man beachte, dass eventuell mehr Marken als angegeben gezeichnet werden, damit diese an sinnvollen Stellen platziert werden können.
- Mit *Ticks = [xWert, yWert]* können die Marken auf den beiden Achsen getrennt spezifiziert werden.

Die Werte *xWert* und *yWert* können *Automatic*, eine nicht-negative ganze Zahl, *Steps = d*, *Steps = [d, n]* oder eine Liste selbstdefinierter Marken sein.

- *Automatic* erzeugt heuristisch gewählte Marken. *Ticks = Automatic* ist äquivalent zu *Ticks = [Automatic, Automatic]*.
- Ein nicht-negativer ganzzahliger Wert setzt die minimale Anzahl der Marken. Die tatsächliche Anzahl sowie die Positionen der Marken werden heuristisch bestimmt. Wird die Zahl 0 angegeben, so werden keine Marken gesetzt. *Ticks = n* ist äquivalent zu *Ticks = [n, n]*.
- *Steps = d* erzeugt Marken an den Stellen jd , wobei j diejenigen ganzzahligen Wert annimmt, für die die Marken im Bereich der graphischen Szene liegen. Die Distanz d zwischen zwei Marken muss ein reeller positiver Wert sein.
- *Steps = [d, n]* erzeugt dieselben „großen“ Marken wie *Steps = d*. Zwischen jeweils zwei solcher Marken werden zusätzlich n kleinere Marken platziert. Der Parameter n muss eine nicht-negative ganze Zahl sein. Die „großen“ Marken tragen eine Beschriftung, falls *Labeling = TRUE*. Die „kleinen“ Marken tragen keine Beschriftung.
- Marken können durch eine Liste `[t1, t2, ...]` an beliebigen Stellen platziert werden. Zulässige Werte für $t1, t2$ etc. sind reelle numerische Ausdrücke, die die Positionen der Marken festlegen. Alternativ kann jedes Element der Markenliste auch eine Gleichung der Form $t = z$ sein, wobei t ein numerischer Wert und z eine Zeichenkette ist. Dies erzeugt eine Marke an der Stelle t mit der durch die Zeichenkette gegebenen Beschriftung. Die Beschriftung wird dargestellt, falls *Labeling = TRUE*. Beispielsweise erzeugt


```
Ticks = [[0.2, PI = "PI"], [sqrt(2), 2, 3]]
```

zwei Marken auf der x -Achse an den Stellen $x = 0.2$ und $x = \pi$, wobei die zweite Marke die Beschriftung "PI" trägt. Auf der y -Achse werden drei Marken ohne Beschriftung platziert.

Werden Marken außerhalb des Sichtbarkeitsbereichs der Szene gesetzt, so wird dieser automatisch ausgedehnt, um alle Marken sichtbar zu machen.

Option **<Title = Titel>**:

- ☞ Diese Option legt die Überschrift der Szene durch die Zeichenkette `Title` fest. In `plot2d`, `plot::Scene` und `plot` ist die Voreinstellung die leere Zeichenkette `Title = ""`, d. h., keine Überschrift. In `plotfunc2d` werden die die zu zeichnenden Funktionen darstellenden Ausdrücke in Zeichenketten verwandelt und als Überschrift verwendet.

Option **<TitlePosition = Wert>**:

- ☞ Diese Option legt die Platzierung der Überschrift fest. Zulässige Werte sind *Above*, *Below* und $[x, y]$. Die Voreinstellung ist `TitlePosition = Above`.
 - Mit `TitlePosition = Above` wird die Überschrift zentriert über die Szene gesetzt.
 - Mit `TitlePosition = Below` wird die Überschrift zentriert unter die Szene gesetzt.
 - Mit `TitlePosition = [x, y]`, kann die Überschrift an jeder Stelle der Szene platziert werden. Die Wert x, y müssen reelle numerische Werte zwischen 0 und 10 sein. Die Position $[0, 0]$ ist die linke obere Ecke der Szene, $[10, 10]$ ist die rechte untere Ecke.
- ☞ Die Überschrift kann innerhalb des angezeigten Bildes mit der Maus bewegt und so interaktiv an eine geeignete Stelle verschoben werden.

Option **<ViewingBox = Wert>**:

- ☞ Diese Option legt den Sichtbarkeitsbereich der Szene fest, d. h., den Bereich der x - und y -Koordinaten, der dargestellt wird. Zulässige Werte sind *Automatic* und $[xWert, yWert]$. Die Voreinstellung ist `ViewingBox = Automatic`.
 - Mit `ViewingBox = Automatic` ist die gesamte Szene sichtbar.

- Die Werte `xWert` und `yWert` können *Automatic* oder ein Bereich `a..b` sein. Zulässige Werte für `a` und `b` sind reelle numerische Ausdrücke sowie die Bezeichner *XMin*, *XMax*, *YMin*, *YMax*. Diese repräsentieren die extremalen Koordinaten der Szene, welche während der numerischen Auswertung der Graphik bestimmt werden.
- ☞ Das Abschneiden graphischer Objekte auf einen Sichtbarkeitsbereich kann sehr zeitaufwendig sein! Man verwende die Voreinstellung *ViewingBox = Automatic*, wenn dies angebracht ist.

Änderungen:

- ☞ Die neuen Optionen *Discont*, *GridLines*, *GridLinesColor*, *GridLinesStyle*, *GridLinesWidth*, *RealValuesOnly* und *ViewingBox* wurden eingeführt. Die Funktionalität der Option *Ticks* wurde erweitert.
- ☞ Die Voreinstellungen diverser Optionen wurden geändert.
- ☞ Überschriften der Szene sowie Objektüberschriften können nun mit der Maus verschoben werden.

plotOptions3d – Szeneoptionen für 3-dimensionale Graphiken

Diese Seite beschreibt die Szeneoptionen, die den Routinen `plot3d`, `plotfunc3d`, `plot::Scene` und `plot` zur Erzeugung 3-dimensionaler Graphiken übergeben werden können. Szeneoptionen sind Attribute, die das allgemeine Aussehen einer graphischen Szene bestimmen, z. B., Kamerastandpunkt, Hintergrundfarbe, Titel, Stil der Achsen etc.

Aufruf(e):

- ☞ `plot3d(<SceneOpt1, SceneOpt2, ...>, graphical objects)`
- ☞ `plotfunc3d(<SceneOpt1, SceneOpt2, ...>, graphical objects)`
- ☞ `plot::Scene(graphical objects, <SceneOpt1, SceneOpt2, ...>)`
- ☞ `plot(graphical objects, <SceneOpt1, SceneOpt2, ...>)`

Parameter:

`graphical objects` — siehe die Hilfeseiten von `plot3d`, `plotfunc3d`, `plot::Scene` und `plot`

Optionen:

SceneOpt1, SceneOpt2, .. — Szeneoptionen: Gleichungen der Form OptionsName = Wert.

OptionsName	zulässige Werte	Standardwert
<i>Arrows</i>	TRUE, FALSE	FALSE
<i>Axes</i>	<i>Box, Corner, None, Origin</i>	<i>Box</i>
<i>AxesOrigin</i>	<i>Automatic, [x0, y0, z0]</i>	<i>Automatic</i>
<i>AxesScaling</i>	<i>[Lin/Log, Lin/Log, Lin/Log]</i>	<i>[Lin, Lin, Lin]</i>
<i>BackGround</i>	<i>[r, g, b]</i>	<i>RGB::White</i>
<i>FontFamily</i>	<i>"helvetica", "lucida", ..</i>	<i>"helvetica"</i>
<i>FontSize</i>	positive ganze Zahlen	8
<i>FontStyle</i>	<i>"bold", ..</i>	<i>"bold"</i>
<i>ForeGround</i>	<i>[r, g, b]</i>	<i>RBG::Black</i>
<i>Labeling</i>	TRUE, FALSE	TRUE
<i>Labels</i>	<i>[string, string, string]</i>	<i>["x", "y", "z"]</i>
<i>LineStyle</i>	<i>SolidLines, DashedLines</i>	<i>SolidLines</i>
<i>LineWidth</i>	positive ganze Zahlen	1
<i>PlotDevice</i>	<i>Screen, "Dateiname", ["Dateiname", Ascii], ["Dateiname", Binary]</i>	<i>Screen</i>
<i>PointStyle</i>	<i>Circles, FilledCircles, FilledSquares, Squares</i>	<i>FilledSquares</i>
<i>PointWidth</i>	positive ganze Zahlen	30
<i>Scaling</i>	<i>Constrained, UnConstrained</i>	<i>UnConstrained</i>
<i>Ticks</i>	<i>Automatic, None, eine ganze Zahl oder [xWert, yWert, zWert]. Zulässige Werte für xWert, yWert, zWert sind Automatic, ganze Zahlen, Steps = d, Steps = [d, n] oder Listen selbstdefinierter Marken.</i>	<i>Automatic</i>
<i>Title</i>	Zeichenketten	<i>" " (plot3d) "f(x, y)" (plotfunc3d) " " (plot::Scene) " " (plot)</i>
<i>TitlePosition</i>	<i>Above, Below, [x, y]</i>	<i>Above</i>
<i>ViewingBox</i>	<i>Automatic</i>	<i>Automatic</i>

Verwandte Funktionen: plot, plot::Scene, plot, plot2d, plotfunc2d, plotOptions2d, plot3d, plotfunc3d

Option <Arrows = Wert>:

- ☞ Diese Option legt fest, ob die Koordinatenachsen mit oder ohne Pfeilspitzen dargestellt werden. Zulässige Werte sind *TRUE* oder *FALSE*, die Voreinstellung ist *Arrows = FALSE*. Diese Option wird bei *Axes = None* oder *Axes = Box* ignoriert.

Option <Axes = Wert>:

- ☞ Diese Option legt den Typ der Koordinatenachsen fest. Zulässige Werte sind *Box*, *Corner*, *None* und *Origin*. Die Voreinstellung ist *Axes = Box*.
 - Mit *Axes = None* werden keine Achsen gezeichnet.
 - Mit *Axes = Box* wird ein quaderförmiger Rahmen um die Szene gezeichnet.
 - Mit *Axes = Corner* wird ein Koordinatenkreuz gezeichnet, die Achsen kreuzen sich in einer Ecke der Szene.
 - Mit *Axes = Origin* wird ein Koordinatenkreuz gezeichnet, die Achsen kreuzen sich in dem durch *AxesOrigin* gegebenen Punkt.

Option <AxesOrigin = Wert>:

- ☞ Diese Option setzt den Punkt, an dem das Koordinatenkreuz zentriert ist. Zulässige Werte sind *Automatic* und $[x0, y0, z0]$, die Voreinstellung ist *AxesOrigin = Automatic*.
 - Mit *AxesOrigin = Automatic* kreuzen sich die Achsen im mathematischen Ursprung (0,0,0), falls dieser innerhalb der Szene liegt. Ist dies nicht der Fall, so kreuzen sich die Achsen in derjenigen Ecke des Bildes, die dem Ursprung am nächsten liegt.
 - Mit *AxesOrigin = $[x0, y0, z0]$* kreuzen sich die Achsen im angegebenen Punkt. Zulässige Werte für die Koordinaten sind reelle numerische Ausdrücke sowie die Bezeichner *XMin*, *XMax*, *YMin*, *YMax*, *ZMin*, *ZMax*. Diese repräsentieren die extremalen Koordinaten der Szene, welche während der numerischen Auswertung der Graphik bestimmt werden.

Option <AxesScaling = [xWert, yWert, zWert]>:

- ☞ Diese Option legt den Skalierungstyp des Graphik fest. Zulässige Werte für *xWert*, *yWert* und *zWert* sind jeweils *Lin* für eine lineare Skala oder *Log* für eine logarithmische Skala. Die Voreinstellung ist *AxesScaling = [Lin, Lin, Lin]*.

- ☞ Bei logarithmischen Skalen darf der Koordinatenbereich nur positive Werte enthalten, sonst wird ein Fehler ausgelöst!

Option `<BackGround = [r, g, b]>`:

- ☞ Diese Option legt die Hintergrundfarbe des Bildes fest. Die Werte *r*, *g*, *b* sind die Rot-, Grün- und Blauanteile gemäß des RGB-Farbmodells. Sie müssen reelle Zahlen zwischen 0 und 1 sein. Vordefinierte Farben werden durch MuPADs RGB-Datenstruktur zur Verfügung gestellt. Die Voreinstellung ist *BackGround* = [1, 1, 1] = RGB::White.

Option `<CameraPoint = Wert>`:

- ☞ Diese Option legt die Position der Kamera des Beobachters fest. Die optische Achse der Kamera ist der Vektor vom Beobachterstandpunkt *CameraPoint* zum Blickpunkt *FocalPoint*. Der Wert von *CameraPoint* kann entweder *Automatic* oder ein Vektor [*x*, *y*, *z*] sein. Mit der Voreinstellung *Automatic* wird der Standpunkt automatisch außerhalb der graphischen Szene gewählt. Ein selbstgewählter Standpunkt sollte sich ebenfalls außerhalb der Szene befinden. Ein (am Durchmesser der Szene gemessen) nahe gelegener Standpunkt führt zu starken perspektivischen Verzerrungen. Ein weit entfernter Standpunkt vermeidet solche Verzerrungen (Parallelprojektion). Die Größe der projizierten Szene ist unabhängig vom Abstand: die Projektion wird automatisch so vergrößert, dass sie die Zeichenfläche ausfüllt.

Option `<FocalPoint = Wert>`:

- ☞ Diese Option legt den Blickpunkt fest, auf den die Kamera des Beobachters gerichtet ist. Die optische Achse der Kamera ist der Vektor vom Beobachterstandpunkt *CameraPoint* zum Blickpunkt *FocalPoint*. Der Wert von *FocalPoint* kann entweder *Automatic* oder ein Vektor [*x*, *y*, *z*] sein. Mit der Voreinstellung *Automatic* wird als Blickpunkt automatisch das Zentrum der graphischen Szene gewählt.

Option `<FontFamily = Fontfamilie>`:

- ☞ Diese Option legt die Fontfamilie für den Titel, die Achsenbeschriftung und die Markenbeschriftung fest. Der Wert *Fontfamilie* kann eine der Zeichenketten "helvetica", "lucida" etc. sein. Die Voreinstellung ist *FontFamily* = "helvetica".

Option <FontSize = n>:

- ☞ Diese Option legt die Fontgröße für den Titel, die Achsenbeschriftung und die Markenbeschriftung fest. Die ganze Zahl *n* kann Werte zwischen 7 und 36 annehmen. Die Voreinstellung ist *FontSize* = 8.

Option <FontStyle = FontStyleString >:

- ☞ Diese Option legt den Fontstil für den Titel, die Achsenbeschriftung und die Markenbeschriftung fest. Der Wert *Fontstil* kann eine der Zeichenketten "bold", ... sein. Die Voreinstellung ist *FontStyle* = "bold".

Option <Foreground = [r, g, b]>:

- ☞ Diese Option legt die „Vordergrundfarbe“ fest, d. h., die Farbe für die Koordinatenachsen, die Achsenbezeichnungen, die Achsenmarken, deren Beschriftung und den Titel. Auch Punkte und Randlinien gefüllter Polygone werden in dieser Farbe gezeichnet. Die Werte *r*, *g*, *b* legen die Rot-, Grün- und Blauanteile gemäß des RGB-Farbmodells fest. Sie müssen reelle Zahlen zwischen 0 und 1 sein. Voreingestellte Farben werden durch MuPADs RGB-Datenstruktur zur Verfügung gestellt. Die Voreinstellung ist *Foreground* = [0, 0, 0] = RGB::Black.
- ☞ Die Vordergrundfarbe bestimmt nicht die Farbe der graphischen Objekte. Diese wird entweder automatisch gewählt oder kann über die Farbeoption der Objekte festgelegt werden.

Option <Labeling = Wert>:

- ☞ Diese Option legt fest, ob die Achsen und die Achsenmarken beschriftet werden. Zulässige Werte sind TRUE oder FALSE, die Voreinstellung ist *Labeling* = TRUE.
- ☞ Die Standardbeschriftung der Achsen kann mittels *Labels* geändert werden. Die Achsenmarken sowie deren Beschriftung können mittels *Ticks* gesetzt werden.

Option <Labels = [xString, yString, zString]>:

- ☞ Diese Option legt über die Zeichenketten *xString*, *yString* und *zString* die Beschriftung der Achsen fest. Die Standardbeschriftung ist *Labels* = ["x", "y", "z"].

Option <LineStyle = Wert>:

- ☞ Diese Option legt den Stil fest, mit der alle Linienobjekte der Szene dargestellt werden. Zulässige Werte sind *SolidLines* und *DashedLines*: es werden entweder durchgezogene oder gestrichelte Linien gezeichnet. Die Voreinstellung ist *LineStyle* = *SolidLines*.
- ☞ Linienobjekte sind Funktionsgraphen, mittels [Mode = Curve, ...] erzeugte Kurven, die Parameterlinien von Flächen sowie mittels [Mode = List, [...Polygone...]] erzeugte Polygonzüge. Man kann die *LineStyle*-Option der graphischen Objekte benutzen, diese Szeneoption zu überschreiben und jedes Linienobjekt in seinem eigenen Linienstil darzustellen.

Option <LineWidth = n>:

- ☞ Diese Option legt die Breite aller Linienobjekte der Szene fest. Zulässige Werte für *n* sind nicht-negative ganze Zahlen, die Voreinstellung ist *LineWidth* = 1.
- ☞ Linienobjekte sind Funktionsgraphen, mittels [Mode = Curve, ...] erzeugte Kurven sowie mittels [Mode = List, [...Polygone...]] erzeugte Polygone. Man kann die *LineWidth*-Option der graphischen Objekte benutzen, diese Szeneoption zu überschreiben und die Linienobjekte mit unterschiedlichen Breiten darzustellen.

Option <PlotDevice = Wert>:

- ☞ Diese Option legt fest, ob die Graphik auf dem Bildschirm erscheint oder in eine Datei geschrieben wird. Zulässige Werte sind *Screen*, eine Zeichenkette "Dateiname", ["Dateiname", *Ascii*] oder ["Dateiname", *Binary*]. Der Standard ist *PlotDevice* = *Screen*.
 - Mit *PlotDevice* = *Screen* wird die Graphik auf dem Bildschirm angezeigt.

- Mit `PlotDevice = ["Dateiname", Format]` wird die Graphik im spezifizierten Format in der Datei namens `Dateiname` gespeichert. Die zur Verfügung stehenden Formate sind *Ascii* und *Binary*. Dies sind MuPAD-spezifische Formate, die von MuPADs Graphikwerkzeug VCam verstanden werden. Die Datei kann später mit VCam geöffnet und angezeigt werden.
 - `PlotDevice = "Dateiname"` entspricht `PlotDevice = ["Dateiname", Binary]`.
- ☞ MuPAD-Graphiken können auch in einer Vielzahl graphischer Standardformate wie z. B. *Postscript*, *JPEG*, *TIFF* etc. gespeichert werden. Die Konvertierung in diese Formate kann jedoch nicht über einen Plot-Befehl innerhalb einer MuPAD-Sitzung geschehen. Man muss stattdessen interaktiv die Oberfläche des graphischen Werkzeugs VCam benutzen: innerhalb eines MuPAD Pro Notebooks wird diese Oberfläche durch einen Doppelklick auf die Graphik aktiviert. Durch Auswahl der Menüpunkte „Bearbeiten/Exportieren...“ wird die Dialogbox „Exportiere Graphik“ geöffnet, in der das gewünschte Format eingestellt werden kann.

Option **<PointSize = Wert>**:

- ☞ Diese Option legt den Stil fest, mit der alle Punktoobjekte der Szene dargestellt werden. Zulässige Werte sind *Circles*, *Squares*, *FilledCircles* und *FilledSquares*. Die Voreinstellung ist `PointSize = FilledSquares`.
- ☞ Punktoobjekte sind graphische Primitive, die mittels der MuPAD-Funktion `point` erzeugt werden. Sie können mit `plot3d` über Objekte des Typs `[Mode = List, [...Punkte...]]` dargestellt werden. Man kann mittels der Objektoption `PointSize` diese Szeneoption überschreiben und jeden Punkt in seinem eigenen Stil darstellen.

Option **<PointWidth = n>**:

- ☞ Diese Option legt die Größe aller Punktoobjekte der Szene fest. Zulässige Werte für `n` sind positive ganze Zahlen, die Voreinstellung ist `PointWidth = 30`.
- ☞ Punktoobjekte sind graphische Primitive, die mittels der MuPAD-Funktion `point` erzeugt werden. Sie können mit `plot3d` über Objekte des Typs `[Mode = List, [...points...]]` dargestellt werden. Man kann mittels der Objektoption `PointWidth` diese Szeneoption überschreiben und Punkte mit unterschiedlichen Größen darstellen.

Option <Scaling = Wert>:

☞ Diese Option legt das Skalierungsverhältnis zwischen den *x*-, *y*-, *z*-Koordinaten fest. Zulässige Werte sind *Constrained* und *UnConstrained*, die Voreinstellung ist *Scaling = UnConstrained*.

- Mit *Scaling = Constrained* wird das Skalierungsverhältnis auf 1 : 1 : 1 festgelegt. Damit werden Kugeln stets als Kugeln gezeichnet. Dieser Modus ist ungeeignet für Szenen, deren Durchmesser in den drei Raumrichtungen von unterschiedlicher Größenordnung ist.
- Mit *Scaling = UnConstrained* wird das Skalierungsverhältnis so gewählt, dass die Szene optimal in das graphische Fenster passt. In diesem Fall können Kugeln zu Ellipsoiden werden.

Option <Ticks = Wert>:

☞ Diese Option legt die Marken auf den Koordinatenachsen fest. Zulässige Werte sind *None*, *Automatic*, eine nicht-negative ganze Zahl oder eine Liste [*xWert*, *yWert*, *zWert*]. Die Voreinstellung ist *Ticks = Automatic*.

- Mit *Ticks = None* werden keine Marken gezeichnet.
- Mit *Ticks = Automatic* werden Marken heuristisch gewählt.
- Mit *Ticks = n* wird eine minimale Anzahl von Marken auf den drei Achsen durch die nicht-negative ganze Zahl *n* spezifiziert. Man beachte, dass eventuell mehr Marken als angegeben gezeichnet werden, damit diese an sinnvollen Stellen platziert werden können.
- Mit *Ticks = [xWert, yWert, zWert]* können die Marken auf den beiden Achsen getrennt spezifiziert werden.

Die Werte *xWert*, *yWert* und *zWert* können *Automatic*, eine nicht-negative ganze Zahl, *Steps = d*, *Steps = [d, n]* oder eine Liste selbstdefinierter Marken sein.

- *Automatic* erzeugt heuristisch gewählte Marken. *Ticks = Automatic* ist äquivalent zu *Ticks = [Automatic, Automatic, Automatic]*.
- Ein nicht-negativer ganzzahliger Wert setzt die minimale Anzahl der Marken. Die tatsächliche Anzahl sowie die Positionen der Marken werden heuristisch bestimmt. Wird die Zahl 0 angegeben, so werden keine Marken gesetzt. *Ticks = n* ist äquivalent zu *Ticks = [n, n, n]*.

- $Steps = d$ erzeugt Marken an den Stellen jd , wobei j diejenigen ganzzahligen Werte annimmt, für die die Marken im Bereich der graphischen Szene liegen. Die Distanz d zwischen zwei Marken muss ein reeller positiver Wert sein.
- $Steps = [d, n]$ erzeugt dieselben „großen“ Marken wie $Steps = d$. Zwischen jeweils zwei solcher Marken werden zusätzlich n kleinere Marken platziert. Der Parameter n muss eine nicht-negative ganze Zahl sein. Die „großen“ Marken tragen eine Beschriftung, falls $Labeling = \text{TRUE}$. Die „kleinen“ Marken tragen keine Beschriftung.
- Marken können durch eine Liste $[t_1, t_2, \dots]$ an beliebigen Stellen platziert werden. Zulässige Werte für t_1, t_2 etc. sind reelle numerische Ausdrücke, die die Positionen der Marken festlegen. Alternativ kann jedes Element der Markenliste auch eine Gleichung der Form $t = z$ sein, wobei t ein numerischer Wert und z eine Zeichenkette ist. Dies erzeugt eine Marke an der Stelle t mit der durch die Zeichenkette gegebenen Beschriftung. Die Beschriftung wird dargestellt, falls $Labeling = \text{TRUE}$. Beispielsweise erzeugt
 $Ticks = [[0.2, \text{PI} = \text{"PI"}], [\text{sqrt}(2), 2, 3], [1, 2, 3]]$

zwei Marken auf der x -Achse an den Stellen $x = 0.2$ und $x = \pi$, wobei die zweite Marke die Beschriftung "PI" trägt. Auf der y - und z -Achse werden jeweils drei Marken ohne Beschriftung platziert.

Werden Marken außerhalb des Sichtbarkeitsbereichs der Szene gesetzt, so wird dieser automatisch ausgedehnt, um alle Marken sichtbar zu machen.

Option **<Title = Titel>**:

- ☞ Diese Option legt die Überschrift der Szene durch die Zeichenkette `Title` fest. In `plot3d, plot::Scene` und `plot` ist die Voreinstellung die leere Zeichenkette `Title = ""`, d. h., keine Überschrift. In `plotfunc3d` werden die zu zeichnenden Funktionen darstellenden Ausdrücke in Zeichenketten verwandelt und als Überschrift verwendet.

Option **<TitlePosition = Wert>**:

- ☞ Diese Option legt die Platzierung der Überschrift fest. Zulässige Werte sind `Above`, `Below` und `[x, y]`. Die Voreinstellung ist `TitlePosition = Above`.
 - Mit `TitlePosition = Above` wird die Überschrift zentriert über die Szene gesetzt.

- Mit `TitlePosition = Below` wird die Überschrift zentriert unter die Szene gesetzt.
 - Mit `TitlePosition = [x, y]`, kann die Überschrift an jeder Stelle der Szene plaziert werden. Die Wert `x, y` müssen reelle numerische Werte zwischen 0 und 10 sein. Die Position `[0, 0]` ist die linke obere Ecke der Szene, `[10, 10]` ist die rechte untere Ecke.
- ☞ Die Überschrift kann innerhalb des angezeigten Bildes mit der Maus bewegt und so interaktiv an eine geeignete Stelle verschoben werden.

Option `<ViewingBox = Wert>`:

- ☞ Diese Option legt den Sichtbarkeitsbereich der Szene fest, d. h., den Bereich der dargestellten *x*-, *y*-, *z*-Koordinaten. Der einzige momentan zulässige Werte ist *Automatic*. Hiermit ist die gesamte Szene sichtbar.

Änderungen:

- ☞ Die Funktionalität der Szeneoption *Ticks* wurde erweitert.
- ☞ Die Voreinstellungen diverser Optionen wurden geändert.
- ☞ Überschriften der Szene sowie Objektüberschriften können nun mit der Maus verschoben werden.
-

point – Erzeugen eines graphischen Punktes

`point(x, y)` definiert einen 2D-Punkt mit den Koordinaten *x* und *y*.

`point(x, y, z)` definiert einen 3D-Punkt mit den Koordinaten *x*, *y* und *z*.

Aufruf(e):

- ☞ `point(x, y <, Color = [r, g, b]>)`
- ☞ `point(x, y, z <, Color = [r, g, b]>)`

Parameter:

x, y, z — reelle Zahlen


Optionen:

`Color = [r, g, b]` — setzt einen Farbwert im RGB-Format mit den Rot-Grün-Blau-Anteilen *r, g, b*. Diese Parameter müssen reelle Zahlenwerte aus dem Intervall von 0 bis 1 sein.


Rückgabewert: ein Objekt vom Typ `DOM_POINT`.

Verwandte Funktionen: `plot`, `plot::Point`, `plot2d`, `plot3d`, `plotfunc2d`, `plotfunc3d`, `polygon`, `RGB`

Details:

- ☞ `point` definiert 2D- oder 3D-Punkte, die graphisch mittels `plot2d` bzw. `plot3d` über das Listenformat `[Mode = List, [...Punkte...]]` dargestellt werden können.
 - ☞ Die Koordinaten und Farbwerte müssen Zahlen oder numerische Ausdrücke sein, die mittels `float` in reelle Gleitpunktzahlen konvertiert werden können. Symbolische Ausdrücke wie z. B. `PI + 1`, `exp(sqrt(2))` etc. werden akzeptiert und automatisch in Gleitpunktwerte verwandelt. Man beachte jedoch, dass Ausdrücke mit symbolischen Bezeichnern nicht akzeptiert werden! Siehe Beispiel ??.
- 
NOTE
- ☞ Die `plot`-Bibliothek stellt mit `plot::Point` ein alternatives graphisches Punktprimitiv zur Verfügung. Dieses Objekt ist wesentlich flexibler als das durch `point` erzeugte Punktobjekt. Das erstere kann in allen Funktionen der `plot`-Bibliothek eingesetzt werden, während das letztere nur in einem Aufruf von `plot2d` bzw. `plot3d` verwendet werden kann.
 - ☞ `point` ist eine Funktion des Systemkerns.
-

Option `<Color = [r, g, b]>`:

- ☞ Die Farbwerte `r`, `g`, `b` müssen Zahlen oder numerische Ausdrücke sein, die mittels `float` in reelle Gleitpunktzahlen aus dem Intervall `[0.0,1.0]` konvertiert werden können. Ein Fehler wird ausgelöst, falls einer der Werte nicht in diesem Bereich liegt. Symbolische Ausdrücke wie z. B. `PI - 3`, `exp(-sqrt(2))` etc. werden akzeptiert und automatisch in Gleitpunktwerte verwandelt. Man beachte jedoch, dass Ausdrücke mit symbolischen Bezeichnern nicht akzeptiert werden! Siehe Beispiel ??.
- 
NOTE
- ☞ Das Domain `RGB` enthält zahlreiche vordefinierte Farbnamen.

Operanden: Die ersten zwei bzw. drei Operanden eines Punktes sind die Koordinaten. Der letzte Operand ist die Liste `[r, g, b]` der Farbwerte. Dieser Operand ist `NIL`, wenn keine Farbe spezifiziert wurde.

Beispiel 1. `point` mit zwei Argumenten definiert einen 2D-Punkt:

```
>> point(1, PI)
```

```
point(1, 3.141592654)
```

Durch `point` erzeugte Punkte sind graphische Primitive, die mittels `plot2d` bzw. `plot3d` über das Listenformat `[Mode = List, [...Punkte...]]` dargestellt werden können:

```
>> plot2d(Scaling = UnConstrained, PointWidth = 30,  
          [Mode = List, [point(i/10, sin(i/10)) $ i=0..63]])
```

Beispiel 2. Punkte können mit einer gegebenen Farbe definiert werden:

```
>> point(0, 1, PI, Color = [1/2, 0, PI - 2*sqrt(2)])
```

```
point(0, 1, 3.141592654, Color = [0.5, 0.0, 0.3131655288])
```

Das Domain RGB enthält zahlreiche vordefinierte Farben:

```
>> point(1.0, 0.0, 1.0, Color = RGB::Red)
```

```
point(1.0, 0.0, 1.0, Color = [1.0, 0.0, 0.0])
```

Beispiel 3. Symbolische Koordinaten oder Farbwerte werden nicht akzeptiert:

```
>> point(x, y, z)
```

```
Error: Illegal argument [point]
```

```
>> point(1, 2, Color = [r, g, b])
```

```
Error: Illegal color specification [point]
```

Man kann jedoch Punktlisten mittels symbolischer Schleifenvariablen erzeugen:

```
>> mypoints := [point(i/40, exp(-i/40),  
                    Color = [1 - 1/i, i/(1 + i), exp(-i/40)])  
               $ i = 1..40]:  
plot2d(PointWidth = 30, [Mode = List, mypoints])
```

```
>> delete mypoints:
```

Änderungen:

- ⌘ Exakte numerische Ausdrücke wie z. B. `PI`, `exp(-sqrt(2))` etc. werden nun akzeptiert und automatisch in Gleitpunktwerte verwandelt.
-

poly – Erzeugen eines Polynoms

`poly(f)` wandelt einen polynomialen Ausdruck in ein Polynom des Kern-Domains `DOM_POLY` um.

Aufruf(e):

- ⌘ `poly(f <, [x1, x2, ...]> <, ring>)`
- ⌘ `poly(p <, [x1, x2, ...]> <, ring>)`
- ⌘ `poly(list, [x1, x2, ...] <, ring>)`

Parameter:

- `f` — ein polynomialer Ausdruck
- `x1, x2, ...` — die Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner.
- `ring` — der Koeffizientenring: entweder *Expr* oder *IntMod(n)* mit einer ganzen Zahl $n > 1$ oder ein Domain vom Typ `DOM_DOMAIN`. Der Standard ist der Ring *Expr* aller MuPAD-Ausdrücke.
- `p` — ein mittels `poly` erzeugtes Polynom vom Typ `DOM_POLY`
- `list` — eine Liste mit Koeffizienten und Exponenten

Rückgabewert: ein Polynom vom Domain-Typ `DOM_POLY`. `FAIL` wird zurückgeliefert, falls eine Konvertierung in ein Polynom nicht möglich ist.

Verwandte Funktionen: `Dom::DistributedPolynomial`,
`Dom::MultivariatePolynomial`, `Dom::Polynomial`,
`Dom::UnivariatePolynomial`, `RootOf`, `coeff`, `collect`, `degree`,
`degreevec`, `divide`, `evalp`, `expr`, `factor`, `gcd`, `ground`, `indets`,
`lcoeff`, `ldegree`, `lmonomial`, `lterm`, `mapcoeffs`, `nterms`, `nthcoeff`,
`nthmonomial`, `nthterm`, `poly2list`, `polylib`, `tcoeff`

Details:

- ⌘ MuPAD stellt zur Darstellung von Polynomen den Kerndatentyp `DOM_POLY` zur Verfügung, dessen Arithmetik effizienter ist als die Arithmetik für Ausdrücke. Weiterhin erlaubt dieser Datentyp die Verwendung spezieller Koeffizientenringe, die nicht mittels Ausdrücken dargestellt werden

können. Die Funktion `poly` dient dazu, Polynome dieses Datentyps zu erzeugen.

☞ `poly(f, [x1, x2, ...], ring)` wandelt den Ausdruck `f` in ein Polynom in den Unbestimmten `x1, x2, ...` über dem angegebenen Koeffizientenring um. Der Ausdruck `f` muss dabei nicht in expandierter Form eingegeben werden, die Expansion übernimmt `poly`.

Werden keine Unbestimmten angegeben, so wird intern in `f` danach gesucht. Ein Fehler wird ausgelöst, wenn keine Unbestimmten gefunden werden.

Ohne Angabe eines Koeffizientenrings wird `Expr` benutzt. In diesem Fall sind beliebige MuPAD-Ausdrücke als Koeffizienten zulässig.

`poly` gibt `FAIL` zurück, wenn der Ausdruck nicht in ein Polynom umgeformt werden kann. Siehe Beispiel ??.

☞ `poly(p, [x1, x2, ...], ring)` konvertiert das Polynom `p` vom Datentyp `DOM_POLY` in ein Polynom in den Unbestimmten `x1, x2, ...` über dem angegebenen Koeffizientenring. Man beachte, dass sowohl die Unbestimmten als auch der Koeffizientenring Teil der Datenstruktur `DOM_POLY` ist. Dieser Aufruf dient dazu, diese Daten zu verändern.

Werden keine Unbestimmten angegeben, so werden die Unbestimmten von `p` übernommen.

Wird kein Koeffizientenring angegeben, so wird der Koeffizientenring von `p` übernommen.

Siehe die Beispiele ?? und ??.

☞ `poly(list, [x1, x2, ...], ring)` wandelt die „Termliste“ `list` in ein Polynom in den Unbestimmten `x1, x2, ...` über dem angegebenen Koeffizientenring um. Siehe die Beispiele ?? und ??. Diese Aufrufsform von `poly` stellt den schnellsten Weg dar, Polynome vom Typ `DOM_POLY` zu erzeugen.

Die Liste muss einen Eintrag für jedes nicht-verschwindende Monom des Polynoms haben: für dünnbesetzte Polynome müssen nur die entsprechende Monome angegeben werden. Das Null-Polynom wird erzeugt, falls die Liste leer ist.

Jedes Element der Termliste muss wiederum eine Liste mit zwei Elementen sein: dem Koeffizienten des Monoms und dem Exponenten bzw. Exponentenvektor. Bei einem univariaten Polynom in der Variable x entspricht die Liste

$$[[c_1, e_1], [c_2, e_2], \dots]$$

dem Polynom $c_1 x^{e_1} + c_2 x^{e_2} + \dots$. Im multivariaten Fall sind die Exponentenvektoren Listen, welche die Exponenten aller Unbestimmten des

Polynoms angeben. Die Exponenten müssen in der gleichen Reihenfolge wie die Unbestimmten angegeben werden. Bei einem multivariaten Polynom in den Unbestimmten x_1, x_2, \dots entspricht die Liste

$$[[c_1, [e_{11}, e_{12}, \dots]], [c_2, [e_{21}, e_{22}, \dots]], \dots]$$

dem Polynom $c_1 x_1^{e_{11}} x_2^{e_{12}} \dots + c_2 x_1^{e_{21}} x_2^{e_{22}} \dots + \dots$.

Die Anordnung der Elemente der Termliste ist irrelevant. Es können auch mehrere Elemente angegeben werden, die zum selben Monom gehören. In diesem Fall werden die Koeffizienten addiert.

Mit diesem Aufruf können von `poly2list` erzeugte Termlisten in Polynome zurückverwandelt werden.

- ☞ Die Anordnung der Unbestimmten ist durch ihre Position in der Eingabeliste `[x1, x2, ...]` bestimmt. Werden sie nicht angegeben, so werden die Unbestimmten automatisch im Ausdruck `f` gesucht, wobei ihre Anordnung durch das System bestimmt wird. Siehe Beispiel ??.
- ☞ Unbestimmte brauchen nicht notwendigerweise Bezeichner oder indizierte Bezeichner zu sein. Auch nicht-rationale Ausdrücke wie z. B. `sin(x)`, `f(x)` oder `y^(1/3)` werden als Unbestimmte akzeptiert. Siehe Beispiel ??.
- ☞ `poly` ist eine Funktion des Systemkerns.

Option `<ring>`:

- ☞ Der Standardring `Expr` repräsentiert beliebige MuPAD-Ausdrücke. Semantisch entspricht dieser Ring dem Ring `Dom::ExpressionField()`. Polynome unterscheiden diese Ringe jedoch. Insbesondere ist die Arithmetik über `Expr` schneller.
- ☞ Der Ring `IntMod(n)` bezeichnet den Restklassenring $\mathbb{Z}/n\mathbb{Z}$ in symmetrischer Darstellung. Der Parameter `n` muss eine natürliche Zahl größer als 1 sein. Mathematisch stimmt dieser Ring mit `Dom::IntegerMod(n)` überein. Polynome unterscheiden diese Ringe jedoch. Insbesondere ist die Arithmetik über `IntMod` schneller. Durch `coeff` bestimmte Koeffizienten werden bei `IntMod` als ganze Zahlen vom Typ `DOM_INT` zurückgeliefert. Siehe die Beispiele ??, ?? und ??.
- ☞ Jedes Domain vom Typ `DOM_DOMAIN` kann als Koeffizientenring benutzt werden, wenn mit diesem Datentyp Arithmetik betrieben werden kann. Details hierzu sind im Abschnitt „Hintergründe“ unten zu finden.
Wird ein Koeffizientenring angegeben, so werden nur Elemente dieses Rings als Koeffizienten zugelassen. `poly` versucht, einen polynomialen Ausdruck `f` in ein Polynom über dem Koeffizientenring zu konvertieren. Für einige Koeffizientenringe kann jedoch kein polynomialer Ausdruck erzeugt werden, falls die Multiplikation mit Unbestimmten keine gültige

Operation des Rings ist. In diesem Fall kann das Polynom über eine Termliste definiert werden.

Siehe Beispiel ??.

Beispiel 1. Ein Aufruf von `poly` erzeugt ein Polynom aus einem polynomialen Ausdruck:

```
>> p := poly(2*x*(x + 3))
```

$$\text{poly}(2x^2 + 6x, [x])$$

Die Operatoren `*`, `+`, `-` und `^` können zum Rechnen mit Polynomen benutzt werden:

```
>> p^2 - p + poly(x, [x])
```

$$\text{poly}(4x^4 + 24x^3 + 34x^2 - 5x, [x])$$

Für die Multiplikation mit einer Konstanten muss entweder die Konstante in ein Polynom des entsprechenden Typs verwandelt werden, oder man verwendet `multcoeffs`:

```
>> poly(c, [x])*p = multcoeffs(p, c)
```

$$\text{poly}((2c)x^2 + (6c)x, [x]) = \text{poly}((2c)x^2 + (6c)x, [x])$$

```
>> delete p:
```

Beispiel 2. Ein Polynom kann auch mit Parametern erzeugt werden. Im folgenden Aufruf ist `y` ein Parameter und nicht eine Unbestimmte:

```
>> poly((x*(y + 1))^2, [x])
```

$$\text{poly}(y^2x^2 + 2yx^2 + x^2, [x])$$

Die Unbestimmten werden intern bestimmt, wenn sie nicht explizit angegeben werden. Der folgende Aufruf erzeugt aus dem oben schon benutzten Ausdruck ein multivariates Polynom:

```
>> poly((x*(y + 1))^2)
```

$$\text{poly}(y^2x^2 + 2yx^2 + x^2, [y, x])$$

Die Reihenfolge der Unbestimmten kann explizit vorgegeben werden:

```
>> poly((x*(y + 1))^2, [x, y])
```

$$\text{poly}(x^2 y^2 + 2 x^2 y + x^2, [x, y])$$

Beispiel 3. Die folgenden Polynome werden durch Termlisten erzeugt:

```
>> poly([[c2, 3], [c1, 7], [c3, 0]], [x])
```

$$\text{poly}(c1 x^7 + c2 x^3 + c3, [x])$$

```
>> poly([[c2, 3], [c1, 7], [c3, 0], [a, 3]], [x])
```

$$\text{poly}(c1 x^7 + (a + c2) x^3 + c3, [x])$$

Für multivariate Polynome müssen Exponentenvektoren als Listen angegeben werden:

```
>> poly([[c1, [2, 2]], [c2, [2, 1]], [c3, [2, 0]]], [x, y])
```

$$\text{poly}(c1 x^2 y^2 + c2 x^2 y + c3 x^2, [x, y])$$

Beispiel 4. Ausdrücke wie $f(x)$ können als Unbestimmte verwendet werden:

```
>> poly(f(x)*(f(x) + x^2))
```

$$\text{poly}(x^2 f(x) + f(x)^2, [x, f(x)])$$

Beispiel 5. Der Restklassenring $\text{IntMod}(7)$ ist ein gültiger Koeffizientenring:

```
>> p := poly(9*x^3 + 4*x - 7, [x], IntMod(7))
```

$$\text{poly}(2 x^3 - 3 x, [x], \text{IntMod}(7))$$

Intern wird bei Rechnungen mit Polynomen über diesem Ring modulare Arithmetik verwendet:

```
>> p^3
```

$$\text{poly}(x^9 - x^7 - 2x^5 + x^3, [x], \text{IntMod}(7))$$

Man beachte jedoch, dass Koeffizienten als einfache ganze Zahlen und nicht als spezielle Domain-Elemente zurückgegeben werden:

```
>> coeff(p)
```

2, -3

```
>> delete p:
```

Beispiel 6. Die Eingabe mittels einer Termliste kann mit der Vorgabe eines Koeffizientenrings kombiniert werden:

```
>> poly([[9, 3], [4, 1], [-2, 0]], [x], IntMod(7))
```

$$\text{poly}(2x^3 - 3x - 2, [x], \text{IntMod}(7))$$

Hierbei werden Eingabekoeffizienten als Elemente des Koeffizientenrings interpretiert, d. h., Konvertierungen wie z. B. $9 \bmod 7 \rightarrow 2$ werden bereits mit der Eingabe durchgeführt. Ein mathematisch äquivalentes Polynom kann auch mittels des Koeffizientenrings `Dom::IntegerMod(7)` definiert werden. Allerdings werden hierbei die Koeffizienten durch die Zahlen $0, \dots, 6$ statt durch $-3, \dots, 3$ dargestellt:

```
>> poly([[9, 3], [4, 1], [-2, 0]], [x], Dom::IntegerMod(7))
```

$$\text{poly}(2x^3 + 4x + 5, [x], \text{Dom::IntegerMod}(7))$$

Der folgende Versuch, ein Polynom mittels eines Ausdrucks zu definieren, scheitert, da die Multiplikation mit einem symbolischen Bezeichner innerhalb des Datentyps `Dom::IntegerMod(7)` nicht zulässig ist:

```
>> c := Dom::IntegerMod(7)(3)
```

3 mod 7

```
>> poly(c*x^2, [x], Dom::IntegerMod(7))
```

FAIL

In einem solchen Fall kann das Polynom unter Vermeidung von Ausdrücken mittels einer Termliste definiert werden:

```
>> poly([[c, 2]], [x], Dom::IntegerMod(7))

      2
poly(3 x , [x], Dom::IntegerMod(7))

>> delete c:
```

Beispiel 7. Es ist möglich, die Unbestimmten eines Polynoms zu ändern:

```
>> p:= poly((a + b)*x - a^2)*x, [x]): p, poly(p, [a, b])

      2      2
poly((a + b) x  + (- a ) x, [x]),

      2      2      2
poly((-x) a  + x  a + x  b, [a, b])
```

Beispiel 8. Es ist möglich, den Koeffizientenrings eines Polynoms zu ändern:

```
>> p := poly(-4*x + 5*y - 5, [x, y], IntMod(7)):
p, poly(p, IntMod(3))

poly(3 x - 2 y + 2, [x, y], IntMod(7)),

poly(y - 1, [x, y], IntMod(3))
```

Beispiel 9. Hier wird ein Polynom über dem Koeffizientenring `Dom::Float` erzeugt:

```
>> poly(3*x - y, Dom::Float)

poly(- 1.0 y + 3.0 x, [y, x], Dom::Float)
```

Über diesem Ring darf der Bezeichner `y` nicht in den Koeffizienten auftauchen, der er nicht in eine Gleitpunktzahl konvertiert werden kann:

```
>> poly(3*x - y, [x], Dom::Float)

FAIL
```

Hintergründe:

- ⇒ Ein Domain muss die folgende Einträge haben, um als Koeffizientenring benutzt werden zu können:
 - Der Eintrag "zero" muss das neutrale Element bezüglich der Addition enthalten.
 - Der Eintrag "one" muss das neutrale Element bezüglich der Multiplikation enthalten.
 - Die Methode "_plus" muss Domain-Elemente addieren.
 - Die Methode "_negate" muss ein Domain-Element negieren, d. h., das inverse Element bezüglich der Addition berechnen.
 - Die Methode "_mult" muss Domain-Elemente multiplizieren.
 - Die Methode "_power" muss eine ganzzahlige Potenz eines Domain-Elementes liefern. Sie wird mit dem Domain-Element als erstem Argument und der Potenz als zweitem Argument aufgerufen.
- ⇒ Weiterhin sollten folgende Methoden implementiert werden, die von Systemfunktionen wie gcd, diff, divide, norm etc. aufgerufen werden:
 - Die Methode "gcd" muss den größten gemeinsamen Teiler mehrerer Domain-Elemente zurückgeben.
 - Die Methode "diff" muss die Ableitung eines Domain-Elementes nach einer Variable liefern.
 - Die Methode "_divide" muss zwei Domain-Elemente dividieren. Sie muss FAIL zurückgeben, wenn dies nicht möglich ist.
 - Die Methode "norm" muss die Norm eines Domain-Elementes berechnen.
 - Die Methode "convert" muss einen Ausdruck in ein Domain-Element umwandeln. Sie muss FAIL zurückgeben, wenn dies nicht möglich ist.

Diese Methode wird aufgerufen, um die Koeffizienten polynomialer Ausdrücke in Elemente des Domains zu konvertieren. Existiert diese Methode nicht, so können nur Domain-Elemente zur Spezifikation der Koeffizienten verwendet werden.
 - Die Methode "expr" muss ein Domain-Element in einen Ausdruck überführen.

Die Systemfunktion expr ruft diese Methode auf, um ein Polynom über diesem Koeffizientenring in einen Ausdruck zu konvertieren. Fehlt diese Methode, wird ein Domain-Element direkt in einen Ausdruck eingesetzt.

Änderungen:

- ⇒ Keine Änderungen.

poly2list – Konvertierung eines Polynoms in eine Liste von Termen

`poly2list(p)` liefert eine Termliste, welche die Koeffizienten und Exponenten-Vektoren des Polynoms `p` enthält.

Aufruf(e):

```
# poly2list(p)
# poly2list(f <, vars>)
```

Parameter:

`p` — ein Polynom vom Typ `DOM_POLY`
`f` — ein polynomialer Ausdruck
`vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner

Rückgabewert: Eine Liste mit den Koeffizienten und Exponenten-Vektoren des Polynoms. `FAIL` wird zurückgeliefert, falls ein gegebener Ausdruck nicht in ein Polynom konvertiert werden kann.

Verwandte Funktionen: `coeff`, `coerce`, `degree`, `degreevec`, `lcoeff`, `poly`, `tcoeff`

Details:

- # Die zurückgelieferte Termliste ist eine Liste, welche diejenigen Monome enthält, deren Koeffizienten nicht 0 sind. Die Monome wiederum sind als Listen mit 2 Elementen dargestellt: Das erste Element ist der Koeffizient des Monoms, das 2-te Element ist entweder der Exponent oder ein Exponenten-Vektor: Ist das Polynom univariat, so ist der Exponent, andernfalls der Exponenten-Vektor enthalten. Die Exponenten-Vektoren haben die gleiche Form, wie sie auch von der Funktion `degreevec` geliefert wird. Ein Null-Polynom resultiert in einer leeren Termliste.
- # Die Elemente der Termliste werden lexikographisch nach den Exponenten-Vektoren sortiert. Das ist auch die Ordnung, die intern für die Terme eines Polynoms verwendet wird.
- # `poly2list(f, vars)` ist äquivalent zu `poly2list(poly(f, vars))`: Zuerst wird der polynomialer Ausdruck `f` in ein Polynom in den Variablen `vars` über den Ausdrücken konvertiert. Dann wird dieses Polynom in eine Termliste konvertiert. Werden keine Variablen angegeben,

so werden die freien Unbestimmten in `f` als Variablen verwendet. Siehe die Funktion `poly` zu den Details, wie ein polynomialer Ausdruck in ein Polynom konvertiert wird. `FAIL` wird zurückgeliefert, falls sich der Ausdruck nicht in ein Polynom konvertieren lässt.

⚡ `poly2list` ist eine Funktion des Systemkerns.

Beispiel 1. Die folgenden Ausdrücke definieren univariate Polynome. Also werden Exponenten und keine Exponenten-Vektoren in den Termlisten zurückgeliefert:

```
>> poly2list(2*x^100 + 3*x^10 + 4)
      [[2, 100], [3, 10], [4, 0]]

>> poly2list(2*x*(x + 1)^2)
      [[2, 3], [4, 2], [2, 1]]
```

Durch Angabe einer Liste von Unbestimmten können symbolische Parameter von den Unbestimmten unterschieden werden:

```
>> poly2list(a*x^2 + b*x + c, [x])
      [[a, 2], [b, 1], [c, 0]]
```

Beispiel 2. Das folgende Polynom hat 2 Variablen, also werden Exponenten-Vektoren geliefert:

```
>> poly2list((x*(y + 1))^2, [x, y])
      [[1, [2, 2]], [2, [2, 1]], [1, [2, 0]]]
```

Beispiel 3. Hier wird ein Polynom vom Domain-Typ `DOM_POLY` angegeben. Dies ist die einzige Möglichkeit, ein Polynom anzugeben, dessen Koeffizienten nicht aus Ausdrücken besteht:

```
>> poly2list(poly(-4*x + 5*y - 5, [x, y], IntMod(7)))
      [[3, [1, 0]], [-2, [0, 1]], [2, [0, 0]]]
```

Änderungen:

⌘ Keine Änderungen.

polygon – Erzeugen eines graphischen Polygons

`polygon(p1, p2, ...)` definiert ein Polygon mit den Knoten `p1, p2` etc.

Aufruf(e):

⌘ `polygon(p1, p2, ... <, Closed = b1> <, Filled = b2> <, Color = [r, g, b]>)`

Parameter:

`p1, p2, ...` — mittels der Funktion `point` erzeugte graphische Punkte. Ein 2D-Polygon wird erzeugt, wenn diese Punkte 2D-Punkte sind. Anderenfalls wird ein 3D-Polygon erzeugt.

Optionen:

`Closed = b1` — `b1` kann entweder `TRUE` oder `FALSE` sein. Bei `TRUE` wird der erste Punkt `p1` intern an die Punkte angefügt, sodass ein geschlossenes Polygon entsteht. Die Standardeinstellung ist `Closed = FALSE`.

`Filled = b2` — `b2` kann entweder `TRUE` oder `FALSE` sein. Bei `FALSE` wird das Polygon als ein aus Linienstücken zusammengesetzter Polygonzug gezeichnet. Bei `TRUE` wird das Polygon als eine gefüllte Fläche gezeichnet. Die Standardeinstellung ist `Filled = FALSE`.

`Color = [r, g, b]` — setzt einen Farbwert im RGB-Format mit den Rot-Grün-Blau-Anteilen `r, g, b`. Diese Parameter müssen reelle Zahlenwerte aus dem Intervall von 0 bis 1 sein.

Rückgabewert: ein Objekt vom Domain-Typ `DOM_POLYGON`.

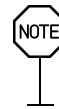
Verwandte Funktionen: `plot`, `plot::Polygon`, `plot2d`, `plot3d`, `plotfunc2d`, `plotfunc3d`, `point`, `RGB`

Details:

- ☞ Durch `polygon` erzeugte Polygone sind graphische Primitive, die mittels `plot2d` bzw. `plot3d` im Listenformat `[Mode = List, [...Primitive...]]` dargestellt werden können.
- ☞ Die `plot`-Bibliothek stellt mit `plot::Polygon` ein alternatives graphisches Polygonprimitiv zur Verfügung. Dieses Objekt ist wesentlich flexibler als das durch `polygon` erzeugte Polygonobjekt. Das erstere kann in allen Funktionen der `plot`-Bibliothek eingesetzt werden, während das letztere nur in einem Aufruf von `plot2d` bzw. `plot3d` verwendet werden kann.
- ☞ `polygon` ist eine Funktion des Systemkerns.

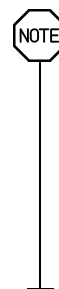
Option **<Filled = b2>**:

- ☞ Mit `Filled = TRUE` wird automatisch ein geschlossenes Polygon erzeugt, d. h., der erste Punkt `p1` wird intern an die Punktfolge angehängt. Das Polygon wird als eine gefüllte Fläche dargestellt.
- ☞ Ist `Closed = FALSE` gesetzt, so werden die Kanten des Polygons mit der Füllfarbe des Inneren dargestellt. Mit `Closed = TRUE` werden die Kanten in der aktuellen Vordergrundfarbe gezeichnet.
- ☞ Gefüllte 3D-Polygone können mit maximal drei Punkten (Dreiecke) definiert werden. Mittels `plot::Polygon` können komplexere gefüllte 3D-Polygone erzeugt werden.



Option **<Color = [r, g, b]>**:

- ☞ Die Farbwerte `r, g, b` müssen Zahlen oder numerische Ausdrücke sein, die mittels `float` in reelle Gleitpunktzahlen aus dem Intervall `[0.0, 1.0]` konvertiert werden können. Ein Fehler wird ausgelöst, falls einer der Werte nicht in diesem Bereich liegt. Symbolische Ausdrücke wie z. B. `PI - 3`, `exp(-sqrt(2))` etc. werden akzeptiert und automatisch in Gleitpunktwerte verwandelt. Man beachte jedoch, dass Ausdrücke mit symbolischen Bezeichnungen nicht akzeptiert werden!
- ☞ In der Definition der Eckpunkte mittels der Funktion `point` können Punktfarben spezifiziert werden. Diese Farben werden ignoriert.
- ☞ Das Domain `RGB` enthält zahlreiche vordefinierte Farbnamen.



Operanden: Die ersten Operanden eines Polygons sind die eingegebenen Punkte. Der drittletzte Operand ist die Liste `[r, g, b]` der Farbwerte. Der zweitletzte Operand ist der boolsche Werte `b1` aus `Closed = b1`. Der letzte Operand ist der boolsche Werte `b2` aus `Filled = b2`.

Beispiel 1. Die Eckpunkte eines 2D-Dreiecks werden definiert:

```
>> p1 := point(0, 0): p2 := point(0, 1): p3 := point(1, 0):
```

Die Kanten des Dreiecks werden mittels `plot2d` dargestellt:

```
>> plot2d(Axes = None, [Mode = List,
      [polygon(p1, p2, p3, Closed = TRUE, Color = RGB::Black)]
    ])
```

Der folgende Aufruf zeichnet die Dreiecksfläche:

```
>> plot2d(Axes = None, [Mode = List,
      [polygon(p1, p2, p3, Filled = TRUE, Color = RGB::Red)]])
```

Der folgende Aufruf zeichnet die Dreiecksfläche zusammen mit den Kanten:

```
>> plot2d(Axes = None, [Mode = List,
      [polygon(p1, p2, p3, Closed = TRUE, Filled = TRUE,
        Color = RGB::Red)]])
```

```
>> delete p1, p2, p3:
```

Beispiel 2. Eine Reihe von 2D-Punkten auf dem Graphen der Kosinus-Funktion wird definiert:

```
>> for i from 0 to 12 do
      p[i] := point(i, cos(i*PI/6)):
end_for:
```

Mit diesen Punkten wird ein Polygonzug erzeugt und gezeichnet:

```
>> plot2d(Scaling = UnConstrained,
      [Mode = List, [polygon(p[i] $ i = 0..12)]])
```

Der folgende Aufruf erzeugt die Fläche zwischen dem Graphen der Cosinus-Funktion und der x -Achse:

```
>> plot2d(Scaling = UnConstrained, [Mode = List,
      [polygon(point(0, 0), p[i] $ i = 0..12, point(12, 0),
        Closed = TRUE, Filled = TRUE)]])
```

Der folgende Aufruf zerlegt die Fläche zwischen dem Graphen der Kosinus-Funktion und der x -Achse in Trapeze. Diese werden als Liste gefüllter Polygone gezeichnet:

```
>> plot2d(Scaling = UnConstrained, [Mode = List,
    [polygon(point(i, 0), p[i], p[i+1], point(i + 1, 0),
        Closed = TRUE, Filled = TRUE) $ i = 0..11]])

>> delete p:
```

Beispiel 3. Die Eckpunkte eines 3D-Dreiecks werden definiert:

```
>> a := point(0, 0, 1): b := point(1, 1, 1): c := point(1, 0, 1):
```

Das Dreieck wird auf unterschiedliche Weisen dargestellt:

```
>> plot3d(Axes = None, [Mode = List, [polygon(a, b, c)] ])

>> plot3d(Axes = None,
    [Mode = List, [polygon(a, b, c, Closed = TRUE)]]])

>> plot3d(Axes = None,
    [Mode = List, [polygon(a, b, c, Filled = TRUE)]]])

>> plot3d(Axes = None, [Mode = List,
    [polygon(a, b, c, Closed = TRUE, Filled = TRUE)]]])

>> plot3d(Axes = None, LineWidth = 30, [Mode = List,
    [polygon(a, b, c, Closed = TRUE, Filled = TRUE)]]])

>> delete a, b, c:
```

Änderungen:

⌘ Keine Änderungen.

polylog – die Polylogarithmusfunktion

`polylog(n, x)` stellt die Polylogarithmusfunktion $Li_n(x)$ vom Index n am Punkt x dar.

Aufruf(e):

⌘ `polylog(n, x)`

Parameter:

- n — ein arithmetischer Ausdruck, der eine ganze Zahl repräsentiert
- x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente x reagiert die Funktion auf die Umgebungsvariable DIGITS, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: dilog, ln

Details:

- ☞ Für eine komplexe Zahl x vom Betrag $|x| < 1$ ist die Polylogarithmusfunktion vom Index n definiert als

$$Li_n(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^n}.$$

Durch analytische Fortsetzung wird diese Funktion auf der ganzen komplexen Ebene definiert.

- ☞ Für ganzzahliges n und Gleitpunktzahlen x werden Gleitpunktapproximationen berechnet.
- ☞ Für ganzzahliges $n \leq 1$ wird für jedes x ein expliziter Ausdruck zurückgeliefert. Für ganzzahliges $n > 1$ und für symbolisches n wird ein unevaluierter Aufruf von polylog zurückgeliefert, falls x keine Gleitpunktzahl ist. Ein Fehler wird ausgelöst, wenn n numerisch, aber keine ganze Zahl ist.
- ☞ Einige spezielle Werte sind für $n = 2$ implementiert (siehe dilog). Die Werte $Li_n(0) = 0$ und $Li_n(1) = \text{zeta}(n)$ werden für jedes n zurückgeliefert. Weiterhin gilt $Li_n(-1) = (2^{1-n} - 1) \text{zeta}(n)$ für jedes $n \neq 1$.
- ☞ $Li_n(x)$ hat für $n \leq 1$ eine Singularität am Punkt $x = 1$. Für Indizes $n \geq 1$ ist der Punkt $x = 1$ ein Verzweigungspunkt. Der Verzweigungsschnitt ist das reelle Intervall $[1, \infty)$. Beim Überschreiten des Schnitts springen die Funktionswerte. Siehe Beispiel ??.
- ☞ $\text{polylog}(2, x)$ stimmt mathematisch mit $\text{dilog}(1-x)$ überein.

Beispiel 1. Für ganzzahliges $n \leq 1$ werden explizite Ergebnisse geliefert:

```
>> polylog(-5, x), polylog(-1, x), polylog(0, x), polylog(1, x)
```

$$\begin{aligned} & \frac{x^2 + 26x^3 + 66x^4 + 26x^5 + x^6}{(1-x)^6}, \frac{x}{(1-x)^2}, \frac{x}{1-x}, -\ln(1-x) \end{aligned}$$

Für ganzzahliges $n > 1$ oder symbolisches n wird ein unevaluierter Aufruf zurückgeliefert:

```
>> polylog(2, x), polylog(n^2 + 1, 2), polylog(n + 1, 2.0)
```

$$\text{polylog}(2, x), \text{polylog}(n^2 + 1, 2), \text{polylog}(n + 1, 2.0)$$

Bei ganzzahligem n wird für Gleitkommazahlen x ein numerische Wert berechnet:

```
>> polylog(-5, -1.2), polylog(10, 100.0 + 3.2*I)
-0.2326930882, 104.9131863 + 11.44600047 I
```

Bei nicht ganzzahligem numerischen n wird ein Fehler ausgelöst:

```
>> polylog(5/2, x)
Error: first argument must be an integer [polylog]
```

Einige spezielle symbolische Werte sind implementiert:

```
>> polylog(4, 1), polylog(5, -1), polylog(2, I)
```

$$\frac{\pi^4}{90}, -\frac{15 \text{ zeta}(5)}{16}, I \text{ CATALAN} - \frac{\pi^2}{48}$$

```
>> assume(n <> 1): polylog(n, -1)
```

$$-\text{zeta}(n) \frac{1-n}{1-2^n}$$

```
>> unassume(n): polylog(n, -1)
```

$$\text{polylog}(n, -1)$$

Beispiel 2. Für Indizes $n \geq 1$ ist das reelle Intervall $[1, \infty)$ ein Verzweigungsschnitt. Die von `polylog` gelieferten Werte springen beim Überschreiten des Schnitts:

```
>> polylog(3, 1.2 + I/10^1000) - polylog(3, 1.2 - I/10^1000)
0.1044301529 I
```

Beispiel 3. Die Funktionen `diff`, `float`, `limit` und `series` verarbeiten `polylog`:

```
>> diff(polylog(n, x), x), float(polylog(4, 3 + I))
polylog(n - 1, x)
-----, 3.177636803 + 1.859135861 I
x

>> series(polylog(4, sin(x)), x = 0)
      2      3      4      5
      x      25 x      13 x      1523 x
x + -- - ---- - ---- + ---- + O(x )
    16    162    768    405000
```

Hintergründe:

- ⌘ Die Polylogarithmen sind durch $\frac{d}{dx} Li_n(x) = \frac{1}{x} Li_{n-1}(x)$, $Li_n(0) = 0$ und $Li_0(x) = -\ln(1-x)$ charakterisiert. Für $n \leq 0$ ist $Li_n(x)$ eine rationale Funktion in x .
- ⌘ Li_n hat einen Verzweigungsschnitt längs des reellen Intervalls $[1, \infty)$ für Indizes $n \geq 1$. Der Wert an einem Punkt x auf dem Schnitt stimmt mit dem Grenzwert „von unten“ überein:

$$Li_n(x) = \lim_{\epsilon \rightarrow 0_+} Li_n(x - \epsilon i) = \lim_{\epsilon \rightarrow 0_+} Li_n(x + \epsilon i) - \frac{2\pi i}{(n-1)!} \ln(x)^{n-1}.$$

- ⌘ Literatur: L. Lewin, "Polylogarithms and Related Functions", North Holland (1981). L. Lewin (ed.), "Structural Properties of Polylogarithms", Mathematical Surveys and Monographs Vol. 37, American Mathematical Society, Providence (1991).

Änderungen:

- ⌘ `polylog` ist eine neue Funktion.

powermod – modulares Potenzieren einer Zahl oder eines Polynoms

`powermod(b, e, m)` berechnet $b^e \bmod m$.

Aufruf(e):

⌘ `powermod(b, e, m)`

Parameter:

- b — die Basis: eine Zahl oder ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- e — die Potenz: eine nicht-negative ganze Zahl
- m — der Modulus: eine Zahl oder ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck

Rückgabewert: Abhängig vom Typ der Basis `b` wird eine Zahl, ein Polynom oder ein polynomialer Ausdruck zurückgeliefert. `FAIL` wird zurückgeliefert, falls ein Ausdruck nicht in ein Polynom konvertiert werden kann.

Überladbar durch: `b`

Verwandte Funktionen: `_mod, divide, modp, mods, poly`

Details:

- ⌘ Wenn `b` und `m` Zahlen sind, so kann die Potenz $b^e \bmod m$ auch durch den direkten Aufruf `b^e mod m` berechnet werden. Die Berechnung über `powermod(b, e, m)` ist jedoch vorzuziehen: unter Vermeidung des Zwischenergebnisses b^e wird die modulare Potenz wesentlich effizienter berechnet.
- ⌘ Ist `b` eine rationale Zahl, so wird das modulare Inverse des Nenners berechnet und dann mit dem Zähler multipliziert.
- ⌘ Ist der Modulus `m` eine ganze Zahl, so muss die Basis `b` entweder eine Zahl sein oder in ein `IntMod(m)`-Polynom konvertierbar sein.
- ⌘ Ist der Modulus `m` ein polynomialer Ausdruck, so muss die Basis `b` entweder eine Zahl, ein polynomialer Ausdruck oder ein Polynom über dem Koeffizientring aller MuPAD-Ausdrücke sein.
- ⌘ Ist der Modulus `m` ein Polynom vom Domain-Typ `DOM_POLY`, so muss die Basis `b` entweder eine Zahl sein, oder ein Polynom, das mit dem Typ von `m` übereinstimmt, oder ein polynomialer Ausdruck, der in ein Polynom vom selben Typ wie `m` konvertierbar ist.
- ⌘ Die für modulare Arithmetik zuständige Systemfunktion `_mod` kann vom Benutzer umdefiniert werden: siehe die Hilfeseite von `_mod`. Die Funktion `powermod` benutzt `_mod` und reagiert entsprechend. Siehe Beispiel ??.
- ⌘ Polynome werden intern mittels der Funktion `divide` dividiert.

Beispiel 1. Wir berechnen $3^{123456} \bmod 7$:

```
>> powermod(3, 123456, 7)
```

1

Ist die Basis eine rationale Zahl, so wird zuerst das modulare Inverse des Nenners berechnet und dann modular mit dem Zähler multipliziert:

```
>> powermod(3/5, 1234567, 7)
```

2

Beispiel 2. Die Koeffizienten des folgenden polynomialen Ausdrucks werden modulo 7 berechnet:

```
>> powermod(x^2 + 7*x - 3, 10, 7)
```

$$3x^2 - x^4 - 3x^6 + x^{14} - x^{16} - 2x^{18} + x^{20} - 3$$

Beispiel 3. Die Potenz des folgenden polynomialen Ausdrucks wird modulo des Polynoms $x^2 + 1$ reduziert:

```
>> powermod(x^2 + 7*x - 3, 10, x^2 + 1)
```

1029668584 x - 534842913

Beispiel 4. Der Typ des Rückgabewerts stimmt mit dem der Basis überein. Ist diese ein Polynom, so wird ein Polynom zurückgeliefert:

```
>> powermod(poly(x^2 + 7*x - 3), 2, x^2 + 1),
    powermod(poly(x^2 + 7*x - 3), 2, poly(x^2 + 1))
```

poly(- 56 x - 33, [x]), poly(- 56 x - 33, [x])

Ist die Basis eine polynomialer Ausdruck, so liefert powermod einen polynomialen Ausdruck:

```
>> powermod(x^2 + 7*x - 3, 2, x^2 + 1),
    powermod(x^2 + 7*x - 3, 2, poly(x^2 + 1))
```

- 56 x - 33, - 56 x - 33

Beispiel 5. Die folgende Umdefinition von `_mod` führt zu einer symmetrischen Darstellung modularer Zahlen:

```
>> alias(R = Dom::IntegerMod(17)):
    _mod := mods: powermod(poly(2*x^2, R), 3, poly(3*x + 1, R))

poly(-4, [x], R)
```

Der folgende Aufruf stellt die Standarddarstellung wieder her:

```
>> _mod := modp: powermod(poly(2*x^2, R), 3, poly(3*x + 1, R))

poly(13, [x], R)

>> unalias(R):
```

Änderungen:

⌘ Keine Änderungen.

print – Bildschirmausgabe von Objekten

`print(object)` gibt `object` auf dem Bildschirm aus.

Aufruf(e):

```
⌘ print(<Unquoted,> <NoNL,> <KeepOrder,> object1,
        object2, ...)
```

Parameter:

`object1, object2, ...` — beliebige MuPAD-Objekte

Optionen:

Unquoted — Zeichenketten werden ohne Anführungsstriche und mit expandierten Steuerzeichen `'\n'`, `'\t'` und `'\\'` ausgegeben.

NoNL — Genau wie *Unquoted*, aber zusätzlich findet am Ende kein Zeilenumbruch statt. `PRETTYPRINT` wird implizit auf `FALSE` gesetzt.

KeepOrder — Operanden von Summen (vom Typ `"_plus"`) behalten ihre interne Ordnung bei.

Rückgabewert: `print` liefert das leere Objekt `null()` vom Typ `DOM_NULL` zurück.

Überladbar durch: `object1, object2, ...`

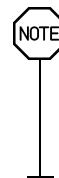
Seiteneffekte: `print` reagiert auf die Umgebungsvariablen `DIGITS`, `PRETTYPRINT` und `TEXTWIDTH` und auf die Ausgabe-Präferenzen `Pref::floatFormat`, `Pref::keepOrder`, `Pref::matrixSeparator`, `Pref::timesDot` und `Pref::trailingZeroes`.

Verwandte Funktionen: `DIGITS`, `DOM_FUNC_ENV`, `expose`, `expr2text`, `fininput`, `fprint`, `fread`, `funcenv`, `input`, `Pref::floatFormat`, `Pref::keepOrder`, `Pref::matrixSeparator`, `Pref::timesDot`, `Pref::trailingZeroes`, `PRETTYPRINT`, `protocol`, `read`, `TEXTWIDTH`, `userinfo`, `write`

Details:

- ☞ Auf interaktiver Ebene wird das Ergebnis eines MuPAD-Kommandos, das am Kommando Prompt eingegeben wird, üblicherweise automatisch auf dem Bildschirm ausgegeben. `print` dient dazu, innerhalb von Schleifen oder Prozeduren weitere Ausgaben zu erzeugen.
- ☞ Außer einigen weiter unten aufgeführten Ausnahmen sind die von `print` erzeugten Ausgaben identisch zu den üblichen Ausgaben von MuPAD Ergebnissen auf interaktiver Ebene.
- ☞ `print` evaluiert seine Argumente der Reihe nach von links nach rechts (siehe Beispiel ??) und zeigt die Ergebnisse auf dem Bildschirm an. Die einzelnen Ausgaben sind durch Kommata getrennt. Am Ende findet ein Zeilenumbruch statt, wenn dies nicht durch die Option `NONL` unterbunden wird.
- ☞ Die Breite einer Ausgabe durch `print` wird durch die Umgebungsvariable `TEXTWIDTH` beschränkt. Siehe Beispiel ??.
- ☞ Die Ausgabe findet in dem durch die Umgebungsvariable `PRETTYPRINT` gegebenen Ausgabeformat statt. Siehe Beispiel ??. `print` erzeugt immer ASCII Ausgaben, selbst unter Windows, wenn die Formelsatzausgabe für Ergebnisausgaben aktiviert ist.
- ☞ `print` steigt rekursiv in die Operanden eines Ausdrucks ab. Für jedes Teilobjekt `s` bestimmt `print` zuerst den Domain-Type `T`. Wenn das Domain `T` einen "print"-Slot hat, dann ruft `print T::print(s)` auf. Im Gegensatz zum Überladungs-Mechanismus für die meisten anderen MuPAD-Funktionen bearbeitet `print` das Ergebnis dieser Evaluierung wieder rekursiv und das Ergebnis dieses rekursiven Prozesses wird anstelle von `s` ausgegeben (siehe Beispiel ??).

Das von der "print"-Methode zurückgelieferte Ergebnis darf das Domainelement *s* selbst nicht als Teilobjekt enthalten, weil dies zu einer Endlos-Rekursion führt (siehe Beispiel ??). Das gilt auch für Ausgabe-Prozeduren von Funktionsumgebungen (siehe unten).



Wenn *T* ein Kern-Domain ohne einen "print"-Slot ist, dann wird die Ausgabe von *s* von *print* selbst erzeugt.

Wenn *T* ein Bibliotheks-Domain ohne einen "print"-Slot ist und die internen Operanden von *s* sind *op1*, *op2*, ..., dann wird *s* als *new(T, op1, op2, ...)* ausgegeben. (Siehe Beispiel ??.)

- ☞ Selbst die Ausgabe von Elementen von Kern-Domains kann durch die Definition einer "print"-Methode verändert werden. Siehe Beispiel ??.
- ☞ "print"-Methoden dürfen Zeichenketten oder Ausdrücke zurückliefern. Zeichenketten werden immer ohne Anführungszeichen ausgegeben. Ausdrücke werden normal ausgegeben. Wenn sie Zeichenketten enthalten, werden diese mit Anführungsstrichen ausgegeben. Siehe Beispiel ??.
- ☞ Die Ausgabe von Ausdrücken wird durch den 0-ten Operanden des Ausdrucks bestimmt. Wenn der 0-te Operand eine Funktionsumgebung ist, so übernimmt dessen zweiter Operand die Ausgabe des Ausdrucks (siehe Beispiele ?? und ??). Andernfalls wird der Ausdruck in funktionaler Notation ausgegeben.
- ☞ Im Gegensatz zur üblichen Ausgabe von MuPAD-Objekten auf interaktiver Ebene, führt *print* keine Rücksubstitution von Aliassen durch (siehe *Pref::alias* zu Details). Weiterhin werden die durch *Pref::output* und *Pref::postOutput* gegebenen Routinen nicht von *print* aufgerufen. Siehe Beispiel ??.
- ☞ Die Ausgabe von Gleitpunktzahlen hängt von der Umgebungsvariablen *DIGITS* und den Einstellungen von *Pref::floatFormat* (Exponential- oder Gleitpunktdarstellung) und *Pref::trailingZeroes* (Ausgabe von nachfolgenden Nullen) ab. Siehe Beispiel ??.
- ☞ *print* ist eine Funktion des Systemkerns.

Option *<Unquoted>*:

- ☞ Mit dieser Option werden Zeichenketten ohne Anführungszeichen ausgegeben. Zusätzlich werden die Steuerzeichen '\n', '\t' und '\\\' in Zeichenketten zu einem Zeilenumbruch, einem Tabulatorvorschub und einem einzelnen Backslash '\' expandiert. Siehe Beispiel ??.
- ☞ Das Steuerzeichen '\t' wird mit der Tabulator-Breite 8 expandiert. Das nächste Zeichen steht dann in der nächsten Spalte *i* mit $i \bmod 8 = 0$.

Option <NoNL>:

- ⌘ Diese Option hat die gleiche Auswirkung wie *Unquoted*. Zusätzlich wird der Zeilenwechsel am Ende der Ausgabe unterdrückt. Siehe Beispiel ??.
- ⌘ Außerdem setzt diese Option `PRETTYPRINT` implizit auf `FALSE`.

Option <KeepOrder>:

- ⌘ Diese Option bestimmt die Termordnung in Summen. Normalerweise sortiert das System die Terme einer Summe so um, dass an der ersten Stelle der Ausgabe ein positiver Term steht. Wenn *KeepOrder* angegeben wird, findet keine solche Umsortierung statt, und Summen werden in der internen Ordnung ausgegeben (siehe Beispiel ??).
- ⌘ Dieses Verhalten kann auch mittels `Pref::keepOrder` gesteuert werden. Genauer gesagt erzeugt `print(KeepOrder, ...)` dieselbe Ausgabe wie der folgende Aufruf:

```
Pref::keepOrder(Always): print(...): Pref::keepOrder(%2):
```

Beispiel 1. Dieses Beispiel zeigt einen einfachen Aufruf von `print` mit Zeichenketten als Argumenten. Sie werden in Anführungszeichen ausgegeben:

```
>> print("Hello", "You"." !"):
      "Hello", "You !"
```

Beispiel 2. Wie die meisten anderen Funktionen evaluiert `print` seine Argumente. Im folgenden Aufruf evaluiert sich `x` zu 0 und `cos(0)` evaluiert sich zu 1:

```
>> a := 0: print(cos(a)^2):
      1
```

Mit `hold` läßt sich der Ausdruck `cos(a)^2` unevaluiert ausgeben:

```
>> print(hold(cos(a)^2)):
      2
      cos(a)
```

```
>> delete a:
```

Beispiel 3. `print` berücksichtigt den aktuellen Wert von `TEXTWIDTH`:

```
>> print(expand((a + b)^4)):
      old := TEXTWIDTH: TEXTWIDTH := 30:
      print(expand((a + b)^4)):
      TEXTWIDTH := old:

          4      4      3      3      2  2
          a  + b  + 4 a b  + 4 a  b + 6 a  b

      4      4      3      3
      a  + b  + 4 a b  + 4 a  b +

          2  2
          6 a  b

>> delete old:
```

Beispiel 4. `print` berücksichtigt den aktuellen Wert von `PRETTYPRINT`:

```
>> print(a/b):
      old := PRETTYPRINT: PRETTYPRINT := FALSE:
      print(a/b):
      PRETTYPRINT := old:

          a
          -
          b

      a/b

>> delete old:
```

Beispiel 5. Hier wird demonstriert, wie man eine formatierte Ausgabe von Elementen eines Nutzer-definierten Domains erhalten kann. Angenommen wir wollen ein neues Domain `Complex` für komplexe Zahlen schreiben. Jedes Element von diesem Domain hat zwei Operanden: den Realteil `r` und den Imaginärteil `s`:

```
>> Complex := newDomain("Complex"): z := new(Complex, 1, 3):
      z + 1;
      print(z + 1):

      (new(Complex, 1, 3)) + 1

      (new(Complex, 1, 3)) + 1
```

Jetzt wollen wir eine schönere Ausgabe der Elemente dieses Domains in der Form $r+s*I$, wobei I die imaginäre Einheit ist. Wir implementieren die Slot-Routine `Complex::print`, die dies tut. Diese Slot-Routine wird jedes Mal mit einem Element des Domains `Complex` als Argument von MuPAD aufgerufen, wenn ein solches Element auf dem Bildschirm ausgegeben werden soll:

```
>> Complex::print := (z -> extop(z, 1) + extop(z, 2)*I):
    z + 1;
    print(z + 1):

(1 + 3 I) + 1

(1 + 3 I) + 1

>> delete Complex, z:
```

Beispiel 6. Der Rückgabewert einer solchen "print"-Methode darf ihr Argument nicht als Teilobjekt enthalten, sonst führt dies zu einer Endlos-Rekursion. Im folgenden Beispiel würde die Slot-Routine `T::print` unendlich oft aufgerufen werden. MuPAD versucht solche Endlos-Rekursionen abzufangen und gibt statt dessen '????' aus:

```
>> T := newDomain(T): T::print := id:
    new(T, 1);
    print(new(T, 1)):

'????'

'????'

>> delete T:
```

Beispiel 7. "print"-Methoden sind auch für Kern-Domains möglich. Dieses Beispiel zeigt, wie man die Ausgabe von Polynomen neu definiert, indem man nur den polynomialen Ausdruck ausgibt:

```
>> poly(x + 1);
    print(poly(x + 1)):

poly(x + 1, [x])

poly(x + 1, [x])

>> unprotect(DOM_POLY): DOM_POLY::print := p -> op(p, 1):
    poly(x + 1);
    print(poly(x + 1)):
    delete DOM_POLY::print: protect(DOM_POLY):
```

```
x + 1
```

```
x + 1
```

Beispiel 8. Wenn eine "print"-Methode eine Zeichenkette liefert, so wird sie ohne Anführungszeichen ausgegeben:

```
>> Example := newDomain("Example"): e := new(Example, 1):  
Example::print := x -> "elementOfExample":  
print(e):
```

```
elementOfExample
```

Wenn eine "print"-Methode einen Ausdruck liefert, dann wird dieser normal ausgegeben. Wenn der Ausdruck Zeichenketten enthält, so werden sie wie üblich mit Anführungszeichen ausgegeben:

```
>> Example::print := x -> ["elementOfExample", extop(x)]:  
print(e):
```

```
["elementOfExample", 1]
```

```
>> delete Example, e:
```

Beispiel 9. Angenommen, eine selbst-definierte Funktion f liefert sich selbst als symbolischen Ausdruck in der Form $f(x, \dots)$ zurück und man möchte diesen in einer speziellen Art ausgeben. Dazu bettet man seine Prozedur f in eine Funktionsumgebung ein und definiert eine Ausgabe-Prozedur als zweites Argument im zugehörigen `funcenv`-Aufruf. Wann immer ein Ausdruck der Form $f(x, \dots)$ ausgegeben werden soll, wird diese Ausgabe-Prozedur mit den Argumenten x, \dots des Ausdrucks aufgerufen:

```
>> f := funcenv(f,  
    proc(x) begin  
        if nops(x) = 2 then  
            "f does strange things with its arguments ".  
            expr2text(op(x, 1))." and ".expr2text(op(x, 2))  
        else  
            FAIL  
        end  
    end):
```

```
>> delete a, b:  
f(a, b)/2;  
f(a, b, c)/2
```

```
f does strange things with its arguments a and b
```

$$\frac{f(a, b, c)}{2}$$

```
>> delete f:
```

Beispiel 10. Für alle vordefinierten Funktionsumgebungen ist der zweite Operand eine interne Ausgabefunktion vom Typ `DOM_EXEC`. Dies ist insbesondere der Fall bei Operatoren wie `+`, `*`, `^` usw. Im folgenden Beispiel ändern wir das Ausgabe-Symbol für den Potenz-Operator `^`, der im dritten Operanden der internen Ausgabe-Funktion in der Funktionsumgebung `_power` gespeichert ist, zu einem doppelten Stern:

```
>> unprotect(_power):
    _power := subsop(_power, [2, 3] = "***"):
    a^b/2;
    print(a^b/2):
    _power := subsop(_power, [2, 3] = "^"):
    protect(_power):
```

$$\frac{a^{**b}}{2}$$

$$\frac{a^{**b}}{2}$$

Beispiel 11. Bei der Option *Unquoted* werden Anführungszeichen weglassen:

```
>> print(Unquoted, "Hello", "You"." !"):
```

```
Hello, You !
```

Mit *Unquoted* werden die speziellen Zeichen `'\t'` und `'\n'` expandiert:

```
>> print(Unquoted, "As you can see\n".
    "'\n' is the newline character\n".
    "\tand '\t' a tabulator"):
```



```
As you can see
'\n' is the newline character
and '\t' a tabulator
```

Beispiel 12. Es ist nützlich, Ausgabe-Zeichenketten mit `expr2text` und dem Konkatenationsoperator `.` zu konstruieren:

```
>> d := 5: print(Unquoted, "d plus 3 = ".expr2text(d + 3)):

d plus 3 = 8

>> delete d:
```

Beispiel 13. Mit der Option `NoNL` wird `PRETTYPRINT` implizit auf `FALSE` gesetzt und es findet kein Zeilenumbruch am Ende der Ausgabe statt. Ansonsten ist das Verhalten genauso wie bei der Option `Unquoted`:

```
>> print(NoNL, "Hello"): print(NoNL, ", You"." !\n"):
print(NoNL, "As you can see PRETTYPRINT is FALSE: "):
print(NoNL, x^2-1): print(NoNL, "\n"):

Hello, You !
As you can see PRETTYPRINT is FALSE: x^2 - 1
```

Beispiel 14. Ist die Option `KeepOrder` gesetzt, so behalten Summen ihre interne Ordnung in der Ausgabe bei:

```
>> print(b - a): print(KeepOrder, b - a):

b - a

- a + b
```

Beispiel 15. Bei normalen Ergebnis-Ausgaben auf interaktiver Ebene wird eine Alias-Rücksubstitution vorgenommen (siehe `Pref::alias`), bei durch `print` erzeugten Ausgaben aber nicht:

```
>> delete a, b: alias(a = b):
a; print(a):
unalias(a):
```

a

b

Im Gegensatz zur üblichen Ergebnis-Ausgabe reagiert `print` nicht auf `Pref::output`:

```
>> old := Pref::output(generate::TeX):
    sin(a)^b; print(sin(a)^b):
    Pref::output(old):

    "\\sin\\left(a\\right)^b"

    b
    sin(a)
```

Dasselbe gilt für `Pref::postOutput`:

```
>> old := Pref::postOutput("postOutput was called"):
    a*b; print(a*b):
    Pref::postOutput(old):

    a b
    postOutput was called

    a b

>> delete old:
```

Beispiel 16. Die Ausgabe von Summanden einer Summe hängt von der Form dieser Summanden ab. Wenn der Summand ein `_mult`-Ausdruck ist, wird nur der erste und letzte Operand dieses Produkts für die Ausgabe untersucht. Wenn einer dieser Terme eine negative Zahl ist, dann wird das "+"-Symbol in der Summe durch ein "-"-Symbol ersetzt:

```
>> print(hold(a + b*c*(-2)),
        hold(a + b*(-2)*c),
        hold(a + (-2)*b*c)):

    a - 2 b c, a + b (-2) c, a - 2 b c
```

Dies muss berücksichtigt werden, wenn man "print"-Methoden für Polynom-Datentypen schreibt.

Beispiel 17. Üblicherweise gibt MuPAD kein Multiplikations-Symbol in Produkten aus und trennt die Faktoren nur durch Leerzeichen. Über `Pref::timesDot` kann man ein Multiplikations-Symbol setzen:

```

>> a*b*c;
    print(a*b*c):

          a b c

          a b c

>> old := Pref::timesDot(" * "):
    a*b*c;
    print(a*b*c):
    Pref::timesDot(old):

          a * b * c

          a * b * c

>> delete old:

```

Beispiel 18. Das Spaltentrennzeichen bei der Ausgabe von Matrizen oder zweidimensionalen Feldern kann mit `Pref::matrixSeparator` verändert werden:

```

>> a := array(1..2, 1..2, [[11, 12], [22, 23]]):
    a; print(a):

```

```

+-      +-
|  11, 12  |
|          |
|  22, 23  |
+-      +-

+-      +-
|  11, 12  |
|          |
|  22, 23  |
+-      +-

```

```

>> old := Pref::matrixSeparator("  "):
    a; print(a):
    Pref::matrixSeparator(old): delete a:

```

```

+-      +-
|  11  12  |
|          |
|  22  23  |
+-      +-

```

```

+-      +-
|   11  12  |
|           |
|   22  23  |
+-      +-

```

Wenn die Ausgabebreite einer Matrix den Wert von `TEXTWIDTH` überschreiten würde, so wird sie in einer Textnotation ausgegeben:

```

>> print(array(1..4, 1..4, (2, 2) = 2)):
    print(array(1..10, 1..10, (5, 5) = 55)):

```

```

+-      +-
|  ?[1, 1], ?[1, 2], ?[1, 3], ?[1, 4]  |
|                                     |
|  ?[2, 1],      2,      ?[2, 3], ?[2, 4]  |
|                                     |
|  ?[3, 1], ?[3, 2], ?[3, 3], ?[3, 4]  |
|                                     |
|  ?[4, 1], ?[4, 2], ?[4, 3], ?[4, 4]  |
+-      +-

    array(1..10, 1..10,
          (5, 5) = 55
    )

```

```

>> delete old, a:

```

Beispiel 19. Gleitpunktzahlen werden üblicherweise in Festpunkt-Notation ausgegeben. Mittels `Pref::floatFormat` kann man die Darstellung durch Mantisse und Exponent einstellen:

```

>> print(0.000001, 1000.0): old := Pref::floatFormat("e"):
    print(0.000001, 1000.0): Pref::floatFormat(old):

    0.000001, 1000.0

    10.0e-7, 1.0e3

```

In der Standard-Ausgabe von Gleitpunktzahlen werden Nullen am Ende abgeschnitten. Dieses Verhalten läßt sich mit `Pref::trailingZeroes` ändern:

```

>> print(0.000001, 1000.0): old := Pref::trailingZeroes(TRUE):
    print(0.000001, 1000.0): Pref::trailingZeroes(old):

    0.000001, 1000.0

    0.0000010000000000, 1000.000000

```

Die Anzahl der Stellen von Gleitpunktzahlen in der Ausgabe hängt von der Umgebungsvariable DIGITS ab:

```
>> print(float(PI)):
DIGITS := 20: print(float(PI)):
DIGITS := 30: print(float(PI)):

3.141592654

3.1415926535897932385

3.14159265358979323846264338328

>> delete old, DIGITS:
```

Beispiel 20. Die Ausgabeordnung von Mengen unterscheidet sich von ihrer internen Ordnung, die man mit `op` erhalten kann:

```
>> s := {a, b, c}:
s;
print(s):
op(s)

{a, b, c}

{a, b, c}

c, b, a
```

Der Index-Operator `[]` kann dazu verwendet werden, um auf die Elemente einer Menge in der Ausgabereihenfolge zuzugreifen:

```
>> s[1], s[2], s[3]

a, b, c

>> delete s:
```

Beispiel 21. Die Ausgabe eines Domains wird durch seinen "Name"-Eintrag bestimmt, sofern dieser existiert, ansonsten durch seinen *key*:

```
>> T := newDomain("T"):
T;
print(T):
```

```

T
T

>> T::Name := "domain T":
T;
print(T):

domain T

domain T

>> delete T:

```

Beispiel 22. Manchmal ist es wünschenswert, Zeichenketten mit „pretty“-Ausdrücken zu kombinieren. Dies geht nicht mit `expr2text`. Andererseits ist eine durch Kommata getrennte Ausgabe hässlich. Zu diesem Zweck kann die folgende Dummy-Ausdrucksfolge benutzt werden. Sie verwendet MuPADs interne Funktion `builtin(1100,...)` zur Ausgabe von Standard-Operatoren, und zwar mit der Priorität 2 (gleich der Priorität von `_exprseq`) und einem leeren Operator-Symbol "":

```

>> myexprseq := funcenv(myexprseq,
                        builtin(1100, 2, "", "myexprseq")):
print(Unquoted,
      myexprseq("String and pretty expression ", a^b, ".")):

b
String and pretty expression a .

>> delete myexprseq:

```

Hintergründe:

- ☞ Die Ausgabeordnung von Mengen unterscheidet sich von ihrer internen Ordnung, die man mit `op` erhalten kann. Für die Umsortierung der Ausgabe ruft der Kern die Methode `DOM_SET::sort` auf, die die Menge als Argument bekommt und eine geordnete Liste zurückliefert. Die Elemente der Menge werden dann entsprechend der Ordnung dieser Liste ausgegeben.

Änderungen:

- ☞ `KeepOrder` funktioniert nun auch dann, wenn `PRETTYPRINT` auf `FALSE` gesetzt ist.

- ⌘ Die Ausgabe ist anders formatiert, wenn PRETTYPRINT auf FALSE gesetzt ist.
 - ⌘ Die Elemente einer Menge werden vor der Ausgabe sortiert.
 - ⌘ Domains und Prozeduren werden in einer Kurzform ausgegeben; mit `expose` bekommt man die volle Implementation.
-

proc – Definieren einer Prozedur

`proc` - `end_proc` definiert eine Prozedur.

Aufruf(e):

```
⌘ (x1, x2, ...) -> body
⌘ proc(
    x1 <= default1> <: type1>,
    x2 <= default2> <: type2>, ...
)<: returntype>
<name pname;>
<option option1, option2, ...;>
<local local1, local2, ...;>
<save global1, global2, ...;>
begin
    body
end_proc
⌘ _procdef(...)
```

Parameter:

<code>x1, x2, ...</code>	— die formalen Parameter der Funktion: Bezeichner
<code>default1, default2, ...</code>	— Standardwerte für die Parameter: beliebige MuPAD-Objekte
<code>type1, type2, ...</code>	— zulässige Typen für die Parameter: Typ-Objekte, die von der Funktion <code>testtype</code> akzeptiert werden.
<code>returntype</code>	— zulässiger Typ für den Rückgabewert: ein Typ-Objekt, das von der Funktion <code>testtype</code> akzeptiert wird.
<code>pname</code>	— der Prozedurname: ein Ausdruck
<code>option1, option2, ...</code>	— die zur Verfügung stehenden Optionen sind: <i>escape, hold, noDebug, remember</i>
<code>local1, local2, ...</code>	— die lokalen Variablen: Bezeichner
<code>global1, global2, ...</code>	— globale Variablen: Bezeichner
<code>body</code>	— der Prozedurrumpf: eine beliebige Folge von Anweisungen

Rückgabewert: eine Prozedur vom Typ `DOM_PROC`.

Verwandte Funktionen: `args, context, debug, expose, hold, MAXDEPTH, newDomain, Pref::ignoreNoDebug, Pref::noProcRemTab, Pref::typeCheck, Pref::warnDeadProcEnv, return, testargs, Type`

Details:

- ☞ Prozeduren `f := proc(x1, x2, ...) ... end_proc` können wie jede andere Systemfunktion in der Form `f(x1, x2, ...)` aufgerufen werden. Der Rückgabewert dieses Aufrufs ist der Wert des zuletzt im Prozedurrumpf evaluierten Kommandos (bzw., der mittels der Funktion `return` vom Rumpf zurückgelieferte Wert).
- ☞ Die Prozedurdeklaration `(x1, x2, ...) -> body` ist äquivalent zu `proc(x1, x2, ...) begin body end_proc`. Sie ist nützlich zur Definition *einfacher* Prozeduren, in denen keine lokalen Variablen benötigt werden. Beispielsweise definiert `f := x -> x^2` die mathematische Abbildung $f: x \mapsto x^2$. Bei Funktionen mit mehreren Argumenten sind Klammern zu benutzen wie z. B. in `f := (x, y) -> x^2 + y^2`. Siehe Beispiel ??.
- ☞ Eine Prozedur kann beliebig viele Parameter haben. Für jeden Parameter kann ein Standardwert definiert werden, der beim Aufruf der Funktion verwendet wird, wenn kein aktueller Wert für diesen Parameter übergeben wird. Beispielsweise definiert


```
f := proc(x = 42) begin body end_proc
```

den Standardwert 42 für x . Der Aufruf $f()$ wird damit äquivalent zu $f(42)$. Siehe Beispiel ??.

- ☞ Zu jedem Parameter kann ein Typ spezifiziert werden. Hierdurch wird bei späteren Prozeduraufrufen eine automatische Typprüfung des Parameters durchgeführt. Beispielsweise ist in

```
f := proc(x : DOM_INT) begin body end_proc
```

der Parameter x auf ganze Zahlen eingeschränkt. Wird beim Aufruf der Funktion ein Argument mit falschem Datentyp übergeben, so wird die Evaluierung mit einer entsprechenden Fehlermeldung abgebrochen. Siehe Beispiel ??.

Jede Prozedur sollte die aktuell übergebenen Eingabeparameter überprüfen. Siehe auch `testargs`.

Auch eine automatische Typprüfung des Rückgabewertes kann implementiert werden, indem `returntype` angegeben wird. Siehe Beispiel ??.

- ☞ Mit dem Schlüsselwort `name` kann einer Prozedur ein Name zugeordnet werden, z. B.,

```
f := proc(...) name myName; begin body end_proc.
```

Eine spezielle Variable `procname` ist Prozeduren zugeordnet, welchen den Prozedurnamen speichert. Wenn der Prozedurrumpf mittels `procname(args())` einen symbolischen Aufruf der Prozedur zurückliefert, so wird der aktuelle Name benutzt. Dies ist der Name, der durch den `name`-Eintrag festgelegt wurde. Wurde dieser nicht festgelegt, so wird der Bezeichner als Name benutzt, dem die Prozedur als Wert zugewiesen wurde (d. h., in diesem Fall f). Siehe Beispiel ??.

- ☞ Mit dem Schlüsselwort `option` können spezielle Eigenschaften für eine Prozedur spezifiziert werden:

escape muss benutzt werden, wenn die Prozedur eine Prozedur als Rückgabewert liefert, welche auf Werte der erzeugenden Prozedur zugreift. Siehe Beispiel ??.

Diese Option sollte nur im Bedarfsfall verwendet werden. Siehe auch `Pref::warnDeadProcEnv`.

hold verhindert, dass die aktuell übergebenen Parameter evaluiert werden. Siehe Beispiel ??.

noDebug verhindert, dass der Source-Code-Debugger MuPADs in diese Prozedur einsteigt. Siehe auch `Pref::ignoreNoDebug`. Siehe Beispiel ??.

remember instruiert die Prozedur, jedes berechnete Ergebnis in einer so genannten „Remember-Tafel“ zu speichern. Wird die Prozedur später mit den gleichen Eingabewerten erneut aufgerufen, so wird das Ergebnis nicht neu berechnet, sondern dieser Tabelle entnommen. Hierdurch können zum Beispiel rekursive Prozeduren drastisch beschleunigt werden. Siehe Beispiel ??.

Aber Vorsicht: die

„Remember-Tafel“ kann schnell sehr groß werden und gegebenenfalls viel Speicherplatz verbrauchen. Siehe auch `Pref::noProcRemTab`.

- ☞ Mit dem Schlüsselwort `local` können lokale Variablen spezifiziert werden, z. B.,

```
f := proc(...) local x, y; begin body end_proc.
```

Siehe Beispiel ??.

Lokale Variablen können nicht als „symbolische Variablen“ (Bezeichner) eingesetzt werden. Man muss ihnen Werte zuweisen, bevor mit ihnen gerechnet werden kann.

Beachten Sie, dass die Namen von globalen MuPAD-Variablen wie etwa `DIGITS`, `READPATH` etc. nicht als lokale Variablen verwendet werden sollten. Siehe hierzu auch das Schlüsselwort `save`.

- ☞ Mit dem Schlüsselwort `save` erhalten globale MuPAD-Variablen in der Prozedur einen lokalen Kontext, z. B.,

```
f := proc(...) save DIGITS; begin DIGITS := newValue; ... end_proc.
```

Damit wird der Wert der Variablen, den sie beim Einstieg in diese Prozedur hatten, beim Verlassen der Prozedur wieder restauriert. Dies geschieht auch dann, wenn die Funktion wegen eines Fehlers abgebrochen wird. Siehe Beispiel ??.

- ☞ Man kann Prozeduren definieren, die eine variable Anzahl von Argumenten akzeptieren. Beispielsweise kann man eine solche Prozedur ohne jegliche formalen Parameter definieren. Auf die beim Aufruf tatsächlich übergebenen Parameter kann im Quell-Code des Rumpfs mittels der Funktion `args` zugegriffen werden. Siehe Beispiel ??.

- ☞ Der Aufruf eines Prozedurnamens `f` schreibt üblicherweise nicht den gesamten Rumpf der Prozedur auf den Bildschirm. Mit `expose(f)` kann der vollständige Quell-Code eingesehen werden. Siehe Beispiel ??.

- ☞ Die Umgebungsvariable `MAXDEPTH` begrenzt die Tiefe rekursiver Prozeduraufrufe. Der voreingestellte Wert ist `MAXDEPTH = 500`. Siehe Beispiel ??.

- ☞ Ist eine Prozedur ein Slot eines Domains, so enthält die spezielle Variable `dom` den Namen des Domains, zu dem der Slot gehört. Ist die Prozedur kein Domain-Slot, ist der Wert von `dom` gleich `NIL`.

- ☞ Anstelle des Schlüsselworts `end_proc` kann auch das Schlüsselwort `end` verwendet werden.

- ☞ Die imperative Deklaration `proc - end_proc` führt intern zu einem Aufruf der Kernfunktion `_procdef`. Für den Anwender besteht keinerlei Bedarf, `_procdef` direkt aufzurufen.

- ☞ `_procdef` ist eine Funktion des Systemkerns.

Beispiel 1. Einfache Prozeduren können mit dem „Pfeiloperator“ \rightarrow erzeugt werden:

```
>> f := x -> x^2 + 2*x + 1:
      f(x), f(y), f(a + b), f(1.5)

      2          2          2
      2 x + x  + 1, 2 y + y  + 1, 2 a + 2 b + (a + b)  + 1, 6.25

>> f := n -> isprime(n) and isprime(n + 2):
      f(i) $ i = 11..18

      TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE
```

Die folgende Anweisung wendet eine „anonyme“ Prozedur auf die Elemente einer Liste an:

```
>> map([1, 2, 3, 4, 5, 6], x -> x^2)

      [1, 4, 9, 16, 25, 36]

>> delete f:
```

Beispiel 2. Die Deklaration von Standardwerten wird demonstriert. Die folgende Prozedur verwendet die Standardwerte, wenn ein Prozeduraufruf nicht alle Parameter festlegt:

```
>> f := proc(x, y = 1, z = 2) begin [x, y, z] end_proc:
      f(x, y, z), f(x, y), f(x)

      [x, y, z], [x, y, 2], [x, 1, 2]
```

Für das erste Argument war kein Standardwert vereinbart worden. Eine Warnung wird ausgelöst, wenn dieses Argument fehlt:

```
>> f()

Warning: Uninitialized variable 'x' used;
during evaluation of 'f'

      [NIL, 1, 2]

>> delete f:
```

Beispiel 3. Die automatische Typprüfung von Prozedurargumenten und Rückgabewerten wird demonstriert. Die folgende Prozedur lässt nur positive ganze Zahlen als Argument zu:

```
>> f := proc(n : Type::PosInt) begin n! end_proc:
```

Ein Fehler wird ausgelöst, wenn ein nicht zulässiger Parameter übergeben wird:

```
>> f(-1)
```

```
Error: Wrong type of 1. argument (type 'Type::PosInt' expected,  
      got argument '-1');  
during evaluation of 'f'
```

In der folgenden Prozedur wird die automatische Typprüfung des Rückgabewerts aktiviert:

```
>> f := proc(n : Type::PosInt) : Type::Integer  
begin  
    n/2  
end_proc:
```

Ein Laufzeitfehler wird ausgelöst, wenn der Rückgabewert keine ganze Zahl ist:

```
>> f(3)
```

```
Error: Wrong type of return value (type 'Type::Integer' expected,  
      value is '3/2');  
during evaluation of 'f'
```

```
>> delete f:
```

Beispiel 4. Der Namenseintrag einer Prozedur wird demonstriert. Eine Prozedur kann einen symbolischen Aufruf von sich selbst zurückliefern, indem sie die Variable `procname` verwendet, die den aktuellen Prozedurnamen enthält:

```
>> f := proc(x)  
begin  
    if testtype(x, Type::Numeric)  
    then return(float(1/x))  
    else return(procname(args()))  
    end_if  
end_proc:  
f(x), f(x + 1), f(3), f(2*I)
```

```
f(x), f(x + 1), 0.3333333333, - 0.5 I
```

Auch Fehlermeldungen benutzen diesen Namen:

```
>> f(0)
```

```
Error: Division by zero;  
during evaluation of 'f'
```

Wenn die Prozedur einen Namenseintrag hat, so wird dieser benutzt:

```
>> f := proc(x)  
  name myName;  
  begin  
    if testtype(x, Type::Numeric)  
      then return(float(1/x))  
      else return(procname(args()))  
    end_if  
  end_proc:  
  f(x), f(x + 1), f(3), f(2*I)  
  
  myName(x), myName(x + 1), 0.3333333333, - 0.5 I
```

```
>> f(0)
```

```
Error: Division by zero;  
during evaluation of 'myName'
```

```
>> delete f:
```

Beispiel 5. Die Option *escape* wird demonstriert. Diese Option muss benutzt werden, wenn die Prozedur eine andere Prozedur zurückliefert, welche auf die formalen Parameter oder lokale Variablen der erzeugenden Prozedur verweist:

```
>> f := proc(n)  
  begin  
    proc(x) begin x^n end_proc  
  end_proc:
```

Ohne die Option *escape* verläßt der formale Parameter *n* der Prozedur *f* seinen Gültigkeitsbereich: *g := f(3)* verweist intern auf *n*. Wenn *g* aufgerufen wird, kann *g* diesen Parameter nicht mehr zum Wert 3 evaluieren, den *n* innerhalb von *f* hatte:

```
>> g := f(3): g(x)
```

```
Warning: Uninitialized variable 'unknown' used;
during evaluation of 'g'
Error: Illegal operand [_power];
during evaluation of 'g'
```

Die Option *escape* weist *f* an, auf Variablen zu achten, die den aktuellen Gültigkeitsbereich verlassen. Nun verweist die Prozedur *g := f(3)* auf den Wert 3 statt auf den formalen Parameter *n* der Prozedur *f* und kann damit korrekt ausgeführt werden:

```
>> f := proc(n)
    option escape;
    begin
        proc(x) begin x^n end_proc
    end_proc:
    g := f(3): g(x), g(y), g(10)

                                3    3
                                x , y , 1000

>> delete f, g;
```

Beispiel 6. Die Option *hold* wird demonstriert. Mit *hold* sieht die Prozedur den Parameter in der übergebenen Form. Ohne *hold* sieht die Prozedur nur den Wert des Parameters:

```
>> f := proc(x) option hold; begin x end_proc:
    g := proc(x) begin x end_proc:
    x := PI/2:
    f(sin(x) + 2) = g(sin(x) + 2), f(1/2 + 1/3) = g(1/2 + 1/3)

                                sin(x) + 2 = 3, 1/2 + 1/3 = 5/6
```

Prozeduren mit der Option *hold* können die Argumente mittels *context* evaluieren:

```
>> f := proc(x) option hold; begin x = context(x) end_proc:
    f(sin(x) + 2), f(1/2 + 1/3)

                                sin(x) + 2 = 3, 1/2 + 1/3 = 5/6

>> delete f, g, x;
```

Beispiel 7. Die Option *noDebug* wird demonstriert. Die *debug*-Anweisung startet den Debugger, der in den Aufruf der Prozedur *f* springt. Nach der Eingabe des Debugger-Kommandos *c* (continue) setzt der Debugger die Ausführung des Prozeduraufrufs fort:

```
>> f := proc(x) begin x end_proc: debug(f(42))
```

```
Activating debugger...
```

```
#0 in f($1=42) at /tmp/debug0.556:4
```

```
mdx> c
```

```
Execution completed.
```

42

Mit der *noDebug*-Option springt der Debugger nicht in die Prozedur hinein:

```
>> f := proc(x) option noDebug; begin x end_proc: debug(f(42))
```

```
Execution completed.
```

42

```
>> delete f:
```

Beispiel 8. Die Option *remember* wird demonstriert. Die *print*-Anweisung in der folgenden Prozedur zeigt an, wenn der Prozedurrumpf durchlaufen wird:

```
>> f:= proc(n : Type::PosInt)
  option remember;
  begin
    print("computing ".expr2text(n)."!");
    n!
  end_proc:
  f(5), f(10)
```

```
"computing 5!"
```

```
"computing 10!"
```

```
120, 3628800
```

Bei erneutem Aufruf werden schon berechnete Werte aus der „Remember-Tafel“ entnommen, ohne dass der Prozedurrumpf erneut durchlaufen wird:

```
>> f(5)*f(10) + f(15)
```

```
"computing 15!"
```

```
1308109824000
```

`option remember` wird in der folgenden Prozedur zur Beschleunigung eingesetzt. Sie berechnet die Fibonacci-Zahlen rekursiv:

```
>> f := proc(n : Type::NonNegInt)
  option remember;
  begin
    if n = 0 or n = 1 then return(n) end_if;
    f(n - 1) + f(n - 2)
  end_proc;

>> f(123)
```

```
22698374052006863956975682
```

Aufgrund der rekursiven Natur sind die Argumente von `f` durch die maximal zulässige Rekursionstiefe eingeschränkt (siehe `MAXDEPTH`):

```
>> f(1000)

Error: Recursive definition [See ?MAXDEPTH];
during evaluation of 'Type::testtype'
```

Ohne `option remember` ist die rekursive Berechnung recht langsam:

```
>> f := proc(n : Type::NonNegInt)
  begin
    if n = 0 or n = 1 then return(n) end_if;
    f(n - 1) + f(n - 2)
  end_proc;

>> f(28)
```

```
317811
```

```
>> delete f;
```

Beispiel 9. Die Benutzung von lokalen Variablen wird demonstriert:

```
>> f := proc(a)
  local x, y;
  begin
    x := a^2;
    y := a^3;
    print("x, y" = (x, y));
    x + y
  end_proc;
```


Die lokalen Variablen `x` und `y` stimmen nicht mit den globalen Variablen `x` und `y` überein. Der Aufruf von `f` ändert nicht die globalen Werte:

```
>> x := 0: y := 0: f(123), x, y

"x, y" = (15129, 1860867)

1875996, 0, 0

>> delete f, x, y:
```

Beispiel 10. Die `save`-Deklaration wird demonstriert. Die folgende Prozedur ändert die Umgebungsvariable `DIGITS` intern ab. Wegen `save DIGITS` wird der ursprüngliche Wert von `DIGITS` bei Rückkehr aus der Prozedur wiederhergestellt:

```
>> myfloat := proc(x, digits)
  save DIGITS;
  begin
    DIGITS := digits;
    float(x);
  end_proc:
```

Der momentane Wert von `DIGITS` ist:

```
>> DIGITS

10
```

Mit der Voreinstellung `DIGITS = 10` leidet die folgende Gleitpunktkonvertierung an numerischer Auslöschung. Wegen der intern erhöhten Genauigkeit liefert `myfloat` ein genaueres Ergebnis:

```
>> x := 10^20*(PI - 21053343141/6701487259):
  float(x), myfloat(x, 20)

-32.0, 0.02616403997
```

Der Wert von `DIGITS` wurde durch den Aufruf von `myfloat` nicht verändert:

```
>> DIGITS

10
```

Die folgende Prozedur benötigt einen globalen Bezeichner, da lokale Variablen nicht als Integrationsvariablen in der Funktion `int` zulässig sind. Intern wird der globale Bezeichner `x` gelöscht, um sicherzustellen, dass er keinen Wert trägt:

```
>> f := proc(n)
  save x;
  begin
    delete x;
    int(x^n*exp(-x), x = 0..1)
  end_proc:

>> x := 3: f(1), f(2), f(3)

      1 - 2 exp(-1), 2 - 5 exp(-1), 6 - 16 exp(-1)
```

Wegen `save x` wurde der ursprüngliche Wert von `x` nach der Integration wiederhergestellt:

```
>> x

      3

>> delete myfloat, x, f:
```

Beispiel 11. Die folgende Prozedur akzeptiert eine beliebige Anzahl von Argumenten. Sie greift mittels `args` auf die beim Aufruf übergebenen Argumente zu, steckt sie in eine lokale Liste, ordnet die Elemente mittels `revert` um und liefert die Argumente in umgekehrter Reihenfolge zurück:

```
>> f := proc()
  local arguments;
  begin
    arguments := [args()];
    op(revert(arguments))
  end_proc:

>> f(a, b, c)

      c, b, a

>> f(1, 2, 3, 4, 5, 6, 7)

      7, 6, 5, 4, 3, 2, 1

>> delete f:
```

Beispiel 12. Mit `expose` erhält man Einblick in den Quell-Code einer Prozedur:

```
>> f := proc(x = 0, n : DOM_INT)
    begin
        sourceCode;
    end_proc

        proc f(x, n) ... end

>> expose(f)

proc(x = 0, n : DOM_INT)
    name f;
begin
    sourceCode
end_proc

>> delete f;
```

Änderungen:

- ☞ Siehe hierzu auch das Kapitel Die neue Sprache in MuPAD 2.0 des beiliegenden Dokuments Von MuPAD 1.4 zu MuPAD 2.0.
- ☞ In früheren Versionen durch `fun` und `func` definierte Funktionen vom Typ `DOM_EXEC` existieren nicht mehr.
- ☞ Man kann nicht mehr „von außen“ auf die lokalen Variablen einer Prozedur zugreifen: die neue MuPAD-Version benutzt „lexikalisches Scoping“ statt des „dynamischen Scopings“ früherer Versionen.
- ☞ Lokale Variablen einer Prozedur können nicht mehr als symbolische Variablen benutzt werden. Lokalen Variablen müssen Werte zugewiesen werden, bevor sie für Rechnungen benutzt werden können.
- ☞ In früheren Versionen konnten globale Umgebungsvariablen wie z. B. `DIGITS` als lokale Variablen deklariert werden, wenn die Prozedur sie lokal abänderte. Dies funktioniert in der neuen Version nicht mehr. Stattdessen wurde das neue Schlüsselwort `save` eingeführt.
- ☞ Die neue Option `escape` wurde eingeführt.
- ☞ `end` kann neben `end_proc` zum Abschluß einer Prozedurdefinition verwendet werden.

product – **Bestimmte und unbestimmte symbolische Produkte**

`product(f, i)` berechnet das unbestimmte Produkt von $f(i)$ bzgl. i , d. h. eine geschlossene Form g mit $g(i+1)/g(i) = f(i)$.

`product(f, i = a..b)` versucht, eine geschlossene Darstellung des Produktes $\prod_{i=a}^b f(i)$ zu finden.

Aufruf(e):

⇒ `product(f, i)`
⇒ `product(f, i = a..b)`

Parameter:

f — ein arithmetischer Ausdruck in i
 i — der Produktindex: ein Bezeichner
 a, b — die Grenzen: arithmetische Ausdrücke

Rückgabewert: ein arithmetischer Ausdruck

Verwandte Funktionen: `_mult`, `*`, `sum`

Details:

⇒ `product` dient zur Vereinfachung *symbolischer* Produkte. `product` sollte nicht benutzt werden, um eine endliche Anzahl von Termen zu multiplizieren: wenn a und b ganze Zahlen vom Typ `DOM_INT` sind, ist der Aufruf `_mult(f $ i = a..b)` wesentlich schneller.

⇒ `product(f, i)` berechnet das unbestimmte Produkt von f bezüglich i . Das ergibt einen Ausdruck g mit $f(i) = g(i+1)/g(i)$.

⇒ `product(f, i = a..b)` berechnet das bestimmte Produkt, wenn i von a bis b läuft.

Wenn $b-a$ eine nichtnegative ganze Zahl ist, wird das explizite Produkt $f(a) \cdot f(a+1) \cdots f(b)$ zurückgegeben.

Wenn $b-a$ eine negative ganze Zahl ist, wird das Reziproke des Ergebnisses von `product(f, i = b+1..a-1)` zurückgegeben. Ist dieses Ergebnis Null, so wird eine Fehlermeldung ausgegeben. Mit dieser Festlegung gilt

`product(f, i = a..b) * product(f, i = b+1..c) = product(f, i = a..c)`

für alle a, b und c .

⇒ `product` gibt einen symbolischen `product`-Aufruf zurück, wenn keine geschlossene Darstellung des Produktes berechnet werden kann.

Beispiel 1. Jeder der folgenden beiden Aufrufe berechnet das Produkt $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$:

```
>> product(i, i = 1..5) = _mult(i $ i = 1..5)
120 = 120
```

Normalerweise ist `_mult` jedoch schneller, wenn die Grenzen ganze Zahlen vom Typ `DOM_INT` sind.

Man kann auch eine geschlossene Form für dieses bestimmte Produkt von 1 bis n bestimmen:

```
>> product(i, i = 1..n)
gamma(n + 1)
```

Da die obere Grenze ein symbolischer Bezeichner ist, kann `_mult` dieses Produkt nicht berechnen:

```
>> _mult(i $ i = 1..n)
Error: Illegal argument [_seggen]
```

Das entsprechende unbestimmte Produkt ist:

```
>> product(i, i);
gamma(i)
```

Das unbestimmte und das bestimmte Produkt von $2i + 1$ sind:

```
>> product(2*i + 1, i)
i
2 gamma(i + 1/2)

>> product(2*i + 1, i = 1..n)
n + 1
2 gamma(n + 3/2)
-----
1/2
PI
```

Die Grenzen können auch symbolische Ausdrücke oder $\pm\infty$ sein:

```
>> product(2*i/(i + 2), i = a..b)
b + 1
gamma(a + 2) gamma(b + 1) 2
-----
a
gamma(a) gamma(b + 3) 2
```

```
>> product(i^2/(i^2 + 2*i + 1), i = 2..infinity)
```

4

Für die folgenden beiden Produkte kann keine geschlossene Form gefunden werden, und es werden die symbolischen product-Aufrufe zurückgegeben:

```
>> delete f: product(f(i), i)
```

```
product(f(i), i)
```

```
>> product((1 + 2^(-i)), i = 1..infinity)
```

```
product | / 1 \
        | -- + 1, i = 1..infinity |
        | i                        |
        | \ 2                      |
        | /
```

Änderungen:

⌘ Keine Änderungen.

protect – Schreibschutz für Bezeichner

protect(x) schützt den Bezeichner x.

Aufruf(e):

⌘ protect(x <, protectionlevel>)

Parameter:

x — ein Bezeichner

Optionen:

protectionlevel — entweder *Error* oder *Warning* oder *None*.
Der Standardwert ist *Warning*.

Rückgabewert: die zuletzt gültige Schutzstufe für x: entweder *Error* oder *Warning* oder *None*.

Verwandte Funktionen: unprotect

Details:

- ⌘ `protect(x, Error)` setzt absoluten Schreibschutz für den Bezeichner. Nachfolgende Zuweisungen an den Bezeichner lösen einen Fehler aus.
- ⌘ `protect(x, Warning)` setzt einen „Warnschutz“ für den Bezeichner: nachfolgende Zuweisungen an `x` führen zu einer Warnung, die Zuweisungen werden jedoch trotzdem ausgeführt.
`protect(x)` ist äquivalent zu `protect(x, Warning)`.
- ⌘ `protect(x, None)` entfernt jeglichen Schutz vom Bezeichner. Dieser Aufruf ist äquivalent zu `unprotect(x)`.
- ⌘ Das Überschreiben vom System geschützter Bezeichner wie beispielsweise der Namen von Systemfunktionen kann die Zerstörung der aktuellen Sitzung zur Folge haben.



Beispiel 1. Der folgende Aufruf schützt den Bezeichner `important` mit der Schutzstufe „*Warning*“:

```
>> protect(important, Warning)
```

```
None
```

Der Bezeichner `important` kann damit noch überschrieben werden:

```
>> important := 1
```

```
Warning: protected variable important overwritten
```

```
1
```

Nun wird der Bezeichner mit der Stufe „*Error*“ geschützt:

```
>> protect(important, Error)
```

```
Warning
```

Damit kann `important` nicht mehr überschrieben werden:

```
>> important := 2
```

```
Error: Identifier 'important' is protected [_assign]
```

Der Bezeichner behält seinen alten Wert:

```
>> important
```

1

Um diesen Wert überschreiben zu können, muss erst der Schutz von `important` entfernt werden:

```
>> protect(important, None)
```

Error

```
>> important := 2
```

2

Der Bezeichner wird mit der Standardstufe „*Warning*“ erneut geschützt:

```
>> protect(important)
```

None

```
>> important := 1
```

Warning: protected variable `important` overwritten

1

```
>> unprotect(important): delete important:
```

Beispiel 2. `protect` wertet sein erstes Argument nicht aus. Hier kann der Bezeichner `x` überschrieben werden, während sein Wert – der Bezeichner `y` – weiterhin schreibgeschützt ist:

```
>> protect(y, Error): x := y: protect(x): x := 1
```

Warning: protected variable `x` overwritten

1

```
>> y := 2
```

Error: Identifier '`y`' is protected [`_assign`]

```
>> unprotect(x): unprotect(y): delete x, y:
```

Hintergründe:

- ☞ `protect` wertet sein erstes Argument nicht aus. Somit können Bezeichner geschützt werden, denen ein Wert zugewiesen wurde.

Änderungen:

☞ Keine Änderungen.

protocol – Protokoll einer MuPAD-Sitzung

`protocol(filename)` beginnt ein Protokoll der aktuellen MuPAD-Sitzung in der Datei namens `filename`.

`protocol(n)` schreibt in die mit dem Dateibezeichner `n` verknüpfte Datei.

`protocol()` beendet das Protokoll.

Aufruf(e):

☞ `protocol(filename <, InputOnly>)`

☞ `protocol(n <, InputOnly>)`

☞ `protocol()`

Parameter:

`filename` — der Dateiname: eine Zeichenkette

`n` — ein von `fopen` erzeugter Dateibezeichner: eine positive ganze Zahl

Optionen:

`InputOnly` — nur Eingaben werden protokolliert

Rückgabewert: das leere Objekt vom Type `DOM_NULL`.

Seiteneffekte: Die Funktion reagiert auf die Umgebungsvariable `WRITEPATH`. Hat diese Variable einen Wert, so wird die Protokolldatei in dem entsprechenden Verzeichnis angelegt, anderenfalls im „aktuellen Arbeitsverzeichnis“.

Verwandte Funktionen: `fclose`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

☞ `protocol` protokolliert alle Ein- und Ausgaben einer MuPAD-Sitzung in einer Textdatei.

☞ Die Datei kann direkt durch ihren Namen spezifiziert werden. Hierdurch wird entweder eine neue Datei geöffnet oder eine existierende Datei dieses Namens wird überschrieben. Dabei öffnet und schließt `protocol` die Datei automatisch.

Wenn `WRITEPATH` keinen Wert hat, interpretiert `protocol` den Dateinamen als Pfadnamen relativ zum „aktuellen Arbeitsverzeichnis“.

Man beachte, dass die Bedeutung des „aktuellen Arbeitsverzeichnisses“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.

Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann. Weiterhin warnt MacMuPAD den Benutzer vor dem Überschreiben existierender Dateien.

Auch absolute Pfadnamen werden von `protocol` verarbeitet.

- ☞ Alternativ kann die Datei durch einen ganzzahligen Bezeichner `n` spezifiziert werden, wobei die Textdatei vorher mittels `fopen(Text, filename, Write)` oder `fopen(Text, filename, Append)` im Schreibmodus geöffnet worden sein muss. Hierbei wird der Dateibezeichner als eine ganze Zahl `n` zurückgeliefert. Man beachte, dass `fopen(filename)` die Datei nur im Lesemodus öffnet, d. h., ohne Schreibberechtigung. Ein folgender `protocol`-Befehl auf diese Datei liefert einen Fehler.

Die Datei wird nicht automatisch von `protocol()` geschlossen und muss durch einen folgenden Aufruf von `fclose` ‚per Hand‘ geschlossen werden.

- ☞ Der Aufruf von `protocol` ohne Argumente beendet ein laufendes Protokoll. Das Schließen einer Protokolldatei mit `fclose` beendet ebenfalls die Protokollierung.
- ☞ Wird während der Durchführung eines Protokolls ein neues Protokoll begonnen, so wird das alte automatisch beendet und die zugehörige Datei wird geschlossen.

Option `<InputOnly>`:

- ☞ Die Protokolldatei enthält nur die Eingabezeilen, alle Ausgaben werden weggelassen.

Beispiel 1. Eine Textdatei „test“ wird durch `fopen` zum Schreiben geöffnet:

```
>> n := fopen(Text, "test", Write):
```

`protocol` beginnt ein Sitzungsprotokoll mit Ein- und Ausgaben in dieser Datei:

```
>> protocol(n):  
1 + 1, a/b;  
solve(x^2 = 2)  
protocol():
```

Diese Datei hat nun den folgenden Inhalt:

```
1 + 1, a/b;

                                a
                                2, -
                                b

solve(x^2 = 2)

                                1/2          1/2
                                { [x = 2    ], [x = - 2    ] }

protocol():
```

Beispiel 2. Hier wird die Protokolldatei von `protocol` direkt geöffnet. Nur die Eingaben werden protokolliert:

```
>> protocol("test", InputOnly):
    1 + 1; a/b;
    solve(x^2 = 2)
    protocol():
```

Diese Datei hat nun den folgenden Inhalt:

```
1 + 1; a/b;
solve(x^2 = 2)
protocol():
```

Änderungen:

☞ Die neue Option *InputOnly* wurde eingeführt.

psi – die Digamma-/Polygammafunktion

`psi(x)` stellt die Digammafunktion dar, d.h., die logarithmische Ableitung `diff(ln(gamma(x)), x)` der gamma-Funktion.

`psi(x, n)` stellt die *n*-te Polygammafunktion dar, d.h., die *n*-te Ableitung `diff(psi(x), x$n)`.

Aufruf(e):

```
☞ psi(x)
☞ psi(x, n)
```

Parameter:

- x — ein arithmetischer Ausdruck
- n — eine nicht-negative ganze Zahl

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Ist x eine Gleitpunktzahl, so reagiert die Funktion auf die Umgebungsvariable DIGITS, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: fact, gamma, zeta

Details:

- ☞ `psi(x, 0)` ist äquivalent zu `psi(x)`.
- ☞ Die Digamma-/Polygammafunktion ist für alle komplexen Argumente außer den singulären Punkten $0, -1, -2, \dots$ definiert.
- ☞ Ist x eine Gleitpunktzahl, so wird eine Gleitpunktzahl zurückgeliefert.

Ist x eine positive ganze Zahl kleiner als 1000 oder ein ungerades ganzzahliges Vielfaches von $1/2$ mit $|x|$ kleiner als 1000, so wird die Rekursionsbeziehung

$$\text{psi}(x+1, n) = \text{psi}(x, n) + (-1)^n * n! / x^{(n+1)}$$

angewendet. Zusammen mit

$$\text{psi}(1) = -\text{EULER},$$

$$\text{psi}(1, n) = (-1)^{(n+1)} * n! * \text{zeta}(n+1), n > 0,$$

$$\text{psi}(1/2) = -2 * \ln(2) - \text{EULER},$$

$$\text{psi}(1/2, n) = (-1)^{(n+1)} * n! * (2^{(n+1)} - 1) * \text{zeta}(n+1), n > 0$$

$$\text{psi}(1/2) = -2 * \ln(2) - \text{EULER},$$

ergeben sich explizite Ausdrücke für den Wert von `psi`.

Die speziellen Werte `psi(infinity) = psi(infinity, 0) = infinity` und `psi(infinity, n) = 0` für $n > 0$ sind implementiert.

Für alle anderen Argumente wird ein symbolischer Aufruf von `psi` zurückgeliefert.

- ☞ Das float-Attribut der Digammafunktion `psi(x)` ist eine Kernfunktion, d. h., die numerische Auswertung ist schnell. Das float-Attribut der Polygammafunktion `psi(x, n)` mit $n > 0$ ist eine Bibliotheksfunktion. Man beachte, dass für ganze Zahlen und ungerade ganzzahlige Vielfache von $1/2$ zur numerischen Auswertung `psi(float(x))` bzw. `psi(float(x), n)`

und nicht `float(psi(x))` bzw. `float(psi(x,n))` benutzt werden sollte: die Berechnung des symbolischen Resultats `psi(x)` bzw. `psi(x,n)` ist zeitaufwendig, weiterhin kann die anschließende float-Evaluierung numerisch instabil sein.

☞ Das `expand`-Attribut schreibt `psi(x,n)` mittels der Beziehung

$$\text{psi}(x+1,n) = \text{psi}(x,n) + (-1)^n * n! / x^{(n+1)}$$

um. Für numerisches `x` wird diese Formel dazu benutzt, das Argument in den Bereich $0 < x < 1$ zu verschieben. Siehe die Beispiele ?? und ??.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> psi(-3/2), psi(4, 1), psi(3/2, 2)

      2
      PI
8/3 - 2 ln(2) - EULER, --- - 49/36, 16 - 14 zeta(3)
      6

>> psi(x + sqrt(2), 4), psi(infinity, 5)

      1/2
psi(x + 2      , 4), 0
```

Für Gleitkommazahlen wird der numerische Wert von `psi` berechnet:

```
>> psi(-5.2), psi(1.0, 3), psi(2.0 + 3.0*I, 10)

6.065773152, 6.493939402, 0.7526409593 - 2.299472238 I
```

Beispiel 2. `psi` ist singulär für nicht-positive ganze Zahlen:

```
>> psi(-2)

Error: singularity [psi]
```

Beispiel 3. Für positive ganze Zahlen und ungerade ganzzahlige Vielfache von $1/2$ wird in mittels `EULER`, `PI`, `ln` bzw. `zeta` ausgedrücktes Resultat geliefert, falls der Absolutbetrag des Argumentes kleiner ist als 1000:

```
>> psi(-5/2), psi(-3/2, 1), psi(4, 3), psi(9/2, 2)
```

$$46/15 - 2 \ln(2) - \text{EULER}, \frac{\pi^2}{2} + 40/9, \frac{\pi^4}{15} - 1393/216,$$

$$19410176/1157625 - 14 \zeta(3)$$

Für größere Argument erhält man solche Ausdrücke über das expand-Attribut:

```
>> psi(1000, 1)
```

```
psi(1000, 1)
```

```
>> expand(%)
```

$$\frac{\pi^2}{6} -$$

```
835458876624295851523752364295.../50820720104325812617835292...
```

Beispiel 4. Die Funktionen diff, float, expand, limit und series verarbeiten symbolische psi-Ausdrücke:

```
>> diff(psi(x^2 + 1, 3), x), float(ln(3 + psi(sqrt(PI))))
```

$$2x \psi(x^2 + 1, 4), 1.183103343$$

```
>> expand(psi(15/4)), expand(psi(x + 3, 2))
```

$$\psi(3/4) + 524/231, \psi(x, 2) + \frac{x^2}{2} + \frac{(x+1)^2}{2} + \frac{(x+2)^2}{2}$$

```
>> limit(x*psi(x), x = 0), limit(psi(x, 3), x = infinity)
```

$$-1, 0$$

```
>> series(psi(x), x = 0), series(psi(x, 3), x = infinity, 3)
```

$$\begin{aligned}
& -\frac{1}{x} - \text{EULER} + \frac{x^2 \text{PI}^2}{6} - x^2 \text{zeta}(3) + \frac{x^3 \text{PI}^4}{90} + O(x^4), \\
& \frac{2}{x^3} + \frac{3}{x^4} + \frac{2}{x^5} + O\left(\frac{1}{x^6}\right)
\end{aligned}$$

Änderungen:

- ☞ Explizite Ausdrücke werden nun für alle ganzen Zahlen und alle ungeraden ganzzahligen Vielfache von $1/2$ für Argumente der Größe $|x| < 1000$ zurückgeliefert. Das `expand`-Attribut schreibt nun auch symbolische Aufrufe mit numerischen Argumenten um. Die speziellen Werte `psi(infinity) = psi(infinity, 0) = 0` und `psi(infinity, n) = 0` für $n > 0$ wurden implementiert.

quit – Beenden einer MuPAD-Sitzung

Auf dem interaktiven Level beendet das Kommando `quit` die laufende MuPAD-Sitzung.

Aufruf(e):

- ☞ quit
- ☞ `_quit()`

Verwandte Funktionen: `break`, `next`, `Pref::callOnExit`, `reset`, `return`

Details:

- ☞ Die `quit`-Anweisung ist äquivalent zum Aufruf `_quit()`.
- ☞ Interaktiv eingegeben beendet `quit` die laufende MuPAD-Sitzung und führt den Anwender zurück zur Betriebssystemebene, von der aus MuPAD gestartet wurde.
- ☞ `quit` sollte nicht innerhalb von Prozeduren verwendet werden. Wird es doch innerhalb einer Prozedur verwendet, so beendet dies nur die

Ausführung dieser Prozedur. Der Rückgabewert der Prozedur ist dabei undefiniert. Zum Verlassen einer Prozedur muss `return` verwendet werden.

- ⌘ Bei Verwendung der MuPAD Pro Notebook-Oberfläche oder anderer graphischer MuPAD-Benutzungsschnittstellen muss statt des `quit`-Befehls die entsprechende Schaltfläche des MuPAD-Hauptfensters zum Beenden der MuPAD-Sitzung verwendet werden. Der Aufruf des `quit`-Befehls führt in diesen Fällen zu einer Fehlermeldung.
- ⌘ Wenn eine MuPAD-Sitzung beendet wird, werden zunächst so genannte „Exit-Handler“ aufgerufen, bevor der MuPAD-Kern abgeschaltet wird. Exit-Handler können über die Funktion `Pref::callOnExit` installiert werden.
- ⌘ `_quit` ist eine Funktion des Systemkerns.

Beispiel 1. In diesem Beispiel wird die Linux/UNIX-Terminalversion von MuPAD gestartet und sofort mit dem `quit`-Befehl beendet:

```
myprompt> mupad

      *-----*      MuPAD 2.0.0 -- The Open Computer Algebra System
    / |      / |
  *-----* |      Copyright (c) 1997 - 2000 by SciFace Software
  | *--|-*      All rights reserved.
  | /      | /
  *-----*      Universität Paderborn, FB-17, Mathematik

>> quit
myprompt>
```

Beispiel 2. In einer MuPAD-Version mit graphischer Benutzeroberfläche, z. B. unter Windows 9x/NT/2000, dem Apple Macintosh-System oder unter Linux/UNIX mit X11/Motif, führt ein `quit`-Befehl zu folgender Fehlermeldung:

```
>> quit

Warning: Quit the kernel via the user interface [quit]
```


Änderungen:

☞ Keine Änderungen.

radsimp – Vereinfachung von Wurzelausdrücken

radsimp vereinfacht arithmetische Ausdrücke, die Wurzeln (Radikale) enthalten.

Aufruf(e):

☞ `radsimp(z)`

Parameter:

`z` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `combine`, `ifactor`, `normal`, `rectform`, `simplify`

Details:

☞ `radsimp(z)` versucht, in `z` enthaltene Wurzelausdrücke zu vereinfachen. Das Ergebnis ist mathematisch äquivalent zu `z`.

☞ Der Aufruf `radsimp(z)` ist äquivalent zu `simplify(z, sqrt)`.

Beispiel 1. Die Vereinfachung einiger konstanter Wurzelausdrücke wird demonstriert:

```
>> ratsimp(2*2^(1/4) + 2^(3/4) - (6*2^(1/2) + 8)^(1/2))
```

0

```
>> ratsimp(
    sqrt(14 + 3*sqrt(3 + 2*sqrt(5 - 12*sqrt(3 - 2*sqrt(2)))))
)
```

$\frac{1}{2}$
2 + 3

```
>> radsimp(3*sqrt(7)/(sqrt(7) - 2))
```

$$\frac{1}{2} \sqrt{7} + 7$$

```
>> radsimp(sqrt(1 + sqrt(3)) + sqrt(3 + 3*sqrt(3))
      - sqrt(10 + 6*sqrt(3)))
```

0

```
>> x := sqrt(3)*I/2 + 1/2: y := x^(1/3) + x^(-1/3): z := y^3 -
3*y
```

$$\frac{\frac{1}{\sqrt{\frac{1}{2} + \frac{1}{2}i\sqrt{3}}} + \left(\frac{1}{2} + \frac{1}{2}i\sqrt{3}\right)^{\frac{1}{3}}}{\left(\frac{1}{2} + \frac{1}{2}i\sqrt{3}\right)^{\frac{1}{3}}} - \frac{3\left(\frac{1}{2} + \frac{1}{2}i\sqrt{3}\right)^{\frac{1}{3}}}{\left(\frac{1}{2} + \frac{1}{2}i\sqrt{3}\right)^{\frac{1}{3}}}$$

```
>> radsimp(z)
```

1

```
>> delete x, y, z:
```

Beispiel 2. radsimp kann auch arithmetische Ausdrücke mit symbolischen Variablen vereinfachen:

```
>> z := x/(sqrt(3) - 1) - x/2
```

$$\frac{x}{\sqrt{3} - 1} - \frac{x}{2}$$

```
>> radsimp(z) = expand(radsimp(z))
```

$$x \frac{\frac{1}{\sqrt{2}} + \frac{1}{2}}{\sqrt{2}} + \frac{1}{2} \frac{x}{\sqrt{2}} = \frac{x \sqrt{2}}{2}$$

```
>> delete z:
```

Hintergründe:

- ☞ Für konstante algebraische Ausdrücke konstruiert `radsimp` einen Turm algebraischer Erweiterungen von \mathbb{Q} (`Dom::AlgebraicExtension`). Es wird versucht, die einfachste Darstellung des Ausdrucks zu finden.
- ☞ Diese Funktion beruht auf einem im folgenden Artikel beschriebenen Algorithmus: Borodin, Fagin, Hopcroft und Tompa, „Decreasing the Nesting Depth of Expressions Involving Square Roots“, JSC 1, 1985, pp. 169-188.

Änderungen:

- ☞ Keine Änderungen.
-

random – Erzeugen von Zufallszahlen

`random()` liefert eine pseudo-zufällige ganze Zahl.

`random(n1..n2)` liefert eine Prozedur, die ganze Pseudo-Zufallszahlen zwischen `n1` und `n2` erzeugt.

Aufruf(e):

- ☞ `random()`
- ☞ `random(n1..n2)`
- ☞ `random(n)`

Parameter:

- `n1, n2` — ganze Zahlen mit $n1 \leq n2$
- `n` — eine positive ganze Zahl

Rückgabewert: `random()` liefert eine nicht-negative ganze Zahl. Die Aufrufe `random(n1..n2)` bzw. `random(n)` liefern Prozeduren vom Typ `DOM_PROC`.

Seiteneffekte: `random` sowie die von dieser Funktion erzeugten Zufallsgeneratoren reagieren auf den Wert der Umgebungsvariablen `SEED` und verändern ihn.

Verwandte Funktionen:

Details:

- ☞ `random()` liefert eine nichtnegative ganze Pseudo-Zufallszahl mit 12 Ziffern, wobei eine Gleichverteilung zugrunde liegt.

- ☞ `r := random(n1..n2)` liefert einen Zufallsgenerator `r`. Jeder folgende Aufruf `r()` erzeugt eine ganze Zahl zwischen `n1` und `n2`.
- ☞ `random(n)` ist äquivalent zu `random(0, n-1)`.
- ☞ Die globale Variable `SEED` wird zur Initialisierung oder Änderung der Folge von Zufallszahlen verwendet. Ihr darf eine beliebige ganze *nicht-verschwindende* Zahl zugewiesen werden. Die Zufallsgeneratoren können hiermit zurückgesetzt werden und liefern zu gegebenem Startwert `SEED` immer dieselbe Zufallsreihe.

`SEED` wird während der Initialisierung von MuPAD auf einen festen Standardwert gesetzt. Damit liefern die Zufallsgeneratoren nach jedem Start einer MuPAD-Sitzung oder nach dem Rücksetzen mit der `reset`-Funktion immer dieselbe Zahlenfolge.
- ☞ Mehrere durch `random` erzeugte Zufallsgeneratoren können simultan benutzt werden. Sie greifen alle auf dieselbe Umgebungsvariable `SEED` zu.

Beispiel 1. Der folgende Aufruf erzeugt eine pseudo-zufällige Folge ganzer Zahlen. Man beachte, dass in der Konstruktion der Folge eine Laufvariable `i` benutzt werden muss. Ein Aufruf wie `random() $ 8` würde 8 Kopien einer einzigen Zahl erzeugen:

```
>> random() $ i = 1..8

427419669081, 321110693270, 343633073697, 474256143563,
558458718976, 746753830538, 32062222085, 722974121768
```

Der folgende Aufruf erzeugt einen „Würfel“, der 20 Mal geworfen wird:

```
>> die := random(1..6): die() $ i = 1..20

2, 2, 2, 4, 4, 3, 3, 2, 1, 4, 4, 6, 1, 1, 1, 2, 4, 2, 1, 3
```

Der folgende Aufruf erzeugt einen „Münze“, die geworfen werden kann und „Kopf“ oder „Zahl“ (engl: „head“ oder „tail“) ergibt:

```
>> coin := random(2): coin() $ i = 1..10

1, 0, 1, 1, 0, 1, 0, 1, 0, 0

>> subs(%, [0 = head, 1 = tail])

tail, head, tail, tail, head, tail, head, tail, head, head
```

Der folgende Aufruf liefert einen Generator für gleichverteilte Gleitpunktzahlen zwischen `-1.0` und `1.0`:

```
>> r := random(-10^DIGITS..10^DIGITS)/float(10^DIGITS):
>> r() $ i = 1..12;
0.1905754559, 0.2075358358, 0.5537108789, 0.1638155425,
0.2610874287, -0.7132768677, -0.7457691643, 0.9053675583,
-0.4759211428, 0.1898567228, 0.6881793744, -0.9192271682
>> delete dice, coin, r:
```

Beispiel 2. `random` ist abhängig von der globalen Variable `SEED`, die bei der (Re-)Initialisierung MuPADs gesetzt wird. Sie kann aber auch vom Benutzer gesetzt werden. Zufallsreihen sind damit reproduzierbar:

```
>> SEED := 1: random() $ i = 1..4
427419669081, 321110693270, 343633073697, 474256143563
>> SEED := 1: random() $ i = 1..4
427419669081, 321110693270, 343633073697, 474256143563
```

Beispiel 3. `random` ermöglicht die Erstellung mehrerer Zufallsgeneratoren für verschiedene Bereiche und deren gleichzeitige Nutzung:

```
>> r1 := random(0..4): r2 := random(2..9): [r1(), r2()] $ i = 1..6
[1, 4], [0, 2], [1, 3], [0, 5], [2, 2], [4, 7]
>> delete r1, r2:
```

Hintergründe:

☞ `random` implementiert einen linearen Kongruenzgenerator: die durch wiederholte Aufrufe von `random()` erzeugte Folge von Pseudozufallszahlen ist $f(x), f(f(x)), \dots$. Hierbei ist x der anfängliche Wert von `SEED`, und f ist die Funktion $x \mapsto ax \bmod m$ mit geeigneten Konstanten a und m .

Änderungen:

⌘ Keine Änderungen.

rationalize – Konvertierung eines Ausdrucks in einen rationalen Ausdruck

`rationalize(object)` transformiert den Ausdruck `object` in einen äquivalenten rationalen Ausdruck, indem nicht-rationale Teilausdrücke durch neu generierte Variablen ersetzt werden.

Aufruf(e):

⌘ `rationalize(object, <, inspect <, stop>>)`

Parameter:

- `object` — ein arithmetischer Ausdruck oder eine Menge oder Liste solcher Ausdrücke
- `inspect` — zu betrachtenden Teilausdrücke: eine Menge von Typen, eine Prozedur oder `NIL`. Der Standardwert ist `NIL`, d. h., *alle* Teilausdrücke werden untersucht.
- `stop` — Teilausdrücke, die nicht ersetzt werden sollen: eine Menge von Typen, eine Prozedur oder `NIL`. Der Standardwert ist die Menge `{DOM_INT, DOM_RAT, DOM_IDENT}`, d. h., ganze Zahlen, rationale Zahlen und Bezeichner werden nicht durch Variablen ersetzt.

Rückgabewert: eine Folge aus dem rationalisierten Objekt und einer Menge von Substitutionsgleichungen.

Verwandte Funktionen: `indets, maprat, rewrite, simplify, subs`

Details:

⌘ Ein Ausdruck oder Teilausdruck wird als „nicht-rational“ angesehen, wenn er weder eine Summe, noch ein Produkt, noch eine Potenz mit ganzzahligem Exponenten ist.

`rationalize(object, inspect, stop)` steigt rekursiv im Ausdrucksbaum von `object` ab, solange die Typen der betrachteten Teilausdrücke in der Menge `inspect` sind. Diejenigen nicht-rationalen Teilausdrücke, deren Typen nicht in der Menge `stop` sind, werden durch neu generierte Variablen `D1, D2` etc. ersetzt.

- ⌘ `rationalize` liefert eine Folge (`rat`, `subsSet`). Hierbei enthält das rationalisierte Objekt `rat` neue Variablen, deren Bedeutung durch die Menge von „Substitutionsgleichungen“ `subsSet` festgelegt ist. Es gilt `object = subs(rat, subsSet)`.
- ⌘ Hat `inspect` den Wert `NIL`, so werden alle Teilausdrücke untersucht. Ist `inspect` eine Menge von Typen, so werden alle Teilausdrücke untersucht, die einem dieser Typen entsprechen. Ist `inspect` eine Prozedur, so werden alle Teilausdrücke `x` untersucht, für die `inspect(x)` den Wert `TRUE` ergibt.
Jeder Teilausdruck, der nicht untersucht wird, wird durch eine Variable ersetzt.
- ⌘ Hat `stop` den Wert `NIL`, so werden alle durchsuchten nicht-rationalen Teilausdrücke durch Variablen ersetzt. Ist `stop` eine Menge von Typen, so wird ein nicht-rationaler Teilausdruck nicht ersetzt, wenn er einem dieser Typen entspricht. Ist `stop` eine Prozedur, so werden nicht-rationale Teilausdrücke `x` nicht ersetzt, wenn `stop(x)` den Wert `TRUE` ergibt.
- ⌘ Die Typen in `inspect` und `stop` können Zeichenketten sein, wie sie von der Funktion `type` geliefert werden, oder Domain-Typen wie z. B. `DOM_INT`, `DOM_RAT` etc.

Beispiel 1. `rationalize` operiert auf einzelnen arithmetischen Ausdrücken sowie auch auf Mengen und Listen von Ausdrücken:

```
>> rationalize(2*sqrt(3) + 0.5*x^3)

          3                      1/2
      2 D2 + D1 x , {D1 = 0.5, D2 = 3 }

>> rationalize([(x - sqrt(2))*(x^2 + sqrt(3)),
                (x - sqrt(2))*(x - sqrt(3))])

          2                      1/2          1/2
[(x - D3) (D4 + x ), (x - D3) (x - D4)], {D3 = 2 , D4 = 3 }
```

Beispiel 2. `rationalize` ermöglicht zu spezifizieren, welche Art von Teilausdrücken untersucht und welche Art von Teilausdrücken nicht verändert werden sollen. Im folgenden Aufruf wird der Teilausdruck `x^3` (vom Typ `"_power"`) nicht untersucht und durch eine Variable ersetzt:

```
>> rationalize(2*sqrt(3) + 0.5*x^3, {"_plus", "_mult"})

          3                      1/2
      2 D5 + D6 D7, {D6 = x , D7 = 0.5, D5 = 3 }
```

Im folgenden Aufruf werden alle Teilausdrücke rekursiv untersucht. Weder Gleitpunktzahlen noch ganze Zahlen noch Bezeichner werden ersetzt:

```
>> rationalize(2*sqrt(3) + 0.5*x^3, NIL,
               {DOM_FLOAT, DOM_INT, DOM_IDENT})
```

$$2 \sqrt{3} + 0.5 x^3, \{ \sqrt{3} = 3^{1/2} \}$$

Änderungen:

☞ Die substituierten Variablen sind nun D1, D2 etc. anstatt X1, X2 etc.

read – Suchen, Lesen und Ausführen einer Datei

`read(filename)` sucht in verschiedenen Verzeichnissen nach der Datei `filename`, liest sie und führt sie aus.

`read(n)` liest die mit dem Dateibezeichner `n` verknüpfte Datei und führt sie aus.

Aufruf(e):

```
☞ read(filename <, Quiet> <, Plain>)
☞ read(n <, Quiet> <, Plain>)
```

Parameter:

`filename` — der Dateiname: eine Zeichenkette
`n` — ein von `fopen` erzeugter Dateibezeichner: eine positive ganze Zahl

Optionen:

`Plain` — mit dieser Option benutzt `read` einen eigenen Parser-Kontext
`Quiet` — unterdrückt Ausgaben während der Ausführung von `read`

Rückgabewert: der Rückgabewert der letzten Anweisung der Datei.

Verwandte Funktionen: `fclose`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `input`, `LIBPATH`, `loadproc`, `pathname`, `print`, `protocol`, `READPATH`, `textinput`, `write`, `WRITEPATH`

Details:

☞ `read(filename)` sucht in verschiedenen Verzeichnissen nach der Datei mit dem Namen `filename`:

- Zunächst wird der Name als relativer Pfadname interpretiert. Er wird dazu der Reihe nach mit den Verzeichnissen konkateniert, die in `READPATH` angegeben sind.
- Danach wird der Name als absoluter Pfadname interpretiert.
- Dann wird der Name relativ zum „aktuellen Arbeitsverzeichnis“ interpretiert.
- Schließlich wird der Name mit jedem der Verzeichnisse konkateniert, die in `LIBPATH` angegeben sind.

Falls dabei mit einem dieser Namen eine Datei geöffnet werden kann, so wird diese Datei mittels `fread` gelesen und ausgeführt.

☞ Man beachte, dass das „aktuelle Arbeitsverzeichnis“, zu welchem relative Dateinamen interpretiert werden, auf unterschiedlichen Betriebssystemen verschieden sein kann. So ist das „aktuelle Arbeitsverzeichnis“ unter Windows das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.

☞ Ein Pfadtrenner („/“ unter UNIX oder Linux, „\“ unter Windows und „:“ auf dem Macintosh) wird, falls erforderlich, beim Konkatenieren von Verzeichnisnamen und `filename` eingefügt.

☞ Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann.

☞ `read(n)` mit einem von `fopen` gelieferten Dateibezeichner `n` ist äquivalent zum Aufruf `fread(n)`.

☞ Details bezüglich des Lesens und Ausführens der Kommandos und zu den Optionen *Plain* und *Quiet* sind auf der Hilfeseite der Funktion `fread` zu finden.

Beispiel 1. Das folgende Beispiel funktioniert so nur unter UNIX und Linux; auf einem anderen Betriebssystem müssen die Dateinamen entsprechend angepasst werden. Zunächst werden mittels `write` Variablenwerte in der Datei „`testfile.mb`“ im Verzeichnis „`/tmp`“ gespeichert:

```
>> a := 3: b := 5: write("/tmp/testfile.mb", a, b):
```

Der folgende Aufruf benutzt den absoluten Pfadnamen. Durch das Einlesen der Datei werden die Werte von `a` und `b` wiederhergestellt:

```
>> delete a, b: read("/tmp/testfile.mb"): a, b
3, 5
```

Alternativ wird ein Suchverzeichnis „/tmp“ festgelegt und ein relativer Pfadname angegeben. Man beachte, dass der Pfadtrenner „/“ von read eingefügt wird:

```
>> delete a, b: READPATH := "/tmp": read("testfile.mb"): a, b
3, 5
```

Man kann die Datei auch mit fopen öffnen und dann einlesen. Da fopen nicht wie read nach der Datei sucht, muss man in diesem Fall allerdings einen absoluten Pfadnamen oder einen Namen relativ zum aktuellen Arbeitsverzeichnis angeben:

```
>> delete a, b:
  n := fopen("/tmp/testfile.mb"): read(n): fclose(n):
  a, b
3, 5

>> delete a, b, READPATH, n
```

Änderungen:

⌘ Die neue Option *Plain* wurde eingeführt.

repeat, while – repeat- und while-Schleife

`repeat` – `end_repeat` ist eine Schleife, die ihren Rumpf solange evaluiert, bis eine Abbruchbedingung erfüllt ist.

`while` – `end_while` ist eine Schleife, die ihren Rumpf evaluiert, solange eine gegebene Bedingung erfüllt ist.

Aufruf(e):

```
⌘ repeat
  body
  until condition end_repeat
⌘ _repeat(body, condition)

⌘ while condition do
  body
  end_while
⌘ _while(condition, body)
```

Parameter:

- `body` — der Schleifenrumpf: eine beliebige Folge von Anweisungen
`condition` — ein boolescher Ausdruck

Rückgabewert: der Wert des letzten innerhalb der Schleife ausgeführten Kommandos. Wurde kein Kommando innerhalb der Schleife ausgeführt, wird `NIL` zurückgegeben. Wird der Rumpf einer `while`-Schleife wegen einer nichterfüllbaren Bedingung gar nicht ausgeführt, so wird das leere Objekt vom Typ `DOM_NULL` zurückgegeben.

Weitere Dokumentation: Kapitel 16 des MuPAD-Tutoriums.

Verwandte Funktionen: `break`, `for`, `next`, `_lazy_and`, `_lazy_or`

Details:

- ☞ In einer `repeat`-Anweisung wird zunächst der Schleifenrumpf `body` und danach die Abbruchbedingung `condition` solange in einer Schleife ausgewertet, bis `condition` den Wert `TRUE` annimmt.
- ☞ In einer `while`-Anweisung wird zunächst die Bedingung `condition` ausgewertet. Wenn sich der Wert `TRUE` ergibt, werden der Schleifenrumpf `body` und die Bedingung `condition` solange in einer Schleife ausgewertet, bis der Ausdruck `condition` den Wert `FALSE` annimmt.
- ☞ Im Gegensatz zur `while`-Schleife wird der Rumpf einer `repeat`-Schleife immer mindestens einmal durchlaufen.
- ☞ Der Schleifenrumpf kann aus beliebig vielen Anweisungen bestehen, die durch einen Doppelpunkt `:` oder ein Semikolon `;` zu trennen sind. Nur das letzte evaluierte Resultat innerhalb des Rumpfs (der Rückgabewert der Schleife) wird auf dem Bildschirm ausgegeben. Zwischenergebnisse können durch `print`-Anweisungen ausgegeben werden.
- ☞ Der boolesche Ausdruck `condition` muss zu `TRUE` oder `FALSE` evaluierbar sein. Intern wird diese Bedingung im Kontext einer so genannten „lazy evaluation“ der Funktionen `_lazy_and` und `_lazy_or` ausgewertet.
- ☞ Die Anweisungen `next` und `break` können in `repeat` und `while`-Schleifen genauso eingesetzt werden wie in `for`-Schleifen.
- ☞ Statt der Schlüsselwörter `end_repeat` und `end_while` kann auch das Schlüsselwort `end` benutzt werden.
- ☞ Die imperativen Formen `repeat - end_repeat` bzw. `while - end_while` sind äquivalent zu entsprechenden Funktionsaufrufen von `_repeat` bzw. `_while`. In der Regel sind die imperativen Formen vorzuziehen, da sie zu leichter lesbarem Code führen.

☞ `_repeat` und `_while` sind Funktionen des Systemkerns.

Beispiel 1. Zwischenergebnisse der Anweisungen innerhalb einer `repeat`- bzw. `while`-Schleife werden nicht auf dem Bildschirm ausgegeben:

```
>> i := 1:
    s := 0:
    while i < 3 do
        s := s + i;
        i := i + 1;
    end_while
```

3

Oben wird allein der Rückgabewert der Schleife angezeigt. Zwischenergebnisse können durch `print`-Anweisungen ausgegeben werden:

```
>> i := 1:
    s := 0:
    while i < 3 do
        print("intermediate sum" = s);
        s := s + i;
        i := i + 1;
    end_while
```

"intermediate sum" = 0

"intermediate sum" = 1

3

```
>> delete i, s:
```

Beispiel 2. Hier wird ein einfaches Beispiel gegeben, wie eine `repeat`-Schleife durch eine äquivalente `while`-Schleife ausgedrückt werden kann. In anderen Beispielen kann dies aufwendiger sein. Gegebenenfalls werden dabei zusätzliche Initialisierungen von Variablen benötigt:

```
>> i := 1:
    repeat
        print(i);
        i := i + 1;
    until i = 3 end:
```

```

1
2

>> i := 1:
  while i < 3 do
    print(i);
    i := i + 1;
  end:

1
2

>> delete i:

```

Beispiel 3. Der boolsche Ausdruck `condition` muss zu `TRUE` oder `FALSE` evaluierbar sein:

```

>> condition := UNKNOWN:
  while not condition do
    print(Condition = condition);
    condition := TRUE;
  end_while:

Error: Unexpected boolean UNKNOWN [while]

```

Um den Fehler zu vermeiden, wird das Abbruchkriterium in `condition <> TRUE` umformuliert:

```

>> condition := UNKNOWN:
  while condition <> TRUE do
    print(Condition = condition);
    condition := TRUE;
  end_while:

Condition = UNKNOWN

>> delete condition:

```

Beispiel 4. Die Äquivalenz zwischen der funktionalen und der imperativen Form der `repeat` bzw. der `while`-Schleife wird demonstriert:

```

>> hold(_repeat((statement1; statement2), condition))

```

```

repeat
    statement1;
    statement2
until condition end_repeat

>> hold(_while(condition, (statement1; statement2)))

while condition do
    statement1;
    statement2
end_while

```

Änderungen:

- ☞ end kann als Alternative zu end_repeat und end_while benutzt werden.

rec – das Domain für Rekurrenzgleichungen

`rec(eq, y(n))` repräsentiert eine Rekurrenzgleichung für die Folge $y(n)$.

Aufruf(e):

- ☞ `rec(eq, y(n) <, cond>)`

Parameter:

- `eq` — eine Gleichung oder ein arithmetischer Ausdruck
- `y` — die Unbestimmte: ein Bezeichner
- `n` — der Folgenindex: ein Bezeichner
- `cond` — eine Menge von Anfangs- oder Randbedingungen

Rückgabewert: ein Objekt vom Typ `rec`.

Verwandte Funktionen: `ode, solve, sum`

Details:

- ☞ `rec(eq, y(n))` erzeugt ein Objekt vom Typ `rec`, das eine Rekurrenzgleichung für die Folge $y(n)$ darstellt.

Die Gleichung `eq` darf nur Verschiebungen $y(n + i)$ um ganzzahlige Werte i enthalten. Wenigstens ein solcher Ausdruck muss in `eq` vorhanden sein. Ein arithmetischer Ausdruck `eq` wird als Gleichung $eq = 0$ interpretiert.

Anfangs- oder Randbedingungen `cond` sind anzugeben als Mengen von Gleichungen der Form $\{y(n_0) = y_0, y(n_1) = y_1, \dots\}$ mit arithmetischen Ausdrücken n_0, n_1, \dots , die den Bezeichner n nicht enthalten dürfen, und arithmetischen Ausdrücken y_0, y_1, \dots , die den Bezeichner y nicht enthalten dürfen.

- ☞ Der Hauptzweck des `rec`-Domains ist es, eine Umgebung zur Überladung der Funktion `solve` zur Verfügung zu stellen. Der Aufruf `solve(r)` liefert für eine Rekurrenzgleichung r vom Typ `rec` eine Menge zurück, die einen affinen Teilraum der vollständigen Lösungsmenge darstellt. Ihr einziges Element ist ein Ausdruck im Folgenindex n , der freie Parameter C_1, C_2 etc. enthalten kann. Siehe die Beispiele ??, ?? und ??.
- ☞ Zur Zeit können nur lineare Rekurrenzgleichungen mit solchen Koeffizienten gelöst werden, die rationale Funktionen in n sind. `solve` behandelt Rekurrenzgleichungen mit konstanten Koeffizienten, findet hypergeometrische Lösungen von Rekurrenzgleichungen erster Ordnung und polynomiale Lösungen von Rekurrenzgleichungen höherer Ordnung mit nicht-konstanten Koeffizienten.
- ☞ `solve` findet nicht immer den vollständigen Lösungsraum. Siehe Beispiel ??. Wenn `solve` keine Lösung findet, dann wird der `solve`-Aufruf symbolisch zurückgeliefert. Für parametrische Rekurrenzgleichungen kann ein `solve`-Aufruf ein Objekt vom Typ `piecewise` zurückliefern. Siehe Beispiel ??.

Beispiel 1. Der erste Befehl definiert die homogene Rekurrenzgleichung erster Ordnung $y(n+1) = 2(n+1)y(n)/n$ für die Folge $y(n)$, die dann mittels `solve` gelöst wird:

```
>> rec(y(n + 1) = 2*y(n)*(n + 1)/n, y(n))
```

$$\text{rec} \left| \begin{array}{c} / \\ y(n + 1) - \frac{2 y(n) (n + 1)}{n} \\ \backslash \end{array} \right., y(n), \{ \} \left| \begin{array}{c} \backslash \\ \\ / \end{array} \right.$$

```
>> solve(%)
```

$$\left\{ \begin{array}{c} n \\ C_1 2^n \end{array} \right\}$$

Die allgemeine Lösung der Rekurrenzgleichung ist demnach $y(n) = C_1 n 2^n$, wobei C_1 für eine beliebige Konstante steht.

Beispiel 2. Im nächsten Beispiel wird die homogene Rekurrenzgleichung erster Ordnung $y(n+1) = 3(n+1)y(n)$ mit der Anfangsbedingung $y(0) = 1$ gelöst:

```
>> solve(rec(y(n + 1) = 3*(n + 1)*y(n), y(n), {y(0) = 1}))
```

$$\{3^n \cdot \Gamma(n+1)\}$$

Die Lösung ist also $y(n) = 3^n \cdot \Gamma(n+1) = 3^n \cdot n!$ für ganzzahliges $n \geq 0$ (Γ ist die Gammafunktion).

Beispiel 3. Im folgenden Beispiel wird die inhomogene Rekurrenzgleichung zweiter Ordnung $y(n+2) - 2y(n+1) + y(n) = 2$ nach der unbekannten Folge $y(n)$ gelöst. Die Anfangsbedingungen $y(0) = -1$ und $y(1) = m$ mit einem Parameter m werden von `solve` automatisch berücksichtigt:

```
>> solve(rec(y(n + 2) - 2*y(n + 1) + y(n) = 2, y(n),
           {y(0) = -1, y(1) = m}))
```

$$\{m \cdot n^2 + n^2 - 1\}$$

Beispiel 4. Die allgemeine Lösung der homogenen Rekurrenzgleichung zweiter Ordnung $y(n+2) + 3y(n+1) + 2y(n) = 0$ wird berechnet:

```
>> solve(rec(y(n + 2) + 3*y(n + 1) + 2*y(n), y(n)))
```

$$\{C6 \cdot (-1)^n + C7 \cdot (-2)^n\}$$

Hierbei sind $C6$ und $C7$ beliebige Konstanten.

Beispiel 5. Für die folgende homogene Rekurrenzgleichung dritter Ordnung findet das System nur die polynomialen Lösungen:

```
>> solve(rec(n*y(n + 3) = (n + 3)*y(n), y(n)))
```

$$\{n \cdot C9\}$$

Beispiel 6. Die folgende homogene Rekurrenzgleichung zweiter Ordnung mit konstanten Koeffizienten enthält einen Parameter a . Die Lösungsmenge der Gleichung hängt davon ab, ob dieser Parameter 0 ist, und `solve` gibt ein `piecewise`-Objekt zurück:

```
>> solve(rec(a*y(n + 2) = y(n), y(n)))
```


$$\text{piecewise} \left\{ \begin{array}{l} \{0\} \text{ if } a = 0, \\ \left\{ C_{11} \frac{1}{a} + C_{10} \frac{1}{a} \right\} \text{ if } a \neq 0 \end{array} \right.$$

Beispiel 7. Die folgende homogene Rekurrenz zweiter Ordnung mit nicht-konstanten Koeffizienten enthält einen Parameter a . Obwohl die Rekurrenz für den speziellen Wert $a = 2$ eine polynomiale Lösung besitzt, wird diese nicht vom System gefunden:

```
>> solve(rec(n*y(n + 2) = (n + a)*y(n), y(n)))
```

$$\{0\}$$

Hintergründe:

- ☞ Für homogene Rekurrenzen mit konstanten Koeffizienten berechnet `solve` zunächst die Nullstellen des charakteristischen Polynoms. Falls diese nicht alle explizit angegeben werden können, z. B. nur durch Verwendung von `RootOf`, dann gibt `solve` keine Lösung zurück. Andernfalls wird die vollständige Lösungsmenge zurückgegeben.
- ☞ Für homogene Rekurrenzen erster Ordnung mit nichtkonstanten Koeffizienten liefert `solve` den vollständigen Lösungsraum, wenn die Koeffizienten in Faktoren vom Grad höchstens zwei zerlegt werden können. Andernfalls liefert `solve` keine Lösung.
- ☞ Für homogene Rekurrenzen mindestens zweiter Ordnung mit nichtkonstanten Koeffizienten findet `solve` die vollständige Menge aller *polynomialen* Lösungen.
- ☞ Gegenwärtig können inhomogene Rekurrenzen nur dann gelöst werden, wenn sie eine polynomiale Lösung besitzen. Obige Bemerkungen gelten entsprechend.
- ☞ Bei parametrischen Rekurrenzen kann es vorkommen, dass das System solche Lösungen nicht findet, die nur für bestimmte Parameterwerte gelten. Siehe Beispiel ??.

Änderungen:

- ☞ `rec` überprüft nun seine Argumente.
 - ☞ `solve` gibt jetzt entweder eine Menge zurück, die genau einen Ausdruck enthält, einen symbolischen `solve`-Aufruf oder ein `piecewise`-Objekt. Die freien Parameter, falls vorhanden, sind nicht mehr symbolische Anfangswerte der Form $y(n_0)$, sondern durchgehend neu generierte Bezeichner.
-

`rectform` – Real- und Imaginärteil eines komplexwertigen Ausdrucks

`rectform(z)` zerlegt den komplexwertigen Ausdruck z in dessen Real- und Imaginärteil, d. h., es bestimmt eine Darstellung von z in der Form $z = \Re(z) + i\Im(z)$.

Aufruf(e):

- ☞ `rectform(z)`

Parameter:

- z — arithmetischer Ausdruck, ein Polynom, eine Reihenentwicklung, ein Feld, eine Liste oder eine Menge

Rückgabewert: ein Element des Domains `rectform`, falls z ein arithmetischer Ausdruck ist, und andernfalls ein Objekt desselben Typs wie z .

Seiteneffekte: Das Ergebnis der Funktion hängt von Eigenschaften von Bezeichnern ab, die über die Funktion `assume` gesetzt worden sind; siehe Beispiel ??.

Überladbar durch: z

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `abs`, `assume`, `collect`, `combine`, `conjugate`, `expand`, `Im`, `normal`, `radsimp`, `Re`, `rewrite`, `sign`, `simplify`

Details:

- ☞ `rectform(z)` versucht, den Ausdruck z in seinen Real- und Imaginärteil zu zerlegen und z in der Form $\Re(z) + i\Im(z)$ zurückzuliefern.

`rectform` arbeitet rekursiv, d. h. es versucht zuerst jeden Teilausdruck in Real- und Imaginärteil zu zerlegen und anschließend Real- und Imaginärteil des gesamten Ausdrucks aus denen der Teilausdrücke zu bestimmen.

Über die Funktionen `Re` und `Im` kann der Real- bzw. Imaginärteil des Ergebnisses von `rectform` extrahiert werden. Siehe Beispiel ??.

`rectform` ist weitaus mächtiger als eine direkte Anwendung der Funktionen `Re` und `Im` auf den Ausdruck `z`, jedoch auch weitaus langsamer. Für konstante arithmetische Ausdrücke wird daher empfohlen, die Funktionen `Re` und `Im` direkt zu verwenden. Siehe Beispiel ??.

`rectform` dient zum Arbeiten mit symbolischen Ausdrücken, wobei Eigenschaften von Bezeichnern berücksichtigt werden (siehe `assume`). Bezeichner ohne Eigenschaften werden als komplexwertig angenommen. Siehe Beispiel ??.

Ist `z` ein Feld, eine Liste oder eine Menge, so wird `rectform` auf jeden Eintrag von `z` angewendet.

Ist `z` ein Polynom oder eine Reihenentwicklung vom Typ `Series::Puisseux` oder `Series::gseries`, so wird `rectform` auf jeden Koeffizienten von `z` angewendet.

Siehe Beispiel ??.

Das Ergebnis `r := rectform(z)` ist ein Element des Domains `rectform`. Solch ein Domain-Element besitzt drei Operanden, die die folgende Gleichung erfüllen:

$$z = \text{op}(r, 1) + I * \text{op}(r, 2) + \text{op}(r, 3).$$

Die ersten beiden Operanden sind reelle arithmetische Ausdrücke. Der dritte Operand ist ein Ausdruck, der nicht in seinen Real- und Imaginärteil zerlegt werden konnte.

Es kann passieren, dass `rectform` nicht in der Lage ist, die geforderte Zerlegung von `z` zu bestimmen. Es versucht jedoch dann, Teile des Real- und Imaginärteils von `z` zu extrahieren. Die extrahierten Teile finden sich in den ersten beiden Operanden. Der dritte Operand enthält den Rest, aus dem keine weiteren Informationen mehr gewonnen werden können. Im schlimmsten Fall können die ersten beiden Operanden gleich Null sein. Beispiel ?? erläutert verschiedene Fälle.

Es können arithmetische Operationen mit Elementen vom Typ `rectform` ausgeführt werden. Das Ergebnis einer solchen Operation ist dann erneut ein Element des Domains `rectform` (siehe Beispiel ??).

Viele der MuPAD-Funktionen, die auf arithmetischen Ausdrücken arbeiten (beispielsweise `expand`, `normal`, `simplify` etc.), können auf Elemente vom Typ `rectform` angewendet werden. Sie werden auf die drei Operanden des Elementes getrennt angewendet.

☞ Mit der Funktion `expr` kann das Ergebnis von `rectform` in ein Element eines Basisdomains konvertiert werden (siehe Beispiel ??).

Beispiel 1. Der Term $\sin(z)$ für komplexwertige z gliedert sich wie folgt in Real- und Imaginärteil auf:

```
>> delete z: r := rectform(sin(z))

      sin(Re(z)) cosh(Im(z)) + (cos(Re(z)) sinh(Im(z))) I
```

Der Real- und Imaginärteil lässt sich daraus unmittelbar wie folgt gewinnen:

```
>> Re(r), Im(r)

      sin(Re(z)) cosh(Im(z)), cos(Re(z)) sinh(Im(z))
```

Die Konjugierte von r ergibt sich daraus unmittelbar:

```
>> conjugate(r)

      sin(Re(z)) cosh(Im(z)) + (-cos(Re(z)) sinh(Im(z))) I
```

Beispiel 2. Der Real- und Imaginärteil von konstanten arithmetischen Ausdrücken kann über die Funktionen `Re` und `Im` bestimmt werden, wie das folgende Beispiel zeigt:

```
>> Re(ln(-4)) + I*Im(ln(-4))

      I PI + ln(4)
```

Diese Funktionen arbeiten wesentlich effizienter als `rectform`, jedoch scheitern sie an der Berechnung des Real- und Imaginärteils von beliebigen symbolischen Ausdrücken, was am Beispiel des Terms $e^{i \sin z}$ gezeigt werden kann:

```
>> delete z: f := exp(I*sin(z)):
      Re(f), Im(f)

      Re(exp(I sin(z))), Im(exp(I sin(z)))
```

Die Funktion `rectform` ist mächtiger. Sie ist in der Lage, den obigen Ausdruck in seinen Real- und Imaginärteil zu zerlegen:

```
>> r := rectform(f)

      cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z))) +
      (sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))) I
```

Nun können wir den Real- und Imaginärteil von f wie folgt gewinnen:

```
>> Re(r)
cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))
>> Im(r)
sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))
```

Beispiel 3. Bezeichner ohne Eigenschaften werden als komplexwertige Variablen betrachtet:

```
>> delete z: rectform(ln(z))
ln(Im(z)2 + Re(z)2)
----- + I arg(Re(z), Im(z))
2
```

Über die Zuweisung von Eigenschaften an Bezeichner kann auf diese Voreinstellung Einfluss genommen werden. Repräsentiert beispielsweise z im obigen Term eine (reelle) negative Zahl, so vereinfacht sich die Berechnung des Real- und Imaginärteils erheblich:

```
>> assume(z < 0): rectform(ln(z))
ln(-z) + I PI
```

Beispiel 4. Wir zerlegen die (komplexwertige) Variable x in ihren Real- und Imaginärteil:

```
>> delete x: a := rectform(x)
Re(x) + I Im(x)
```

Das gleiche machen wir für die reellwertige Variable y :

```
>> delete y: assume(y, Type::Real): b := rectform(y)
y
>> domtype(a), domtype(b)
rectform, rectform
```

Die Ergebnisse, die Elemente vom Domain-Typ `rectform` sind, werden in den Bezeichnern `a` und `b` gespeichert. Berechnen wir dann die Summe `a` und `b`, so ist das Ergebnis erneut ein Element vom Domain-Typ `rectform`, d. h. ein Ausdruck, der in Real- und Imaginärteil zerlegt ist:

```
>> c := a + b
```

$$(y + \operatorname{Re}(x)) + I \operatorname{Im}(x)$$

```
>> domtype(c)
```

`rectform`

Selbst wenn wir beliebige arithmetische Ausdrücke mit einbeziehen, wird das Ergebnis ein Element vom Domain-Typ `rectform` sein:

```
>> delete z: d := a + 2*b + exp(z)
```

$$(2y + \operatorname{Re}(x) + \cos(\operatorname{Im}(z)) \exp(\operatorname{Re}(z))) +$$

$$I (\operatorname{Im}(x) + \sin(\operatorname{Im}(z)) \exp(\operatorname{Re}(z)))$$

```
>> domtype(d)
```

`rectform`

Über die Funktion `expr` werden solche Elemente in ein entsprechendes Element eines Basisdomains konvertiert:

```
>> expr(d)
```

$$2y + I \operatorname{Im}(x) + \operatorname{Re}(x) + \cos(\operatorname{Im}(z)) \exp(\operatorname{Re}(z)) +$$

$$I \sin(\operatorname{Im}(z)) \exp(\operatorname{Re}(z))$$

```
>> domtype(%)
```

`DOM_EXPR`

Beispiel 5. `rectform` wirkt auf Polynome und Reihenentwicklungen, indem `rectform` auf die Koeffizienten solcher Objekte angewandt wird:

```
>> delete x, y: p := poly(ln(-4) + y*x, [x]):  
rectform(p)
```

$$\operatorname{poly}((\operatorname{Re}(y) + I \operatorname{Im}(y)) x + (\ln(4) + I \pi), [x])$$

Ebenso wirkt `rectform` auf Listen, Mengen und Felder, indem es auf jeden Eintrag angewandt wird:

```
>> a := array(1..2, [x, y]):
      rectform(a)
```

$$\begin{array}{cc} + - & - + \\ | \operatorname{Re}(x) + I \operatorname{Im}(x), \operatorname{Re}(y) + I \operatorname{Im}(y) | \\ + - & - + \end{array}$$

Man beachte, dass `rectform` nicht auf alle elementare Datentypen anwendbar ist. Wird beispielsweise `rectform` auf eine Tabelle angewendet, so wird eine Fehlermeldung ausgegeben:

```
>> a := table("1st" = x, "2nd" = y):
      rectform(a)
```

```
Error: invalid argument, expecting an arithmetical expression \
[rectform::new]
```

Man kann die Funktion `map` verwenden, um `rectform` auf die Operanden eines solchen Objekts anzuwenden:

```
>> map(a, rectform)

      table(
        "2nd" = Re(y) + I Im(y),
        "1st" = Re(x) + I Im(x)
      )
```

Beispiel 6. Dieses Beispiel verdeutlicht die Bedeutung der drei Operanden von Objekten, die von `rectform` zurückgeliefert wird.

Wir beginnen mit dem Term $x + \sin(y)$, den `rectform` vollständig in seinen Real- und Imaginärteil zerlegen kann:

```
>> delete x, y: r := rectform(x + sin(y))

      (Re(x) + sin(Re(y)) cosh(Im(y))) +
      I (Im(x) + cos(Re(y)) sinh(Im(y)))
```

Die ersten beiden Operanden von `r` sind der Real- und Imaginärteil des Terms, und der dritte Operand trägt den Wert 0:

```
>> op(r)

      Re(x) + sin(Re(y)) cosh(Im(y)),
      Im(x) + cos(Re(y)) sinh(Im(y)), 0
```

Nun betrachten wir den Term $x + f(y)$, wobei $f(y)$ eine unbekannte Funktion in einer komplexen Variable repräsentiert. `rectform` kann den Real- und Imaginärteil von x bestimmen, jedoch nicht für den Teilterm $f(y)$:

```
>> delete f: r := rectform(x + f(y))

      Re(x) + I Im(x) + f(y)
```

Die ersten beiden Operanden bestehen aus dem Real- und dem Imaginärteil von x . Der dritte Operand ist der Restterm $f(y)$, für den `rectform` keine weiteren Informationen über den Real- und Imaginärteil bestimmen konnte:

```
>> op(r)

      Re(x), Im(x), f(y)

>> Re(r), Im(r)

      Re(x) + Re(f(y)), Im(x) + Im(f(y))
```

Im schlimmsten Fall kann `rectform` keinerlei Informationen über den Real- und Imaginärteil des Eingabeterms finden. Dann besteht der dritte Operand aus dem vollständigen Eingabeterm, möglicherweise durch das rekursive Wirken von `rectform` in einer umgeschriebenen Form. Die ersten beiden Operanden sind 0. Hier ist ein Beispiel für einen solchen Fall:

```
>> r := rectform(sin(x + f(y)))

      sin(f(y) + I Im(x) + Re(x))

>> op(r)

      0, 0, sin(f(y) + I Im(x) + Re(x))

>> Re(r), Im(r)

      Re(sin(f(y) + I Im(x) + Re(x))),

      Im(sin(f(y) + I Im(x) + Re(x)))
```

Beispiel 7. Fortgeschrittene Benutzer können die Funktion `rectform` für ihre eigenen speziellen mathematischen Funktionen erweitern (siehe „Hintergründe“ unten). Zu diesem Zweck bettet man die mathematische Funktion in eine Funktionsumgebung `f` ein und implementiert deren Slot "`rectform`", der das Verhalten der Funktion `rectform` für die Funktionsumgebung `f` beschreibt.

Enthält nun ein Ausdruck z einen Teilausdruck der Form $f(u, \dots)$, so bewirkt die Funktion `rectform` den Aufruf `f::rectform(u, \dots)` der Slot-Routine, um die Zerlegung von $f(u, \dots)$ in seinen Real- und Imaginärteil zu bestimmen.

Wir zeigen diese Funktionsweise am Beispiel der Sinus-Funktion. Die Funktionsumgebung `sin` besitzt natürlich schon den Slot `"rectform"`. Wir nennen unsere Funktionsumgebung daher `Sin`, um nicht die Systemfunktion `sin` zu überschreiben:

```
>> Sin := funcenv(Sin):
      Sin::rectform := proc(u) // Berechne rectform(Sin(u)):
        local r, a, b;
        begin
          // Wende rekursiv rectform auf das Argument an:
          r := rectform(u);

          if op(r, 3) <> 0 then
            // Der Real- und Imaginärteil von Sin(u) kann nicht
            // bestimmt werden:
            new(rectform, 0, 0, Sin(u))
          else
            a := op(r, 1); // der Realteil von u
            b := op(r, 2); // der Imaginärteil von u
            new(rectform, Sin(a)*cosh(b), cos(a)*sinh(b), 0)
          end_if
        end:
      end:

>> delete z: rectform(Sin(z))

      Sin(Re(z)) cosh(Im(z)) + (cos(Re(z)) sinh(Im(z))) I
```

Falls die `if`-Bedingung wahr wird, ist `rectform` nicht in der Lage, `u` vollständig in seinen Real- und Imaginärteil zu zerlegen. In diesem Fall ist auch `Sin::rectform` nicht in der Lage, `Sin(u)` in seinen Real- und Imaginärteil zu zerlegen und liefert daher ein Objekt vom Typ `rectform` zurück, dessen dritter Operand aus dem gesamten Ausdruck `Sin(u)` besteht:

```
>> delete f: rectform(Sin(f(z)))

      Sin(f(z))

>> op(%)

      0, 0, Sin(f(z))
```

Hintergründe:

- ⌘ Wenn in `z` ein Teilausdruck der Form `f(u, . . .)` vorkommt, wobei `f` eine Funktionsumgebung ist, so versucht `rectform` den Slot `"rectform"` von `f` aufzurufen, um die Zerlegung von `f(u, . . .)` in seinen Real- und Imaginärteil zu bestimmen. Damit läßt sich die Funktionalität von `rectform` für benutzereigene mathematische Funktionen erweitern.

Der Slot "rectform" von f wird mit den Argumenten u, \dots von f aufgerufen. Falls die Routine $f::\text{rectform}$ nicht in der Lage ist, die geforderte Zerlegung von $f(u, \dots)$ vorzunehmen, sollte sie das Objekt $\text{new}(\text{rectform}(0, 0, f(u, \dots)))$ zurückliefern. Siehe Beispiel ??.

Falls f keinen Slot "rectform" hat, so gibt rectform das Objekt $\text{new}(\text{rectform}(0, 0, f(u, \dots)))$ für den entsprechenden Teilausdruck zurück.

- ⌘ Entsprechendes gilt für Domainelemente: Tritt ein Domainelement d eines Bibliotheks-Domains T als Teilausdruck in z auf, so versucht rectform den Slot "rectform" dieses Domains mit dem Element d als Argument aufzurufen, um die Zerlegung von d in seinen Real- und Imaginärteil zu bestimmen.

Falls die Routine $T::\text{rectform}$ nicht in der Lage ist, die geforderte Zerlegung von d vorzunehmen, sollte sie das Element $\text{new}(\text{rectform}(0, 0, d))$ zurückliefern.

Falls das Domain T keinen Slot "rectform" hat, so gibt rectform das Objekt $\text{new}(\text{rectform}(0, 0, d))$ für den entsprechenden Teilausdruck zurück.

Änderungen:

- ⌘ rectform berücksichtigt Eigenschaften von Bezeichnern (siehe `assume`). Der zweite Parameter (eine Menge von reellwertigen Variablen) ist dadurch obsolet geworden.

register – Speicherbegrenzung der Demo-Version aufheben

`register(Name, Key)` registriert die MuPAD-Installation auf UNIX-Plattformen.

Aufruf(e):

- ⌘ `register(Name, Key)`

Parameter:

- Name — der Namenseintrag eines Registrierungs-Codes: eine Zeichenkette
- Key — der Registrierungsschlüssel: eine Zeichenkette

Rückgabewert: `TRUE`, wenn die Registrierung erfolgreich war, und sonst `FALSE`.

Weitere Dokumentation: Siehe das MuPAD Lizenzabkommen, das unter der Internet-Adresse http://www.sciface.com/mupad_download/reg_form.html erhältlich ist.

Details:

- ☞ Die freie MuPAD-Version, die vom Internet heruntergeladen werden kann, hat eine eingebaute Speicherbegrenzung von 6 Megabytes. Mit `register` kann die Speicherbegrenzung unter UNIX aufgehoben werden.

Unter Windows kann man die MuPAD-Version über den Menüeintrag „Registrieren“ im Hilfe-Menü registrieren.

Auf Macintosh-Plattformen erreicht man diesen Menüeintrag unter „Über MuPAD“ im Apple-Menü.

- ☞ Sie können über die folgende Internetseite einen Registrierungsschlüssel bekommen:

`http://www.sciface.com/mupad_download/reg_form.html`

- ☞ Sie benötigen für die Registrierung Schreibrechte auf den Verzeichnisbaum, in dem MuPAD installiert wurde. Im Zweifelsfall registrieren Sie als Benutzer *root*. Siehe auch Beispiel ??.

Beispiel 1. Ist der Schlüssel korrekt und die Registrierung war erfolgreich, dann liefert `register` `TRUE`:

```
>> register("My name", "12345-67890-ABCDE")
```

```
Memory limitation removed.
```

`TRUE`

Beispiel 2. Ein ungültiger Schlüssel führt zu der folgenden Nachricht:

```
>> register("My name", "invalid key")
```

```
Wrong password or not registered user.
```

`FALSE`

Beispiel 3. Ist der Schlüssel in Ordnung, aber Sie haben keine Schreibrechte auf dem Verzeichnisbaum, in dem MuPAD installiert wurde, dann passiert folgendes:

```
>> register("My name", "12345-67890-ABCDE")
```

Cannot remove memory limitation.

FALSE

Änderungen:

☞ Keine Änderungen.

reset – Rücksetzen einer MuPAD-Sitzung

`reset()` reinitialisiert eine MuPAD-Sitzung, so dass sie sich anschließend wie eine neu gestartete Sitzung verhält.

Aufruf(e):

☞ `reset()`

Rückgabewert: das leere Objekt `null()` vom Typ `DOM_NULL`.

Verwandte Funktionen: `delete`, `quit`

Details:

- ☞ `reset` initialisiert eine MuPAD-Sitzung neu. Nach dem Aufruf von `reset()` verhält sich die aktuelle Sitzung wie eine frisch gestartete Sitzung. Die Werte aller Bezeichner werden gelöscht und die Umgebungsvariablen auf ihre Standardwerte gesetzt. Schließlich werden die Initialisierungsdateien `sysinit.mu` und `userinit.mu` neu gelesen.
 - ☞ `reset` darf nur im interaktiven Modus verwendet werden. In einer Prozedur löst der Aufruf einen Fehler aus.
 - ☞ `reset` ist eine Funktion des Systemkerns.
-

Beispiel 1. `reset` löscht die Werte aller Bezeichner und setzt Umgebungsvariablen auf ihre Standardwerte zurück:

```
>> a := 1: DIGITS := 5: reset(): a, DIGITS  
a, 10
```

Änderungen:

- ☞ Keine Änderungen.
-

return – Beenden einer Prozedur

`return(x)` beendet die Ausführung einer Prozedur und liefert `x` als Rückgabewert.

Aufruf(e):

- ☞ `return(x)`

Parameter:

`x` — ein beliebiges MuPAD-Objekt

Rückgabewert: `x`.

Verwandte Funktionen: `DOM_PROC`, `proc`, `->`

Details:

- ☞ Üblicherweise beendet MuPAD eine Prozedur, wenn alle Anweisungen innerhalb der Prozedur ausgeführt wurden. Der Rückgabewert ist das Ergebnis der zuletzt ausgeführten Anweisung.
Alternativ verursacht der Aufruf `return(x)` innerhalb einer Prozedur den unmittelbaren Ausstieg aus der Prozedur. Das Argument `x` wird dabei zum Rückgabewert der Prozedur. Die Evaluierung setzt direkt nach dem Punkt wieder ein, wo die Prozedur aufgerufen wurde.
 - ☞ `x` darf eine Ausdrucksfolge sein, d. h., Aufrufe der Form `return(x1, x2, ...)` sind möglich.
 - ☞ `return()` liefert das leere Objekt vom Typ `DOM_NULL` als Rückgabewert.
 - ☞ Man beachte, dass `return` eine Funktion ist und nicht ein Schlüsselwort der MuPAD-Sprache. Ein Aufruf der Form `return x;` (wie beispielsweise in der Programmiersprache C) verursacht in MuPAD einen Syntaxfehler.
 - ☞ Außerhalb einer Prozedur liefert `return(x)` lediglich den Wert `x` zurück.
 - ☞ `return` ist eine Funktion des Systemkerns.
-

Beispiel 1. Dieses Beispiel zeigt die Implementation einer Maximumfunktion (die im Gegensatz zur Systemfunktion `max` nur zwei Argumente erlaubt). Ist `x` größer als `y`, so wird der Wert von `x` zurückgegeben und die Ausführung der Prozedur dabei beendet. Im anderen Fall wird `return` nicht aufgerufen, und `y` ist als zuletzt ausgewertetes Objekt der Rückgabewert:

```
>> mymax := proc(x : Type::Real, y : Type::Real)
  begin
    if x > y then
      return(x)
    end_if;
    y
  end_proc;

>> mymax(3, 2), mymax(4, 5)

3, 5

>> delete mymax:
```

Beispiel 2. `return()` liefert das leere Objekt:

```
>> f := x -> return(): type(f(anything))

DOM_NULL

>> delete f:
```

Beispiel 3. Wird `return` auf interaktiver Ebene aufgerufen, so werden lediglich die evaluierten Argumente zurückgegeben:

```
>> x := 1: return(x, y)

1, y

>> delete x:
```

Änderungen:

⌘ Keine Änderungen.

revert – Umkehrung von Listen und Zeichenketten, Invertierung von Reihenentwicklungen

`revert` kehrt die Reihenfolge der Elemente einer Liste oder der Buchstaben einer Zeichenkette um. Für Reihenentwicklungen wird die funktionale Inverse berechnet.

Aufruf(e):

⌘ `revert(object)`

Parameter:

`object` — eine Liste, eine Zeichenkette oder eine Reihenentwicklung vom Typ `Series::Puisseux`

Rückgabewert: ein Objekt desselben Typs wie das Eingabeobjekt oder ein symbolischer Funktionsaufruf vom Typ `"revert"`.

Überladbar durch: `object`

Verwandte Funktionen: `series, substring`

Details:

- ⌘ `revert` ist eine allgemeine Funktion zur Berechnung von Kompositionsinversen oder zur Umkehrung der Reihenfolge von Elementen. Diese Funktionalität kann per Überladung auf weitere Typen von Eingabeobjekten erweitert werden.
 - ⌘ Derzeit stellt die MuPAD-Bibliothek Funktionalität für Listen und Zeichenketten bereit, für die `revert` die Reihenfolge der Elemente bzw. Zeichen umdreht. Weiterhin wird für Reihenentwicklungen die funktionale Inverse zurückgeliefert.
 - ⌘ Für alle anderen Typen von Eingabeobjekten, welche `revert` nicht überladen, wird der symbolische Aufruf `revert(object)` zurückgeliefert.
-

Beispiel 1. `revert` operiert auf Listen und Zeichenketten:

```
>> revert([1, 2, 3, 4, 5])  
  
[5, 4, 3, 2, 1]  
  
>> revert("nuf si DAPuM ni gnimmargorP")  
  
"Programming in MuPAD is fun"
```

`revert` operiert auf Reihen:

```
>> revert(series(sin(x), x)) = series(arcsin(x), x)
```

$$x + \frac{x^3}{6} + \frac{3x^5}{40} + O(x^6) = x + \frac{x^3}{6} + \frac{3x^5}{40} + O(x^6)$$

Die Inverse einer Reihenentwicklung von \exp um den Punkt $x = 0$ ist die Reihenentwicklung der inversen Funktion \ln um den Punkt $x = \exp(0) = 1$:

```
>> revert(series(exp(x), x, 3)) = series(ln(x), x = 1, 3)
```

$$(x - 1) - \frac{(x - 1)^2}{2} + O((x - 1)^3) =$$

$$(x - 1) - \frac{(x - 1)^2}{2} + O((x - 1)^3)$$

Beispiel 2. Für alle anderen Eingabetypen wird ein symbolischer Funktionsaufruf zurückgeliefert:

```
>> revert(x + y)
```

```
revert(x + y)
```

Die folgende Reihenentwicklung liefert keine Reihe vom Typ `Series::Puisseux`, sondern eine verallgemeinerte Entwicklung vom Typ `Series::gseries`, die von `revert` nicht invertiert wird:

```
>> revert(series(exp(-x)/(1 + x), x = infinity, 3))
```

$$\text{revert} \left| \frac{1}{x \exp(x)} - \frac{1}{2 x^2 \exp(x)} + O\left(\frac{1}{x^3 \exp(x)}\right) \right|$$

Änderungen:

☞ Keine Änderungen.

rewrite – Umformung eines Ausdrucks

`rewrite(f, target)` schreibt den Ausdruck `expr` in eine mathematisch äquivalente Form um, die die Zielfunktion `target` benutzt.

Aufruf(e):

⌘ `rewrite(f, target)`

Parameter:

`f` — ein arithmetischer Ausdruck
`target` — die Zielfunktion, mit der dargestellt werden soll: `cot`,
`coth`, `diff`, `exp`, `fact`, `gamma`, `heaviside`, `ln`,
`piecewise`, `sign`, `sincos`, `sinhcosh`, `tan` oder `tanh`

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `f`

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `collect`, `combine`, `expand`, `factor`, `normal`,
`partfrac`, `rationalize`, `rectform`, `simplify`

Details:

- ⌘ Das Ziel `target` stellt die Funktion dar, die in der angestrebten Darstellung verwendet werden soll. Unevaluierte Funktionsaufrufe in `f` werden durch diese Zielfunktion ausgedrückt, wenn dies mathematisch möglich ist.
- ⌘ Mit dem Ziel `exp` werden alle trigonometrischen und hyperbolischen Funktionen mittels `exp` umgeschrieben. Weiterhin werden die inversen Funktionen sowie `arg` mittels `ln` ausgedrückt.
- ⌘ Mit dem Ziel `sincos` werden die Funktionen `tan`, `cot`, `exp`, `sinh`, `cosh`, `tanh`, and `coth` mittels `sin` und `cos` ausgedrückt.
- ⌘ Mit dem Ziel `sincos` werden die Funktionen `exp`, `tanh`, `coth`, `sin`, `cos`, `tan` und `cot` mittels `sinh` und `cosh` ausgedrückt.

Beispiel 1. Dieses Beispiel demonstriert den Gebrauch von `rewrite`:

```
>> rewrite(D(D(y))(x), diff)
               diff(y(x), x, x)
>> rewrite(fact(n), gamma), rewrite(gamma(n), fact);
```

```

gamma(n + 1), fact(n - 1)
>> rewrite(sign(x), heaviside), rewrite(heaviside(x), sign);
          sign(x)
2 heaviside(x) - 1, ----- + 1/2
                    2
>> rewrite(heaviside(x), piecewise)
piecewise(1 if 0 < x, heaviside(0) if x = 0, -1 if x < 0)

```

Beispiel 2. Trigonometrische Funktionen können mittels exp, sin, cos etc. dargestellt werden:

```

>> rewrite(tan(x), exp), rewrite(cot(x), sincos),
rewrite(sin(x), tan)

```

$$\begin{aligned}
& \frac{I \exp(I x)^2 - I \cos(x)}{\exp(I x)^2 + 1}, \frac{\sin(x)}{\cos(x)}, \frac{\tan(x)}{\sqrt{2}} \\
& \frac{I \exp(I x)^2 - I \cos(x)}{\exp(I x)^2 + 1}, \frac{\sin(x)}{\cos(x)}, \frac{\tan(x)}{\sqrt{2}}
\end{aligned}$$

```

>> rewrite(arcsinh(x), ln)
ln(x + (x^2 + 1)^(1/2))

```

Änderungen:

- Die Zielfunktionen cot, coth, tanh und piecewise wurden hinzugefügt.

RGB – vordefinierte Farbnamen

RGB::Name wertet sich zu einer Liste [r, g, b] von Rot-, Grün- und Blauanteilen der Farbe „Name“ gemäß des RGB-Farbmodells aus.

RGB::ColorNames() liefert eine Liste aller vordefinierten Farbnamen.

RGB::ColorNames(subname) liefert eine Liste aller vordefinierten Farbnamen, welche den angegebenen Teilnamen subname enthalten.

Aufruf(e):

```
# RGB::Name
# RGB::ColorNames()
# RGB::ColorNames(subname)
```

Parameter:

Name — der Farbname: ein Bezeichner
subname — ein Teil eines Farbnamens: ein Bezeichner

Rückgabewert: RGB::Name wertet sich zu einer Liste [r, g, b] reeller Gleitpunktwerte zwischen 0.0 und 1.0 aus. RGB::ColorNames liefert eine Liste von Farbnamen.

Verwandte Funktionen: plot2d, plotfunc2d, plot3d, plotfunc3d

Details:

```
# RGB-Werte können in Plot-Kommandos benutzt werden.
```

Beispiel 1. Die Grundfarben des RGB-Modells sind Rot, Grün und Blau:

```
>> RGB::Red, RGB::Green, RGB::Blue
      [1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]
```

Der folgende Aufruf liefert alle vordefinierten Farbnamen, die ‚Olive‘ enthalten:

```
>> RGB::ColorNames(Olive)
      [OliveDrab, Olive, OliveGreenDark]
```

Die RGB-Werte dieser Farben sind:

```
>> RGB::OliveDrab, RGB::Olive, RGB::OliveGreenDark
      [0.419599, 0.556902, 0.137303],
      [0.230003, 0.370006, 0.170003],
      [0.333293, 0.419599, 0.184301]
```

Beispiel 2. Das folgende Kommando zeichnet ein gefülltes graues Dreieck mit schwarzen Randlinien auf weißem Hintergrund:

```
>> plot2d(BackGround = RGB::White,
          ForeGround = RGB::Black,
          Labeling = TRUE,
          [Mode = List, [polygon(point(0, 0),
                                point(1, 0),
                                point(0, 1),
                                Closed = TRUE,
                                Filled = TRUE,
                                Color = RGB::LightGrey)]
          ])
```

Änderungen:

⌘ Keine Änderungen.

select – Auswählen von Operanden

`select(object, f)` liefert eine Kopie des Objekts, in der alle Operanden entfernt wurden, die ein durch die Prozedur `f` definiertes Kriterium nicht erfüllen.

Aufruf(e):

⌘ `select(object, f <, p1, p2, ...>)`

Parameter:

<code>object</code>	— eine Liste, eine Menge, eine Tabelle, eine Ausdrucksfolge oder ein Ausdruck vom Typ <code>DOM_EXPR</code>
<code>f</code>	— eine Prozedur, die einen boolschen Wert zurückliefert
<code>p1, p2, ...</code>	— beliebige MuPAD-Objekte, die von <code>f</code> als zusätzliche Argumente akzeptiert werden

Rückgabewert: ein Objekt vom selben Typ wie das Eingabeobjekt.

Überladbar durch: `object`

Verwandte Funktionen: `map, op, split, zip`

Details:

- ⇒ `select` ist eine schnelle, nützliche Funktion, um Elemente aus Listen, Mengen, Tabellen etc. zu extrahieren, die ein durch die Funktion `f` definiertes Kriterium erfüllen.
- ⇒ Die Funktion `f` muss einen Wert liefern, der sich zu einem der boolschen Werte `TRUE`, `FALSE` oder `UNKNOWN` auswerten läßt. Sie kann einen dieser Werte direkt liefern, aber auch eine Gleichung oder eine Ungleichung, die von der Funktion `bool` zu einem dieser Werte vereinfacht werden kann.
- ⇒ Intern wird die Prozedur `f` mit dem Aufruf `f(x, p1, p2, ...)` auf alle Operanden `x` des Eingabeobjekts angewendet. Ist das Ergebnis nicht `TRUE`, so wird dieser Operand entfernt. Das Eingabeobjekt selbst wird dadurch nicht verändert.
Das Ausgabeobjekt ist vom selben Typ wie das Eingabeobjekt, d. h., eine Liste liefert eine Liste, eine Menge liefert eine Menge etc.
- ⇒ Eine Ausdrucksfolge als Eingabeobjekt wird nicht ausgeglichen. Siehe Beispiel ??.
- ⇒ Auch „atomare“ Objekte wie z.B. Zahlen oder Bezeichner können als erstes Argument an `select` übergeben werden. Sie werden wie Folgen mit einem einzigen Element behandelt.
- ⇒ `select` ist eine Funktion des Systemkerns.

Beispiel 1. `select` arbeitet mit Listen und Mengen. Im erste Beispiel werden alle wahren Aussagen aus einer Liste von logischen Aussagen ausgewählt. Das Ergebnis ist wieder eine Liste:

```
>> select([1 = 1, 1 = 2, 2 = 1, 2 = 2], bool)
[1 = 1, 2 = 2]
```

Im folgenden Beispiel wird aus einer Menge die Teilmenge aller Elemente ausgewählt, die `iszero` als Null erkennt:

```
>> select({0, 1, x, 0.0, 4*x}, iszero)
{0, 0.0}
```

`select` arbeitet auch mit Tabellen:

```
>> T:= table(1 = "y", 2 = "n", 3 = "n", 4 = "y", 5 = "y"):
select(T, has, "y")
```

```

table(
  5 = "y",
  4 = "y",
  1 = "y"
)

```

Der folgende Ausdruck ist eine Summe, d. h., ein Ausdruck vom Typ `"_plus"`. Es werden diejenigen Summanden ausgewählt, die `x` nicht enthalten:

```

>> select(x^5 + 2*x + y - 4, _not@has, x)

      y - 4

```

Aus dem folgenden Produkt werden alle Faktoren ausgewählt, die den Bezeichner `x` enthalten. Das Ergebnis ist ein Produkt mit genau einem Faktor. Rein syntaktisch ist dieses Ergebnis daher nicht vom Typ `"_mult"`:

```

>> select(11*x^2*y*(1 - y), has, x)

      2
     x

```

```

>> delete T:

```

Beispiel 2. `select` arbeitet mit Ausdrucksfolgen:

```

>> select((1, -4, 3, 0, -5, -2), testtype, Type::Negative)

      -4, -5, -2

```

Der `$`-Befehl erzeugt solche Ausdrucksfolgen:

```

>> select(i $ i = 1..20, isprime)

      2, 3, 5, 7, 11, 13, 17, 19

```

„Atomare“ Objekte werden wie Ausdrucksfolgen mit einem Element behandelt:

```

>> select(5, isprime)

      5

```

Im folgenden Beispiel wird das leere Objekt `null()` vom Typ `DOM_NULL` zurückgegeben:

```

>> domtype(select(6, isprime))

      DOM_NULL

```

Beispiel 3. Es ist möglich, als Argument *f* eine „anonyme“ Prozedur direkt anzugeben. Hiermit können komplexere Aktionen mit einem einzigen Aufruf ausgeführt werden, ohne die Prozedur vorher einem Bezeichner zuweisen zu müssen. Im folgenden Beispiel liefert `anames(All)` eine Menge mit allen Bezeichnern, die in der laufenden MuPAD-Sitzung einen Werte haben. Mittels `select` werden hieraus diejenigen Bezeichner ausgewählt, die mit dem Buchstaben "h" beginnen:

```
>> select(anames(All), x -> expr2text(x)[0] = "h")
      {has, hold, help, hastype, history, heaviside}
```

Änderungen:

⌘ Keine Änderungen.

series – Berechnung einer (verallgemeinerten) Reihenentwicklung

`series(f, x = x0)` berechnet die ersten Terme einer Reihenentwicklung von *f* bezüglich der Variablen *x* um den Punkt *x0*.

Aufruf(e):

⌘ `series(f, x <= x0> <, order> <, dir> <, NoWarning>)`

Parameter:

- f* — ein arithmetischer Ausdruck, als Funktion in *x* zu interpretieren
- x* — ein Bezeichner
- x0* — der Entwicklungspunkt: ein arithmetischer Ausdruck. Ohne Angabe dieses Punktes wird der Entwicklungspunkt 0 benutzt.
- order* — die Anzahl der zu berechnenden Terme: eine nicht-negative ganze Zahl oder *infinity*. Die Standardordnung ist durch die Umgebungsvariable `ORDER` mit dem voreingestellten Wert 6 gegeben.

Optionen:

- `dir` — *Left*, *Right*, oder *Real*. Falls keine in der komplexen Ebene gültige Entwicklung existiert, können hiermit Entwicklungen angefordert werden, die nur längs der reellen Achse gültig zu sein brauchen.
- `NoWarning` — unterdrückt Warnungen, die während der Reihenberechnung ausgegeben werden. Das kann nützlich sein, wenn `series` von benutzerdefinierten Prozeduren aus aufgerufen wird.

Rückgabewert: Falls `order` eine natürliche Zahl ist, dann gibt `series` ein Objekt vom Domain-Typ `Series::Puisseux` oder `Series::gseries` oder einen Ausdruck vom Typ `"series"` zurück. Für `order = infinity` wird ein arithmetischer Ausdruck zurückgegeben.

Seiteneffekte: Die Ergebnisse der Funktion hängen vom Wert der Umgebungsvariablen `ORDER` ab, die standardmäßig die Anzahl der Terme in Reihenentwicklungen bestimmt.

Überladbar durch: `f`

Verwandte Funktionen: `asympt`, `limit`, `O`, `ORDER`, `Series::gseries`, `Series::Puisseux`, `taylor`, `Type::Series`

Details:

- ☞ `series` versucht entweder die Taylor-Reihe, die Laurent-Reihe, die Puiseux-Reihe oder eine verallgemeinerte Reihenentwicklung von `f` um `x = x0` zu bestimmen. Siehe `Series::gseries` für weitere Informationen zu verallgemeinerten Reihenentwicklungen.
Mittels des Typ-Ausdrucks `Type::Series` kann der mathematische Typ einer von `series` gelieferten Reihe festgestellt werden.
- ☞ Kann `series` eine Reihenentwicklung von `f` nicht durchführen, so wird der Funktionsaufruf mit evaluierten Argumenten symbolisch zurückgeliefert. Das Ergebnis ist dann ein Ausdruck vom Typ `"series"`. Siehe Beispiel ??.
- ☞ Mathematisch ist die von `series` berechnete Entwicklung in einer Umgebung des Entwicklungspunktes in der komplexen Ebene gültig. Mit den Optionen *Left* oder *Right* kann man gerichtete Entwicklungen bestimmen, die nur entlang der reellen Achse gültig zu sein brauchen. Mit der Option *Real* wird eine zweiseitige Entwicklung entlang der reellen Achse berechnet. Siehe die Beispiele ?? und ??.
- ☞ Falls `x0` den Wert `infinity` bzw. `-infinity` hat, dann wird eine Reihenentwicklung längs der reellen Achse von links an das positive reelle

Unendliche bzw. von rechts an das negative reelle Unendliche berechnet. Siehe Beispiel ??.

Solche eine Reihe wird wie folgt bestimmt: Die Variable x in f wird durch $x = 1/u$ ersetzt. Anschließend wird eine gerichtete Reihenentwicklung von f um $u = 0+$ bestimmt und zuletzt die Variable u wieder durch $u = 1/x$ ersetzt.

Das Ergebnis einer solchen Reihenberechnung ist im mathematischen Sinn eine Potenzreihe in $1/x$. Es kann allerdings passieren, dass die Koeffizienten der Entwicklung von der Reihenvariable abhängen. Siehe den entsprechenden Punkt weiter unten.

- ☞ Das optionale Argument `order` bestimmt die Anzahl der Terme der Entwicklung. Ohne Angabe von `order` wird der Wert der Umgebungsvariablen `ORDER` benutzt, deren Standardwert 6 durch Zuweisung an `ORDER` verändert werden kann.

Die Anzahl der Terme der Entwicklung wird vom Term mit dem kleinsten Grad an gezählt, d. h., „`order`“ ist als „relative Abbruchordnung“ anzusehen.

Es kann vorkommen, dass die Anzahl der Terme in der berechneten Reihenentwicklung von der geforderten Anzahl abweicht. Siehe die Beispiele ?? und ??.



- ☞ In manchen Fällen kann es bei der Reihenberechnung zu Auslöschungseffekten kommen, so dass die angegebene Ordnung zu klein ist, um eine Reihenentwicklung durchzuführen. Dann wird die Berechnung mit einer Fehlermeldung abgebrochen. Siehe Beispiel ??.
 - ☞ Falls `order` den Wert `infinity` hat, dann versucht das System, das erste Argument in eine formale unendliche Reihe zu konvertieren. Das heißt, dass eine allgemeine Formel für den n -ten Koeffizienten in der Taylor-Entwicklung von f berechnet wird. Das Ergebnis ist dann eine symbolische Summe. Siehe Beispiel ??.
 - ☞ Falls `series` eine Reihenentwicklung vom Domain-Typ `Series::Puiseux` zurückliefert, so kann es passieren, dass die Koeffizienten der Entwicklung von der Reihenvariable abhängen. In so einem Fall ist die Entwicklung keine Puiseux-Reihe im mathematischen Sinne. Siehe Beispiel ??.
- Wenn die Reihenvariable x ist und der Entwicklungspunkt x_0 , dann gilt jedoch die folgende Aussage für jede Koeffizientenfunktion $c(x)$ und alle positiven ε : Die Funktion $c(x)(x - x_0)^\varepsilon$ konvergiert für $x \rightarrow x_0$ gegen 0, während die Funktion $c(x)(x - x_0)^{-\varepsilon}$ in der Nähe von x_0 unbeschränkt ist. In ähnlicher Weise gilt für den Entwicklungspunkt ∞ und alle positiven ε , dass für große reelle Werte von x die Funktion $c(x)x^{-\varepsilon}$ verschwindet und die Funktion $c(x)x^\varepsilon$ unbeschränkt ist.
- ☞ Die zurückgelieferten Reihenentwicklungen können mit den arithmetischen Standard-Operationen weiterverarbeitet werden. Weiterhin sind

folgende Methoden vorhanden: `ldegree` gibt den Exponenten des Leitterms zurück, `Series::Puisseux::order` den Exponenten des Fehlerterms, `expr` wandelt in einen arithmetischen Ausdruck um und schneidet dabei den Fehlerterm ab, `coeff(s, n)` gibt den Koeffizienten desjenigen Terms von `s` mit Exponent `n` zurück, `lcoeff` gibt den führenden Koeffizienten zurück, `revert` berechnet das Kompositionsinverse der Reihenentwicklung, `diff` differenziert eine Reihenentwicklung, `map` wendet eine Funktion auf alle Koeffizienten an. Weitere Details sind auf den Hilfeseiten zu `Series::Puisseux` bzw. `Series::gseries` zu finden.

Beispiel 1. Wir berechnen eine Reihenentwicklung von $\sin(x)$ um $x = 0$. Dies ist eine Taylor-Reihe:

```
>> s := series(sin(x), x)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^6)$$

Syntaktisch ist das Ergebnis ein Objekt vom Datentyp `Series::Puisseux`:

```
>> domtype(s)
```

`Series::Puisseux`

Der mathematische Typ der Reihenentwicklung kann mit dem Typausdruck `Type::Series` untersucht werden:

```
>> testtype(s, Type::Series(Taylor))
```

TRUE

Diverse Systemfunktionen wurden überladen, um auf Reihen-Objekte angewendet werden zu können. Beispielsweise können mit der Funktion `coeff` die Koeffizienten einer Reihenentwicklung extrahiert werden:

```
>> coeff(s, 5)
```

1/120

Summen und Produkte von Reihen können mit den üblichen Operatoren berechnet werden:

```
>> s + 2*s, s*s
```

$$3x - \frac{x^3}{2} + \frac{x^5}{40} + O(x^6), \quad x^2 - \frac{x^4}{3} + \frac{2x^6}{45} + O(x^7)$$

```
>> delete s:
```

Beispiel 2. Dieses Beispiel berechnet die Komposition einer Reihe s mit sich selbst, d. h., die Reihenentwicklung von $\sin(\sin(x))$.

```
>> s := series(sin(x), x): s @ s = series(sin(sin(x)), x)
```

$$x - \frac{x^3}{3} + \frac{x^5}{10} + O(x^6) = x - \frac{x^3}{3} + \frac{x^5}{10} + O(x^6)$$

```
>> delete s:
```

Beispiel 3. Die Reihenentwicklung des Tangens am Ursprung wird auf zwei verschiedene Arten berechnet:

```
>> series(sin(x), x) / series(cos(x), x) = series(tan(x), x)
```

$$x + \frac{x^3}{3} + \frac{2x^5}{15} + O(x^6) = x + \frac{x^3}{3} + \frac{2x^5}{15} + O(x^6)$$

```
>> bool(%)
```

TRUE

Beispiel 4. Ohne ein optionales Argument wird die Funktion sign nicht weiter entwickelt:

```
>> series(x*sign(x^2 + x), x)
```

$$x \text{ sign}(x + x^2) + O(x^7)$$

Das Ergebnis ist ein wenig einfacher, wenn man an einer Entwicklung interessiert ist, die nur entlang der reellen Achse gültig zu sein braucht:

```
>> series(x*sign(x^2 + x), x, Real)
```

$$x \text{ sign}(x) + O(x^6)$$

Die Funktion sign verschwindet aus dem Ergebnis, wenn man eine einseitig gerichtete Entwicklung entlang der reellen Achse anfordert:

```
>> series(x*sign(x^2 + x), x, Right),  
series(x*sign(x^2 + x), x, Left)
```

$$x + O(x^6), -x + O(x^6)$$

Beispiel 5. In MuPAD ist die Funktion `heaviside` nur auf der reellen Achse definiert. Eine ungerichtete Entwicklung in der komplexen Ebene macht daher keinen Sinn:

```
>> series(x*heaviside(x + 1), x)

Warning: Could not find undirected series expansion; try option
'Left', 'Right', or 'Real' [Series::main]

series(x heaviside(x + 1), x)
```

Mit einer solchen Option wird eine Entwicklung entlang der reellen Achse berechnet:

```
>> series(x*heaviside(x + 1), x, Real),
series(x*heaviside(x + 1), x, Right)

7 7
x + O(x ), x + O(x )
```

Die Funktion `heaviside` ist am Punkt i in der komplexen Ebene nicht definiert. Daher ist dort auch keine Reihenentwicklung möglich:

```
>> series(heaviside(x), x = I, Real)

Error: heaviside is not defined for non-real expansion points \
[heaviside::series]
```

Beispiel 6. Hier wird eine Laurent-Entwicklung um den Punkt 1 berechnet:

```
>> series(1/(x^2 - 1), x = 1)

1 2 3
----- - 1/4 + | - - 1/8 | - ---- + ---- +
2 (x - 1) \ 8 / (x - 1) (x - 1)
16 32

4
O((x - 1) )
```

Beispiel 7. Reihenentwicklungen um Unendlich werden berechnet:

```
>> s1 := series((x + 1)/(x - 1), x = infinity)

2 2 2 2 / 1 \
1 + - + -- + -- + -- + 0 | -- |
x 2 3 4 | 5 |
x x x \ x /
```

```
>> s2 := series(psi(x), x = infinity)
```

$$\ln(x) - \frac{1}{2x} - \frac{1}{12x^2} + \frac{1}{120x^4} + O\left(\frac{1}{x^5}\right)$$

```
>> domtype(s1), domtype(s2)
```

```
Series::Puisseux, Series::Puisseux
```

Obwohl beide Entwicklungen vom Domain-Typ `Series::Puisseux` sind, stellt `s2` keine Puiseux-Reihe im mathematischen Sinn dar, denn der erste Term enthält die Logarithmusfunktion, welche eine wesentliche Singularität bei Unendlich hat:

```
>> coeff(s2)
```

```
ln(x), -1/2, -1/12, 0, 1/120
```

Die folgende Entwicklung ist vom Domain-Typ `Series::gseries`:

```
>> s3 := series(exp(x)/(1 - x), x = infinity, 4)
```

$$-\frac{\exp(x)}{x} - \frac{\exp(x)}{x^2} - \frac{\exp(x)}{x^3} + O\left(\frac{\exp(x)}{x^4}\right)$$

```
>> domtype(s3)
```

```
Series::gseries
```

```
>> delete s1, s2, s3:
```

Beispiel 8. In diesem Beispiel wird eine allgemeine Formel für den n -ten Koeffizienten a_n in der Taylor-Entwicklung der Funktion $\exp(-x) = \sum_{n \geq 0} a_n x^n$ um den Nullpunkt berechnet, indem für den Parameter `order` der Wert `infinity` angegeben wird. Das Ergebnis ist eine symbolische Summe:

```
>> series(exp(-x), x, infinity)
```

$$\sum_{n1=0}^{\infty} \frac{(-1)^{n1}}{n1!} x^{n1}$$

Beispiel 9. Die Sinusfunktion hat eine wesentliche Singularität bei Unendlich. Dort kann `series` keine Reihenentwicklung berechnen und liefert einen symbolischen Funktionsaufruf zurück:

```
>> series(sin(x), x = infinity)
      series(sin(x), x = infinity)
>> domtype(%), type(%)
      DOM_EXPR, "series"
```

Beispiel 10. Im folgenden Beispiel ist wegen Auslöschungseffekten die angeforderte Anzahl von Termen für die Reihenentwicklung zu klein, um die reziproke Reihe zu berechnen:

```
>> series(exp(x), x, 3)
      2
      x      3
      1 + x + -- + O(x )
      2
```

```
>> series(1/(exp(x) - 1 - x - x^2/2), x, 3)
```

Error: order too small [Series::Puisseux::_invert]

Mit höherer Ordnung kann eine Entwicklung berechnet werden, allerdings mit einer kleineren Anzahl von Termen:

```
>> series(1/(exp(x) - 1 - x - x^2/2), x, 5)
```

$$\frac{1}{x^3} - \frac{3}{2x^2} + \frac{1}{x} - \frac{1}{2} + O(x)$$

Beispiel 11. Hier sind einige Beispiele, in denen die Anzahl der berechneten Terme von der geforderten Anzahl abweicht:

```
>> series(sin(x^2), x, 5)
      2      5
      x + O(x )
```

```
>> series((sin(x^4) - tan(x^4)) / x^10, x, 15)
```

$$-\frac{x^2}{2} + O(x^5)$$

Beispiel 12. Man kann die Fähigkeiten von `series` dadurch erweitern, dass man `series`-Attribute (Slots) für benutzerdefinierte spezielle mathematische Funktionen implementiert. Dies wird im Folgenden anhand der Exponentialfunktion illustriert. (Natürlich hat diese Funktion bereits ein `series`-Attribut in MuPAD, welches man sich mit `expose(exp::series)` anzeigen lassen kann.) Damit das bereits existierende Attribut nicht überschrieben wird, wird mit einer Kopie der Exponentialfunktion namens `Exp` gearbeitet.

Das `series`-Attribut ist eine Prozedur mit vier Argumenten, die immer dann aufgerufen wird, wenn eine Reihenentwicklung der speziellen Funktion mit einem beliebigen Argument berechnet werden soll. Das erste Argument dieser Prozedur ist genau das Argument der speziellen Funktion in dem `series`-Aufruf. Das zweite Argument ist die Reihenvariable; der Entwicklungspunkt ist dabei immer der Ursprung 0. (Andere Entwicklungspunkte werden intern durch eine Variablentransformation in den Ursprung verschoben.) Das dritte und das vierte Argument sind identisch mit den Argumenten `order` und `dir` von `series`.

Der Befehl `series(Exp(x^2 + 2), x, 5)` zum Beispiel wird intern in den Aufruf `Exp::series(x^2 + x, x, 5, FAIL)` umgewandelt. In diesem Beispiel zeigt das vierte Argument `FAIL` an, dass eine ungerichtete Reihenentwicklung berechnet werden soll. Es folgt nun ein Beispiel eines `series`-Attributs für `Exp`.

```
>> // series-Attribut für Exp. Zuständig für den
// Aufruf series(Exp(f), x = 0, order, dir)
ExpSeries := proc(f, x, order, dir)
    local t, x0, s, r, i;
    begin
        // Entwickle das Argument rekursiv in eine Reihe.
        t := series(f, x, order, dir);

        // Bestimme die Ordnung k des kleinsten Terms in t, so
        // dass t = c*x^k + Terme höherer Ordnung ist, wobei
        // c eine von 0 verschiedene Konstante ist.
        k := ldegree(t);

        if k = FAIL then
            // t besteht nur aus einem Fehlerterm O(..)
            error("order too small");

        elif k < 0 then
            // Dies entspricht einer Entwicklung von exp um Unendlich
            // oder minus Unendlich. So eine Entwicklung existiert aber
            // nicht für die Exponentialfunktion, da sie im
            // Unendlichen eine wesentliche Singularität besitzt.
            // Deswegen wird FAIL zurückgegeben, was dazu führt, dass
            // der series-Aufruf unevaluiert zurückgegeben wird.
            // Für andere spezielle Funktionen kann hier eine
```

```

// asymptotische Entwicklung implementiert werden.
return(FAIL);

else // k >= 0
// Dies entspricht einer Entwicklung von exp um
// einen endlichen Punkt x0. Wir schreiben t = x0 + y,
// wobei alle Terme in y positive Ordnung haben.
// Dann gilt exp(x0 + y) = exp(x0)*exp(y), und die
// Entwicklung von exp(y) wird als Komposition der
// Reihenentwicklung von exp(x) um x = 0 mit der Reihe
// t - x0 berechnet. Sollte die spezielle Funktion
// endliche Singularitäten haben, dann sind diese
// hier separat zu behandeln.
x0 := coeff(t, x, 0);
s := Series::Puisseux::create(1, 0, order,
    [1/i! $ i = 0..(order - 1)], x, 0);
return(Series::Puisseux::scalmult(s @ (t - x0), Exp(x0), 0))
end_if
end_proc:

```

Die spezielle Funktion muss in eine Funktionsumgebung eingebettet werden, damit ihr ein `series`-Attribut zugeordnet werden kann. Der folgende Befehl definiert `Exp` als eine solche Funktionsumgebung und kopiert denjenigen Teil der Systemfunktion `exp`, der für deren Auswertung zuständig ist. Der `subsop`-Befehl bewirkt, dass `Exp` mit symbolischen Argumenten auch als `Exp` und nicht als `exp` zurückgegeben wird, siehe die Hilfeseite zu `DOM_PROC`.

```

>> Exp := funcenv(subsop(op(exp, 1), 6 = hold(Exp)), NIL, NIL):
    Exp(1), Exp(-1.0), Exp(x^2 + x)

```

```

                                2
    Exp(1), 0.3678794412, Exp(x + x )

```

`series` kann bereits mit dieser „neuen“ Funktion umgehen, es wird aber lediglich eine Taylor-Entwicklung mit symbolischen Ableitungen berechnet:

```

>> ORDER := 3: series(Exp(x), x = 0)

```

```

                                2
                                x  D(D(Exp))(0)          3
    1 + x D(Exp)(0) + ----- + O(x )
                                2

```

Nun wird die Prozedur `ExpSeries` dem `series`-Attribut von `Exp` zugeordnet:

```

>> Exp::series := ExpSeries:

```

Schließlich kann das neue Attribut getestet werden:


```
>> series(Exp(x^2 + x), x = 0) = series(exp(x^2 + x), x = 0)

      2      2
      3 x      3      3 x      3
      1 + x + ---- + O(x ) = 1 + x + ---- + O(x )
      2      2

>> series(Exp(x^2 + x), x = 2) = series(exp(x^2 + x), x = 2)

      2
      27 Exp(6) (x - 2)      3
Exp(6) + 5 Exp(6) (x - 2) + ---- + O((x - 2) ) =
      2

      2
      27 exp(6) (x - 2)      3
exp(6) + 5 exp(6) (x - 2) + ---- + O((x -
2) )
      2

>> series(Exp(x^2 + x), x = 0, 0)
Error: order too small [ExpSeries]

>> series(Exp(x^2 + x), x = infinity)

      2
series(Exp(x + x ), x = infinity)
```

Eine weitere Möglichkeit, Reihenentwicklungen für benutzerdefinierte Funktionen zu erhalten, besteht darin, das `diff`-Attribut der entsprechenden Funktionsumgebung zu implementieren. Dieses wird von `series` für die Berechnung einer Taylor-Entwicklung verwendet, wenn kein `series`-Attribut definiert ist. Dementsprechend erhält man in solchen Fällen nur dann eine Reihenentwicklung, wenn eine Taylor-Entwicklung existiert, während ein `series`-Attribut auch allgemeinere Reihenentwicklungen liefern kann.

```
>> delete ExpSeries, Exp:
```

Änderungen:

☞ Die neuen Optionen *Real* und *NoWarning* wurden eingeführt.

setuserinfo – Setzen eines Informationsgrades

`setuserinfo(f, n)` setzt den Informationsgrad für die Funktion `f` auf `n` und schaltet dadurch in `f` eingebaute `userinfo`-Befehle ein oder aus.

Aufruf(e):

```
# setuserinfo(f, n <, style>)  
# setuserinfo(f)  
# setuserinfo(n)  
# setuserinfo(NIL)  
# setuserinfo()
```

Parameter:

f — eine Prozedur, ein Domainname oder *Any*
n — der „Informationsgrad“: eine nichtnegative ganze Zahl
style — entweder *Name* oder *Quiet*

Optionen:

Name — veranlasst `userinfo`, den Namen der aufrufenden Funktion an die ausgegebenen Informationen anzuhängen
Quiet — veranlasst `userinfo`, den Präfix „Info:“ am Anfang einer Zeile zu unterdrücken

Rückgabewert: der zuvor gültige Informationsgrad.

Verwandte Funktionen: `print`, `userinfo`, `warning`

Details:

- # Der Informationsgrad kontrolliert die Ausgabe der Funktion `userinfo`. Diese Funktion ist in viele Bibliotheksfunktionen eingebaut, um bei Bedarf Informationen über den internen Ablauf von Algorithmen auszugeben.
- # `setuserinfo(f, n <, style>)` setzt den Informationsgrad von *f* auf den Wert *n* und liefert den zuvor gesetzten Wert. Das Setzen des Informationsgrades eines Domains verändert nicht die gesetzten Informationsgrade der Methoden dieses Domains.
- # `setuserinfo(f)` liefert den momentan gültigen Informationsgrad von *f*, ohne ihn zu verändern.
- # `setuserinfo(Any, n <, style>)` setzt global den Informationsgrad aller Funktionen auf den Wert *n*. Hierbei werden gesetzte Informationsgrade von Domains und Prozeduren nicht verändert.
- # `setuserinfo(n)` ist äquivalent zu `setuserinfo(Any, n)`.
- # `setuserinfo(Any)` liefert den momentan gültigen globalen Informationsgrad, ohne ihn zu verändern.

☞ `setuserinfo(NIL)` setzt den Informationsgrad *aller* Funktionen und Domains auf den Standardwert 0. Üblicherweise gibt `userinfo` mit diesem Wert keine Informationen aus.

☞ `setuserinfo()` liefert eine Tabelle aller gesetzten Informationsgrade. Diese Tabelle wird durch den Aufruf `setuserinfo(NIL)` gelöscht.

Beispiel 1. Eine Prozedur `f` wird definiert, die Informationen mittels `userinfo` ausgibt:

```
>> f := proc(x)
      begin
        userinfo(1, "enter 'f'");
        userinfo(2, "the argument is " . expr2text(x));
        x^2
      end_proc;
```

Nachdem die `userinfo`-Anweisungen innerhalb von `f` mittels `setuserinfo` aktiviert sind, liefert jeder Aufruf von `f` Statusinformationen:

```
>> setuserinfo(f, 1, Name): f(5)
Info: enter 'f' [f]
```

25

Der Informationsgrad von `f` wird erhöht:

```
>> setuserinfo(f, 2): f(4)
Info: enter 'f'
Info: the argument is 4
```

16

Der Präfix „Info:“ soll nicht angezeigt werden:

```
>> setuserinfo(f, 2, Quiet): f(3)
enter 'f'
the argument is 3
```

9

Die `userinfo`-Anweisungen werden deaktiviert, indem global alle Informationsgrade gelöscht werden:

```
>> setuserinfo(NIL): f(2)
```

4

```
>> delete f:
```

Änderungen:

- ☞ Die neuen Optionen *Name* und *Quiet* wurden eingeführt.
-

share – Erzeugen einer „Unique Data Representation“

`share()` erzeugt eine „Unique Data Representation“ für jedes MuPAD-Objekt. Diese Funktion dient einem sehr technischen Zweck und wird in der Regel vom Benutzer nicht benötigt.

Aufruf(e):

- ☞ `share()`

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Verwandte Funktionen: `bytes`

Details:

- ☞ `share` bewirkt, dass vor der Ausführung des nächsten Kommandos auf interaktiver Ebene eine „Unique Data Representation“ für jedes MuPAD-Objekt hergestellt wird. Das heißt, dass alle MuPAD-Objekte, die identisch sind, physikalisch nur einmal im Speicher gehalten werden. Damit verringert `share` den logisch benutzten Speicherbereich.
 - ☞ `share` ist langsam und braucht selbst viel Speicher während seiner Ausführung.
 - ☞ `share` wird nicht sofort ausgeführt, sondern erst dann, wenn man auf den interaktiven Level der MuPAD-Sitzung zurückkehrt. `share` kann also nicht dazu benutzt werden, um während einer längeren Rechnung Speicher freizugeben.
 - ☞ `share` ist eine Funktion des Systemkerns.
-

Beispiel 1. Das folgende Beispiel wurde in einer frisch gestarteten MuPAD-Sitzung ausgeführt. Man erkennt, dass durch `share` zwar logischer Speicher freigegeben wird, der MuPAD-Kern benötigt jedoch physikalisch neuen Speicher zur Herstellung der „Unique Data Representation“. Die Ausgabe des Beispiels wird auf unterschiedlichen Maschinen voneinander abweichen:

```
>> int(x, x): bytes()
```

```
1980600, 2191872, 2147483647
```

```
>> share(): bytes()
```

```
1201076, 2830848, 2147483647
```

Änderungen:

☞ share ist eine neue Funktion.

sign – das Vorzeichen einer reellen oder komplexen Zahl

sign(z) liefert das Vorzeichen der Zahl z.

Aufruf(e):

☞ sign(z)

Parameter:

z — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: z

Seiteneffekte: sign berücksichtigt Eigenschaften von Bezeichnern. Für reelle Ausdrücke kann das Resultat vom Wert der Umgebungsvariablen DIGITS abhängen.

Verwandte Funktionen: abs, conjugate, Im, Re

Details:

- ☞ Das Signum einer komplexen Zahl $z \neq 0$ ist mathematisch definiert als $z/|z|$. Für reelle Zahlen ist dies 1 oder -1 .
- ☞ sign(0) und sign(0.0) liefern 0. Der Anwender kann diesen Wert durch eine Zuweisung umdefinieren, z. B.
`unprotect(sign): sign(0) := 1: protect(sign):`
- ☞ Ist z vom Typ DOM_INT, DOM_RAT oder DOM_FLOAT, so wird eine schnelle Kernfunktion benutzt, um das Vorzeichen zu bestimmen. Das Ergebnis ist entweder -1 , 0 oder 1.

- ☞ Kann das Signum eines Ausdrucks nicht bestimmt werden, so wird ein symbolischer Funktionsaufruf zurückgeliefert, wobei einige Vereinfachungsregeln angewendet werden. Insbesondere werden numerische Faktoren in symbolischen Produkten vereinfacht. Siehe Beispiel ??.
- ☞ Die `expand`-Funktion schreibt das Signum eines Produkts in ein Produkt von Signumwerten um (`expand(sign(x*y))` liefert `sign(x)*sign(y)`). Siehe Beispiel ??.
- ☞ Konstante Ausdrücke wie z. B. `PI - sqrt(2)`, `exp(I*3) - I*sin(3)` etc. werden intern durch Gleitpunktauswertung daraufhin überprüft, ob sie reell und ungleich 0 sind. Ist dies der Fall, so wird `-1` oder `1` zurückgeliefert, wobei intern die Güte der Gleitpunktapproximation getestet wird. Siehe Beispiel ??.

Beispiel 1. Die Vorzeichen einiger reeller Zahlen und Ausdrücke werden bestimmt:

```
>> sign(-8/3), sign(3.2), sign(exp(3) - sqrt(2)*PI), sign(0)
-1, 1, 1, 0
```

Das Signum einer komplexen Zahl z ist die komplexe Zahl $z/abs(z)$:

```
>> sign(0.5 + 1.1*I), sign(2 + 3*I), sign(exp(sin(2 + 3*I)))
0.4138029443 + 0.9103664775 I, (2/13 + 3/13 I)^(1/2),
exp(I cos(2) sinh(3))
```

Beispiel 2. `sign` liefert einen symbolischen Funktionsaufruf zurück, wenn der Ausdruck Unbekannte enthält:

```
>> sign(x), sign(2*x*y), sign(2*x + y), sign(PI*exp(2 + y))
sign(x), sign(x y), sign(2 x + y), sign(exp(y + 2))
```

Mittels `expand` können in speziellen Fällen weitere Vereinfachungen erreicht werden:

```
>> expand(sign(2*x*y)), expand(sign(PI*exp(2 + y)))
sign(x) sign(y), sign(exp(y))
```

Beispiel 3. `sign` reagiert auf Eigenschaften von Bezeichnern:

```
>> sign(x + PI)
sign(x + PI)
>> assume(x > -3): sign(x + PI)
1
>> unassume(x):
```

Beispiel 4. Die folgende rationale Zahl approximiert π auf etwa 20 Stellen genau:

```
>> p := 157079632679489661923/50000000000000000000:
```

Mit der Standardgenauigkeit `DIGITS = 10` kann `sign` mit seinem internen Gleitpunkttest nicht entscheiden, ob `p` kleiner oder größer als π ist:

```
>> float(PI - p)
0.0
```

Dieses Ergebnis wird durch numerische Auslöschung bestimmt und erlaubt keinen Schluss auf das Vorzeichen von `PI - p`. Der Gleitpunkttest innerhalb von `sign` überprüft die Verlässlichkeit numerischer Approximationen und liefert in diesem Fall kein vereinfachtes Ergebnis:

```
>> sign(PI - p)
sign(PI - 157079632679489661923/50000000000000000000)
```

Mit erhöhten `DIGITS` kann eine verlässliche Entscheidung gefällt werden:

```
>> DIGITS := 20: sign(PI - p)
1
>> delete p, DIGITS:
```

Änderungen:

⌘ Einige Vereinfachungsregeln wurden geändert.

`signIm` – das Vorzeichen des Imaginärteils einer komplexen Zahl

`signIm(z)` repräsentiert das Vorzeichen von `Im(z)`.

Aufruf(e):

$\#$ `signIm(z)`

Parameter:

z — ein arithmetischer Ausdruck, der eine komplexe Zahl repräsentiert

Rückgabewert: entweder ± 1 , 0 oder eine symbolischer Aufruf.

Überladbar durch: z

Details:

$\#$ `signIm(z)` liefert die Information, ob eine komplexe Zahl z in der oberen oder der unteren Halbebene liegt: `signIm(z)` liefert 1, falls $\text{Im}(z) > 0$ gilt oder falls z reell und $z < 0$ ist. Am Ursprung gilt `signIm(0)=0`. Für alle anderen numerischen Argumente wird -1 zurückgeliefert. Damit gilt `signIm(z)=sign(Im(z))`, falls z nicht auf der reellen Achse liegt.

$\#$ Kann die Lage des Argumentes in der komplexen Ebene nicht bestimmt werden, so wird ein unevaluierter Aufruf zurückgeliefert.

$\#$ Die Funktionen `diff` und `series` behandeln `signIm` als konstante Funktion. Siehe Beispiel ??.

$\#$ Die folgende Beziehung gilt für beliebiges komplexes z und p :

$$(-z)^p = z^p (-1)^{-p \text{ signIm}(z)}.$$

Beispiel 1. Für numerische Argumente kann die Lage in der komplexen Ebenen immer bestimmt werden:

```
>> signIm(2 + I), signIm(- 4 - I*PI), signIm(0.3), signIm(-
2/7),
    signIm(-sqrt(2) + 3*I*PI)

1, -1, -1, 1, 1
```

Symbolische Argumente ohne Eigenschaften führen zu unevaluierten Aufrufen:

```
>> signIm(x), signIm(x - I*sqrt(2))

1/2
signIm(x), signIm(x - I 2 )
```

Durch `assume` gesetzte Eigenschaften werden berücksichtigt:


```

>> assume(x, Type::Real): signIm(x - I*sqrt(2))
-1
>> assume(x > 0): signIm(x)
-1
>> assume(x < 0): signIm(x)
1
>> assume(x = 0): signIm(x)
0
>> unassume(x):

```

Beispiel 2. Abgesehen von den Sprungunstetigkeiten längs der reellen Achse ist `signIm` eine konstante Funktion. Die Unstetigkeiten werden von `diff` ignoriert:

```

>> diff(signIm(z), z)
0

```

Auch `series` behandelt `signIm` als konstante Funktion:

```

>> series(signIm(z/(1 - z)), z = 0)
      /      z      \      6
signIm| ----- | + O(z )
      \ - z + 1 /

```

Änderungen:

⚡ `signIm` ist eine neue Funktion.

`simplify` – Vereinfachung von Ausdrücken

`simplify(f)` wendet Termersetzungsregeln an, um den Ausdruck `f` zu vereinfachen.

`simplify(f, target)` schränkt die Vereinfachungen auf Termersetzungsregeln ein, die auf die durch `target` angegebene(n) Zielfunktion(en) anwendbar sind.

Aufruf(e):

`simplify(f <, target>)`
`simplify(l <, target>)`

Parameter:

`f` — ein arithmetischer Ausdruck
`l` — eine Menge, eine Liste, ein Array oder ein Polynom vom Typ `DOM_POLY`

Optionen:

`target` — einer der Bezeichner `cos`, `sin`, `exp`, `ln`, `sqrt`, `logic` oder `relation`

Rückgabewert: ein Objekt vom selben Typ wie das Eingabeobjekt `f` bzw. `l`.

Überladbar durch: `f`, `l`

Seiteneffekte: Ohne eine `target`-Option reagiert `simplify` auf Eigenschaften von Bezeichnern.

Weitere Dokumentation: Kapitel „Manipulation von Ausdrücken“ des Tutoriums.

Verwandte Funktionen: `collect`, `combine`, `expand`, `factor`, `match`, `normal`, `radsimp`, `rectform`, `rewrite`

Details:

- ☞ In einem Aufruf ohne `target`-Option wird zunächst versucht, Vereinfachungen des Gesamtausdrucks zu erreichen. Dies beinhaltet insbesondere das Umschreiben von Produkten trigonometrischer und exponentieller Terme. Dann wird `simplify` rekursiv auf die Operanden des Ausdrucks angewendet. Dabei werden die "`simplify`"-Methoden der speziellen Funktionen aufgerufen, die im Ausdruck enthalten sind.
- ☞ Der Aufruf `simplify(f)` impliziert alle Vereinfachungen, die mit den Optionen `sin`, `cos`, `exp` und `ln` erreicht werden können.
- ☞ Die Aufrufe `simplify(f, sqrt)` und `radsimp(f)` sind äquivalent. Hiermit werden konstante Radikalausdrücke vereinfacht.
- ☞ Im Aufruf `simplify(l <, target>)` wird die Vereinfachung auf die Operanden des Objekts `l` angewendet.

Option <target>:

- ☞ Mit den Optionen `sin`, `cos`, `exp` und `ln` werden nur spezifische Vereinfachungen wie das Umschreiben von Produkten trigonometrischer oder exponentieller Terme durchgeführt.
- ☞ Mit der Option `sqrt` wird `radsimp` aufgerufen, d. h., Radikale werden vereinfacht. Die Hilfeseite von `radsimp` liefert weitere Details.
- ☞ Mit der Option `logic` werden Regeln der booleschen Algebra auf boolesche Ausdrücke angewendet. Der property-Mechanismus wird nicht eingesetzt, um die Wahrheit von Atomen zu entscheiden.
- ☞ Die Option `relation` ist obsolet, da nunmehr auch arithmetische Operationen für Gleichungen und Ungleichungen in MuPAD definiert sind. Sie wird nur noch aus Gründen der Abwärtskompatibilität unterstützt.

Beispiel 1. `simplify` versucht, arithmetische Ausdrücke zu vereinfachen:

```
>> simplify(exp(x)-exp(x/2)^2)
0
```

```
>> f := sin(x)^2 + cos(x)^2 + (exp(x) - 1)/(exp(x/2) + 1):
simplify(f)
```

$$\frac{\exp(x) - 1}{\exp\left(\frac{x}{2}\right) + 1}$$

Bei Angabe von Optionen werden nur spezielle Vereinfachungen durchgeführt:

```
>> simplify(f, sin)
```

$$\frac{\exp(x) + \exp\left(\frac{x}{2}\right) - 1}{\exp\left(\frac{x}{2}\right) + 1}$$

```
>> simplify(f, exp)
```

$$\cos^2(x) + \sin^2(x) + \exp\left(\frac{x}{2}\right) - 1$$

```
>> delete f:
```

Beispiel 2. Die Option *sqrt* dient zur Vereinfachung von Radikalen:

```
>> simplify(sqrt(4 + 2*sqrt(3)), sqrt)
```

$$\frac{1}{3} + 1$$

```
>> x := 1/2 + sqrt(23/108):
    y := x^(1/3) + 1/3/x^(1/3):
    z := y^3 - y
```

$$\frac{\frac{1}{\sqrt[3]{3 \sqrt[3]{23} + 18}} + \frac{1}{2} \sqrt[3]{\frac{1}{3}}}{\frac{\sqrt[3]{3 \sqrt[3]{23} + 18}}{3} + \frac{1}{2} \sqrt[3]{\frac{1}{3}}} - \frac{1}{\frac{\sqrt[3]{3 \sqrt[3]{23} + 18}}{3} + \frac{1}{2} \sqrt[3]{\frac{1}{3}}}$$

```
>> simplify(z, sqrt)
```

$$1$$

```
>> delete x, y, z:
```

Beispiel 3. Die Option *logic* dient zur Vereinfachung boolescher Ausdrücke:

```
>> simplify((a and b) or (a and (not b)), logic)
```

$$a$$

Beispiel 4. Benutzerdefinierte Funktionen können ein "simplify"-Attribut haben. Sei z. B. f eine Funktion, von der nur bekannt ist, dass sie additiv ist. Somit kann der Funktionswert von f nur an der Stelle 0 berechnet werden, aber MuPAD kann veranlasst werden, die Additivität von f zu benutzen:

```

>> f := funcenv( x -> if iszero(x) then 0 else procname(x) end):
      f::simplify := proc(F)
        local argument;
        begin
          argument := op(F,1);
          if type(argument) = "_plus" then
            map(argument, f)
          else
            F
          end
        end
      end:

>> f(x + 3*y) - f(3*y) = simplify(f(x + 3*y) - f(3*y))

      f(x + 3 y) - f(3 y) = f(x)

```

Es wäre möglich, das "simplify"-Attribut von f weiter zu verfeinern, so dass auch $f(3*y)$ als $3*f(y)$ geschrieben wird. Auf der anderen Seite ist es natürlich Geschmackssache, ob man $f(x) + f(y)$ tatsächlich für einfacher hält als $f(x + y)$. Die umgekehrte Regel (Umschreiben von $f(x) + f(y)$ als $f(x + y)$) ist nicht kontextfrei und kann daher nicht in einem "simplify"-Attribut implementiert werden.

Änderungen:

☞ Keine Änderungen.

sin, cos, tan, csc, sec, cot – die trigonometrischen Funktionen

$\sin(x)$ stellt die Sinus-Funktion dar.

$\cos(x)$ stellt die Kosinus-Funktion dar.

$\tan(x)$ stellt die Tangens-Funktion $\sin(x)/\cos(x)$ dar.

$\csc(x)$ stellt die Kosekans-Funktion $1/\sin(x)$ dar.

$\sec(x)$ stellt die Sekans-Funktion $1/\cos(x)$ dar.

$\cot(x)$ stellt die Kotangens-Funktion $\cos(x)/\sin(x)$ dar.

Aufruf(e):

☞ $\sin(x)$

☞ $\cos(x)$

☞ $\tan(x)$

↻ `csc(x)`

↻ `sec(x)`

↻ `cot(x)`

Parameter:

`x` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `x`

Seiteneffekte: Für Gleitpunktargumente reagieren die Funktionen auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `arcsin`, `arccos`, `arctan`, `arccsc`, `arcsec`, `arccot`

Details:

- ↻ Die Argumente sind im Bogenmaß anzugeben. Beispielsweise ist ein Winkel von 180° als π einzugeben.
- ↻ Alle trigonometrischen Funktionen sind für komplexe Argumente definiert.
- ↻ Für Gleitpunktargumente werden Gleitpunktwerte berechnet. Für die meisten exakten Argumente werden unevaluierte Funktionsaufrufe zurückgeliefert.
- ↻ Verschiebungen um ganzzahlige Vielfache von π werden aus additiven Argumenten entfernt. Weiterhin führen rationale Vielfache von π zu vereinfachten Ergebnissen: Symmetriebeziehungen werden benutzt, um das Argument auf das Standardintervall $[0, \pi/2)$ zu verschieben. Explizite Ausdrücke werden für die folgenden Argumente zurückgeliefert:

$$0, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}, \frac{2\pi}{5}, \frac{\pi}{6}, \frac{\pi}{8}, \frac{3\pi}{8}, \frac{\pi}{10}, \frac{3\pi}{10}, \frac{\pi}{12}, \frac{5\pi}{12}.$$

Siehe Beispiel ??.

- ↻ Das Ergebnis wird durch Hyperbel-Funktionen ausgedrückt, wenn das Argument ein rationales Vielfaches von \mathbb{I} ist. Siehe Beispiel ??.
- ↻ Die Funktionen `expand` und `combine` benutzen die trigonometrischen Additionstheoreme. Siehe Beispiel ??.

- ⌘ Die trigonometrischen Funktionen reagieren nicht unmittelbar auf mittels assume gesetzte Eigenschaften von Bezeichnern. Diese können jedoch über simplify berücksichtigt werden. Siehe Beispiel ??.
- ⌘ $\sec(x)$ und $\csc(x)$ werden direkt in $1/\cos(x)$ bzw. $1/\sin(x)$ umgewandelt. Mittels expand oder rewrite können Ausdrücke in tan und cot durch sin und cos ausgedrückt werden. Siehe Beispiel ??.
- ⌘ Die inversen Funktionen sind durch arcsin, arccos, arctan, arc-csc, arcsec bzw. arccot implementiert. Siehe Beispiel ??.
- ⌘ Die float-Attribute sind Kernfunktionen, d. h., die Gleitpunktauswertung ist schnell.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)
```

```

0, cos(1), tan(5 + I), 1, -----, 21/2 + 1
                        /  PI \
                      cos| -- |
                        \ 11  /

```

```
>> sin(-x), cos(x + PI), tan(x^2 - 4)
```

```

                2
            -sin(x), -cos(x), tan(x  - 4)

```

Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)

-0.7693905459, 946.4239673 + 770.3351731 I, 1.0e20
```

Beispiel 2. Einige spezielle Werte sind implementiert:

```
>> sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)
```

```

    1/2      1/2
    5        5      1/2      1/2
---- - 1/4, ---- - 1/4, 2  + 1, - 3  - 2
    4        4

```

Verschiebungen um ganzzahlige Vielfache von π werden aus den Argumenten eliminiert:

```
>> sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)
```

$$\sin(x), -\cos(3), \tan(x), \cot(2)$$

Alle Argumente, die rationale Vielfache von π sind, werden auf das Intervall $[0, \pi/2)$ transformiert:

```
>> sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)
```

$$\sin\left|\frac{4}{7}\pi\right|, \cos\left|\frac{-20}{9}\pi\right|, \tan\left|\frac{123}{11}\pi\right|, -\cot\left|\frac{-1}{13}\pi\right|$$

Beispiel 3. Ist das Argument ein rationales Vielfaches von I , so wird das Ergebnis durch Hyperbel-Funktionen ausgedrückt:

```
>> sin(5*I), cos(5/4*I), tan(-3*I)
```

$$I \sinh(5), \cosh(5/4), -I \tanh(3)$$

Für andere komplexe Argumente kann `expand` zur Umformulierung verwendet werden:

```
>> sin(5*I + 2*PI/3), cos(5/4*I - PI/4), tan(-3*I + PI/2)
```

$$\sin\left|\frac{2}{3}\pi + 5I\right|, \cos\left|\frac{5}{4}I - \frac{1}{4}\pi\right|, \tan\left|\frac{-3}{2}I + \frac{1}{2}\pi\right|$$

```
>> expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
    expand(tan(-3*I + PI/2))
```

$$\frac{1}{2} \cosh(5) - \frac{1}{2} I \sinh(5), \frac{1}{2} \cosh(5/4) - \frac{1}{2} I \sinh(5/4), -\frac{I \cosh(3)}{\sinh(3)}$$

Beispiel 4. Die `expand`-Funktion benutzt die Additionstheoreme:

```
>> expand(sin(x + PI/2)), expand(cos(x + y))
```

$$\cos(x), \cos(x) \cos(y) - \sin(x) \sin(y)$$

Die `combine`-Funktion benutzt diese Regeln in umgekehrter Richtung. Sie versucht, Produkte trigonometrischer Funktionen umzuformen:


```
>> combine(sin(x)*sin(y), sincos)
```

$$\frac{\cos(x - y)}{2} - \frac{\cos(x + y)}{2}$$

Die trigonometrischen Funktionen reagieren nicht unmittelbar auf mittels `assume` gesetzte Eigenschaften von Bezeichnern:

```
>> assume(n, Type::Integer): sin(n*PI), cos(n*PI)
```

$$\sin(n \text{ PI}), \cos(n \text{ PI})$$

Solche Eigenschaften werden jedoch von `simplify` berücksichtigt:

```
>> simplify(sin(n*PI)), simplify(cos(n*PI))
```

$$0, (-1)^n$$

```
>> assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))
```

$$\sin(x + n \text{ PI}), -\sin(x)$$

```
>> y := cos(x - n*PI) + cos(n*PI - x): y , simplify(y)
```

$$\cos(x - n \text{ PI}) + \cos(n \text{ PI} - x), -2 \cos(x)$$

```
>> delete n, y:
```

Beispiel 5. Es gibt zahlreiche Zusammenhänge zwischen den trigonometrischen Funktionen:

```
>> csc(x), sec(x)
```

$$\frac{1}{\sin(x)}, \frac{1}{\cos(x)}$$

Die `expand`-Funktion schreibt alle trigonometrischen Funktionen mittels `sin` und `cos` um:

```
>> expand(tan(x)), expand(cot(x))
```

$$\frac{\sin(x)}{\cos(x)}, \frac{\cos(x)}{\sin(x)}$$

Mittels `rewrite` kann gezielt umgeformt werden:

```
>> rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

$$\frac{\sin(x) (\cos(2x) + I \sin(2x))}{\cos(x)}, \frac{\frac{\sqrt{x} \sqrt{\cot^2 x - 1}}{\sqrt{2}}}{\frac{\sqrt{x} \sqrt{2}}{\cot^2 x - 1} + 1}$$

Beispiel 6. Die inversen Funktionen sind durch arcsin, arccos etc. implementiert:

```
>> sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))
```

$$x, (1 - x^2)^{1/2}, \frac{1}{(x^2 + 1)^{1/2}}$$

Man beachte, dass arcsin(sin(x)) nicht notwendigerweise x zurückliefert, da der Realteil der von arcsin gelieferten Werte im Intervall $[-\pi/2, \pi/2]$ liegt:

```
>> arcsin(sin(3)), arcsin(sin(1.6 + I))
PI - 3, 1.541592654 - 1.0 I
```

Beispiel 7. Systemfunktionen wie z.B. diff, float, limit oder series verarbeiten die trigonometrischen Funktionen:

```
>> diff(sin(x^2), x), float(sin(3)*cot(5 + I))
```

$$2x \cos(x^2), -0.01668502608 - 0.1112351327 I$$

```
>> limit(x*sin(x)/tan(x^2), x = 0)
```

$$1$$

```
>> series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0, 10)
```

$$\frac{1}{30} + \frac{x^2}{756} + O(x^3)$$

Änderungen:

- ⌘ Weitere spezielle Werte und Vereinfachungen wurden implementiert.
-

\sinh , \cosh , \tanh , csch , sech , coth – die Hyperbel-Funktionen

$\sinh(x)$ stellt die Sinus-Hyperbolicus-Funktion dar.

$\cosh(x)$ stellt die Kosinus-Hyperbolicus-Funktion dar.

$\tanh(x)$ stellt die Tangens-Hyperbolicus-Funktion $\sinh(x)/\cosh(x)$ dar.

$\operatorname{csch}(x)$ stellt die Kosekans-Hyperbolicus-Funktion $1/\sinh(x)$ dar.

$\operatorname{sech}(x)$ stellt die Sekans-Hyperbolicus-Funktion $1/\cosh(x)$ dar.

$\operatorname{coth}(x)$ stellt die Kotangens-Hyperbolicus-Funktion $\cosh(x)/\sinh(x)$ dar.

Aufruf(e):

- ⌘ $\sinh(x)$
- ⌘ $\cosh(x)$
- ⌘ $\tanh(x)$
- ⌘ $\operatorname{csch}(x)$
- ⌘ $\operatorname{sech}(x)$
- ⌘ $\operatorname{coth}(x)$

Parameter:

x — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: x

Seiteneffekte: Für Gleitpunktargumente reagieren die Funktionen auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `arcsinh`, `arccosh`, `arctanh`, `arccsch`, `arcsech`, `arccoth`

Details:

- ⌘ Diese Funktionen sind für komplexe Argumente definiert.

- ☞ Für Gleitpunktargumente werden Gleitpunktwerte berechnet. Für die meisten exakten Argumente werden unevaluierte Funktionsaufrufe zurückgeliefert.
- ☞ Argumente, die rationale Vielfache von $i\pi/2$ sind, führen zu vereinfachten Ergebnissen. Hat das Argument einen negativen reellen Faktor vom Typ `Type::Real`, so werden Symmetriebeziehungen benutzt, um diesen Faktor positiv zu machen. Siehe Beispiel ??.
- ☞ Die speziellen Werte
 $\sinh(0) = 0, \sinh(\pm\infty) = \pm\infty,$
 $\cosh(0) = 1, \cosh(\pm\infty) = \infty,$
 $\tanh(0) = 0, \tanh(\pm\infty) = \pm 1,$
 $\coth(\pm\infty) = \pm 1$
sind implementiert.
- ☞ Die Funktionen `expand` und `combine` benutzen die hyperbolischen Additionstheoreme. Siehe Beispiel ??.
- ☞ `sech(x)` und `csch(x)` werden direkt in $1/\cosh(x)$ bzw. $1/\sinh(x)$ umgewandelt. Mittels `expand` oder `rewrite` können Ausdrücke in `tan` und `cot` durch `sin` und `cos` ausgedrückt werden. Siehe Beispiel ??.
- ☞ Die inversen Funktionen sind durch `arcsin`, `arccos`, `arctan`, `arc-csc`, `arcsec` bzw. `arccot` implementiert. Siehe Beispiel ??.
- ☞ Die float-Attribute sind Kernfunktionen, d. h., die Gleitpunktauswertung ist schnell.

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)
0, cosh(1), tanh(5 + I),  $\frac{1}{\sinh(\pi)}$ ,  $\frac{1}{\cosh(1/11)}$ , coth(8)
>> sinh(x), cosh(x + I*PI), tan(x^2 - 4)
sinh(x), cosh(x + I PI),  $\tan(x^2 - 4)$ 
```

Für Gleitpunktargumente werden numerische Werte berechnet:

```
>> sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
1.953930316e53, 7.295585032 + 135.0143985 I, 1.0e20
```

Beispiel 2. Für Argumente, die ganzzahlige Vielfache von $i\pi/2$ sind, werden vereinfachte Ergebnisse zurückgeliefert:

```
>> sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
      coth(-17/2*I*PI)
```

$I, 1, 0, 0$

Reelle numerische Faktoren im Argument werden über Symmetriebeziehungen umgeschrieben:

```
>> sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)
```

```

      / 3 x \      / x PI \      / 12 x y PI \
-sinh(5), cosh| --- |, - tanh| ---- |, - coth| -----
- |          \ 2 /          \ 12 /          \ 17      /
```

Beispiel 3. Die expand-Funktion benutzt die Additionstheoreme:

```
>> expand(sinh(x + PI*I)), expand(cosh(x + y))
```

$-\sinh(x), \cosh(x) \cosh(y) + \sinh(x) \sinh(y)$

Die combine-Funktion benutzt diese Regeln in umgekehrter Richtung. Sie versucht, Produkte von Hyperbel-Funktionen umzuformen:

```
>> combine(sinh(x)*sinh(y), sinhcosh)
```

$$\frac{\cosh(x + y)}{2} - \frac{\cosh(x - y)}{2}$$

Beispiel 4. Es gibt zahlreiche Zusammenhänge zwischen den Hyperbel-Funktionen:

```
>> csch(x), sech(x)
```

$$\frac{1}{\sinh(x)}, \frac{1}{\cosh(x)}$$

Die expand-Funktion schreibt alle Funktionen mittels sinh und cosh um:

```
>> expand(tanh(x)), expand(coth(x))
```

$$\frac{\sinh(x)}{\cosh(x)}, \frac{\cosh(x)}{\sinh(x)}$$

Mittels rewrite kann gezielt umgeformt werden:

```
>> rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)
```

$$\frac{\sinh(x) (\cosh(2x) + \sinh(2x))}{\cosh(x)}, \frac{\frac{\sqrt{x}}{2} \tanh\left|\frac{x}{2}\right|}{1 - \tanh\left|\frac{x}{2}\right|}$$

```
>> rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)
```

$$\frac{(\exp(y)^2 + 1) \left| \frac{\exp(x)}{\sqrt{2}} - \frac{\exp(-x)}{\sqrt{2}} \right| \coth\left|\frac{x}{2}\right| + 1}{\exp(y)^2 - 1 \coth\left|\frac{x}{2}\right| - 1}$$

Beispiel 5. Die inversen Funktionen sind durch arcsinh, arccosh etc. implementiert:

```
>> sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))
```

$$x, (x^2 - 1)^{1/2}, \frac{1}{(x+1)^{1/2} (1-x)^{1/2}}$$

Man beachte, dass arcsinh(sinh(x)) nicht notwendigerweise x zurückliefert, da der Imaginärteil der von arcsinh gelieferten Werte im Intervall $[-\pi/2, \pi/2]$ liegt:

```
>> arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))
```

$$3, 1.6 - 0.5309649149 I$$

Beispiel 6. Systemfunktionen wie z.B. `diff`, `float`, `limit` oder `series` verarbeiten die Hyperbel-Funktionen:

```
>> diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))

      2
      2 x cosh(x ), 10.01749636 - 0.0008270853591 I

>> limit(x*sinh(x)/tanh(x^2), x = 0)

      1

>> series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0, 10)

      2
      29 x      3
- 1/30 + ---- + O(x )
      756
```

Änderungen:

- ⌘ Einige spezielle Werte wurden implementiert. Die Konvertierung mittels `rewrite` wurde vervollständigt. Die Gleitpunktauswertung von `tanh` und `coth` wurde für große Argument gegen numerischen Über-/Unterlauf gesichert.

slot – Methode oder Eintrag eines Domains oder einer Funktionsumgebung

`slot(d, "n")` liefert den Wert des Slots "n" des Objektes d.

`slot(d, "n", v)` erzeugt oder ändert den Slot "n". Ihm wird der Wert v zugewiesen.

Aufruf(e):

```
⌘ d :: n
⌘ slot(d, "n")
⌘ d :: n := v
⌘ slot(d, "n", v)
⌘ object :: dom
⌘ slot(object, "dom")
```

Parameter:

- `d` — ein Domain oder eine Funktionsumgebung
- `n` — der Name des Slots: ein Bezeichner
- `v` — der neue Wert des Slots: ein beliebiges MuPAD-Objekt
- `object` — ein beliebiges MuPAD-Objekt

Rückgabewert: `slot(d, "n")` liefert den Wert des Slots, `slot(d, "n", v)` liefert das Objekt `d` mit dem neuen oder geänderten Slot, `slot(object, "dom")` liefert den Domain-Typ des Objekts.

Überladbar durch: `d`

Weitere Dokumentation: Abschnitt 5.7 des Dokumentes "From MuPAD 1.4 to MuPAD 2.0".

Verwandte Funktionen: `DOM_DOMAIN`, `DOM_FUNC_ENV`, `domain`, `funcenv`, `newDomain`

Details:

- ☞ `slot` wird zum Definieren von Methoden und Einträgen von benutzerdefinierte Datentypen (Domains) und zum Definieren von Attributen von Funktionsumgebungen benutzt. Die so definierten Methoden, Einträge und Attribute werden *Slots* genannt. Sie ermöglichen das Überladen von Systemfunktionen durch benutzerdefinierte Domains und Funktionsumgebungen. Siehe den Abschnitt „Hintergründe“ für weitere Details.
 - ☞ Jedes MuPAD-Objekt besitzt den speziellen Slot `"dom"`. Er enthält das Domain, zu welchem das Objekt gehört: `slot(object, "dom")` ist äquivalent zu `domtype(object)`. Der Wert dieses speziellen Slots kann nicht verändert werden. Siehe Beispiel ??.
 - ☞ Neben dem speziellen Slot `"dom"` können nur Domains und Funktionsumgebungen weitere Slots besitzen.
Der Aufruf `slot(d, "n")` ist äquivalent zu `d := n`. Er liefert den Wert des Slots.
Der Aufruf `slot(d, "n", v)` fügt einen neuen Slot `"n"` mit dem Wert `v` zu `d` hinzu bzw. ändert einen existierenden Slot-Wert ab und liefert das modifizierte Objekt `d`.
 - ☞ Ist `d` eine *Funktionsumgebung*, so liefert der Aufruf `slot(d, n, v)` eine geänderte *Kopie* von `d`. Die Funktionsumgebung `d` selbst wird dabei nicht verändert! Erst die Zuweisung `d := slot(d, "n", v)` verändert `d`. Siehe Beispiel ??.
- Ist `d` jedoch ein Domain, so ändert der Aufruf `slot(d, n, v)` das Domain `d` selbst als Seiteneffekt! Dies ist der so genannte „Referenzeffekt“ bei Domains. Siehe Beispiel ??.

- ☞ Wird auf einen nicht existierenden Slot lesend zugegriffen, so wird der Rückgabewert `FAIL` geliefert. Siehe Beispiel ??.
- ☞ Mit dem `::`-Operator kann auch auf Slots zugegriffen werden.
 Sofern der Ausdruck `d::n` nicht auf der linken Seite einer Zuweisung auftaucht, ist er äquivalent zu `slot(d, "n")`.
 Mit der Zuweisung `d::n := v` wird dem Slot "n" des Objektes `d` der Wert `v` zugewiesen. Diese Zuweisung ist beinahe äquivalent zu `d := slot(d, "n", v)`. Man beachte den folgenden feinen semantischen Unterschied zwischen den Zuweisungen: bei `d::n := v` wird `d` nicht vollständig evaluiert, sondern nur eine Stufe weit. Damit wird der Slot "n" dem Wert von `d` zugewiesen. Im Aufruf `slot(d, "n", v)` wird der Bezeichner `d` *vollständig evaluiert*. Siehe Beispiel ??.
- ☞ Mit `delete d::n` oder `delete slot(d, "n")` wird der Slot "n" der Funktionsumgebung oder des Domains `d` gelöscht. Siehe Beispiel ??.
 Der spezielle Slot "dom" kann nicht gelöscht werden.
- ☞ Das erste Argument von `slot` wird nicht ausgeglichen. Dies erlaubt den Zugriff auf die Slots von Ausdrucksfolgen und `null()`-Objekten. Siehe hierzu Beispiel ??.
- ☞ Für Domains existiert ein spezieller Mechanismus, um neue Slots bei Bedarf anzulegen. Wird auf einen nicht existierenden Slot zugegriffen, so wird die Domain-Methode "make_slot" aufgerufen, um diesen Slot zu erzeugen. Existiert diese Methode nicht, dann wird der Rückgabewert `FAIL` geliefert. Siehe Beispiel ??.
- ☞ `slot` ist eine Funktion des Systemkerns.

Beispiel 1. Jedes Objekt besitzt den Slot "dom":

```
>> slot(x, "dom") = domtype(x),
    slot(45, "dom") = domtype(45),
    slot(sin, "dom") = domtype(sin)

DOM_IDENT = DOM_IDENT, DOM_INT = DOM_INT,

DOM_FUNC_ENV = DOM_FUNC_ENV
```

Beispiel 2. Es wird der "float"-Slot der Funktionsumgebung `sin` ausgelesen, die die Sinus-Funktion implementiert. Der "float"-Slot ist wiederum eine Funktionsumgebung, die wie jede MuPAD-Funktion aufgerufen werden kann. Man beachte jedoch die unterschiedliche Funktionalität: der "float"-Slot versucht stets, eine Gleitpunktapproximation zu berechnen:

```
>> s := slot(sin, "float"): s(1) , sin(1)
                                0.8414709848, sin(1)
```

Der folgende Aufruf weist dem Bezeichner `s` die die vollständige Funktionsumgebung von `sin` zu, wobei das "float"-Attribut abgeändert wird: Der `slot`-Aufruf hat dabei keinen Einfluss auf die ursprüngliche `sin`-Funktion:

```
>> s := slot(sin, "float", x -> float(x - x^3/3!)):
    s(PI/3) = sin(PI/3), s::float(1) <> sin::float(1)

          1/2      1/2
          3        3
    ---- = ----, 0.8333333333 <> 0.8414709848
          2        2
```

```
>> delete s:
```

Beispiel 3. Wird die `slot` Funktion zum Ändern von Slots eines Domains benutzt, so wird hierdurch das Domain als Seiteneffekt verändert. Bei Funktionsumgebungen ist dies nicht der Fall (siehe Beispiel ??):

```
>> old_one := slot(Dom::Float, "one")
                                1.0

>> newDomFloat := slot(Dom::Float, "one", 1):
    slot(newDomFloat, "one"), slot(Dom::Float, "one")
                                1, 1
```

Der Originalzustand wird wiederhergestellt:

```
>> slot(Dom::Float, "one", old_one): slot(Dom::Float, "one")
                                1.0

>> delete old_one, newDomFloat:
```

Beispiel 4. Die Funktionsumgebung `sin` besitzt keinen "sign" slot. Beim Zugriff auf diesen Slot wird daher `FAIL` als Ergebnis geliefert:

```
>> slot(sin, "sign"), sin::sign
                                FAIL, FAIL
```

Beispiel 5. Es wird eine Funktionsumgebung, die den Logarithmus zur Basis 10 berechnet definiert:

```
>> log10 := funcenv(x -> log(10, x)):
```

Soll die Funktion `info` eine Kurzinformation zur Funktion `log10` ausgeben, so muss der noch nicht existierende Slot "info" definiert werden. Da `slot` für Funktionsumgebungen eine Kopie des eigentlichen Objektes zurückgibt, muss das Ergebnis des `slot`-Aufrufs dem Bezeichner `log10` zugewiesen werden:

```
>> log10 := slot(log10, "info",  
                  "log10 -- Logarithmus zur Basis 10"):  
      info(log10)  
  
log10 -- Logarithmus zur Basis 10
```

Slots werden mit dem `delete` Kommando gelöscht:

```
>> delete log10::info: info(log10)  
  
Sorry, no information available.
```

Es ist nicht möglich, den speziellen Slot "dom" zu löschen:

```
>> delete log10::dom  
  
Error: Illegal argument [delete]  
  
>> delete log10:
```

Beispiel 6. Wird auf einen Slot auf der linken Seite einer Zuweisung mit dem `::`-Operator zugegriffen, so ergibt sich ein feiner Unterschied zu dem Zugriff über die `slot` Funktion. Der folgende Aufruf fügt dem Datentyp `DOM_INT` der ganzen Zahlen einen Slot "xyz" hinzu:

```
>> delete b: d := b: b := DOM_INT: slot(d, "xyz", 42):
```

Der Slot "xyz" von `DOM_INT` wurde abgeändert, da `d` vollständig zum Domain `DOM_INT` evaluiert wurde, so dass der Slot `DOM_INT::xyz` mit dem Wert 42 erzeugt wurde:

```
>> slot(d, "xyz") , slot(DOM_INT, "xyz")  
  
42, 42
```

Die analoge Aufrufsequenz mit `::` verhält sich anders: `d` wird nur zu seinem Wert evaluiert, dieser ist der Bezeichner `b`. Da kein Slot `b::xyz` existiert, wird ein Fehler ausgelöst.

```
>> delete b: d := b: b := DOM_INT: d::xyz := 42
```

```
Error: Unknown slot "d::xyz" [slot]
```

```
>> delete b, d:
```

Beispiel 7. Das erste Argument von `slot` wird nicht ausgeglichen. Dies erlaubt den Zugriff auf die Slots von Ausdrucksfolgen und `null()`-Objekten.

```
>> slot((a, b), "dom"), slot(null(), "dom")
```

```
DOM_EXPR, DOM_NULL
```

Beispiel 8. Es folgt ein Beispiel für die Anwendung der Funktion `make_slot`. Das Element `undefined` des Domains `stdlib::Undefined` repräsentiert einen undefinierten Wert. Jede Funktion f sollte $f(\text{undefined}) = \text{undefined}$ liefern. In der Definition von `stdlib::Undefined` findet sich in etwa folgender Code:

```
>> undef := newDomain("stdlib::Undefined"):
    undefined := new(undef):
    undef::func_call := proc() begin undefined end_proc;
    undef::make_slot := undef::func_call:
```

Bei einer Funktion f , die durch ihr erstes Argument überladbar ist, überprüft der Funktionsaufruf $f(\text{undefined})$, ob der Slot `undef::f` existiert. Ist dies nicht der Fall, so wird dieser Slot automatisch von der Funktion `undef::make_slot` erzeugt, die als Rückgabewert den gewünschten Wert des neuen Slots liefert. Damit liefert der Funktionsaufruf $f(\text{undefined})$ als Ergebnis den Wert `undefined`.

Beispiel 9. Dieses Beispiel ist sehr fortgeschritten und technisch. Es zeigt die Möglichkeiten auf, die sich durch das Überladen der `slot` Funktion ergeben. Hierdurch kann der Zugriff auf Slots auch für andere MuPAD-Objekte als Domains (`DOM_DOMAIN`) und Funktionsumgebungen (`DOM_FUNC_ENV`) realisiert werden. Das folgende Beispiel definiert z. B. die beiden Slots "numer" und "denom" für rationalen Zahlen. In MuPAD gehören rationale Zahlen dem Kerntyp `DOM_RAT` an, der keine Slots "numer" und "denom" besitzt:

```
>> domtype(3/4)
```

```
DOM_RAT
```

```
>> slot(3/4, "numer");
```

```
Error: Unknown slot "(3/4)::numer" [slot]
```

Es ist aber möglich, DOM_RAT zu ändern. Dafür müssen wir temporär den protect-Status von DOM_RAT aufheben:

```
>> unprotect(DOM_RAT):
    _assign(DOM_RAT::slot,
      proc(r : DOM_RAT, n : DOM_STRING, v=null(): DOM_INT)
        local i : DOM_INT;
      begin
        i := contains(["numer", "denom"], n);
        if i = 0 then
          error("Unknown slot \"%.expr2text(r).\"::\".n.\"")
        end;
        if args(0) = 3 then
          subsop(r, i = v)
        else
          op(r, i)
        end
      end_proc):
```

Nach dieser Zuweisung stehen uns neue Slots zur Verfügung, mit denen wir auf die Operanden rationaler Zahlen (Zähler und Nenner) zugreifen können:

```
>> slot(3/4, "numer"), (3/4)::numer,
    slot(3/4, "denom"), (3/4)::denom

3, 3, 4, 4

>> a := 3/4: slot(a, "numer", 7)

7/4

>> a::numer := 11: a

11/4
```

Die obigen Änderungen an DOM_RAT werden rückgängig gemacht:

```
>> delete DOM_RAT::slot, a: protect(DOM_RAT, Error):
```

Hintergründe:

☞ Ist x ein Domain-Element und die Bibliotheksfunktion f soll für den benutzerdefinierten Datentyp überladen werden, so geschieht dies wie folgt. Es wird getestet, ob das Domain von x eine Domain-Methode f besitzt. Trifft dies zu, dann wird diese Domain-Methode aufgerufen. Technisch wird dies wie folgt realisiert:

```

f:= proc(x)
begin
    // prüfe, ob f durch x überladen wird
    if x::dom::f <> FAIL then
        // benutze die Methode des Domains von x
        return(x::dom::f(args()))
    else
        // führe den Code für f aus
    endif
end_proc:

```

☞ Durch Überladen der Funktion `slot` können Slots auch für andere Objekte als Domains und Funktionsumgebungen realisiert werden. Siehe Beispiel ??.

☞ Im Prinzip darf der Name `n` eines Slots ein beliebiges MuPAD-Objekt sein.

Man beachte jedoch, dass mit dem `::`-Operator nicht auf Slots zugegriffen werden kann, die mittels `slot(d, n, v)` definiert wurden und deren Name `n` keine Zeichenkette ist.

☞ Der `::`-Operator kann auch mit Zeichenketten benutzt werden: die Aufrufe `d::"n"` und `d::n` sind äquivalent.

Änderungen:

☞ `slot` ist eine neue Funktion.

☞ `slot` ersetzt und vereint die Funktionen `domattr` and `funcattr` älterer MuPAD-Versionen. Die Domain-Methode `"make_slot"` hat die gleiche Funktionalität wie die frühere Domain-Methode `domattr`.

`solve` – Lösen von Gleichungen und Ungleichungen

`solve(eq, x)` liefert die Menge aller komplexen Lösungen der Gleichung oder Ungleichung `eq` bezüglich `x`.

`solve(system, vars)` löst ein System von Gleichungen nach den Variablen `vars` auf.

`solve(eq, vars)` bewirkt dasselbe wie `solve([eq], vars)`.

`solve(system, x)` bewirkt dasselbe wie `solve(system, [x])`.

`solve(eq)` ohne zweites Argument bewirkt dasselbe wie `solve(eq, S)`, wo `S` die Menge aller Unbestimmten in `eq` ist. Dasselbe gilt für `solve(system)`.

Aufruf(e):

```
# solve(eq, x <, options>)  
# solve(eq, vars <, options>)  
# solve(eq <, options>)  
# solve(system, x <, options>)  
# solve(system, vars <, options>)  
# solve(system <, options>)  
# solve(ODE)  
# solve(REC)
```

Parameter:

eq	— eine einzelne Gleichung oder eine Ungleichung vom Typ " <code>_equal</code> ", " <code>_less</code> ", " <code>_leequal</code> ", oder " <code>_unequal</code> ". Auch ein arithmetischer Ausdruck wird akzeptiert und als Gleichung mit verschwindender rechter Seite interpretiert.
x	— die Unbestimmte, nach der aufgelöst werden soll: ein Bezeichner oder ein indizierter Bezeichner
vars	— eine nichtleere Menge oder Liste von Unbestimmten, nach denen aufgelöst werden soll
system	— eine Menge, Liste, Tabelle oder ein Array von Gleichungen bzw. arithmetischen Ausdrücken. Letztere werden als Gleichungen mit verschwindender rechter Seite aufgefasst.
ODE	— eine gewöhnliche Differentialgleichung: ein Objekt vom Typ <code>ode</code> .
REC	— eine Rekurrenzgleichung: ein Objekt vom Typ <code>rec</code> .

Optionen:

- | | |
|------------------------------------|---|
| <i>MaxDegree</i> = <i>n</i> | — verwendet keine expliziten Formeln, um Polynomgleichungen vom Grad größer als <i>n</i> durch Radikale zu lösen. Die Voreinstellung der positiven ganzen Zahl <i>n</i> ist 2. |
| <i>BackSubstitution</i> = <i>b</i> | — Legt fest, ob beim Lösen algebraischer Systeme eine Rücksubstitution durchgeführt wird oder nicht; <i>b</i> muss TRUE oder FALSE sein. Die Voreinstellung ist TRUE. |
| <i>Multiple</i> | — liefert Lösungsmengen als Objekte vom Typ <code>Dom::Multiset</code> , die zu jeder Nullstelle eines Polynoms auch die Vielfachheit mit angeben. Diese Option ist nur für Polynomgleichungen und polynomiale Ausdrücke zulässig. |
| <i>PrincipalValue</i> | — es wird nur eine Lösung als einelementige Menge zurückgeliefert |
| <i>Domain</i> = <i>d</i> | — Löst die Gleichung über dem Domain <i>d</i> . <i>d</i> muss für eine Teilmenge der komplexen Zahlen (z. B. die reellen oder ganzen Zahlen) stehen oder ein Domain sein, über dem Polynome faktorisiert werden können (z. B. ein endlicher Körper). Im letzteren Fall ist die Option nur für Polynomgleichungen zulässig. Fehlt diese Option, so werden alle Lösungen im Bereich der komplexen Zahlen zurückgeliefert. |
| <i>IgnoreSpecialCases</i> | — Falls eine Fallunterscheidung nötig wird, behandle jede Bedingung als falsch, die impliziert, dass ein Parameter der Gleichung Element einer festen endlichen Menge ist. |

Rückgabewert: `solve(eq, x)` liefert stets ein Objekt, das mathematisch als Menge anzusehen ist. Ein Aufruf von `solve` liefert eine Menge von Listen zurück, wenn eines der Argumente eine Menge oder Liste ist, oder wenn das erste Argument ein Array oder eine Tabelle ist, oder wenn keine Variablen angegeben werden, nach denen aufgelöst werden soll. Jede dieser Listen besteht aus Gleichungen, auf deren linker Seite eine Variable steht, nach der zu lösen war. Weiterhin kann `solve` auch einen Ausdruck der Form `x in S` zurückliefern; hierbei ist `x` eine Variable, nach der zu lösen war, und `S` irgendeine Menge.

Überladbar durch: `eq`

Seiteneffekte: `solve` reagiert auf Eigenschaften von Bezeichnern.

Verwandte Funktionen: `linsolve`, `numeric::linsolve`,
`numeric::solve`, `RootOf`, `solvers`

Details:

- ⌘ Die `solve`-Funktion bietet eine einheitliche Schnittstelle zu einer Vielzahl spezialisierter Lösungsalgorithmen. Mittels `?solvers` erhält man einen Überblick über alle zur Verfügung stehenden Löser.
- ⌘ Werden keine Unbestimmten übergeben, so wird automatisch nach allen in `eq` bzw. `system` auftretenden Unbestimmten aufgelöst. Unbestimmte sind alle Bezeichner (ausgenommen mathematische Konstanten wie `PI`, `EULER` etc.) sowie alle indizierten Bezeichner. Unbestimmte, die als Funktionsnamen oder Indizes vorkommen, werden nicht berücksichtigt. Siehe Beispiel ??.
- ⌘ Ist das zweite Argument eine Liste von Unbestimmten, nach denen aufgelöst werden soll, so werden die Komponenten der Lösungsvektoren in der Reihenfolge zurückgegeben, in der sie in der Liste angegeben werden. Werden die Unbestimmten als Menge angegeben, so benutzt `solve` eine intern gewählte Anordnung der Unbestimmten.
- ⌘ Die von `solve` zurückgelieferten Mengen können viele verschiedene Typen haben (siehe den Abschnitt „Hintergründe“ dieser Hilfeseite). Sie bieten aber zur Weiterverarbeitung gemeinsame Schnittstellenfunktionen. Hierzu gehören die mengentheoretischen Operationen `intersect`, `union`, `minus`. Siehe Beispiel ??. Weiterhin sind die arithmetischen Operationen `+`, `*`, `^` etc. auf Lösungsmengen anwendbar (sie sind punktweise definiert). Die Funktion `solveLib::getElement` dient zur Auswahl einzelner Elemente der von `solve` gelieferten Lösungsmenge. Mit der Funktion `solveLib::isFinite` kann überprüft werden, ob sie eine endliche Menge darstellt.
- ⌘ `solve(eq, x)` liefert nur diejenigen Lösungen zurück, die den Eigenschaften von `x` nicht widersprechen. Siehe Beispiel ??. Beim Lösen eines Systems mit mehreren Unbestimmten werden die die Eigenschaften der Variablen, nach denen gelöst wird, nur in manchen Fällen berücksichtigt.
- ⌘ Da die Funktion `solve` überladbar ist, kann sie vom Benutzer auf eigene Typen von Gleichungen erweitert werden. `solve` ist bereits für die Domains `ode` und `rec` überladen und bietet damit eine Schnittstelle zur Lösung von Differential- und Rekurrenzgleichungen. Beispiele dazu sind auf den Hilfeseiten von `ode` und `rec` zu finden.
- ⌘ Der Aufruf `float(hold(solve)(Gleichungen, Unbestimmte <, Optionen>))` liefert eine numerische Lösung. Er ist äquivalent zum direkten Aufruf des numerischen Löfers `numeric::solve(Gleichungen,`

Unbestimmte $\langle, \text{Optionen} \rangle$). Die Hilfeseite von `numeric::solve` liefert Details zu den zur Verfügung stehenden Optionen. Insbesondere können Startwerte und Suchbereiche für die numerische Suche angegeben werden. Man beachte, dass bei nichtpolynomialen Gleichungen nur eine einzelne numerische Lösung gesucht wird. Siehe Beispiel ??.

Im Gegensatz zu `solve` reagiert `numeric::solve` nicht auf mittels `assume` gesetzte Eigenschaften von Bezeichnern.



Option **<Multiple>**:

- ☞ Mit dieser Option liefert `solve` eine Menge vom Typ `Dom::Multiset` zurück.
- ☞ Der Versuch, die Nullstellen des Nullpolynoms mit dieser Option zu bestimmen, endet mit einem Fehler, da es in MuPAD keine unendlichen Mengen vom Typ `Dom::Multiset` gibt.
- ☞ Ist die Lösungsmenge vom Typ `RootOf`, so wird diese Option ignoriert.

Option **<PrincipalValue>**:

- ☞ Mit dieser Option wird nur eine einzige Lösung zurückgeliefert, auch wenn mehrere Lösungen existieren. Hat die Gleichung keine Lösung, so ist das Ergebnis die leere Menge.
- ☞ Kann kein Element der Lösungsmenge gefunden werden, so ist das Ergebnis ein symbolischer Aufruf von `solve`. Dies ist insbesondere der Fall, falls die Lösungsmenge fallweise definiert ist und kein Element existiert, das in allen Lösungsästen enthalten ist.
- ☞ Diese Option darf auch im Fall mehrerer Unbestimmten verwendet werden, nach den aufgelöst werden soll. In diesem Fall wird eine Menge mit einer einzigen Liste (die einen Lösungsvektor darstellt) zurückgegeben.

Option **<MaxDegree = n>**:

- ☞ Diese Option schaltet die Verwendung der expliziten Cardano-Lösungsformeln für Polynome an oder ab; andere Methoden wie z. B. Faktorisierung werden immer angewendet. Der angegebene Maximalgrad `n` bezieht sich dabei auf die Faktoren des Polynoms, nicht auf das Eingabepolynom.
- ☞ Für Gleichungen von einem Grad größer als 4 existieren keine allgemeinen Lösungsformeln. Es macht daher keinen Unterschied, ob `MaxDegree` auf 4 oder auf einen höheren Wert gesetzt wird.

Option <BackSubstitution = b>:

- Ein Objekt des Typs "RootOf" wird niemals in eine Variable eingesetzt. Daher kann, selbst wenn *BackSubstitution* auf TRUE gesetzt wird, die Lösung für eine Variable gleich der Nullstellenmenge eines Polynoms sein, das von einer anderen Variablen abhängt.

Option <Domain = d>:

- Gleichungen und Systeme in mehr als einer Variablen können nicht über Domains gelöst werden.
- Zwei Arten von Domains *d* sind erlaubt: einerseits Teilmengen von \mathbb{C} ; andererseits Domains, über denen Polynome faktorisiert werden können (diese aber nur für Polynomgleichungen).
- Eine Teilmenge von \mathbb{C} kann von einem der Typen sein, die *solve* zurückliefert (siehe hierzu den Abschnitt „Hintergründe“.) Statt \mathbb{C} , \mathbb{R} , \mathbb{Q} und \mathbb{Z} dürfen auch die entsprechenden Domains des Domainspaketes `Dom::Complex`, `Dom::Real`, `Dom::Rational` und `Dom::Integer` verwendet werden.

Option <IgnoreSpecialCases>:

- Diese Option veranlasst *solve*, die folgende Heuristik anzuwenden, um die Anzahl der Zweige in fallweise definierten Objekten zu verringern: jede Gleichheit wird als falsch angenommen, falls sie nicht vom Property-Mechanismus bewiesen werden kann; z. B. gilt jeder Nenner als ungleich Null, außer es kann das Gegenteil bewiesen werden. Diese Option führt gewöhnlich zu einer deutlichen Verringerung der Anzahl fallweise definierter Objekte.

Beispiel 1. Hat eine Gleichung nur endlich viele Lösungen, so wird meist eine Menge vom Typ `DOM_SET` zurückgegeben:

```
>> solve(x^4 - 5*x^2 + 6*x = 2, x)
```

$$\left\{1, 3^{\frac{1}{2}} - 1, -3^{\frac{1}{2}} - 1\right\}$$

Beispiel 2. Die Lösungsmenge ist in manchen Fällen eine unendliche diskrete Menge:

```
>> S := solve(sin(x*PI/7) = 0, x)
```

$$\{ 7 \cdot x_4 \mid x_4 \in \mathbb{Z}_- \}$$

Um nur die Lösungen in einem bestimmten endlichen Intervall zu bekommen, schneidet man einfach die Lösungsmenge mit diesem Intervall:

```
>> S intersect Dom::Interval(-22, 22)
```

$$\{-21, -14, -7, 0, 7, 14, 21\}$$

```
>> delete S:
```

Beispiel 3. Die Lösungsmenge einer Ungleichung ist meist ein Intervall oder eine Vereinigung von Intervallen:

```
>> solve(x^2 > 5, x)
```

$$]5^{(1/2)}, \text{infinity}[\cup]-\text{infinity}, -5^{(1/2)}[$$

Beispiel 4. Alle bis auf zwei Zahlen haben nicht das Quadrat 7:

```
>> solve(x^2 <> 7, x)
```

$$\mathbb{C}_- \text{ minus } \{7^{1/2}, -7^{1/2}\}$$

Beispiel 5. Fallweise definierte Objekte treten dann auf, wenn eine Gleichung neben der Variable, nach der sie gelöst werden soll, noch weitere Parameter enthält. So ist die bekannte Lösungsformel für die quadratische Gleichung $ax^2 + bx + c = 0$ nur für $a \neq 0$ gültig; andernfalls erhält man die Gleichung $bx = -c$, die wiederum eine Fallunterscheidung ($b = 0$ oder $b \neq 0$) erfordert. Falls auch $b = 0$ gilt, ist die Gleichung $c = 0$ entweder wahr (jedes x ist Lösung) oder falsch (kein x ist Lösung).

```
>> S := solve(a*x^2 + b*x + c, x)
```


Innerhalb von Operatoren und Indizes wird nicht nach Unbestimmten gesucht. Daher ist weder f noch y eine Unbestimmte der folgenden Gleichung:

```
>> solve(f(x[y]) = 7)

solve({f(x[y]) = 7}, [x[y]])
```

Beispiel 7. Trägt die Unbekannte, nach der aufgelöst werden soll eine mathematische Eigenschaft, so werden nur die mit dieser Eigenschaft verträglichen Lösungen geliefert. Im Folgenden wird x als positiv (und damit implizit als reell) vorausgesetzt:

```
>> assume(x, Type::Positive): solve(x^4 = 1, x)

{1}
```

Ohne Eigenschaften werden alle komplexen Lösungen zurückgeliefert:

```
>> unassume(x): solve(x^4 = 1, x)

{-1, 1, -I, I}
```

Beispiel 8. Mittels der Option *Multiple* wird auch die Vielfachheit der Nullstellen eines Polynoms sichtbar. Im folgenden Beispiel ist -1 eine doppelte Nullstelle von $x^3 + 2x^2 + x$, während 0 nur eine einfache Nullstelle ist:

```
>> solve(x^3 + 2*x^2 + x, x, Multiple)

{[0, 1], [-1, 2]}
```

Beispiel 9. Wurde *BackSubstitution* auf `FALSE` gesetzt, so kann die Lösung für eine Variable y eine andere Variable x enthalten, nach der ebenfalls zu lösen war; dies jedoch nur, wenn x in der Liste der Unbestimmten hinter y auftritt.

```
>> solve({x^2 + y = 1, x - y = 2}, [y, x], BackSubstitution = FALSE)

{ --                1/2                --
{ |                13                |
{ |  y = x - 2, x = - ---- - 1/2  |,
{ --                2                --

--                1/2                -- }
|                13                | }
|  y = x - 2, x = ---- - 1/2  | }
--                2                -- }
```

```
>> solve({x^2 + y = 1, x - y = 2}, {x, y})
```

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} x = -\frac{13}{2} - \frac{1}{2}, y = -\frac{13}{2} - \frac{5}{2} \end{array} \right\}, \\ \left\{ \begin{array}{l} x = -\frac{13}{2} - \frac{1}{2}, y = -\frac{13}{2} - \frac{5}{2} \end{array} \right\} \end{array} \right.$$

Obwohl *BackSubstitution* im folgenden Beispiel eingeschaltet ist, enthält die Lösung für y die Variable x , da Variablen niemals durch Ausdrücke vom Typ "RootOf" ersetzt werden.

```
>> solve({x^2 + y = 1, x - y = 2}, [y, x], MaxDegree = 1)
```

$$\{[y = x - 2, x = \text{RootOf}(X^2 + X^2 - 3, X^2)]\}$$

Beispiel 10. MuPADs *solve* findet keine exakte symbolische Lösung der folgenden Gleichung:

```
>> solve(2^x = x^2, x)
```

$$\text{solve}(2^x - x^2 = 0, x)$$

Mittels *float* wird der numerische Gleichungslöser aufgerufen:

```
>> float(%)
```

$$\{-0.7666646959\}$$

Unter Vermeidung eines eventuell zeitaufwendigen Versuchs einer symbolischen Lösung kann der numerische Löser über das folgende Kommando direkt aufgerufen werden:

```
>> float(hold(solve)(2^x - x^2, x))
```

$$\{-0.7666646959\}$$

Für nichtpolynomiale Gleichungen liefert der numerische Gleichungslöser höchstens eine Lösung, auch wenn es mehrere gibt. Es können Suchintervalle angegeben werden, um weitere Lösungen zu finden:

```
>> float(hold(solve)(2^x - x^2, x = 0..3))
```

{2.0}

Die Anweisung `float(hold(solve)(...))` ruft intern den numerischen Gleichungslöser `numeric::solve` auf. Dieser kann auch direkt aufgerufen werden:

```
>> numeric::solve(2^x - x^2, x = 3..6)
```

{4.0}

Für polynomiale Gleichungen liefert der numerische Löser *alle* Lösungen:

```
>> solve(x^4 + x^3 = 3*x, x)
```

$\{0\} \cup \text{RootOf}(X^2 + X^3 - 3, X)$

```
>> float(%)
```

$\{0.0\} \cup \{1.17455941, -1.087279705 + 1.171312111 i, \\ -1.087279705 - 1.171312111 i\}$

```
>> eval(%)
```

$\{0.0, 1.17455941, -1.087279705 + 1.171312111 i, \\ -1.087279705 - 1.171312111 i\}$

Allgemein wird empfohlen, keine symbolischen Zwischenergebnisse zu verwenden, wenn numerische Lösungen angestrebt werden. Symbolische Teilrechnungen können zeitaufwendig sein, weiterhin kann die numerische Auswertung symbolischer Resultate numerisch instabil sein. Ein direkter Aufruf des numerischen Lösert `numeric::solve` vermeidet solche Probleme.

Hintergründe:

⌘ Die folgenden Typen von Lösungsmengen können von `solve` zurückgeliefert werden:

- endliche Mengen vom Typ `DOM_SET`;
- symbolische Aufrufe von `solve`;
- Nullstellenmengen von Polynomen (`RootOf`). Dieser Ergebnistyp tritt auf, wenn `solve` ein Polynom nicht durch Radikale auflösen kann oder darf, weil `MaxDegree` überschritten ist.
- Mengenausdrücke der Typen `"_union"`, `"_intersect"` und `"_minus"`;
- symbolische Aufrufe von `solveLib::Union`. Diese repräsentieren Vereinigungen über parametrisierte Familien von Mengen;

- die Mengen $\mathbb{C}, \mathbb{R}, \mathbb{Q}$ und \mathbb{Z} vom Typ `solveLib::BasicSet`;
- Intervalle (`Dom::Interval`);
- Bildmengen von Funktionen (`Dom::ImageSet`);
- fallweise definierte Objekte (`piecewise`), wobei jeder Zweig eine Menge von einem der zuvor genannten Typen enthält.

Änderungen:

- ⇒ Die Syntax für das Lösen über einem Domain wurde von `solve(eq, x = d)` in `solve(eq, x, Domain = d)` geändert.
- ⇒ Neue Typen von Lösungsmengen wurden implementiert.
- ⇒ `solve` reagiert nun auf Eigenschaften sowohl von Variablen, nach denen gelöst werden soll, als auch von freien Parametern.
- ⇒ `solve` führt nun Fallunterscheidungen durch und liefert gegebenenfalls Objekte vom Typ `piecewise` zurück, wenn dies notwendig ist.
- ⇒ Eine neue Option `IgnoreSpecialCases` wurde eingeführt.
- ⇒ Die Voreinstellung für `MaxDegree` wurde von 4 auf 2 geändert.
- ⇒ Die Voreinstellung für `BackSubstitution` wurde von `FALSE` auf `TRUE` geändert.
- ⇒ Die von `solve` aufgerufenen numerischen Gleichungslöser wurden neu geschrieben und verbessert.

solvers – Gleichungslöser im Überblick

Neben dem universellen `solve`-Kommando stellt MuPAD eine Reihe spezialisierter Routinen zur Lösung spezieller Typen von Gleichungen zur Verfügung. Diese speziellen Löser können jeweils nur gewisse Klassen von Problemen behandeln, sind in diesem Rahmen aber effizienter als das universelle `solve`.

Aufruf(e):

- ⇒ `detools::pdesolve(...)`
- ⇒ `linalg::matlinsolve(...)`
- ⇒ `linalg::matlinsolveLU(...)`
- ⇒ `linalg::vandermondeSolve(...)`
- ⇒ `linsolve(...)`
- ⇒ `numeric::linsolve(...)`
- ⇒ `numeric::matlinsolve(...)`

```
numeric::fsolve(..)
numeric::odesolve(..)
numeric::odesolve2(..)
numeric::polyroots(..)
numeric::polysysroots(..)
numeric::realroot(..)
numeric::realroots(..)
numeric::solve(..)
numlib::mroots(..)
numlib::lincongruence(..)
numlib::msqrts(..)
polylib::realroots(..)
solve(..)
```

Details:

Die folgenden Gleichungstypen können gelöst werden:

Gleichungstyp	verfügbare Löser
Systeme linearer Gleichungen	<code>solve</code> <code>linsolve</code> <code>linalg::matlinsolve</code> <code>linalg::matlinsolveLU</code> <code>linalg::vandermondeSolve</code> <code>numeric::linsolve</code> <code>numeric::matlinsolve</code>
univariate polynomiale Gleichungen	<code>solve</code> <code>polylib::realroots</code> <code>numeric::solve</code> <code>numeric::polyroots</code>
Systeme polynomialer Gleichungen	<code>solve</code> <code>numeric::solve</code> <code>numeric::polysysroots</code>
beliebige univariate Gleichungen	<code>solve</code> <code>numeric::solve</code> <code>numeric::realroot</code> <code>numeric::realroots</code>
Systeme beliebiger Gleichungen	<code>solve</code> <code>numeric::solve</code> <code>numeric::fsolve</code>
Ungleichungen	<code>solve</code>
Systeme gewöhnlicher Differentialgleichungen	<code>solve</code> <code>numeric::odesolve</code> <code>numeric::odesolve2</code>
Systeme von Rekurrenzgleichungen	<code>solve</code>
partielle Differentialgleichungen	<code>detools::pdesolve</code>
Kongruenzen	<code>numlib::lincongruence</code> <code>numlib::mroots</code> <code>numlib::msqrts</code>

☞ `detools::pdesolve` gestattet das Lösen partieller Differentialgleichungen. Diese erste Version des Löser ist noch nicht sehr mächtig; im wesentlichen wurde nur die Methode der Charakteristiken für quasi-lineare Gleichungen erster Ordnung implementiert.

☞ `linalg::vandermondeSolve` ist die empfohlene Routine für Gleichungssysteme, deren Koeffizienten eine Vandermonde-Matrix bilden.

☞ `linalg::matlinsolve` löst lineare Gleichungssysteme, die durch eine Koeffizientenmatrix über einem beliebigen Koeffizientenring gegeben sind.

Für Berechnungen über elementaren Koeffizientenkörpern (MuPAD-Ausdrücke, ganze oder rationale Zahlen, Gleitpunktzahlen etc.) wird empfohlen, stattdessen `numeric::matlinsolve` zu benutzen.

☞ `linalg::matlinsolveLU` löst lineare Gleichungssysteme über belie-

bigen Körpern, deren Koeffizientenmatrix in Form einer LR-Zerlegung gegeben ist.

- ⌘ `linsolve` ist die empfohlene Routine für lineare Gleichungssystemen über speziellen Koeffizientenringen. Bei elementaren Koeffizienten wie z. B. Ausdrücken, ganzen oder rationalen Zahlen, Gleitpunktzahlen etc. wird empfohlen, stattdessen `numeric::linsolve` zu benutzen.
- ⌘ `numeric::linsolve` ist ein schneller numerischer Löser für linearer Gleichungen. Diese Routine kann auch exakte symbolische Ergebnisse berechnen, wenn die Koeffizienten MuPAD-Ausdrücke sind. Sollen Lösungen über nicht-elementaren Koeffizientenringen berechnet werden, so ist `linsolve` zu benutzen.
- ⌘ `numeric::matlinsolve` ist ein schneller numerischer Löser für lineare Gleichungssysteme, die durch eine Koeffizientenmatrix spezifiziert sind. Diese Routine kann auch exakte symbolische Ergebnisse berechnen, wenn die Koeffizienten MuPAD-Ausdrücke sind. Sollen Lösungen über nicht-elementaren speziellen Koeffizientenringen berechnet werden, so ist `linalg::matlinsolve` zu benutzen.
- ⌘ `numeric::fsolve` ist der numerische Löser für Systeme beliebiger Gleichungen. Diese Routine sucht nur nach einer Lösung.
- ⌘ `numeric::odesolve` ist der numerische Löser für das Anfangswertproblem von Systemen gewöhnlicher Differentialgleichungen.
- ⌘ `numeric::odesolve2` liefert einen anwenderfreundlich verpackten Aufruf von `numeric::odesolve` als MuPAD-Funktion.
- ⌘ `numeric::polyroots` berechnet numerisch alle Wurzeln eines einzelnen univariaten Polynoms.
- ⌘ `numeric::polysysroots` berechnet alle Wurzeln eines Systems multivariater polynomialer Gleichungen. Dies ist ein hybrider Algorithmus, der symbolische Gröbner-Berechnungen mit einer numerischen Nachbearbeitung verbindet.
- ⌘ `numeric::realroot` berechnet eine einzelne reelle Lösung einer beliebigen reellen Gleichung.
- ⌘ `numeric::realroots` isoliert alle reellen Lösungen einer einzelnen beliebigen reellen Gleichung mittels Intervallarithmetik.
- ⌘ `numeric::solve` kombiniert `numeric::fsolve`, `numeric::polyroots` und `numeric::polysysroots`. Diese Funktion bestimmt selbständig den Typ des Gleichungssystems und ruft eine dieser Routinen auf.
- ⌘ `numlib::lincongruence` löst lineare Kongruenzen.
- ⌘ `numlib::mroots` löst Polynomkongruenzen.

- ☞ `numlib::msqrts` berechnet die Quadratwurzeln einer ganzen Zahl modulo einer anderen ganzen Zahl.
 - ☞ `polylib::realroots` liefert eine exakte Isolation aller reellen Wurzeln eines einzelnen univariaten reellen Polynoms.
 - ☞ `solve` ist der universelle Löser für Gleichungen und Gleichungssysteme sowie für Ungleichungen; diese Funktion löst auch Differenzialgleichungen (`ode`) sowie Rekurrenzgleichungen (`rec`) durch Aufruf der entsprechenden Löser.
- Die Bibliothek `solve` stellt einige Hilfsfunktionen zur Behandlung der Ergebnisse von `solve` bereit.
-

`sort` – Sortieren einer Liste

`sort(list)` liefert eine sortierte Kopie der Liste `list`.

Aufruf(e):

- ☞ `sort(list <, f>)`

Parameter:

- `list` — eine Liste mit beliebigen MuPAD-Objekten
- `f` — eine Prozedur, die die Anordnung festlegt

Rückgabewert: eine Liste.

Überladbar durch: `list`

Verwandte Funktionen: `sysorder`

Details:

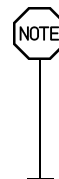
- ☞ `sort` sortiert die Liste in „aufsteigender Ordnung“.
- ☞ Ist keine Prozedur `f` angegeben, so werden Listen wie folgt sortiert:
 - Eine Liste reeller Zahlen vom syntaktischen Typ `Type::Real` wird numerisch sortiert.
 - Eine Liste aus Zeichenketten werden lexikographisch sortiert.
 - In allen anderen Fällen wird eine Liste nach der systeminternen Ordnung sortiert, d. h. `sort(list)` ist äquivalent zu `sort(list, sysorder)`.

Die interne Ordnung hängt nicht von der aktuellen MuPAD-Sitzung ab, kann sich aber zwischen verschiedenen MuPAD-Versionen unterscheiden. Der Benutzer sollte keinerlei Annahmen über den internen Sortiermechanismus voraussetzen. Eine solche Sortierung kann z. B. dazu dienen, zu Vergleichszwecken eindeutige Anordnungen von Listen zu erzeugen.

- ☞ Bei Zeichenketten werden große Buchstaben vor kleinen Buchstaben einsortiert. Beispielsweise wird "Z" vor "abc" eingeordnet.



- ☞ Mengen und Tabellen haben keine eindeutige interne Ordnung (`sysorder`). Dementsprechend führt das Sortieren von Listen, in denen Elemente Mengen oder Tabellen sind oder solche als (Sub-) Operanden enthalten, nicht immer zu einer eindeutig bestimmten Anordnung. Siehe Beispiel ??.



- ☞ Durch Angabe einer Prozedur f können beliebige Sortierkriterien vorgegeben werden. Diese Funktion wird dazu benutzt, jeweils zwei Elemente der Liste zu vergleichen. Sie wird in der Form $f(x, y)$ mit Elementen x, y aus der Menge aufgerufen und muss einen Ausdruck liefern, der zu `TRUE` oder `FALSE` ausgewertet werden kann. Bei `TRUE` wird x vor y einsortiert. Dementsprechend gilt für die geordnete Menge $L := \text{sort}(\text{list}, f)$ die Aussage $\text{bool}(f(L[i], L[j])) = \text{TRUE}$ für $i < j$.

Falls die durch f gegebene Ordnung keine Wohlordnung ist, kann die Sortierung „instabil“ sein, d. h. Elemente gleicher Ordnung können vertauscht werden.

- ☞ Die mittlere Laufzeit zum Sortieren von n Elementen ist $O(n \log n)$.
- ☞ `sort` ist eine Funktion des Systemkerns.

Beispiel 1. Reelle Zahlen vom Typ `Type::Real` werden numerisch sortiert:

```
>> sort([4, -1, 2/3, 0.5])
      [-1, 0.5, 2/3, 4]
```

Zeichenketten werden lexikographisch sortiert:

```
>> sort(["chip", "alpha", "Zip"])
      ["Zip", "alpha", "chip"]
```

Andere Objekte werden gemäß der internen Ordnung sortiert. Dies schließt auch Listen ein, die Elemente unterschiedlichen Typs enthalten:

```
>> sort([4, -1, 2/3, 0.5, "alpha"])
```

```

["alpha", -1, 4, 0.5, 2/3]

>> sort([4, -1, 2/3, 0.5, I])

[-1, 4, 0.5, 2/3, I]

```

Beispiel 2. Für Mengen und Tabellen existiert keine eindeutige interne Ordnung:

```

>> sort([ {1}, {2} ]) <> sort([ {2}, {1} ])

[ {1}, {2} ] <> [ {2}, {1} ]

>> sort([ table("a" = 42), table("a" = 43) ]) <>
sort([ table("a" = 43), table("a" = 42) ])

-- table(          table(          --
|      "a" = 42 ,    "a" = 43      | <>
-- )                )              --

-- table(          table(          --
|      "a" = 43 ,    "a" = 42      |
-- )                )              --

```

Beispiel 3. Die folgende Liste wird nach einem benutzerdefinierten Kriterium sortiert:

```

>> sort([-2, 1, -3, 4], (x, y) -> abs(x) < abs(y))

[1, -2, -3, 4]

```

Hintergründe:

- ⌘ Eine Variante des „Quicksort“-Algorithmus wird benutzt.

Änderungen:

- ⌘ Die systeminterne Ordnung hat sich mit dieser MuPAD-Version geändert.

split – Zerlegen eines Objekts

`split(object, f)` zerlegt das Eingabeobjekt `object` in eine Liste aus drei Objekten desselben Typs wie `object`. Das erste Ausgabeobjekt besteht aus denjenigen Operanden des Eingabeobjekts, die ein durch die Prozedur `f` definiertes Kriterium erfüllen. Das zweite Ausgabeobjekt besteht aus den Operanden, die das Kriterium nicht erfüllen. Das dritte Ausgabeobjekt besteht aus den Operanden, für die nicht entschieden werden kann, ob das Kriterium erfüllt ist.

Aufruf(e):

```
# split(object, f <, p1, p2, ...>)
```

Parameter:

- | | |
|--------------------------|---|
| <code>object</code> | — eine Liste, eine Menge, eine Tabelle, eine Ausdrucksfolge oder ein Ausdruck vom Typ <code>DOM_EXPR</code> |
| <code>f</code> | — eine Prozedur, die einen boolschen Wert zurückliefert |
| <code>p1, p2, ...</code> | — beliebige MuPAD-Objekte, die von <code>f</code> als zusätzliche Argumente akzeptiert werden |

Rückgabewert: eine Liste mit drei Objekten vom gleichen Typ wie das Eingabeobjekt.

Überladbar durch: `object`

Verwandte Funktionen: `map, op, select, zip`

Details:

- # Die Funktion `f` muss einen Wert liefern, der sich zu einem der boolschen Werte `TRUE`, `FALSE` oder `UNKNOWN` auswerten läßt. Sie kann einen dieser Werte direkt liefern, aber auch eine Gleichung oder eine Ungleichung, die von der Funktion `bool` zu einem dieser Werte vereinfacht werden kann.
- # Die Prozedur `f` wird mit dem Aufruf `f(x, p1, p2, ...)` auf alle Operanden `x` des Eingabeobjekts angewendet. Abhängig vom Ergebnis `TRUE`, `FALSE` bzw. `UNKNOWN` wird dieser Operand in das erste, zweite bzw. dritte Ausgabeobjekt eingetragen.
Die Ausgabeobjekte sind vom selben Typ wie das Eingabeobjekt, d. h. eine Liste wird in drei Listen aufgespalten, eine Menge in drei Mengen, eine Tabelle in drei Tabellen etc.
- # Ist das Eingabeobjekt eine Ausdrucksfolge, so wird weder die Eingabe noch die Ausgabe (eine Liste dreier Folgen) ausgeglichen.

☞ Auch „atomare“ Objekte wie z.B. Zahlen oder Bezeichner können als erstes Argument an `split` übergeben werden. Sie werden wie Folgen mit einem einzigen Element behandelt.

☞ `split` ist eine Funktion des Systemkerns.

Beispiel 1. Der folgende Befehl testet, welche der Listenelemente Primzahlen sind:

```
>> split([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], isprime)

[[2, 3, 5, 7], [1, 4, 6, 8, 9, 10], []]
```

Das Ergebnis ist eine Liste mit drei Listen. Die erste Liste enthält die Primzahlen, die zweite Liste alle anderen Zahlen. Die dritte Liste ist leer, da für jede Zahl der Eingabeliste entschieden werden kann, ob sie prim ist oder nicht.

Beispiel 2. Mit den optionalen Argumenten `p1`, `p2`, ... können Funktionen zur Zerlegung benutzt werden, die mehr als nur ein Argument benötigen. Beispielsweise ist `contains` eine solche Funktion, die sich bequem mit `split` einsetzen läßt. Der folgende Aufruf zerlegt eine Liste von Mengen in diejenigen Mengen, die `x` enthalten, und die anderen Mengen, die `x` nicht enthalten:

```
>> split([a, x, b], {a}, {x, 1}], contains, x)

[[{a, b, x}, {x, 1}], [{a}], []]
```

Die Elemente der zurückgegebenen Liste sind vom Typ `DOM_LIST`, da das Eingabeobjekt eine Liste ist. Ist das übergebene Objekt eine Menge (vom Typ `DOM_SET`), so sind die Elemente der Ergebnisliste auch von diesem Typ:

```
>> split([a, x, b], {a}, {x, 1}], contains, x)

[{{x, 1}, {a, b, x}}, {{a}}, {}]
```

Beispiel 3. Wir benutzen `is` als Entscheidungsfunktion, um eine Ausdrucksfolge zu zerlegen. Diese Funktion liefert `UNKNOWN`, falls sie keine Entscheidung über die abgefragte Eigenschaft fällen kann:

```
>> split((-2, -1, a, 0, b, 1, 2), is, Type::Positive)

[(1, 2), (-2, -1, 0), (a, b)]
```

Beispiel 4. Eine Tabelle wird zerlegt, die als männlich ("m") bzw. weiblich ("f") gekennzeichnete Personen enthält:

```
>> people := table("Tom" = "m", "Rita" = "f", "Joe" = "m"):
      [male, female, dummy] := split(people, has, "m"):
```

```
>> male
```

```
table(
  "Joe" = "m",
  "Tom" = "m"
)
```

```
>> female
```

```
table(
  "Rita" = "f"
)
```

```
>> dummy
```

```
table()
```

```
>> delete people, male, female, dummy:
```

Änderungen:

☞ Keine Änderungen.

`sqrt` – die Wurzelfunktion

`sqrt(z)` stellt die Wurzel von z dar.

Aufruf(e):

☞ `sqrt(z)`

Parameter:

z — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: z

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `_power`, `isqrt`, `numlib::issqr`

Details:

- ☞ $x = \text{sqrt}(z)$ stellt diejenige der beiden Lösungen von $x^2 = z$ dar, die einen nicht-negativen Realteil hat. Insbesondere stellt $\text{sqrt}(z)$ für reelles $z > 0$ die positive Wurzel dar. Für reelles $z < 0$ stellt $\text{sqrt}(z)$ die Wurzel mit positivem Imaginärteil dar.
 - ☞ Für Gleitpunktargumente wird eine Gleitpunktzahl berechnet. Man beachte, dass die negative reelle Halbachse als Verzweigungsschnitt gewählt ist. Dementsprechend springen die von `sqrt` gelieferten Werte, wenn man diesen Schnitt überschreitet. Siehe Beispiel ??.
 - ☞ `sqrt` versucht, gewisse Vereinfachungen zu erreichen. Insbesondere werden positive ganzzahlige Faktoren aus einigen symbolischen Produkten herausgezogen. Siehe Beispiel ??.
 - ☞ Man beachte, dass $\text{sqrt}(x^2)$ nicht für alle komplexen Werte zu x vereinfacht werden kann (z. B., $\text{sqrt}(x^2) = -x$ für reelles $x < 0$). Siehe Beispiel ??.
 - ☞ Mathematisch stimmen $\text{sqrt}(z)$ und $z^{(1/2)} = \text{power}(z, 1/2)$ überein. Allerdings versucht `sqrt` stärkere Vereinfachungen zu erreichen als `_power`. Siehe Beispiel ??.
-

Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> sqrt(2), sqrt(4), sqrt(36*7), sqrt(127)
      1/2      1/2      1/2
      2      , 2, 6 7      , 127

>> sqrt(1/4), sqrt(1/2), sqrt(3/4), sqrt(25/36/7), sqrt(4/127)
      1/2      1/2      1/2      1/2
      2      3      5 7      2 127
1/2, ----, ----, ----, ----
      2      2      42      127

>> sqrt(-4), sqrt(-1/2), sqrt(1 + I)
      1/2      1/2
      2 I, 1/2 I 2      , (1 + I)
```

```
>> sqrt(x), sqrt(4*x^(4/7)), sqrt(4*x/3), sqrt(4*(x + I))
```

$$x^{1/2}, 2 x^{2/7} \sqrt[4]{x}, \sqrt{\frac{4x}{3}}, (4x + 4I)^{1/2}$$

Beispiel 2. Für Gleitkommazahlen wird der numerische Wert der Wurzel berechnet:

```
>> sqrt(1234.5), sqrt(-1234.5), sqrt(-2.0 + 3.0*I)
35.13545218, 35.13545218 I, 0.8959774761 + 1.674149228 I
```

Die Werte springen, wenn man die negative reelle Halbachse überschreitet:

```
>> sqrt(-4.0), sqrt(-4.0 + I/10^100), sqrt(-4.0 - I/10^100)
2.0 I, 2.5e-101 + 2.0 I, 2.5e-101 - 2.0 I
```

Beispiel 3. Die Wurzel symbolischer Produkte mit positiven ganzzahligen Faktoren wird vereinfacht:

```
>> sqrt(20*x*y*z)
2 (x y z)^{1/2}
```

Beispiel 4. Die Wurzel von Quadraten wird nur vereinfacht, wenn das Argument reell ist und das Vorzeichen bekannt ist:

```
>> sqrt(x^2*y^4)
(x^2 y^4)^{1/2}
```

```
>> assume(x > 0): sqrt(x^2*y^4)
x^2 y^4^{1/2}
```

```
>> assume(x < 0): sqrt(x^2*y^4)
-x^2 y^4^{1/2}
```

Beispiel 5. `sqrt` vereinfacht stärker als die `_power`-Funktion:

```
>> sqrt(4*x), (4*x)^(1/2) = _power(4*x, 1/2)
```

$$2 x^{1/2}, (4 x)^{1/2} = (4 x)^{1/2}$$

Änderungen:

⌘ Keine Änderungen.

`strmatch` – Suche nach Mustern in einer Zeichenkette

`strmatch(text, pattern)` vergleicht die Zeichenketten `text` und `pattern` auf Gleichheit, wobei das Muster `pattern` „Stellvertreterzeichen“ enthalten darf.

`strmatch(text, pattern, Index)` überprüft, ob `pattern` als Teilzeichenkette in `text` enthalten ist, und liefert die Position des ersten Auftretens des Musters `pattern` zurück.

Aufruf(e):

⌘ `strmatch(text, pattern)`
⌘ `strmatch(text, pattern, Index)`

Parameter:

`text, pattern` — Zeichenketten

Optionen:

Index — veranlasst `strmatch` nach Teilzeichenketten in `text` zu suchen, die `pattern` enthalten. `FALSE` wird zurückgeliefert, wenn keine Entsprechung gefunden wird. Anderenfalls wird die Position des ersten Auftauchens von `pattern` durch eine Liste aus zwei ganzen Zahlen angezeigt.

Rückgabewert: Ohne *Index* wird `TRUE` oder `FALSE` geliefert. Mit *Index* wird eine Liste zweier nichtnegativer ganzer Zahlen oder `FALSE` geliefert.


Überladbar durch: `text, pattern`

Verwandte Funktionen: `_concat`, `length`, `substring`, `stringlib::contains`, `stringlib::pos`

Details:

- ☞ Das Muster `pattern` kann die Stellvertreterzeichen `\?` und `*` enthalten. Der Stellvertreter `\?` steht für ein beliebiges einzelnes Zeichen oder kein Zeichen. Der Stellvertreter `*` steht für eine beliebige Teilzeichenkette, die auch leer sein kann.
 - ☞ Die Zeichenkette `text` darf keine Stellvertreterzeichen enthalten.
 - ☞ In MuPAD Zeichenketten wird das Zeichen `\` mittels `\\` repräsentiert. Siehe Beispiel ??.
 - ☞ Die Bibliothek `stringlib` stellt weitere Funktionen zum Arbeiten mit Zeichenketten zur Verfügung.
 - ☞ `strmatch` ist eine Funktion des Systemkerns.
-

Option `<Index>`:

- ☞ `strmatch(text, pattern, Index)` überprüft, ob `text` das Muster `pattern` enthält und liefert die Liste `[i, j]` zurück. Die ganze Zahl `i` ist der Index des ersten Zeichens der Zeichenfolge, die dem Muster entspricht, `j` ist der Index des letzten Zeichens. Damit gilt `substring(text, a, b-a+1) = pattern`. Es wird nur das erste Auftreten von `pattern` in `text` gefunden. `FALSE` wird zurückgeliefert, falls `pattern` nicht gefunden wird.
 - ☞ Man beachte, dass die Zählung der Zeichen von `text` mit dem Index 0 beginnt. 
 - ☞ Werden Stellvertreterzeichen in `pattern` verwendet, so wird die *längste* gefundene Übereinstimmung zurückgeliefert. Beispielsweise wird im Text `"XXabcbXX"` das Muster `"a*b"` mit der Teilzeichenkette `"abcb"` identifiziert, nicht mit `"ab"`.
-

Beispiel 1. Es werden einfache Vergleiche von Zeichenketten durchgeführt:

```
>> s := "Hamburg": strmatch(s, "Hamburg")  
  
TRUE  
  
>> strmatch(s, "Ham"), strmatch(s, "burg")  
  
FALSE, FALSE  
  
>> delete s:
```

Beispiel 2. Dieses Beispiel demonstriert die Benutzung von Stellvertreterzeichen. Der Stellvertreter `\?` steht für ein einzelnes Zeichen oder auch für kein Zeichen:

```
>> strmatch("Mississippi", "Miss\?issip\?i")  
  
TRUE
```

Der Stellvertreter `*` repräsentiert eine beliebige (eventuell auch leere) Zeichenkette:

```
>> strmatch("Mississippi", "Mi\*i\*pp\*i\*")  
  
TRUE
```

Im folgenden Aufruf wird das Muster nicht gefunden:

```
>> strmatch("Mississippi", "Mis\?i\*ppi\*i")  
  
FALSE
```

Beispiel 3. Das Zeichen `?` ist kein Stellvertreterzeichen:

```
>> strmatch("Mississippi", "Miss?issip\?i")  
  
FALSE
```

Das Zeichen `\` innerhalb von Zeichenketten ist über die Zeichenfolge `\\` repräsentiert. Dementsprechend wird `\\` als einzelnes Zeichen angesehen:

```
>> s := "a\\b": s[0], s[1], s[2]  
  
"a", "\\ ", "b"  
  
>> strmatch("Missi\\ssippi", "Missi\\?ssippi")  
  
TRUE  
  
>> delete s:
```

Beispiel 4. Mit der Option *Index* kann überprüft werden, ob eine Zeichenkette Bestandteil einer anderen Zeichenkette ist. In diesem Fall wird auch die Position des Teilstrings zurückgeliefert:

```
>> strmatch("cdxxcd", "xx", Index)  
  
[2, 3]
```

Es wird nur das erste Auftreten des Musters innerhalb des Textes angezeigt:

```
>> strmatch("cdxxcd", "cd", Index)

[0, 1]
```

Es wird die längste passende Übereinstimmung angegeben:

```
>> strmatch("cdxxcxcd", "x*x", Index)

[2, 5]
```

```
>> strmatch("cdxxcd", "xx*", Index)

[2, 5]
```

```
>> strmatch("cdxxcd", "*xx*", Index)

[0, 5]
```

Änderungen:

- ☞ In früheren MuPAD-Versionen durfte die Zeichenkette `text` Stellvertreterzeichen enthalten.
-

subs – Substitution von Operanden

`subs(f, old = new)` liefert eine Kopie des Objektes `f`, in der alle mit `old` übereinstimmenden Operanden durch den Wert `new` ersetzt sind.

Aufruf(e):

- ☞ `subs(f, old = new <, Unsimplified>)`
- ☞ `subs(f, old1 = new1, old2 = new2, ... <, Unsimplified>)`
- ☞ `subs(f, [old1 = new1, old2 = new2, ...] <, Unsimplified>)`
- ☞ `subs(f, {old1 = new1, old2 = new2, ...} <, Unsimplified>)`
- ☞ `subs(f, table(old1 = new1, old2 = new2, ...) <, Unsimplified>)`
- ☞ `subs(f, s1, s2, ... <, Unsimplified>)`

Parameter:

- `f` — ein beliebiges MuPAD-Objekt
- `old, old1, old2, ...` — beliebige MuPAD-Objekte
- `new, new1, new2, ...` — beliebige MuPAD-Objekte
- `s1, s2, ...` — „Substitutionsgleichungen“: entweder Gleichungen `old = new`, oder Mengen oder Listen solcher Gleichungen, oder Tabellen, deren Einträge als solche Gleichungen interpretiert werden.

Optionen:

- `Unsimplified` — unterdrückt die Vereinfachung des zurückgelieferten Objekts nach der Substitution

Rückgabewert: eine Kopie des Eingabeobjekts mit ersetzten Operanden.

Überladbar durch: `f`

Verwandte Funktionen: `extnops, extop, extsubsop, has, map, match, op, subsex, subsop`

Details:

- ☞ `subs` liefert eine veränderte Kopie des Objektes ohne das Objekt selbst zu verändern.
- ☞ `subs(f, old = new)` ersetzt jeden Operanden in `f`, der mit `old` übereinstimmt, durch den Wert `new`. Siehe Beispiel ??.
- ☞ Der Aufruf `subs(f, old1 = new1, old2 = new2, ...)` führt eine „sequentielle Substitution“ durch: die Substitutionen werden von links nach rechts bearbeitet, wobei jeweils das Ergebnis der letzten Substitution benutzt wird. Siehe Beispiel ??.
- ☞ Der Aufruf `subs(f, [old1 = new1, old2 = new2, ...])` führt eine „parallele Substitution“ durch: jede einzelne Substitution bezieht sich auf die Operanden des Eingabeobjekts `f` und nicht auf die Operanden von „Zwischenergebnissen“ vorheriger Substitutionen. Bei mehrfachen Substitutionen ein und desselben Operanden wird nur die erste Substitution durchgeführt. Übergabe der Substitutionsgleichungen als Menge oder Tabelle führt ebenfalls zu paralleler Substitution. Siehe Beispiel ??.
- ☞ Der Aufruf `subs(f, s1, s2, ...)` beschreibt die allgemeinste Form einer Substitution, in der sequentielle und parallele Teils substitutionen gemischt werden können. Dieser Aufruf entspricht `subs(... subs(subs(f, s1), s2), ...)`. Abhängig von der Form von `s1, s2, ...` werden hierbei jeweils sequentielle oder parallele Substitutionen wie oben beschrieben ausgeführt. Siehe Beispiel ??.

- ⌘ Es werden nur Operanden ersetzt, die mittels `op` gefunden werden („syntaktische Substitution“). Die Funktion `subsex` dient zur Durchführung „semantischer“ Substitutionen, in der auch Teilsummen oder -produkte gefunden und ersetzt werden. Siehe Beispiel ??.
- ⌘ Nach der Substitution wird das Ergebnis nicht evaluiert. Vollständige Evaluierung kann mittels `eval` erzwungen werden. Siehe Beispiel ??.
- ⌘ Operanden von Ausdrucksfolgen können von `subs` ersetzt werden: Folgen werden nicht ausgeglichen. Siehe Beispiel ??.
- ⌘ Der Aufruf `subs(f)` ist zulässig. Er liefert `f` ohne Änderungen zurück.
- ⌘ `subs` ist eine Funktion des Systemkerns.

Option *<Unsimpified>*:

- ⌘ Nach der Substitution wird eine Vereinfachung (aber keine Evaluierung) des resultierenden Objekts durch den MuPAD-Kern vorgenommen. Diese abschließende Vereinfachung kann mit dieser Option unterdrückt werden. Siehe Beispiel ??.

Beispiel 1. Einige einfache Substitutionen werden durchgeführt:

```
>> subs(a + b*a, a = 4)

      4 b + 4

>> subs([a * (b + c), sin(b + c)], b + c = a)

      2
[a , sin(a)]
```

Beispiel 2. Um in einem Ausdruck die Sinus-Funktion zu ersetzen, muss die Evaluierung des Bezeichners `sin` mittels `hold` verhindert werden. Sonst würde `sin` durch seinen Wert ersetzt. Dieser Wert ist die Funktionsumgebung, welche MuPADs Sinus-Funktion definiert. Im Ausdruck `sin(x)` ist der 0-te Operand `sin` der Bezeichner, nicht die Funktionsumgebung:

```
>> domtype(sin), domtype(hold(sin)), domtype(op(sin(x), 0));

DOM_FUNC_ENV, DOM_IDENT, DOM_IDENT

>> subs(sin(x), sin = cos), subs(sin(x), hold(sin) = cos)

sin(x), cos(x)
```

Beispiel 3. Der folgende Aufruf führt zur sequentiellen Substitution $x \rightarrow y \rightarrow z$:

```
>> subs(x^3 + y*z, x = y, y = z)
```

$$z^2 + z^3$$

Beispiel 4. Der Unterschied zwischen sequentieller und paralleler Substitution wird demonstriert. Sequentielle Substitutionen ergeben die folgenden Resultate:

```
>> subs(a^2 + b^3, a = b, b = a)
```

$$a^2 + a^3$$

```
>> subs(a^2 + b^3, b = a, a = b)
```

$$b^2 + b^3$$

Im Gegensatz dazu tauscht parallele Substitution die Bezeichner aus:

```
>> subs(a^2 + b^3, [a = b, b = a])
```

$$a^3 + b^2$$

Im folgenden Aufruf wird zunächst mit der Ersetzung von a durch $x + y$ das Zwischenergebnis $y + 2*x$ erzeugt. Hieraus entsteht dann beim Ersetzen von x durch z das Ergebnis $y + 2*z$:

```
>> subs(a + x, a = x + y, x = z)
```

$$y + 2*z$$

Parallele Substitution führt zu einem anderen Ergebnis. Im nächsten Aufruf wird a durch $x + y$ ersetzt, wobei gleichzeitig der Operand x *des Originalausdrucks* $a + x$ durch z ersetzt wird:

```
>> subs(a + x, [a = x + y, x = z])
```

$$x + y + z$$

Parallele Substitution wird auch durchgeführt, wenn die Substitutionsgleichungen als Menge übergeben werden:

```
>> subs(a + x, {a = x + y, x = z})
```

$$x + y + z$$

Weiterhin wird parallele Substitution bei Angabe der Substitutionsgleichungen als Tabelle benutzt:

```
>> T := table(): T[a] := x + y: T[x] := z: T
```

```
table(
  x = z,
  a = x + y
)
```

```
>> subs(a + x, T)
```

$$x + y + z$$

```
>> delete T:
```

Beispiel 5. Sequentielle und parallele Substitutionen werden kombiniert:

```
>> subs(a + x, {a = x + y, x = z}, x = y)
```

$$2 y + z$$

Beispiel 6. Nur Operanden, die durch `op` gefunden werden, können ersetzt werden. Der folgende Ausdruck enthält den Teilausdruck $x + y$ als den Operanden `op(f, [1, 2])`:

```
>> f := sin(z*(x + y)): op(f, [1, 2]);
```

$$x + y$$

Dementsprechend kann dieser Teilausdruck ersetzt werden:

```
>> subs(f, x + y = z)
```

$$\sin(z^2)$$

Rein syntaktisch enthält die folgende Summe den Teilausdruck $x + y$ nicht. Er wird daher nicht ersetzt:

```
>> subs(x + y + z, x + y = z)
```

$$x + y + z$$

Im Gegensatz zu `subs` findet die Funktion `subsex` jedoch auch Teilsummen und -produkte und ersetzt sie:

```
>> subsex(x + y + z, x + y = z)

      2 z

>> subs(a*b*c, a*c = 5), subsex(a*b*c, a*c = 5)

      a b c, 5 b

>> delete f:
```

Beispiel 7. Das Ergebnis von `subs` wird nicht evaluiert. Im folgenden Beispiel wird der Bezeichner `sin` nicht durch seinen Wert ersetzt. Dieser Wert ist die Prozedur, die das Verhalten der Sinus-Funktion des Systems definiert. Dementsprechend wird `sin(PI)` nicht durch diese Prozedur zu 0 vereinfacht:

```
>> subs(sin(x), x = PI)

      sin(PI)
```

Die Evaluierung kann durch `eval` erzwungen werden:

```
>> eval(subs(sin(x), x = PI))

      0
```

Beispiel 8. Die Operanden von Ausdrucksfolgen können ersetzt werden. Zur Eingabe müssen Folgen jedoch geklammert werden:

```
>> subs((a, b, a*b), a = x)

      x, b, b x
```

Beispiel 9. Die Option *Unsimpilified* unterdrückt Vereinfachungen:

```
>> subs(a + b + 2, a = 1, b = 0, Unsimpilified)

      1 + 0 + 2
```

Beispiel 10. In Domains werden mit `subs` angeforderte Substitutionen ignoriert. Ein neues Domain mit den Methoden "foo" und "bar" wird definiert:

```
>> mydomain := newDomain("Test"):
      mydomain::foo := x -> 4*x:
      mydomain::bar := x -> 4*x^2:
```

Innerhalb des Domains soll nun der Wert 4 überall durch den Wert 3 ersetzt werden:

```
>> mydomain := subs(mydomain, 4 = 3):
```

Diese Substitution hatte jedoch keinerlei Effekt:

```
>> mydomain::foo(x), mydomain::bar(x)

                2
            4 x, 4 x
```

Ersetzungen in Domain-Methoden müssen direkt in den Methoden vorgenommen werden:

```
>> mydomain::foo := subs(mydomain::foo, 4 = 3):
      mydomain::bar := subs(mydomain::bar, 4 = 3):
      mydomain::foo(x), mydomain::bar(x)

                2
            3 x, 3 x
```

```
>> delete mydomain:
```

Änderungen:

☞ `subs` substituiert nicht mehr innerhalb von Domains. Siehe Beispiel ??.

subsex – erweiterte Substitution

`subsex(f, old = new)` liefert eine Kopie des Objektes `f`, in der alle mit `old` übereinstimmenden Ausdrücke durch den Wert `new` ersetzt sind. Im Gegensatz zur Funktion `subs` ersetzt `subsex` auch „unvollständige“ Teilausdrücke.

Aufruf(e):

```

# subsex(f, old = new <, Unsimplified>)
# subsex(f, old1 = new1, old2 = new2, ... <, Unsimplified>)
# subsex(f, [old1 = new1, old2 = new2, ...] <, Unsimplified>)
# subsex(f, {old1 = new1, old2 = new2, ...} <, Unsimplified>)
# subsex(f, table(old1 = new1, old2 = new2, ...) <, Unsimplified>)
# subsex(f, s1, s2, ... <, Unsimplified>)

```

Parameter:

<code>f</code>	— ein beliebiges MuPAD-Objekt
<code>old, old1, old2, ...</code>	— beliebige MuPAD-Objekte
<code>new, new1, new2, ...</code>	— beliebige MuPAD-Objekte
<code>s1, s2, ...</code>	— „Substitutionsgleichungen“: entweder Gleichungen <code>old = new</code> , oder Mengen oder Listen solcher Gleichungen, oder Tabellen, deren Einträge als solche Gleichungen interpretiert werden.

Optionen:

`Unsimplified` — unterdrückt die Vereinfachung des zurückgelieferten Objekts nach der Substitution

Rückgabewert: eine Kopie des Eingabeobjekts mit ersetzten Operanden.

Überladbar durch: `f`

Verwandte Funktionen: `extnops`, `extop`, `extsubsop`, `has`, `map`, `match`, `op`, `subs`, `subsop`

Details:

- # `subsex` liefert eine veränderte Kopie des Objektes ohne das Objekt selbst zu verändern.
- # `subsex(f, old = new)` ersetzt jeden Teilausdruck in `f`, der mit `old` übereinstimmt, durch den Wert `new`.
- # In den meisten Fällen liefert `subsex` dasselbe Ergebnis wie `subs`. Im Gegensatz zu `subs` ist es mit `subsex` jedoch möglich, auch „unvollständige“ Teilausdrücke wie beispielsweise `a + b` in einer Summe `a + b + c` zu ersetzen. Allgemein können Kombinationen von Operanden der n -ären „Operatoren“ `+`, `*`, `and`, `_exprseq`, `intersect`, `or`,

`_lazy_and`, `_lazy_or` und `union` ersetzt werden. Speziell können also Teilsummen und Teilprodukte ersetzt werden. Man beachte, dass diese Operationen als kommutativ vorausgesetzt werden. Z. B. wird das Teilprodukt $a \cdot c$ durch `subsex(a*b*c, a*c = new)` ersetzt. Siehe Beispiel ?? und Beispiel ??.

- ☞ `subsex` ist wesentlich langsamer als `subs` und sollte daher nur in Situationen benutzt werden, wo `subs` die benötigte Funktionalität nicht zur Verfügung stellt.
- ☞ Der Aufruf `subsex(f, old1 = new1, old2 = new2, ...)` führt eine „sequentielle Substitution“ durch. Siehe die `subs`-Hilfeseite für Details.
- ☞ Der Aufruf `subsex(f, [old1 = new1, old2 = new2, ...])` führt eine „parallele Substitution“ durch. Siehe die `subs`-Hilfeseite für Details.
- ☞ Der Aufruf `subsex(f, s1, s2, ...)` beschreibt die allgemeinste Form einer Substitution, in der sequentielle und parallele Teilsubstitutionen gemischt werden können. Dieser Aufruf entspricht `subsex(... subsex(subsex(f, s1), s2), ...)`. Abhängig von der Form von `s1, s2, ...` werden hierbei jeweils sequentielle oder parallele Substitutionen ausgeführt. Ein Beispiel hierzu findet man auf der `subs`-Hilfeseite.
- ☞ Nach der Substitution wird das Ergebnis nicht evaluiert. Vollständige Evaluierung kann mittels `eval` erzwungen werden. Siehe Beispiel ??.
- ☞ Operanden von Ausdrucksfolgen können durch `subsex` ersetzt werden: Folgen werden nicht ausgeglichen. Siehe Beispiel ??.
- ☞ Der Aufruf `subsex(f)` ist zulässig. Er liefert `f` ohne Änderungen zurück.
- ☞ `subsex` ist eine Funktion des Systemkerns.

Option **<Unsimplified>**:

- ☞ Nach der Substitution wird eine Vereinfachung (aber keine Evaluierung) des resultierenden Objekts durch den MuPAD-Kern vorgenommen. Diese abschließende Vereinfachung kann mit dieser Option unterdrückt werden. Ein Beispiel hierzu findet man auf der `subs`-Hilfeseite.

Beispiel 1. Es werden einige einfache Substitutionen durchgeführt; `subsex` findet und ersetzt Teilsummen und -produkte:

```
>> subsex(a + b + c, a + c = x)
```

$b + x$


```
>> subsex(a*b*c, a*c = x)
```

$b\ x$

```
>> subsex(a * (b + c) + b + c, b + c = a)
```

$a + a^2$

```
>> subsex(a + b*c*d + b*d, b*d = c);
```

$a + c + c^2$

Beispiel 2. Es werden Teilausdrücke in Ausdrucksfolgen und symbolischen Vereinigungsmengen ersetzt:

```
>> subsex((a, b, c, d), (b, d) = w)
```

a, c, w

```
>> subsex(a union b union c, a union b = w)
```

$c \text{ union } w$

Dasselbe Ergebnis kann mittels der dem Operator union äquivalenten Funktion `_union` erzielt werden:

```
>> subsex(_union(a, b, c), _union(a, b) = w)
```

$c \text{ union } w$

Beispiel 3. Das Ergebnis von `subsex` wird nicht evaluiert. Im folgenden Beispiel wird der Bezeichner `sin` nicht durch seinen Wert ersetzt. Dieser Wert ist die Prozedur, die das Verhalten der Sinus-Funktion des Systems definiert. Dementsprechend wird `sin(2*PI)` nicht durch diese Prozedur zu 0 vereinfacht:

```
>> subsex(sin(2*x*y), x*y = PI)
```

$\sin(2\ PI)$

Die Evaluierung kann durch `eval` erzwungen werden:

```
>> eval(subsex(sin(2*x*y), x*y = PI))
```

0

Beispiel 4. Die Operanden von Ausdrucksfolgen können ersetzt werden. Zur Eingabe müssen Folgen geklammert werden:

```
>> subsex((a, b, a*b*c), a*b = x)
a, b, c x
```

Beispiel 5. Die Option *Unsimplified* unterdrückt Vereinfachungen:

```
>> subsex(2 + a + b, a + b = 0, Unsimplified)
2 + 0
```

Änderungen:

⌘ `subsex` substituiert nicht mehr innerhalb von Domains. Weitere Informationen entnehme man der `subs`-Hilfeseite.

`subsop` – Ersetzung von Operanden

`subsop(object, i = new)` liefert eine Kopie des Objektes `object`, in der der *i*-te Operand durch den Wert `new` ersetzt ist.

Aufruf(e):

⌘ `subsop(object, i1 = new1, i2 = new2, ... <, Unsimplified>)`

Parameter:

<code>object</code>	— ein beliebiges MuPAD-Objekt
<code>i1, i2, ...</code>	— ganze Zahlen oder Listen ganzer Zahlen
<code>new1, new2, ...</code>	— beliebige MuPAD-Objekte

Optionen:

Unsimplified — unterdrückt die Vereinfachung des Rückgabewerts nach der Substitution

Rückgabewert: das Eingabeobjekt mit ersetzten Operanden oder `FAIL`.

Überladbar durch: `object`

Verwandte Funktionen: `extnops`, `extop`, `extsubsop`, `map`, `match`, `op`, `subs`, `subsex`

Details:

- ⇒ `subsop` liefert eine geänderte Kopie des Objektes, ohne das Objekt selbst zu verändern.
- ⇒ `subsop(object, i = new)` ersetzt den Operanden `op(object, i)` durch `new`. Operanden werden dabei wie bei der Funktion `op` angegeben: `i` kann entweder eine ganze Zahl oder eine Liste ganzer Zahlen sein. So ersetzt beispielsweise `subsop(object, [j, k] = new)` den (Sub-)Operanden `op(op(object, j), k)`. Siehe Beispiel ??.
Im Gegensatz zur Funktion `op` können in `subsop` keine Bereiche angegeben werden, um mehrere Operanden zu ersetzen. Stattdessen sind mehrere Substitutionsgleichungen anzugeben.
- ⇒ Wenn mehrere Operanden zu ersetzen sind, werden die angegebenen Substitutionen von links nach rechts bearbeitet. Die Substitutionen werden nacheinander durchgeführt, die Zwischenergebnisse werden dabei nicht vereinfacht.
- ⇒ Das Ergebnis von `subsop` wird nicht weiter evaluiert. Vollständige Evaluierung kann mittels `eval` erzwungen werden. Siehe Beispiel ??.
- ⇒ Operanden von Ausdrucksfolgen können mittels `subsop` ersetzt werden: Folgen werden nicht ausgeglichen.
- ⇒ Man beachte, dass sich die Reihenfolge von Operanden durch eine Substitution ändern kann. Siehe Beispiel ??.
- ⇒ Ist der Zugriff auf einen Operanden nicht möglich, so wird `FAIL` zurückgegeben.
- ⇒ Substitutionen mittels `subsop` sind schneller als mit `subs` oder `subsex`.
- ⇒ Der Aufruf `subsop(object)` ist zulässig und liefert das Objekt ohne Änderungen zurück.
- ⇒ `subsop` ist eine Funktion des Systemkerns.

Option *<Unsimpified>*:

- ⇒ Nach der Substitution wird eine Vereinfachung (aber keine Evaluierung) des resultierenden Objekts durch den MuPAD-Kern vorgenommen. Diese abschließende Vereinfachung kann mit dieser Option unterdrückt werden. Siehe Beispiel ??.
-

Beispiel 1. Das Ersetzen von einem bzw. mehreren Operanden eines Ausdrucks wird demonstriert:

```
>> x := a + b: subsop(x, 2 = c)

a + c

>> subsop(x, 1 = 2, 2 = c)

c + 2
```

Auch der 0-te Operand eines Ausdrucks (der „Operator“) kann ersetzt werden:

```
>> subsop(x, 0 = _mult)

a b
```

Die Variable `x` selbst wurde durch die Substitutionen nicht verändert:

```
>> x

a + b

>> delete x:
```

Beispiel 2. Der folgende Aufruf benutzt eine Liste ganzer Zahlen, um den Suboperanden `c` zu spezifizieren:

```
>> subsop([a, b, f(c)], [3, 1] = x)

[a, b, f(x)]
```

Beispiel 3. Dieses Beispiel demonstriert die internen Vereinfachungen von Ausdrücken. Der folgende Aufruf ersetzt den ersten Operanden `a` durch `2`. Das Ergebnis wird zu `3` vereinfacht:

```
>> subsop(a + 1, 1 = 2)

3
```

Die Option *Unsimplified* verhindert die Vereinfachung:

```
>> subsop(a + 1, 1 = 2, Unsimplified)

2 + 1
```

Das folgende Aufruf demonstriert den Unterschied zwischen *Vereinfachung* und *Evaluierung*. Nach Ersetzen von `x` durch `PI` wird der Bezeichner `sin` nicht evaluiert, d. h., die Systemfunktion `sin` wird nicht aufgerufen:

```
>> subsop(sin(x), 1 = PI)

sin(PI)
```

Evaluation von `sin` vereinfacht das Ergebnis:

```
>> eval(%)

0
```

Beispiel 4. Die Anordnung von Operanden kann sich durch Substitutionen ändern. Die Ersetzung des Bezeichners `b` durch `z` ändert die interne Reihenfolge der Summanden in `x`:

```
>> x := a + b + c: op(x)

a, b, c

>> x := subsop(x, 2 = z): op(x)

a, c, z

>> delete x:
```

Hintergründe:

- ☞ Zur Überladung von `subsop` reicht es aus, die Fälle `subsop(object)` und `subsop(object, i = new)` zu behandeln.

Änderungen:

- ☞ Keine Änderungen.

substring – Extrahieren eines Teils einer Zeichenkette

`substring(string, i)` liefert das $(i + 1)$ -te Zeichen der Zeichenkette `string`.

`substring(string, i, l)` liefert `l` Zeichen startend vom $(i + 1)$ -ten Zeichen.

`substring(string, i..j)` liefert die aus dem $(i + 1)$ -ten bis zum $(j + 1)$ -ten Zeichen bestehende Teilzeichenkette.


Aufruf(e):

```
# substring(string, i)
# substring(string, i, l)
# substring(string, i..j)
```

Parameter:

string — eine nicht-leere Zeichenkette
 i — eine ganze Zahl zwischen 0 und `length(string) - 1`
 l — eine ganze Zahl zwischen 0 und `length(string) - i`
 j — eine ganze Zahl zwischen i und `length(string) - 1`

Rückgabewert: eine Zeichenkette.**Verwandte Funktionen:** `length`, `strmatch`, `stringlib::subs`**Details:**

- # Die Zeichen einer Zeichenkette sind mit den Zahlen von 0 bis `length(string) - 1` indiziert. 
- # Die leere Zeichenkette "" wird zurückgeliefert, falls die Länge `l = 0` angegeben wird.
- # `substring` ist eine Funktion des Systemkerns.

Beispiel 1. Einzelne Zeichen einer Zeichenkette werden extrahiert:

```
>> substring("0123456789", i) $ i = 0..9
      "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
```

Teilzeichenketten vorgegebener Länge werden extrahiert:

```
>> substring("0123456789", 0, 2), substring("0123456789", 4, 4)
      "01", "4567"
```

Die leere Zeichenkette hat die Länge 0:

```
>> substring("0123456789", 4, 0)
      ""
```

Teilzeichenketten können durch Indexbereiche festgelegt werden:

```
>> substring("0123456789", 0..9)
      "0123456789"
```

Man beachte, dass die Zeichenindizes mit 0 beginnend gezählt werden:

```
>> substring("123456789", 4..8)

"56789"
```

Beispiel 2. Die folgende while-Schleife löscht alle Leerzeichen am Ende einer Zeichenkette:

```
>> string := "MuPAD      ":
  while substring(string, length(string) - 1) = " " do
    string := substring(string, 0..length(string) - 2)
  end_while

"MuPAD"
```

Die folgende for-Schleife sucht nach aufeinanderfolgenden Leerzeichen in einer Zeichenkette und verkleinert solche Zwischenräume auf ein einzelnes Leerzeichen:

```
>> string := "MuPAD - the open computer algebra system":
  result := substring(string, 0):
  space_count := 0:
  for i from 1 to length(string) - 1 do
    if substring(string, i) <> " " then
      result := result . substring(string, i);
      space_count := 0
    elif space_count = 0 then
      result := result . substring(string, i);
      space_count := space_count + 1
    end_if
  end_for:
  result

"MuPAD - the open computer algebra system"

>> delete string, result, space_count, i:
```

Änderungen:

⌘ Keine Änderungen.

sum – bestimmte und unbestimmte symbolische Summation

`sum(f, i)` bestimmt eine symbolische Antidifferenz von $f(i)$ bezüglich i .

`sum(f, i = a..b)` sucht nach einer geschlossenen Darstellung der Summe $\sum_{i=a}^b f(i)$.

Aufruf(e):

```
⌘ sum(f, i)
⌘ sum(f, i = a..b)
⌘ sum(f, i = RootOf(p, x))
```

Parameter:

`f` — ein von i abhängender arithmetischer Ausdruck
`i` — der Summationsindex: ein Bezeichner
`a, b` — die Grenzen: arithmetische Ausdrücke
`p` — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
`x` — eine Unbestimmte von `p`

Rückgabewert: ein arithmetischer Ausdruck.

Verwandte Funktionen: `_plus, +, int, numeric::sum, product, rec`

Details:

⌘ `sum` dient zur *symbolischen* Summation von Ausdrücken (das diskrete Analogon zur Integration). Diese Funktion sollte *nicht* zur einfachen Addition endlich vieler Terme verwendet werden: für ganzzahliges a, b vom Typ `DOM_INT` ist der Aufruf `_plus(f $ i = a..b)` effizienter als `sum(f, i = a..b)`. Siehe Beispiel ??.

⌘ `sum(f, i)` berechnet die unbestimmte Summe von f bzgl. i . Dies ist ein Ausdruck g mit $f(i) = g(i+1) - g(i)$.

⌘ `sum(f, i = a..b)` berechnet die definite Summe über i von a bis b . Ist $b - a$ eine nicht-negative ganze Zahl, so wird die explizite Summe $f(a) + f(a+1) + \dots + f(b)$ zurückgeliefert.

Ist $b - a$ eine negative ganze Zahl, so wird das Negative des Ergebnisses von `sum(f, i = b+1..a-1)` zurückgeliefert. Mit dieser Konvention ist die Regel

$$\text{sum}(f, i = a..b) + \text{sum}(f, i = b+1..c) = \text{sum}(f, i = a..c)$$
für beliebige a, b und c erfüllt.

⌘ `sum(f, i = RootOf(p, x))` berechnet die Summe, in der sich i über alle Wurzeln des Polynoms p bezüglich x erstreckt.

Ist f eine rationale Funktion in i , so wird eine explizite Darstellung der Summe gefunden.

Siehe Beispiel ??.

- ☞ Das System liefert einen symbolischen `sum`-Aufruf zurück, wenn keine geschlossene Darstellung der Summe gefunden wurde.
- ☞ Unendliche symbolische Reihen ohne symbolische Parameter können mittels `float` oder `numeric::sum` numerisch approximiert werden. Siehe Beispiel ??.

Beispiel 1. Einige unbestimmte Summen werden berechnet:

```
>> sum(1/(i^2 - 1), i)
```

$$-\frac{1}{2i} - \frac{1}{2(i-1)}$$

```
>> sum(1/i/(i + 2)^2, i)
```

$$\frac{\text{psi}(i+2, 1)}{2} - \frac{1}{4i} - \frac{1}{4i+4}$$

```
>> sum(binomial(n + i, i), i)
```

$$\frac{i \text{ binomial}(i+n, i)}{n+1}$$

```
>> sum(binomial(n, i)/2^n - binomial(n + 1, i)/2^(n + 1), i)
```

$$\frac{2i \text{ binomial}(n, i) - i \text{ binomial}(n+1, i)}{2^{\frac{n}{2}} - 4i 2^{\frac{n}{2}} + 2n 2^{\frac{n}{2}}}$$

Einige bestimmte Summen werden berechnet. Man beachte, dass $\pm\infty$ zulässige Summationsgrenzen sind:

```
>> sum(1/(i^2 + 21*i), i = 1..infinity)
```

18858053/108636528

```
>> sum(1/i, i = a .. a + 3)
```

$$-\frac{1}{a} + \frac{1}{a+1} + \frac{1}{a+2} + \frac{1}{a+3}$$

Beispiel 2. Wir berechnen einige Summen über alle Wurzeln eines Polynoms:

```
>> sum(i^2, i = RootOf(x^3 + a*x^2 + b*x + c, x))
```

$$a^2 - 2 b$$

```
>> sum(1/(z + i), i = RootOf(x^4 - y*x + 1, x))
```

$$\frac{y^3 + 4 z^3}{y^4 z + z^4 + 1}$$

Beispiel 3. sum kann endliche Summen berechnen, deren Grenzen ganze Zahlen vom Typ DOM_INT sind:

```
>> sum(1/(i^2 + i), i = 1..100)
```

$$100/101$$

```
>> sum(binomial(n, i), i = 0..4)
```

$$n + \text{binomial}(n, 2) + \text{binomial}(n, 3) + \text{binomial}(n, 4) + 1$$

```
>> expand(%)
```

$$\frac{7}{12} n^7 + \frac{11}{24} n^6 - \frac{1}{12} n^3 + \frac{1}{24} n^4 + 1$$

In solchen Fällen ist allerdings die Verwendung von `_plus` meist effizienter:

```
>> _plus(1/(i^2 + i) $ i = 1..100)
```

$$100/101$$

```
>> _plus(binomial(n, i) $ i = 0..4)
```

$$n + \text{binomial}(n, 2) + \text{binomial}(n, 3) + \text{binomial}(n, 4) + 1$$

Bei symbolischen Grenzen kann `_plus` nicht verwendet werden:

```
>> _plus(1/(i^2 + i) $ i = 1..n)
```

```
Error: Illegal argument [_seggen]
```

```
>> _plus(binomial(n, i) $ i = 0..n)
```

Error: Illegal argument [_seqgen]

```
>> sum(1/(i^2 + i), i = 1..n), sum(binomial(n, i), i = 0..n)
```

$$\frac{n}{n+1}, \frac{n}{2}$$

Beispiel 4. Die folgende unbestimmte Summe kann nicht symbolisch berechnet werden:

```
>> sum(ln(i)/i^5, i = 1..infinity)
```

$$\text{sum} \left| \frac{\ln(i)}{i^5}, i = 1..infinity \right|$$

Mittels `float` erhält man eine Gleitpunktnäherung:

```
>> float(%)
```

0.0285737805

Alternativ kann die Funktion `numeric::sum` aufgerufen werden. Dies ist üblicherweise viel effizienter als die Anwendung von `float`, da hierbei die meist kostenaufwendige symbolische Berechnung mittels `sum` vermieden wird:

```
>> numeric::sum(ln(i)/i^5, i = 1..infinity)
```

0.0285737805

Beispiel 5. `sum` findet keine geschlossene Form für die folgende definite Summe und liefert eine Rekursionsformel (siehe `rec`) zurück:

```
>> sum(binomial(n, i)^3, i = 0..n)
```

$$\text{solve} \left| \text{rec} \left| u2(n+2) - \frac{8 u2(n) (n+1)^2}{(n+2)^2} \right. \right|$$

$$\frac{u2(n+1) (21 n^2 + 7 n + 16)}{-----}, u2(n), \{u2(0) = 1, u2(1) = 2\}$$

$$(n + 2)^2$$

$$\begin{array}{c} \backslash \quad \backslash \\ | \quad | \\ | \quad | \\ | \quad | \\ / \quad / \end{array}$$

Hintergründe:

- ☞ Die Funktion `sum` implementiert die Algorithmen von Abramov für rationale Ausdrücke, von Gosper für hypergeometrische Ausdrücke und von Zeilberger für die bestimmte Summation holonomer Ausdrücke.

Änderungen:

- ☞ Keine Änderungen.

`sysname` – der Name des Betriebssystems

`sysname()` liefert den Typ des Betriebssystems, unter dem MuPAD aktuell ausgeführt wird.

Aufruf(e):

- ☞ `sysname(<Arch>)`

Optionen:

Arch — veranlasst die Rückgabe genauerer Angaben zur Architektur

Rückgabewert: eine Zeichenkette.

Verwandte Funktionen: `system`

Details:

- ☞ `sysname()` liefert eine der folgenden Zeichenketten:
 - "UNIX" für UNIX- und Linux-Betriebssysteme,
 - "MSDOS" für MSDOS-Betriebssysteme, eingeschlossen MS-Windows,
 - "MACOS" für Apple Macintosh-Betriebssysteme.

☞ `sysname(Arch)` liefert einen spezifischeren Namen des Betriebssystems als Zeichenkette. Steht diese Information auf der benutzten Architektur nicht zur Verfügung, so wird dieselbe Zeichenkette wie beim Aufruf `sysname()` geliefert.

☞ `sysname` ist eine Funktion des Systemkerns.

Beispiel 1. Unter einem MS-DOS- oder MS-Windows-Betriebssystem liefert `sysname` die folgenden Werte:

```
>> sysname(), sysname(Arch)
      "MSDOS", "MSDOS"
```

Beispiel 2. Unter einem aktuellen Linux-Betriebssystem wie z. B. Linux 2.0 mit libc-6.0 liefert `sysname` die folgenden Werte:

```
>> sysname(), sysname(Arch)
      "UNIX", "linux"
```

Beispiel 3. Unter einem Solaris-Betriebssystem (SunOS, Sun Microsystems) liefert `sysname` die folgenden Werte:

```
>> sysname(), sysname(Arch)
      "UNIX", "Solaris"
```

Beispiel 4. Unter einem Apple Macintosh-Betriebssystem liefert `sysname` die folgenden Werte:

```
>> sysname(), sysname(Arch)
      "MACOS", "MACOS"
```

Änderungen:

☞ Keine Änderungen.

`sysorder` – Vergleich von Objekten bezüglich der internen Ordnung

`sysorder(object1, object2)` liefert `TRUE`, wenn MuPADs interne Ordnung des Objekts `object1` kleiner oder gleich der Ordnung des Objekts `object2` ist. Anderenfalls wird `FALSE` geliefert.

Aufruf(e):

```
# sysorder(object1, object2)
```

Parameter:

object1, object2 — beliebige MuPAD-Objekte

Rückgabewert: TRUE oder FALSE.

Verwandte Funktionen: `_less`, `listlib::removeDupSorted`, `sort`

Details:

- # Es existiert eine eindeutige interne Ordnung für fast alle Objekte, die in einer MuPAD Sitzung erzeugt werden. `sysorder` vergleicht zwei Objekte bezüglich dieser Ordnung.

Die Ausnahmen sind Mengen, Domains und Tabellen. Für sie existiert keine eindeutige interne Ordnung. Damit haben auch Objekte, die Mengen, Domains oder Tabellen als (Sub-)Operanden enthalten, keine eindeutige interne Ordnung. Siehe Beispiel ??.



- # Man sollte nicht versuchen, Objekte mit dieser internen Ordnung nach spezifischen Kriterien zu sortieren. So spiegelt z. B. die interne Ordnung nicht die natürliche Anordnung von Zahlen oder Zeichenketten wieder. Weiterhin kann sich diese Ordnung zwischen verschiedenen MuPAD-Versionen unterscheiden.

Die Eindeutigkeit ist die einzige Eigenschaft, die verwendet werden sollte. Siehe Beispiel ??.

- # `sysorder` ist eine Funktion des Systemkerns.

Beispiel 1. Es wird hier an ein paar Beispielen gezeigt, wie sich die interne Ordnung in der aktuellen MuPAD Version verhält. Bei nicht-negativen ganzen Zahlen entspricht sie der natürlichen Ordnung, während dies bei rationalen oder negativen ganzen Zahlen nicht zutrifft:

```
>> sysorder(3, 4) = bool(3 <= 4),
    sysorder(45, 33) = bool(45 <= 33),
    sysorder(0, 4) = bool(0 <= 4)

    TRUE = TRUE, FALSE = FALSE, TRUE = TRUE

>> sysorder(1/3, 1/4) <> bool(1/3 <= 1/4),
    sysorder(-4, 2) <> bool(-4 <= 2),
    sysorder(-4, -2) <> bool(-4 <= -2)

    TRUE <> FALSE, FALSE <> TRUE, FALSE <> TRUE
```

Beispiel 2. Für Zeichenketten oder Bezeichnernamen ist die interne Ordnung nicht lexikographisch:

```
>> sysorder("abc", "baa"), sysorder("abc", "bab"),
      sysorder("abc", "bac")

      FALSE, FALSE, TRUE

>> sysorder(abc, baa), sysorder(abc, bab), sysorder(abc, bac)

      FALSE, FALSE, TRUE
```

Beispiel 3. Für Mengen und Tabellen existiert keine eindeutige interne Ordnung:

```
>> sysorder({1, 2, 3}, {4, 5, 6}), sysorder({4, 5, 6}, {1, 2, 3})

      FALSE, FALSE

>> sysorder(table("a" = 42), table("a" = 43)),
      sysorder(table("a" = 43), table("a" = 42))

      FALSE, FALSE
```

Beispiel 4. Eine einfache Anwendung von `sysorder` wird gezeigt. Es soll eine Funktion `f` implementiert werden, deren einzige bekannte Eigenschaft die Schiefsymmetrie $f(-x) = -f(x)$ ist. Ausdrücke, die Aufrufe von `f` enthalten, sollten soweit wie möglich automatisch vereinfacht werden, z. B. sollte $f(x) + f(-x)$ für jedes Argument x automatisch zu 0 werden. Zur Implementation einer entsprechenden „Normalform“ benutzen wir `sysorder` zur Entscheidung, ob der Aufruf $f(x)$ das Ergebnis $f(x)$ oder $-f(-x)$ zurückliefert:

```
>> f := proc(x) begin
      if sysorder(x, -x) then
        return(-procname(-x))
      else return(procname(x))
      end_if;
    end_proc;
```

Für numerische Argumente scheint `f` positive Werte zu bevorzugen:

```
>> f(-3), f(3), f(-4.5), f(4.5), f(-2/3), f(2/3)

      -f(3), f(3), -f(4.5), f(4.5), -f(2/3), f(2/3)
```

Für andere Argumente ist das Ergebnis schwer vorhersagbar:

```
>> f(x), f(-x), f(sqrt(2) + 1), f(-sqrt(2) - 1)
      1/2      1/2
-f(-x), f(-x), - f(- 2 - 1), f(- 2 - 1)
```

Mit dieser Implementation werden Ausdrücke mit `f` automatisch vereinfacht:

```
>> f(x) + f(-x) - f(3)*f(x) + f(-3)*f(-x) + sin(f(7)) + sin(f(-
7))
```

0

```
>> delete f:
```

Änderungen:

☞ Die interne Ordnung hat sich mit der neuen Version geändert.

system – Ausführen eines Betriebssystemkommandos

`system("command")` führt das Betriebssystemkommando oder Programm `command` aus.

Aufruf(e):

☞ `system("command")`
☞ `!command`

Parameter:

"command" — ein Kommando des Betriebssystems oder ein Programmname als MuPAD-Zeichenkette

Rückgabewert: der „Fehlercode“: eine ganze Zahl.

Verwandte Funktionen: `sysname`

Details:

☞ `!command` ist fast äquivalent zu `system("command")`: der Aufruf `!command` liefert jedoch keinen Wert an die MuPAD-Sitzung zurück.

☞ `system` ist nicht in allen MuPAD-Versionen verfügbar. Speziell wird diese Funktion unter Windows-Betriebssystemen nicht unterstützt. Gegebenenfalls wird folgende Fehlermeldung ausgegeben:

```
Error: Function not available for this client [system].
```

☞ `system("command")` leitet das Kommando "command" an das Betriebssystem weiter. Beispielsweise können hiermit andere Anwendungsprogramme aus MuPAD heraus gestartet werden.

Für die Terminal-Version gilt weiterhin: Ein erfolgreicher Aufruf liefert den Wert 0. Anderenfalls wird ein ganzzahliger Fehlercode zurückgegeben, dessen Wert vom Betriebssystem und vom Kommando abhängt.

☞ Wenn das aufgerufene Kommando auf UNIX Systemen nach `stderr` schreibt, so gehen diese Ausgaben auf MuPADs `stderr`. Wenn `system` in XMuPAD aufgerufen wird, so wird die Ausgabe in die Shell umgeleitet, aus der heraus XMuPAD aufgerufen wurde.

☞ `system` ist eine Funktion des Systemkerns.

Beispiel 1. Unter UNIX oder Linux wird das Systemkommando `date` aufgerufen. Die Kommandoausgabe wird auf dem Bildschirm angezeigt, der Fehlercode 0 für erfolgreiches Durchführen des Kommandos wird in die MuPAD-Sitzung zurückgegeben:

```
>> errorcode := system("date"):
```

```
Fri Sep 29 14:42:13 MEST 2000
```

```
>> errorcode
```

```
0
```

Nun wird das `date`-Kommando mit der Kommandozeilenoption `'+%m'` aufgerufen, um nur den aktuellen Monat anzeigen zu lassen:

```
>> errorcode := system("date '+%m'"):
```

```
09
```

Vergisst man den Präfix `'+'` in der Kommandozeilenoption von `date`, so liefert `date` und somit auch `system` einen Fehlercode. Beachten Sie, dass die Fehlerausgabe nach `stderr` geht:

```
>> system("date '%m'")
```

```
date: invalid date '%m'
```

```
1
```

```
>> delete errorcode:
```

Beispiel 2. Auf die Ausgabe eines Programms, das mit dem `system`-Befehl gestartet wird, kann in MuPAD nicht direkt zugegriffen werden, aber die Ausgabe kann in eine Datei umgeleitet und dann mit dem `read` oder `ftextinput`-Befehl gelesen werden:

```
>> system("echo communication example > comm_file"):
      ftextinput("comm_file")

      "communication example"

>> system("rm -f comm_file"):
```

Änderungen:

☞ Keine Änderungen.

table – Erzeugen einer Tabelle

`table()` erzeugt eine neue leere Tabelle.

`table(index1 = entry1, index2 = entry2, ...)` erzeugt eine neue Tabelle mit den gegebenen Indizes und den zugeordneten Einträgen `entry1` etc.

Aufruf(e):

☞ `table()`
☞ `table(index1 = entry1, index2 = entry2, ...)`

Parameter:

`index1, index2, ...` — die Indizes: beliebige MuPAD-Objekte
`entry1, entry2, ...` — die zugeordneten Werte: beliebige MuPAD-Objekte

Rückgabewert: ein Objekt vom Typ `DOM_TABLE`.

Verwandte Funktionen: `_assign`, `_index`, `array`, `assignElements`, `delete`, `DOM_ARRAY`, `DOM_LIST`, `DOM_TABLE`, `indexval`

Details:

☞ Tabellen sind die wohl flexibelsten MuPAD-Objekte zur Speicherung von Daten. Im Gegensatz zu Arrays oder Listen können beliebige Objekte als Indizes verwendet werden. Indizierte Zugriffe sind schnell und fast unabhängig von der Tabellengröße. Damit sind Tabellen auch für die Speicherung großer Datenmengen geeignet.

- ⌘ Ein indizierter Aufruf der Form `T[index]` liefert den zum Index gehörenden Eintrag einer Tabelle `T`. Existiert kein Eintrag zum angegebenen Index, so wird der indizierte Ausdruck `T[index]` symbolisch zurückgeliefert.
- ⌘ Eine indizierte Zuweisung der Form `T[index] := entry` fügt einen weiteren Eintrag in eine existierende Tabelle `T` ein bzw. überschreibt einen bereits vorhandenen Eintrag.
- ⌘ `table` dient zur „expliziten“ Erzeugung von Tabellen. Alternativ existiert folgender Mechanismus zur „impliziten“ Erzeugung von Tabellen:
Ist der Wert eines Bezeichners `T` weder eine Tabelle noch ein Array oder eine Liste, so ist eine indizierte Zuweisung `T[index] := entry` äquivalent zu `T := table(index = entry)`. d.h., durch die indizierte Zuweisung wird implizit eine neue Tabelle mit einem Eintrag erzeugt. Siehe Beispiel ??.
Falls der Wert von `T` vor einer indizierten Zuweisung eine Tabelle, ein Array oder eine Liste war, so trägt die Zuweisung nur den neuen Wert ein, ohne den Typ von `T` implizit zu verändern.
- ⌘ Tabelleneinträge können mit der Funktion `delete` gelöscht werden. Siehe Beispiel ??.
- ⌘ `table` ist eine Funktion des Systemkerns.

Beispiel 1. Der folgende Aufruf erzeugt eine Tabelle mit zwei Einträgen:

```
>> T := table(a = 13, c = 42)

      table(
        c = 42,
        a = 13
      )
```

Die eingetragenen Daten können durch indizierte Aufrufe abgerufen werden. Man beachte das symbolische Ergebnis für den Index `b`, zum dem kein Tabelleneintrag existiert:

```
>> T[a], T[b], T[c]

      13, T[b], 42
```

Tabelleneinträge können durch indizierte Zuweisungen verändert werden:

```
>> T[a] := T[a] + 10: T

      table(
        c = 42,
        a = 23
      )
```

Auch Ausdrucksfolgen können als Indizes bzw. Werte benutzt werden. Sie müssen bei Eingabe mittels `table` jedoch zusätzlich geklammert werden:

```
>> T := table((a, b) = "hello", a + b = (50, 70))

      table(
        a + b = (50, 70),
        (a, b) = "hello"
      )

>> T[a + b]

      50, 70
```

Indizierter Zugriff braucht keine zusätzliche Klammerung:

```
>> T[a, b] := T[a, b]." world": T

      table(
        a + b = (50, 70),
        (a, b) = "hello world"
      )

>> delete T:
```

Beispiel 2. Die folgende indizierte Zuweisung mit einem Bezeichner `T` ohne Wert verwandelt `T` implizit in eine Tabelle:

```
>> delete T: T[4] := 7: T

      table(
        4 = 7
      )

>> delete T:
```

Beispiel 3. Tabelleneinträge können mit `delete` gelöscht werden:

```
>> T := table(a = 1, b = 2, (a, b) = (1, 2))

      table(
        (a, b) = (1, 2),
        b = 2,
        a = 1
      )

>> delete T[b], T[a, b]: T
```

```
table(
  a = 1
)
```

```
>> delete T:
```

Änderungen:

- ⌘ In früheren Versionen führte eine Zuweisung des Wertes `NIL` an einen Tabelleneintrag zum Löschen des Eintrags. Nun wird `NIL` dem Tabelleneintrag zugewiesen, gelöscht wird mittels `delete`.
-

taylor – Berechnung einer Taylor-Reihenentwicklung

`taylor(f, x = x0)` berechnet die ersten Terme der Taylor-Reihe von `f` bezüglich der Variablen `x` um den Punkt `x0`.

Aufruf(e):

⌘ `taylor(f, x < = x0> <, order>)`

Parameter:

- `f` — ein arithmetischer Ausdruck, als Funktion in `x` zu interpretieren
- `x` — ein Bezeichner
- `x0` — der Entwicklungspunkt: ein arithmetischer Ausdruck; ohne Angabe dieses Punktes wird der Entwicklungspunkt 0 benutzt.
- `order` — die Anzahl der zu berechnenden Terme: eine nicht-negative ganze Zahl; die Standardordnung ist durch die Umgebungsvariable `ORDER` mit dem voreingestellten Wert 6 gegeben.

Rückgabewert: ein Objekt vom Domain-Typ `Series::Puisseux` oder ein Ausdruck vom Typ `"taylor"`.

Seiteneffekte: Die Ergebnisse der Funktion hängen vom Wert der Umgebungsvariablen `ORDER` ab, die standardmäßig die Anzahl der Terme in Reihenentwicklungen bestimmt.

Überladbar durch: `f`

Verwandte Funktionen: `asympt`, `diff`, `limit`, `O`, `series`, `Series::Puisseux`, `Type::Series`

Details:

- ☞ `taylor` versucht, die Taylor-Entwicklung von `f` um `x = x0` zu bestimmen. Drei Fälle können dabei auftreten:
 1. `taylor` ist in der Lage, die entsprechende Taylor-Entwicklung zu bestimmen. In diesem Fall ist das Ergebnis ein Ausdruck vom Domain-Typ `"taylor"`. Mittels `expr` kann das Ergebnis in einen arithmetischen Ausdruck vom Domain-Typ `DOM_EXPR` konvertiert werden. Siehe Beispiel ??.
 2. `taylor` konnte feststellen, dass die entsprechende Taylor-Entwicklung nicht existiert. In diesem Fall wird eine Fehlermeldung ausgegeben. Siehe Beispiel ??.
 3. `taylor` ist nicht in der Lage festzustellen, ob die entsprechende Taylor-Entwicklung existiert oder nicht. In diesem Fall wird der Funktionsaufruf mit evaluierten Argumenten zurückgeliefert. Das Ergebnis ist dann ein Ausdruck vom Typ `"taylor"`. Siehe Beispiel ??.

☞ Mathematisch ist die von `taylor` berechnete Entwicklung in einer Umgebung des Entwicklungspunktes in der komplexen Ebene gültig.

☞ Falls `x0` den Wert `infinity` bzw. `-infinity` hat, wird eine Reihenentwicklung längs der reellen Achse von links an das positive reelle Unendliche bzw. von rechts an das negative reelle Unendliche berechnet. Siehe Beispiel ??.

Solche eine Reihenberechnung wird wie folgt bestimmt: Die Variable `x` in `f` wird durch $x = 1/u$ ersetzt. Anschließend wird eine gerichtete Reihenentwicklung von `f` um $u = 0+$ bestimmt und zuletzt die Variable `u` im Ergebnis durch $u = 1/x$ rücks substituiert.

Das Ergebnis einer solchen Reihenberechnung ist im mathematischen Sinn eine Potenzreihe in $1/x$.

☞ Das optionale Argument `order` bestimmt die Anzahl der Terme der Entwicklung. Ohne Angabe von `order` wird der Wert der Umgebungsvariablen `ORDER` benutzt, deren Standardwert 6 durch Zuweisung an `ORDER` verändert werden kann.

Die Anzahl der Terme der Entwicklung wird vom Term mit dem kleinsten Grad an gezählt, d.h. „order“ ist als „relative Abbruchordnung“ anzusehen.

Es kann vorkommen, dass die Anzahl der Terme in der berechneten Reihenentwicklung von der geforderten Anzahl abweicht (siehe Beispiel ??). Siehe `series` für Details.



☞ `taylor` verwendet die Systemfunktion `series` zur Taylor-Entwicklung. Für eine detaillierte Beschreibung der Parameter und der zurückgelieferten Datenstruktur sei auf die Hilfeseite von `series` verwiesen.

Beispiel 1. Eine Taylor-Entwicklung um den Nullpunkt wird berechnet:

```
>> s := taylor(exp(x^2), x)
```

$$1 + x^2 + \frac{x^4}{2} + O(x^6)$$

Das Ergebnis von `taylor` ist vom folgenden Domain-Typ:

```
>> domtype(s)
```

Series::Puisseux

Wenn wir die Funktion `expr` auf eine Reihe anwenden, erhalten wir den entsprechenden arithmetischen Ausdruck ohne den Fehlerterm zurück:

```
>> expr(s); domtype(%)
```

$$x^2 + \frac{x^4}{2} + 1$$

DOM_EXPR

```
>> delete s:
```

Beispiel 2. Eine Taylor-Entwicklung von $f(x) = \frac{1}{x^2-1}$ um den Punkt $x = 1$ existiert nicht, und `taylor` antwortet daher mit einer entsprechenden Fehlermeldung:

```
>> taylor(1/(x^2 - 1), x = 1)
```

```
Error: does not have a Taylor series expansion, try 'series' [\ntaylor]
```

Dem Ratschlag der Fehlermeldung folgend wird versucht, mittels `series` eine verallgemeinerte Reihenentwicklung zu berechnen. In der Tat existiert eine Laurent-Entwicklung:

```
>> series(1/(x^2 - 1), x = 1)
```

$$\frac{1}{2(x-1)} - \frac{1}{4} + \frac{x}{8} - \frac{1}{8} + \frac{(x-1)^2}{16} + \frac{(x-1)^3}{32} + O((x-1)^4)$$

Beispiel 3. Kann eine Taylor-Entwicklung nicht berechnet werden, so wird der Funktionsaufruf symbolisch mit einem Hinweis an den Benutzer zurückgeliefert:

```
>> taylor(1/exp(x^a), x = 0)

Warning: could not compute Taylor series expansion; try 'serie\
s' with option 'Left', 'Right', or 'Real' for a more gene-
ral e\
xpansion [taylor]
```

$$\text{taylor} \left| \frac{1}{\exp(x^a)}, x = 0 \right|$$

In diesem Beispiel liefert auch `series` unter Angabe einer der vorgeschlagenen Optionen nur einen symbolischen Funktionsaufruf zurück.

Es folgt ein weiteres Beispiel, für das keine Taylor-Entwicklung berechnet werden kann. In diesem Fall jedoch ergibt `series` mit optionalem Argument eine allgemeinere Entwicklung:

```
>> taylor(psi(1/x), x = 0)

Warning: could not compute Taylor series expansion; try 'serie\
s' with option 'Left', 'Right', or 'Real' for a more gene-
ral e\
xpansion [taylor]
```

$$\text{taylor} \left| \psi \left(\frac{1}{x} \right), x = 0 \right|$$

```
>> series(psi(1/x), x = 0, Right)
```

$$\ln \left| \frac{1}{x} \right| - \frac{x}{2} - \frac{x^2}{12} - \frac{x^4}{120} + O(x^5)$$

Beispiel 4. Hier ist ein Beispiel für eine „gerichtete“ Taylor-Entwicklung längs der reellen Achse um infinity:

```
>> taylor(exp(1/x), x = infinity)
```

$$1 + \frac{1}{x} + \frac{1}{2x^2} + \frac{1}{6x^3} + \frac{1}{24x^4} + \frac{1}{120x^5} + O\left(\frac{1}{x^6}\right)$$

Beispiel 5. Hier ist ein Beispiel, für das die Anzahl der berechneten Terme von der angeforderten Anzahl abweicht:

```
>> taylor((sin(x^4) - tan(x^4)) / x^10, x, 15)
```

$$-\frac{x^2}{2} + O(x^5)$$

Änderungen:

- ⌘ taylor liefert einen Ausdruck vom Typ "taylor", falls keine Reihenentwicklung berechnet werden konnte (was nicht notwendigerweise bedeutet, dass eine Taylor-Entwicklung nicht existiert).
- ⌘ Ein Fehler wird ausgelöst, falls eine angeforderte Taylor-Entwicklung nicht existiert.

tbl2text – Konkatenation von Zeichenketten in einer Tabelle

tbl2text konkateniert alle Einträge einer Tabelle von Zeichenketten.

Aufruf(e):

```
⌘ tbl2text(strtab)
```

Parameter:

strtab — eine Tabelle von Zeichenketten

Rückgabewert: eine Zeichenkette.

Verwandte Funktionen: `_concat`, `coerce`, `expr2text`, `int2text`, `text2expr`, `text2list`, `text2tbl`

Details:

- ⌘ Die Tabelle muss durch 1, 2, 3 etc. indiziert sein, alle Einträge müssen Zeichenketten sein. Sie werden in der Reihenfolge ihrer Indizes verkettet.
 - ⌘ `tbl2text` fügt durch `text2tbl` zerlegte Texte wieder zusammen.
 - ⌘ `tbl2text` ist eine Funktion des Systemkerns.
-

Beispiel 1. Eine Zeichenkette kann aus beliebig vielen Tabelleneinträgen aufgebaut werden:

```
>> tbl2text(table(1 = "Hell", 2 = "o", 3 = " ", 4 = "world."))  
"Hello world."
```

Änderungen:

- ⌘ Keine Änderungen.
-

tcoeff – der niedrigste Koeffizient eines Polynoms

`tcoeff(p)` gibt den niedrigsten Koeffizienten des Polynoms `p` zurück.

Aufruf(e):

- ⌘ `tcoeff(p <, vars> <, order>)`

Parameter:

- `p` — ein Polynom vom Typ `DOM_POLY` oder ein polynomialer Ausdruck
- `vars` — eine Liste der Unbestimmten des Polynoms: typischerweise Bezeichner oder indizierte Bezeichner
- `order` — die Termordnung: entweder *LexOrder* oder *DegreeOrder* oder *DegInvLexOrder* oder eine benutzerdefinierte Termordnung vom Typ `Dom::MonomOrdering`. Der Standard ist die lexikographische Ordnung *LexOrder*.

Rückgabewert: ein Element des Koeffizientenrings des Polynoms oder `FAIL`.

Überladbar durch: `p`

Verwandte Funktionen: `coeff`, `collect`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`

Details:

- ⌘ Das Argument `p` kann entweder ein polynomialer Ausdruck sein oder ein mittels `poly` erzeugtes Polynom oder ein Element eines Polynom-Datentyps, der `tcoeff` überlädt.
 - ⌘ Wird eine Liste von Unbestimmten übergeben, so wird `p` als Polynom in diesen Unbestimmten angesehen. Man beachte, dass diese Liste nicht mit den Unbestimmten in `p` übereinzustimmen braucht. Siehe Beispiel ??.
 - ⌘ Der zurückgelieferte Koeffizient ist der „niedrigste“ bezüglich der lexikographischen Ordnung, falls nicht mittels des Arguments `order` eine andere Ordnung angegeben wird. Siehe Beispiel ??.
 - ⌘ Das Ergebnis von `tcoeff` wird nicht weiter evaluiert. Vollständige Evaluation kann mit der Funktion `eval` erzwungen werden. Siehe Beispiel ??.
 - ⌘ `tcoeff` liefert `FAIL`, wenn das Eingabepolynom nicht in ein Polynom in den angegebenen Unbekannten konvertiert werden kann. Siehe Beispiel ??.
 - ⌘ Für die Ordnung *LexOrder* ruft `tcoeff` eine schnelle Kernfunktion auf. Andere Ordnungen werden durch langsamere Bibliotheksfunktionen behandelt.
-

Beispiel 1. Es wird gezeigt, wie die Unbestimmten das Ergebnis beeinflussen:

```
>> p := 2*x^2*y + 3*x*y^2:
      tcoeff(p), tcoeff(p, [x, y]), tcoeff(p, [y, x])
                                     2, 3, 2
```

Beachten Sie, dass die an `tcoeff` übergebenen Unbestimmten verwendet werden, auch wenn das Polynom sie gar nicht enthält:

```
>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
      tcoeff(p), tcoeff(p, [x, y]), tcoeff(p, [y, x]),
      tcoeff(p, [y]), tcoeff(p, [z])
                                     2      2      2
      3, 3, 2, 2 x , 2 x y + 3 x y
>> delete p:
```

Beispiel 2. Es wird demonstriert, wie verschiedene Ordnungen das Ergebnis beeinflussen:

```
>> p := poly(5*x^4 + 4*x^3*y + 3*x^2*y^3*z, [x, y, z]):
      tcoeff(p), tcoeff(p, DegreeOrder), tcoeff(p, DegInvLexOrder)

      3, 4, 5
```

Der folgende Aufruf benutzt die umgekehrte lexikographische Termordnung in 3 Unbestimmten:

```
>> tcoeff(p, Dom::MonomOrdering(RevLex(3)))

      5

>> delete p:
```

Beispiel 3. tcoeff evaluiert sein Ergebnis nicht vollständig:

```
>> p := poly(27*x^2 + a*x, [x]): a := 5:
      tcoeff(p, [x]), eval(tcoeff(p, [x]))

      a, 5

>> delete p, a:
```

Beispiel 4. Es wird ein Polynom über dem Restklassenring modulo 7 definiert:

```
>> p := poly(3*x, [x], Dom::IntegerMod(7)): tcoeff(p)

      3 mod 7
```

Dieses Polynom kann nicht als Polynom in einer anderen Unbestimmten angesehen werden, da der „Koeffizient“ $3 \cdot x$ nicht als Element des Koeffizientenrings `Dom::IntegerMod(7)` interpretiert werden kann:

```
>> tcoeff(p, [y])

      FAIL

>> delete p:
```

Änderungen:

- ☞ Selbstdefinierte Termordnungen können nun vorgegeben werden.
 - ☞ Unbestimmte können nun auch für Polynome vom Domain-Typ `DOM_POLY` angegeben werden.
 - ☞ In früheren MuPAD-Versionen war `tcoeff` eine Kernfunktion.
-

`testargs` – Entscheidung, ob Prozedurargumente zu testen sind

Innerhalb einer Prozedur gibt `testargs` an, ob die Prozedur interaktiv aufgerufen wurde.

Aufruf(e):

- ☞ `testargs()`
- ☞ `testargs(b)`

Parameter:

`b` — `TRUE` oder `FALSE`

Rückgabewert: `TRUE` oder `FALSE`.

Verwandte Funktionen: `proc`, `testtype`, `Pref::typeCheck`

Details:

- ☞ Das Überprüfen von Argumenten eines Prozeduraufrufs kann kostspielig sein. Die meisten Funktionen der MuPAD-Bibliotheken verfahren daher gemäß der folgenden Philosophie:

Wird eine Prozedur auf interaktiver Ebene aufgerufen, so sind die Argumente zu überprüfen, da diese interaktiv vom Nutzer eingegeben wurden. Entsprechen die Argumente nicht der dokumentierten Eingabespezifikation, so sind aussagekräftige Fehlermeldungen auszugeben, um den Nutzer über den falschen Gebrauch der Funktion zu informieren.

Wird eine Prozedur von einer anderen Prozedur aufgerufen, so brauchen die Argumente zur Effizienzsteigerung nicht geprüft zu werden. Die aufrufende Funktion ist dafür verantwortlich, nur zulässige Parameter zu übergeben.

`testargs` ist dabei das Hilfsmittel, um festzustellen, ob die Argumente zu testen sind: bei einem Aufruf innerhalb einer Prozedur liefert `testargs()` den Wert `TRUE`, wenn die Prozedur interaktiv aufgerufen wurde. Anderenfalls wird `FALSE` geliefert.

☞ `testargs` kennt zwei Modi: im „Standard-Modus“ ist die Funktionalität wie oben beschrieben. Im „Debug-Modus“ liefert ein Aufruf `testargs()` immer `TRUE`. Dies unterstützt das Debuggen von Prozeduren: jede `testargs` benutzende Funktion überprüft seine Argumente und liefert aussagekräftigere Fehlermeldungen, wenn falsche Argumente übergeben wurden.

Der Aufruf `testargs(TRUE)` schaltet in den „Debug-Modus“, d. h., Parametertests werden global aktiviert.

Der Aufruf `testargs(FALSE)` schaltet in den „Standard-Modus“, d. h., Parametertests werden nur bei interaktiven Prozeduraufrufen durchgeführt.

Der Aufruf `testargs(b)` liefert dabei den vorher gesetzten Wert zurück.

☞ Das Überprüfen von Argumenten eines Prozeduraufrufs kann auch mit der Funktion `Pref::typeCheck` gesteuert werden.

☞ `testargs` ist eine Funktion des Systemkerns.

Beispiel 1. Das folgende Beispiel demonstriert, wie `testargs` innerhalb eines Prozedurrumpfs benutzt werden sollte. Die Funktion `p` soll eine Folge von `n` Nullen erzeugen und erwartet dazu eine positive ganze Zahl `n` als Argument:

```
>> p := proc(n)
  begin
    if testargs() then
      if not testtype(n, Type::PosInt) then
        error("expecting a positive integer");
      end_if;
    end_if;
    return(0 $ n)
  end_proc;
```

Das tatsächlich übergebene Argument wird überprüft, wenn `p` interaktiv aufgerufen wird:

```
>> p(13/2)

Error: expecting a positive integer [p]
```

Das Argument wird nicht getestet, wenn `p` aus einer Prozedur heraus aufgerufen wird. Die folgende Fehlermeldung stammt nicht von `p`, sondern entsteht beim Versuch, `0 $ n` zu erzeugen:

```
>> f := proc(n) begin p(n) end_proc: f(13/2)
```

```
Error: Illegal argument [_seqgen];  
during evaluation of 'p'
```

Der „Debug-Modus“ von `testargs` wird aktiviert:

```
>> testargs(TRUE):
```

Nun wird auch bei einem nicht-interaktiven Aufruf von `p` eine aussagekräftige Fehlermeldung ausgegeben:

```
>> f(13/2)
```

```
Error: expecting a positive integer [p]
```

Beim Aufräumen wird der „Standard-Modus“ von `testargs` wieder eingeschaltet:

```
>> testargs(FALSE): delete f, g:
```

Änderungen:

☞ Keine Änderungen.

`testtype` – syntaktische Typüberprüfung

`testtype(object, T)` prüft, ob das Objekt `object` syntaktisch vom Typ `T` ist.

Aufruf(e):

☞ `testtype(object, T)`

Parameter:


`object` — ein beliebiges MuPAD-Objekt
`T` — ein Typobjekt

Rückgabewert: `TRUE` oder `FALSE`.

Überladbar durch: `object, T`

Verwandte Funktionen: `coerce, domtype, hastype, is, type, Type`

Details:

- ⌘ Das Typobjekt `T` kann ein Domain-Typ wie z. B. `DOM_INT`, `DOM_EXPR` etc. sein, oder eine Zeichenkette, wie sie von der Funktion `type` geliefert wird, oder ein Objekt der Type-Bibliothek. Letztere sind im allgemeinen die sinnvollsten Objekte für das Argument `T`.
- ⌘ `testtype` führt einen rein syntaktischen Test durch. Mittels `is` können semantische Tests durchgeführt werden, die auch Eigenschaften von Bezeichnern berücksichtigen. 
- ⌘ Details zum Überladungsmechanismus sind in der Sektion „Hintergründe“ dieser Hilfeseite aufgeführt.
- ⌘ `testtype` ist eine Funktion des Systemkerns.

Beispiel 1. Der folgende Aufruf testet, ob das erste Argument ein Ausdruck ist. Ausdrücke sind elementare Objekte des MuPAD-Kerns vom Domain-Typ `DOM_EXPR`:

```
>> testtype(x + y, DOM_EXPR)

TRUE
```

Die Funktion `type` unterscheidet Ausdrücke nach ihrem 0-ten Operanden. Die entsprechende Zeichenkette ist ein gültiges Typobjekt für `testtype`:

```
>> type(x + y), testtype(x + y, "_plus")

"_plus", TRUE
```

Der folgende Aufruf testet, ob das erste Argument eine ganze Zahl ist. Hierzu wird abgefragt, ob der Domain-Typ `DOM_INT` ist:

```
>> testtype(7, DOM_INT)

TRUE
```

Man beachte, dass `testtype` einen rein syntaktischen Test durchführt. Zwar ist 7 mathematisch als ganze Zahl auch eine rationale Zahl, der entsprechende Domain-Typ `DOM_RAT` umfasst jedoch nicht `DOM_INT`:

```
>> testtype(7, DOM_RAT)

FALSE
```

Die Typobjekte der Type-Bibliothek sind flexibler. Beispielsweise stellt der Typ `Type::Rational` die Vereinigung von `DOM_INT` und `DOM_RAT` dar:


```
>> testtype(7, Type::Rational)
```

```
TRUE
```

Auch andere Typisierungen sind für die Zahl 7 gültig:

```
>> testtype(7, Type::PosInt), testtype(7, Type::Prime),  
    testtype(7, Type::Numeric), testtype(7, Type::Odd)
```

```
TRUE, TRUE, TRUE, TRUE
```

Beispiel 2. Untertypen von Ausdrücken können mittels Zeichenketten spezifiziert werden:

```
>> type(f(x)), type(sin(x))
```

```
"function", "sin"
```

```
>> testtype(sin(x), "function"), testtype(sin(x), "sin"),  
    testtype(sin(x), "cos")
```

```
TRUE, TRUE, FALSE
```

Beispiel 3. Es wird demonstriert, wie ein eigenes Typobjekt implementiert werden kann. Dazu wird eine neues Domain `div3` erzeugt, dessen „testtype“-Attribut testet, ob ein Objekt ein ganzzahliges Vielfaches von 3 ist:

```
>> div3 := newDomain("divisible by 3?"):  
    div3 := slot(div3, "testtype",  
                proc(x) begin  
                    return(testtype(x/3, Type::Integer))  
                end_proc):
```

Durch den Überladungsmechanismus ruft das Kommando `testtype(object, div3)` dieses Attribut auf:

```
>> testtype(5, div3), testtype(6, div3), testtype(sin(1), div3)
```

```
FALSE, TRUE, FALSE
```

```
>> delete div3:
```

Hintergründe:

- ⌘ Die Überladung von `testtype` arbeitet folgendermaßen: Zunächst wird geprüft, ob `domtype(object) = T` oder `type(object) = T` gilt. Ist dies der Fall, so wird `TRUE` zurückgegeben.
- ⌘ Als nächstes wird die Methode "`testtype`" des Domains `object::dom` mit den Argumenten `object`, `T` aufgerufen. Liefert diese Methode einen anderen Wert als `FAIL`, so wird dieser Wert von `testtype` zurückgegeben.
- ⌘ Existiert die Methode `object::dom::testtype` nicht bzw. liefert sie `FAIL`, so wird Überladung durch das zweite Argument benutzt:
 - Ist `T` ein Domain, so wird die Methode "`testtype`" von `T` mit den Argumenten `object`, `T` aufgerufen.
 - Ist `T` kein Domain, so wird die Methode "`testtype`" von `T::dom` mit den Argumenten `object`, `T` aufgerufen.

Änderungen:

- ⌘ Keine Änderungen.
-

`text2expr` – Umwandlung eines Textes in einen Ausdruck

`text2expr(text)` interpretiert die Zeichenkette `text` als MuPAD-Eingabe und erzeugt das entsprechende Objekt.

Aufruf(e):

- ⌘ `text2expr(text)`

Parameter:

`text` — eine Zeichenkette

Rückgabewert: ein MuPAD-Objekt.

Verwandte Funktionen: `coerce`, `expr2text`, `input`, `int2text`, `tbl2text`, `text2int`, `text2list`, `text2tbl`

Details:

- ⌘ Die Zeichenkette muss einer syntaktisch korrekten MuPAD-Eingabe entsprechen, anderenfalls liefert `text2expr` einen Fehler. Typischerweise können Zeichenketten, die mittels `expr2text` aus MuPAD-Objekten erzeugt wurden, durch `text2expr` wieder in entsprechende Objekte zurückverwandelt werden.

- ☞ Das Objekt wird zurückgegeben, ohne weiter evaluiert zu werden. Eine Evaluation kann durch die Funktion `eval` erzwungen werden.
- ☞ Die Zeichenkette `text` braucht nicht mit einem „`i`“ oder einem „`:`“ abgeschlossen sein.
- ☞ `text2expr` ist eine Funktion des Systemkerns.

Beispiel 1. Eine Zeichenkette wird in einen einfachen Ausdruck konvertiert. Der erzeugte Ausdruck wird nicht automatisch evaluiert:

```
>> text2expr("21 + 21")
```

$$21 + 21$$

Die Evaluation wird durch eval erzwungen:

```
>> eval(%)
```

42

Beispiel 2. Eine Zeichenkette wird in eine Anweisungsfolge konvertiert:

```
>> text2expr("x:= 3; x + 2 + 1"); eval(%)
```

```
(x := 3;  
x + 2 + 1)
```

6

>> x

3

```
>> delete x:
```

Beispiel 3. Eine Matrix wird in eine Zeichenkette verwandelt:

```
>> matrix([[a11, a12], [a21, a22]])
```

```

+-          +-
|   a11, a12 |
|             |
|   a21, a22 |
+-          +-

```

```
>> expr2text(%)
```

```
"Dom::Matrix()(array(1..2, 1..2, (1,1) = a11, (1,2) = a12, (2,\n1) = a21, (2,2) = a22))"
```

Die Zeichenkette wird in eine Matrix zurück verwandelt:

```
>> text2expr(%)
```

```
Dom::Matrix()(array(1..2, 1..2, (1, 1) = a11, (1, 2) = a12,\n\n(2, 1) = a21, (2, 2) = a22))
```

```
>> eval(%)
```

```
+-      +-
|  a11, a12  |
|            |
|  a21, a22  |
+-      +-
```

Änderungen:

☞ Keine Änderungen.

text2int – Umwandlung eines Textes in eine ganze Zahl

text2int(text, b) konvertiert eine Zeichenkette, die einer ganzen Zahl in b-adischer Darstellung entspricht, in eine ganze Zahl vom Typ DOM_INT.

Aufruf(e):

☞ text2int(text <, b>)

Parameter:

text — eine Zeichenkette
b — die Basis: eine ganze Zahl zwischen 2 und 36. Die Standardbasis ist 10.

Rückgabewert: eine ganze Zahl.

Verwandte Funktionen: coerce, expr2text, genpoly, int2text, numlib::g_adic, tbl2text, text2expr, text2list, text2tbl

Details:

☞ Der Text wird als b -adische Darstellung interpretiert, die Zeichen müssen aus den ersten b Elementen der Folge $0, 1, \dots, 9, A, B, \dots, Z$ bestehen. Auch kleine Buchstaben a, b, \dots, z werden akzeptiert. Für Basen größer als 10 dienen die Buchstaben zur Darstellung der b -adischen Ziffern größer als 9: $a = A = 10, \dots, z = Z = 35$.

☞ `text2int` ist die Inverse der Funktion `int2text`.

☞ `text2int` ist eine Funktion des Systemkerns.

Beispiel 1. Bezüglich der Standardbasis 10 führt `text2int` eine reine Konvertierung des Datentyps von `DOM_STRING` nach `DOM_INT` aus:

```
>> text2int("123"), text2int("-45678")  
  
123, -45678
```

Beispiel 2. Die Zeichen der Zeichenkette werden als Ziffern zur angegebenen Basis interpretiert, der Rückgabewert ist eine im Dezimalsystem ausgegebene übliche MuPAD-Zahl. Das folgende Beispiel konvertiert eine zur Basis 2 bzw. 16 dargestellte Zahl in eine „normale“ Zahl, die im Dezimalsystem ausgegeben wird:

```
>> text2int("101", 2), text2int("101", 16)  
  
5, 257
```

Die Ziffer „3“ existiert in einer Binärdarstellung nicht:

```
>> text2int("103", 2)  
  
Error: Illegal argument [text2int]
```

Beispiel 3. Für Basen größer als 10 stellen Buchstaben die b -adischen Ziffern größer 9 dar:

```
>> text2int("3B9ACA00", 16), text2int("Z", 36) = text2int("z", 36)  
  
1000000000, 35 = 35
```

Änderungen:

☞ Keine Änderungen.

`text2list`, `text2tbl` – Zerlegen eines Textes

`text2list` zerlegt eine Zeichenkette in eine Liste von Teilzeichenketten.

`text2tbl` zerlegt eine Zeichenkette in eine Tabelle von Teilzeichenketten.

Aufruf(e):

☞ `text2list(text, separators <, Cyclic>)`

☞ `text2tbl(text, separators <, Cyclic>)`

Parameter:

`text` — der zu analysierende Text: eine Zeichenkette
`separators` — Trennzeichen: eine Liste von Zeichenketten. Die leere Zeichenkette "" wird nicht als Trennzeichen akzeptiert.

Optionen:

Cyclic — die Trennzeichenliste wird zyklisch benutzt

Rückgabewert: eine Liste bzw. eine Tabelle von Zeichenketten.

Verwandte Funktionen: `coerce`, `expr2text`, `int2text`, `tbl2text`, `text2expr`, `text2int`

Details:

☞ Beide Funktionen zerlegen eine Zeichenkette in Teilzeichenketten, wobei die Zeichenketten in der Liste `separators` als Trennzeichen verwendet werden. `text2list` liefert eine Liste mit den Teilzeichenketten zurück, `text2tbl` eine durch 1, 2, 3 etc. indizierte Tabelle.

☞ Ohne die Option *Cyclic* wird der Text wie folgt zerlegt. Das erste Auftreten eines Trennzeichens aus der Liste `separators` wird gesucht. Wird kein Trennzeichen gefunden, wird der gesamte Text als einzige Teilzeichenkette zurückgeliefert. Anderenfalls definiert die Teilzeichenkette bis zum Trennzeichen die erste Teilzeichenkette, das Trennzeichen selbst ist die zweite Teilzeichenkette. Der verbleibende Text wird analog zerlegt.

Ohne die Option *Cyclic* hängt das Ergebnis nicht von der Reihenfolge der Trennzeichen ab.

☞ Mit der Option *Cyclic* wird zunächst das erste Trennzeichen in der Liste *separators* benutzt, um die erste Teilzeichenkette zu finden. Das Trennzeichen selbst ist die zweite Teilzeichenkette. Dann wird das zweite Trennzeichen in der Liste *separators* benutzt, um die dritte Teilzeichenkette zu finden usw.

Nach dem letzten Trennzeichen der *separators*-Liste wird die Suche erneut mit dem ersten Trennzeichen fortgesetzt, bis der Text vollständig zerlegt ist oder das aktuelle Trennzeichen nicht auftritt.

Mit der Option *Cyclic* hängt das Ergebnis von der Reihenfolge der Trennzeichen ab.

☞ `tbl2text` fügt durch `text2tbl` zerlegte Texte wieder zusammen.

☞ `text2list`, `text2tbl` sind Funktionen des Systemkerns.

Beispiel 1. Das folgende Beispiel demonstriert den Unterschied zwischen einem Aufruf der Funktion `text2list` mit und ohne *Cyclic*:

```
>> text2list("This is a simple example!", ["is", "mp"])
["Th", "is", " ", "is", " a si", "mp", "le exa", "mp", "le!"]
>> text2list("This is a simple example!", ["is", "mp"], Cyclic)
["Th", "is", " is a si", "mp", "le example!"]
```

Beispiel 2. Das folgende Beispiel demonstriert den Unterschied zwischen einem Aufruf der Funktion `text2tbl` mit und ohne *Cyclic*:

```
>> text2tbl("This is a simple example!", ["is", "mp"])
      table(
        9 = "le!",
        8 = "mp",
        7 = "le exa",
        6 = "mp",
        5 = " a si",
        4 = "is",
        3 = " ",
        2 = "is",
        1 = "Th"
      )
>> text2tbl("This is a simple example!", ["is", "mp"], Cyclic)
```

```

table(
    5 = "le example!",
    4 = "mp",
    3 = " is a si",
    2 = "is",
    1 = "Th"
)

```

Änderungen:

☞ Keine Änderungen.

textinput – interaktive Eingabe von Text

textinput ermöglicht die interaktive Eingabe von Text.

Aufruf(e):

```

☞ textinput(<prompt1>)
☞ textinput(<prompt1,> x1, <prompt2,> x2, ...)

```

Parameter:

prompt1, prompt2, ...	— Eingabeaufforderungen: Zeichenketten
x1, x2, ...	— Bezeichner

Rückgabewert: die letzte Eingabe, konvertiert in eine Zeichenkette.

Verwandte Funktionen: finput, fprintf, fread, ftextinput, input, print, read, text2expr, write

Details:

- ☞ textinput() liefert die Eingabeaufforderung „Please enter text :“ und wartet auf Eingaben vom Benutzer. Die Eingabe wird in eine Zeichenkette verwandelt und als Funktionswert zurückgegeben.
- ☞ textinput(prompt1) benutzt die Zeichenkette prompt1 statt der Standardeingabeaufforderung „Please enter text :“.
- ☞ textinput(<prompt1,> x1) konvertiert die Eingabe in eine Zeichenkette und weist diese dem Bezeichner x1 zu. Hierbei wird die Standardeingabeaufforderung verwendet, falls keine Zeichenkette prompt1 übergeben wird.

- ☞ Mehrere Eingabewerte können mit einem `textinput`-Befehl eingelesen werden. Jeder Bezeichner in der Argumentenfolge veranlasst `textinput`, eine Eingabeaufforderung auszugeben und auf eine Eingabe zu warten, die dann dem Bezeichner zugewiesen wird. Eine direkt vor dem Bezeichner übergebene Zeichenkette ersetzt dabei die Standardeingabeaufforderung. Siehe Beispiel ?? . Argumente, die weder Zeichenketten noch Bezeichner sind, werden ignoriert.
- ☞ In der Terminal-Version MuPADs muss die Eingabe mit dem Steuerzeichen `<CTRL-D>` bzw. `<STRG-D>` beendet werden (dabei muss der Cursor hinter dem letzten Zeichen der aktuellen Eingabezeile stehen). Graphische Benutzeroberflächen von MuPAD öffnen gegebenenfalls ein separates Fenster für die Eingabe.
- ☞ Die Eingabe kann über mehrere Zeilen gehen. MuPAD benutzt das Zeichen `\n` (Zeilenvorschub) in der Ausgabezeichenkette, um den Zeilenvorschub darzustellen.
- ☞ Eingabezeichen mit einem führenden `\` werden nicht als Steuerzeichen, sondern als zwei einzelne Zeichen interpretiert.
- ☞ Die Bezeichner `x1` etc. dürfen Werte haben. Diese werden durch `textinput` überschrieben.
- ☞ `textinput` ist eine Funktion des Systemkerns.

Beispiel 1. Die Standardeingabeaufforderung wird ausgegeben, die Eingabe wird als Zeichenkette zurückgeliefert:

```
>> textinput()

Please enter text input: << myinput >>

"myinput"
```

Beispiel 2. Eine selbstdefinierte Eingabeaufforderung wird benutzt, die Eingabe wird dem Bezeichner `x` zugewiesen:

```
>> textinput("enter your name: ", x)

enter your name: << Turing >>

"Turing"

>> x

"Turing"

>> delete x:
```

Beispiel 3. Beim Einlesen mehrerer Werte kann für jeden Wert eine eigene Eingabeaufforderung angegeben werden:

```
>> textinput("She: ", hername, "He:  ", hisname)
```

```
She: << Bonnie >>
```

```
He:  << Clyde >>
```

```
"Clyde"
```

```
>> hername, hisname
```

```
"Bonnie", "Clyde"
```

```
>> delete hername, hisname:
```

Änderungen:

⌘ Keine Änderungen.

mtime, time – Messen von Realzeit und CPU-Zeit

`mtime()` liefert die gesamte Realzeit in Millisekunden, die in der gegenwärtigen MuPAD-Sitzung verbraucht wurde.

`mtime(a1, a2, ...)` liefert die Realzeit für die Auswertung aller Argumente.

`time()` liefert die CPU-Gesamtrechenzeit, die in der gegenwärtigen MuPAD-Sitzung verbraucht wurde.

`time(a1, a2, ...)` liefert die CPU-Zeit für die Auswertung aller Argumente.

Aufruf(e):

⌘ `mtime()`

⌘ `mtime(a1, a2, ...)`

⌘ `time()`

⌘ `time(a1, a2, ...)`

Parameter:

`a1, a2, ...` — beliebige MuPAD-Objekte

Rückgabewert: eine nichtnegative ganz Zahl

Verwandte Funktionen: `prog::profile`

Details:

- ☞ `rtime` liefert die vergangene Realzeit in Millisekunden. Die letzten drei Ziffern sind im Moment jedoch immer 0, d. h., die von `rtime` gemessene Zeit ist nur auf eine Sekunde genau.
 - ☞ Der von `time()` zurückgelieferte Wert umfasst die gesamte CPU-Rechenzeit, die von MuPAD in irgendeiner Weise verbraucht wurde. Dies schließt Systeminitialisierung sowie das Einlesen von Eingaben ein. Die Rechenzeit anderer zur gleichen Zeit laufenden Programme ist nicht enthalten, selbst wenn sie mittels `system` von MuPAD aus gestartet wurden.
 - ☞ Der von `time` zurückgegebene Wert wird abhängig vom verwendeten Betriebssystem meist dadurch berechnet, dass die in die Ausführungszeit fallenden Takte der Systemuhr gezählt werden. Das Ergebnis ist damit ein Vielfaches einer Taktlänge, welche die Genauigkeit der Zeitmessung festlegt. Auf vielen UNIX-Rechnern ist diese Taktlänge 10 Millisekunden.
 - ☞ Auf Rechnern ohne „time sharing“, wie z. B. dem Macintosh, stimmen CPU-Zeit und Realzeit im Wesentlichen überein.
 - ☞ `rtime` und `time` sind Funktionen des Systemkerns.
-

Beispiel 1. Dieses Beispiel zeigt, wie man Zeitmessungen und den Zugriff auf Ergebnisse kombinieren kann. Man beachte, dass eine Zuweisung in zusätzliche Klammern eingeschlossen werden muss, wenn sie als Argument an eine Funktion übergeben wird:

```
>> rtime((a := int(exp(x)*sin(x), x)))
                        9000

>> a
               sin(x) exp(x)   cos(x) exp(x)
               ----- - -----
                   2           2

>> delete a:
```

Alternativ können Zeitmessungen für mehrere Befehle in der folgenden Weise durchgeführt werden:

```
>> t0 := rtime():
    command1
    command2
    ...
    rtime() - t0
```

Beispiel 2. Hier wird `rttime` verwendet, um die Stunden, Minuten und Sekunden zu berechnen, die seit dem Beginn dieser Sitzung vergangen sind:

```
>> t := rttime()/1000:
    h := trunc(t/3600):
    m := trunc(t/60 - h*60):
    s := t - m*60 - h*3600:

>> print(Unquoted, "This session is running for " .
          h . " hours, " . m . " minutes and " .
          s . " seconds.")

This session is running for 0 hours, 0 minutes and 10 seconds.

>> delete t, h, m, s:
```

Beispiel 3. Um eine aussagekräftigere Ausgabe zu erhalten, kann man die gemessene Zeit mit der passenden Zeiteinheit multiplizieren:

```
>> time((a := isprime(2^1000 - 1)))*msec

700 msec

>> time((a := isprime(2^1000 - 1))*sec/1000.0

0.7 sec

>> delete a:
```

Hintergründe:

- ☞ Auf UNIX-Systemen wird die Zeit mittels Aufrufs der Systemfunktion `time` gemessen.

Änderungen:

- ☞ Keine Änderungen.

traperror – Abfangen von Fehlern

`traperror(object)` fängt bei der Auswertung von `object` auftretende Fehler ab.

`traperror(object, t)` tut dasselbe, bricht die Auswertung jedoch ab, falls sie nach `t` Sekunden Realzeit noch nicht beendet ist.

Aufruf(e):

```
# traperror(object)
# traperror(object, t)
```

Parameter:

object — ein beliebiges MuPAD-Objekt
 t — die Zeitvorgabe: eine nichtnegative ganze Zahl

Rückgabewert: eine nichtnegative ganze Zahl.

Verwandte Funktionen: error, prog::error, lasterror

Details:

- # traperror fängt etwaige Fehler ab, die bei der Auswertung von object auftreten. Syntaktische Fehler, d. h., Fehler beim Einlesen von object, sowie Fehler, die zur Beendigung von MuPAD führen, können nicht abgefangen werden.
- # traperror liefert den Fehlercode 0, falls kein Fehler aufgetreten ist. Bei Überschreiten der Zeit t wird der Fehlercode 1320 geliefert (Execution time exceeded). Der Fehlercode ist 1028, wenn der Fehler mittels der Funktion error ausgelöst wurde.
- # Wenn traperror keine Zeitvorgabe gesetzt hat und ein 'Execution time exceeded'-Fehler durch eine umschließende traperror(..., t)-Anweisung ausgelöst wird, dann wird dieser Fehler nicht durch das innere traperror aufgefangen. Er wird erst von demjenigen traperror-Aufruf abgefangen, der die Zeitvorgabe gesetzt hat. Siehe Beispiel ??.
- # Das auszuwertende Objekt kann eine Zuweisung sein, die aus syntaktischen Gründen in zusätzlichen Klammern zu übergeben ist. Das folgende Codestück demonstriert eine typische Anwendung von traperror:

```
if traperror((x := EineFehleranfaelligeFunktion()) <> 0) then
    MacheEtwasMit(x);
else ReagiereAufDenFehler();
end_if;
```

- # Mit lasterror können abgefangene Fehler erneut ausgelöst werden.
 - # traperror ist eine Funktion des Systemkerns.
-

Beispiel 1. Fehler, die während der Ausführung von Kernfunktionen auftreten, können unterschiedliche Fehlercodes haben, die vom Problem abhängen. Beispielsweise erzeugt ‚Division by zero‘ den Fehlercode 1025:

```
>> y := 1/x: traperror(subs(y, x = 0))
```

1025

```
>> lasterror()
```

Error: Division by zero [_power]

Der folgende Versuch, eine riesige Gleitpunktzahl zu berechnen, scheitert an numerischem Überlauf. Der entsprechende Fehlercode ist 20:

```
>> traperror(exp(12345678.9))
```

20

```
>> lasterror()
```

Error: Overflow/underflow in arithmetical operation;
during evaluation of 'exp::float'

Beispiel 2. Alle mittels der Funktion `error` ausgelösten Fehler haben den Fehlercode 1028. Dies sind die Fehler, die während der Ausführung von Bibliotheksfunktionen auftreten:

```
>> traperror(error("My error!"))
```

1028

```
>> lasterror()
```

Error: My error!

Beispiel 3. Es wird versucht, ein Polynom zu faktorisieren; die Berechnung wird nach 10 Sekunden abgebrochen:

```
>> traperror(factor(x^1000 + 4*x + 1), 10)
```

1320

```
>> lasterror()
```

Error: Execution time exceeded;
during evaluation of 'faclib::univ_mod_gcd'

Beispiel 4. Hier sind zwei geschachtelte `traperror`-Aufrufe. Der innere Aufruf enthält eine nicht-terminierte Schleife und der äußere Aufruf hat eine Zeitvorgabe von 2 Sekunden gesetzt. Wenn die Ausführungszeit abgelaufen ist, wird dieser spezielle Fehler nicht vom inneren `traperror`-Aufruf abgefangen. Wegen des Fehlers wird `print(1)` nie aufgerufen:

```
>> traperror((traperror((while TRUE do 1 end)); print(1)), 2)

1320

>> lasterror()

Error: Execution time exceeded
```

Änderungen:

⌘ Keine Änderungen.

type – der Typ eines Objekts

`type(object)` liefert den Typ des Objektes `object`.

Aufruf(e):

⌘ `type(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: ein Domain-Typ vom Typ `DOM_DOMAIN` oder eine Zeichenkette.

Überladbar durch: `object`

Verwandte Funktionen: `coerce`, `domtype`, `hastype`, `testtype`, `Type`

Details:

⌘ Ist `object` nicht ein Ausdruck vom Domain-Typ `DOM_EXPR`, so sind `type(object)` und `domtype(object)` äquivalent, d. h., `type` liefert den Datentyp des Objekts.

- ⌘ Ist `object` ein Ausdruck vom Datentyp `DOM_EXPR`, so bestimmt der 0-te Operand (der „Operator“) den Typ. Hat der Operator ein `"type"`-Attribut, so wird dieses von `type` zurückgeliefert: Dies ist für gewöhnlich eine Zeichenkette. Hat der Operator kein solches Attribut, so liefert `type` die Zeichenkette `"function"`.
 - ⌘ Im Gegensatz zu den meisten anderen Funktionen werden als Argument an `type` übergebene Ausdrucksfolgen nicht ausgeglichen. Siehe Beispiel ??.
 - ⌘ `type` ist eine Funktion des Systemkerns.
-

Beispiel 1. Ist ein Objekt kein Ausdruck, so ist sein Typ gleich seinem Datentyp:

```
>> type(3)

DOM_INT
```

Beispiel 2. Der Operator einer Summe ist `_plus`; dessen Typattribut ist `"_plus"`:

```
>> type(x + y*z)

"_plus"
```

Die Funktion `type` wertet ihr Argument aus. Dadurch wird aus der Differenz von `x` und `y` die Summe von `x` und `(-1)*y`; ihr Typ ist nicht mehr `"_subtract"`, sondern `"_plus"`:

```
>> type(x - y)

"_plus"
```

Beispiel 3. Ist der Operator eines Ausdrucks keine Funktionsumgebung, die ein Typattribut hat, so ist der Ausdruck vom Typ `"function"`:

```
>> type(f(2))

"function"
```

Beispiel 4. Der folgende Aufruf von `type` ist kein Aufruf mit zwei Argumenten, da Ausdrucksfolgen im Argument nicht ausgeglichen werden:

```
>> type((2, 3))

"_exprseq"
```


Änderungen:

- ☞ Keine Änderungen.
-

unassume – Löschen der Eigenschaften von Bezeichnern

`unassume(x)` löscht die Eigenschaften des Bezeichners `x`.

Aufruf(e):

- ☞ `unassume(x)`
- ☞ `unassume(<Global>)`

Parameter:

`x` — ein Bezeichner oder eine Liste oder Menge von Bezeichnern

Optionen:

`Global` — löscht die „globale Eigenschaft“

Rückgabewert: das leere Objekt `null()`.

Verwandte Funktionen: `assume`, `delete`, `getprop`, `is`

Details:

- ☞ `unassume` dient zum Löschen von mittels `assume` gesetzten Eigenschaften von Bezeichnern. `?property` liefert eine kurze Beschreibung des „property“-Mechanismus.
 - ☞ Wird eine Liste oder Menge von Bezeichnern angegeben, so werden die Eigenschaften aller angegebenen Bezeichner gelöscht.
 - ☞ Die Aufrufe `unassume()` und `unassume(Global)` sind äquivalent. Hiermit wird die „globale Eigenschaft“ gelöscht, die allen Bezeichnern zugeordnet wird. Siehe `assume` für weitere Details zum Setzen einer globalen Eigenschaft.
 - ☞ Der Befehl `delete x` löscht neben dem Wert des Bezeichners `x` auch seine Eigenschaften.
-

Beispiel 1. Den Bezeichnern x und y werden Eigenschaften zugeordnet:

```
>> assume(x > 0): assume(y < 0): getprop(x), getprop(y)
                                     > 0, < 0
```

```
>> sign(x), sign(y)
                                     1, -1
```

unassume oder delete löschen die Eigenschaften:

```
>> unassume(x): delete y: getprop(x), getprop(y)
                                     x, y
```

```
>> sign(x), sign(y)
                                     sign(x), sign(y)
```

Durch die Angabe einer Liste oder Menge können die Eigenschaften mehrerer Bezeichner mit einem Aufruf von unassume gelöscht werden:

```
>> assume(x > y): unassume([x, y]): getprop(x), getprop(y)
                                     x, y
```

Beispiel 2. Alle Bezeichner sollen reelle Zahlen darstellen. Die entsprechende globale Eigenschaft wird gesetzt:

```
>> assume(Global, Type::Real): getprop(x), getprop(y), getprop(z)
                                     Type::Real, Type::Real, Type::Real
```

```
>> Re(x), Im(y), Re(x*y*z)
                                     x, 0, x y z
```

unassume() oder unassume(Global) löscht die globale Eigenschaft:

```
>> unassume(): Re(x), Im(y), Re(x*y*z)
                                     Re(x), Im(y), Re(x y z)
```

Änderungen:

- ⌘ Ein Aufruf ohne Argumente löscht die globale Eigenschaft.
-

universe – die Allklasse (die Menge aller Objekte)

universe repräsentiert die Allklasse, d. h. die Menge aller Objekte.

Verwandte Funktionen: `_union`, `_intersect`, `_minus`, `DOM_SET`

Details:

- ⌘ universe ist ein Objekt vom Domain-Typ `stdlib::Universe`. Es gibt keine weiteren Objekte diesen Typs.
 - ⌘ Die üblichen Mengenoperationen wie Vereinigung, Schnitt und Subtraktion von Mengen verarbeiten universe.
-

Beispiel 1. Wir zeigen einige elementare Mengenoperationen mit universe:

```
>> universe union {a}
                                universe
>> universe intersect {a}
                                {a}
>> {a} minus universe
                                {}
```

Änderungen:

- ⌘ universe ist ein neues Objekt.
-

unloadmod – Ausladen eines Moduls

`unloadmod("modulename")` lädt das dynamische Modul namens `modulename` aus.

`unloadmod()` versucht, alle derzeit geladenen dynamischen Module auszuladen.

Aufruf(e):

```
# unloadmod("modulename" <, Force>)  
# unloadmod()
```

Parameter:

`modulename` — der Name eines Moduls: eine Zeichenkette

Optionen:

Force — zwingt den Modul-Manager, ein *statisches* Modul auszuladen.

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Seiteneffekte: Beim Ausladen des Maschinencodes eines Moduls bleibt das Modul-Domain unangetastet. Bei einem späteren Zugriff auf dieses Modul-Domain wird der entsprechende Maschinencode bei Bedarf automatisch wieder eingeladen. Die Funktion `reset` lädt alle dynamischen Module aus.

Weitere Dokumentation: Dynamic Modules - User's Manual and Programming Guide for MuPAD 1.4, Andreas Sorgatz, Okt 1998, Springer Verlag, Heidelberg, mit CD-ROM, ISBN 3-540-65043-1.

Verwandte Funktionen: `external`, `loadmod`, `module::displace`, `module::new`, `unexport`

Details:

- # `unloadmod("modulename")` lädt den Maschinencode des Moduls aus dem MuPAD-Prozess und dem Arbeitsspeicher.
- # `unloadmod` bricht mit einer Fehlermeldung ab, wenn versucht wird, ein *statisches* Modul ohne die Option *Force* auszuladen.
- # `unloadmod` ist eine Funktion des Systemkerns.

Beispiel 1. Dynamische Module können zur Laufzeit ausgeladen werden, um Speicher-Ressourcen zu sparen oder um die Module zu ändern und neu zu übersetzen.

```
>> loadmod("stdmod"): unloadmod():
```

Der Maschinencode wird bei Bedarf automatisch wieder eingeladen:

```
>> stdmod::which("stdmod")  
  
"/usr/local/mupad/linux/modules/stdmod.mdm"
```

Hintergründe:

- ☞ Die Kernfunktionen `external`, `loadmod` und `unloadmod` sind Basisfunktionen zum Zugriff auf Module. Weitere Funktionen stehen in der Bibliothek `module` zur Verfügung.
- ☞ Wird eine Modulfunktion nach dem Ausladen oder Verdrängen ihres Maschinencodes aufgerufen, so wird der entsprechende Maschinencode automatisch wieder eingeladen. Hierbei wird, im Gegensatz zum erneuten Laden des Moduls mittels `loadmod`, das Modul-Domain nicht verändert.
- ☞ Einige wenige Betriebssysteme unterstützen das Ausladen von Maschinencode zur Laufzeit nicht. Der Einsatz von dynamischen Modulen wird hierdurch aber nicht eingeschränkt.

Änderungen:

- ☞ Keine Änderungen.
-

`unprotect` – Aufheben des Schreibschutzes von Bezeichnern

`unprotect(x)` entfernt jeglichen Schreibschutz des Bezeichners `x`.

Aufruf(e):

- ☞ `unprotect(x)`

Parameter:

`x` — ein Bezeichner

Rückgabewert: die zuletzt gültige Schutzstufe für `x`: entweder *Error* oder *Warning* oder *None* (siehe `protect`).

Verwandte Funktionen: `protect`

Details:

- ☞ `unprotect(x)` ist äquivalent zu `protect(x, None)`.
 - ☞ `unprotect` evaluiert seine Argumente nicht. Siehe Beispiel ??.
-

Beispiel 1. `unprotect` erlaubt es, Systemfunktionen Werte zuzuweisen:

```
>> unprotect(sign): sign(x) := 1
```

1

Es wird jedoch davon abgeraten, vom System geschützte Bezeichner mit Werten zu versehen. Die obige Zuweisung wird rückgängig gemacht:

```
>> delete sign(x): protect(sign, Error):
```

Beispiel 2. `unprotect` evaluiert seine Argumente nicht. Hier wird der Schreibschutz von Bezeichner `x` entfernt und nicht von dessen Wert `y`:

```
>> x := y: protect(y): unprotect(x): y := 1
```

Warning: protected variable y overwritten

1

```
>> unprotect(y): delete x, y:
```

Änderungen:

☞ Keine Änderungen.

`userinfo` – Ausgabe von Informationen über den Programmablauf

`userinfo(n, message)` gibt Informationen auf dem Bildschirm aus, falls mittels `setuserinfo` ein Informationsgrad von mindestens `n` für die gegenwärtig ausgeführte Prozedur gesetzt wurde.

`userinfo(n1..n2, message)` gibt Informationen aus, wenn der mittels `setuserinfo` gesetzte Informationsgrad zwischen `n1` und `n2` liegt.

Aufruf(e):

☞ `userinfo(<Text,> n, message1, message2, ...)`

☞ `userinfo(<Text,> n1..n2, message1, message2, ...)`

Parameter:

`n, n1, n2`

— die Informationsgrade:
nichtnegative ganze Zahlen

`message1, message2, ...`

— beliebige MuPAD-Objekte.
Typischerweise Zeichenketten.

Optionen:

Text — die Argumente werden bei der Ausgabe nicht durch Kommata getrennt

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Seiteneffekte: Die Ausgabeformatierung von `userinfo` ist von der Umgebungsvariablen `TEXTWIDTH` abhängig.

Verwandte Funktionen: `print`, `setuserinfo`, `warning`

Details:

- ☞ `userinfo` darf nicht auf interaktiver Ebene, sondern sollte im Rumpf einer Prozedur oder einer Domain-Methode verwendet werden, um Statusinformationen über benutzte Algorithmen, Zwischenergebnisse etc. auszugeben. Ist ein `userinfo`-Befehl in eine Funktion namens `f` eingebaut, so wird er durch Setzen eines geeigneten Informationsgrades mittels `setuserinfo(f, n)` aktiviert. Die Informationen werden bei folgenden Aufrufen der Funktion `f` ausgegeben.
 - ☞ Die Ausgabe besteht aus den evaluierten Argumenten `message1` etc., eventuell gefolgt vom Namen der Prozedur (siehe `setuserinfo`). Zeichenketten werden ohne Anführungszeichen ausgegeben. Der Pretty-Printer wird nicht benutzt. Die Argumente werden in der Ausgabe durch Kommata getrennt, wenn die Option *Text* nicht angegeben ist.
 - ☞ Der Informationsgrad einer einzelnen Prozedur, aller Methoden eines Domains oder aller Prozeduren überhaupt kann mittels `setuserinfo` gesetzt werden. Demnach können bis zu drei verschiedene Informationsgrade auf eine Prozedur zutreffen.
 - ☞ Die meisten Bibliotheksfunktionen von **MuPAD** verwenden `userinfo`, um Informationen auszugeben. Siehe Beispiel ??.
 - ☞ `userinfo` ist eine Funktion des Systemkerns.
-

Beispiel 1. Die Funktion `expr2text` ist nützlich, um **MuPAD**-Objekte in einen Text einzufügen:

```
>> f := proc(x)
      begin
        userinfo(2, "the argument is " . expr2text(x));
        x^2
      end_proc:

>> setuserinfo(f, 2, Name): f(12)
```

```
Info: the argument is 12 [f]
```

144

```
>> setuserinfo(f, 0): delete f:
```

Beispiel 2. Ein Aufruf von `userinfo(n, message)` bewirkt, dass `message` auch dann angezeigt wird, falls der Informationsgrad höher als `n` ist; soll sie nur angezeigt werden, falls der Informationsgrad genau gleich `n` ist, kann man `userinfo` mit einem Bereich aufrufen, der nur aus einem Punkt besteht:

```
>> f := proc()
      begin
        userinfo(2..2, "Infolevel = 2");
        userinfo(2, "Infolevel >= 2");
      end_proc:
```

```
>> setuserinfo(f, 2): f():
```

```
Info: Infolevel = 2
Info: Infolevel >= 2
```

```
>> setuserinfo(f, 3): f():
```

```
Info: Infolevel >= 2
```

```
>> setuserinfo(f, 0): delete f:
```

Beispiel 3. Setzt man den Informationsgrad von `faclib` auf 5, so erhält man Informationen über die aufgerufenen Faktorisierungsalgorithmen:

```
>> setuserinfo(faclib, 5): factor(x^2 + 2*x + 1)

Info: faclib::monomial called with poly(x^2 + 2*x + 1, [x])
Info: Squarefree factorization (Yun's algorithm) called
```

$$(x + 1)^2$$

```
>> setuserinfo(faclib, 0):
```

Hintergründe:

- ⌘ Die Informationen werden nur ausgewertet, wenn sie ausgegeben werden.
- ⌘ Durch den Aufruf `setuserinfo()` erhält man die globale Tabelle aller gesetzten Informationsgrade.

Änderungen:

- ⌘ `userinfo` gibt jetzt den Prozedurnamen in einem anderem Format aus.
 - ⌘ Die Option *Pretty* wurde entfernt. `userinfo` verwendet den Pretty-Printer nicht.
 - ⌘ Die Option *Text* wurde eingeführt.
 - ⌘ `userinfo` ist jetzt eine Kernfunktion.
-

`val` – der Wert eines Objektes

`val(object)` ersetzt jeden Bezeichner in `object` durch seinen Wert.

Aufruf(e):

- ⌘ `val(object)`

Parameter:

`object` — ein beliebiges MuPAD-Objekt

Rückgabewert: das „evaluierte“ Objekt.

Verwandte Funktionen: `eval`, `hold`, `level`, `LEVEL`, `MAXLEVEL`

Details:

- ⌘ Das resultierende Objekt wird nicht vereinfacht.
 - ⌘ Ist das Ergebnis eine Menge, so werden mehrfach auftretende Elemente entfernt.
 - ⌘ `val` arbeitet nicht rekursiv, d. h., wenn der Wert eines Bezeichners selbst wieder Bezeichner enthält, dann werden diese nicht durch ihre Werte ersetzt. Siehe Beispiel ??.
 - ⌘ Die Funktion `val` gleicht ihr Argument nicht aus. Daher darf auch eine Ausdruckssequenz als Argument übergeben werden. Siehe Beispiel ??.
 - ⌘ `val` ist eine Funktion des Systemkerns.
-

Beispiel 1. `val` ersetzt Bezeichner durch ihre Werte, ruft jedoch keine arithmetischen Funktionen wie z. B. `_plus` auf:

```
>> a := 0: val(a*b + 4 + 0)
      0 b + 4 + 0
```

In Mengen werden mehrfach vorkommende Elemente entfernt:

```
>> a := b: val({a, b, a*0})
      {b, 0 b}

>> delete a:
```

Beispiel 2. `val` gleicht weder das Argument aus, noch werden leere Objekte vom Typ `DOM_NULL` entfernt:

```
>> a := null(): val((a, null()))
      null(), null()
```

Jedoch darf nicht mehr als ein Argument übergeben werden:

```
>> val(a, null())
      Error: Wrong number of arguments [val]

>> delete a:
```

Beispiel 3. `val` ersetzt Bezeichner nicht rekursiv durch ihre Werte:

```
>> delete a, b: a := b: b := c: val(a)
      b
```

Änderungen:

☞ Keine Änderungen.

version – die Versionsnummer der MuPAD-Bibliothek

`version()` gibt die Versionsnummer der installierten MuPAD-Bibliothek zurück.

Aufruf(e):

```
# version()
```

Rückgabewert: die Versionsnummer: eine Liste dreier nichtnegativer ganzer Zahlen.

Verwandte Funktionen: `patchlevel`, `Pref::kernel`

Details:

- # Der Aufruf `Pref::kernel()` liefert die Versionsnummer des installierten MuPAD-Kerns.
 - # Die Versionsnummern von Kern und Bibliothek stimmen nicht notwendigerweise überein: `version` bezieht sich auf die Versionsnummer der Bibliothek, während der Aufruf `Pref::kernel()` die Versionsnummer des Kerns liefert.
-

Beispiel 1. Die Versionsnummer der hier verwendeten MuPAD-Bibliothek ist:

```
>> version()

[2, 0, 0]
```

Änderungen:

- # Keine Änderungen.
-

warning – Ausgabe einer Warnung

`warning(message)` gibt die Warnung `message` aus.

Aufruf(e):

```
# warning(message)
```

Parameter:

`message` — eine Zeichenkette

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Seiteneffekte: Die Ausgabeformatierung von `warning` ist von der Umgebungsvariablen `TEXTWIDTH` abhängig.

Verwandte Funktionen: `error, print, userinfo`

Details:

- ⌘ `warning(message)` gibt die mit dem Präfix „Warning: “ versehene Zeichenkette `message` aus.
 - ⌘ `warning` kann dazu verwendet werden, den Anwender auf mögliche Probleme in einem Algorithmus hinzuweisen. Beispielsweise verwendet `limit` Warnungen, um Hinweise zu geben, wie sich von Parametern abhängende Grenzwerte berechnen lassen. Siehe Beispiel ??.
 - ⌘ `warning` ist eine Funktion des Systemkerns.
-

Beispiel 1. Eine Warnung:

```
>> warning("You should not do this!"):

Warning: You should not do this!
```

Beispiel 2. Dieses Beispiel zeigt eine einfache Prozedur, die zwei Zahlen dividiert. Wenn das zweite Argument weggelassen wird, dann wird eine Warnung ausgegeben, aber die Berechnung geht weiter:

```
>> mydivide := proc(x, y)
  begin
    if args(0) < 2 then
      warning("Denominator not given, using 1.");
      y := 1;
    end_if:
    x/y
  end_proc:
mydivide(10)

Warning: Denominator not given, using 1. [mydivide]
```

10

Beispiel 3. Im folgenden Aufruf hängt der angeforderte Grenzwert vom Parameter `c` ab:

```
>> limit(exp(c*x), x = infinity);
```

Warning: cannot determine sign of c [stdlib::limit::limitMRV]

```
limit(exp(c x), x = infinity)
```

Der Nutzer kann auf die Warnung reagieren, indem er `c` mittels `assume` eine Eigenschaft zuordnet:

```
>> assume(c < 0): limit(exp(c*x), x = infinity);
```

0

```
>> assume(c > 0): limit(exp(c*x), x = infinity);
```

infinity

```
>> unassume(c):
```

Änderungen:

⌘ `warning` ist eine neue Funktion.

write – Schreiben von Variablenwerten in eine Datei

`write(filename)` speichert alle Bezeichner einer MuPAD-Sitzung, die einen Wert haben, zusammen mit ihren Werten in der Datei namens `filename`.

`write(filename, x1, x2, ...)` speichert die momentanen Werte der Bezeichner `x1, x2` etc.

`write(n)` und `write(n, x1, x2, ...)` speichert die Daten in der mit dem Dateibezeichner `n` verknüpften Datei.

Aufruf(e):

⌘ `write(<format>, filename)`

⌘ `write(<format>, filename, x1, x2, ...)`

⌘ `write(n)`

⌘ `write(n, x1, x2, ...)`

Parameter:

`filename` — der Dateiname: eine Zeichenkette

`x1, x2, ...` — Bezeichner

`n` — ein von `fopen` erzeugter Dateibezeichner: eine nichtnegative ganze Zahl

Optionen:

`format` — das Schreibformat: Entweder *Bin* oder *Text*. Mit *Bin* werden die Daten in MuPADs Binärformat gespeichert, mit *Text* im üblichen ASCII-Format. Die Voreinstellung ist *Bin*.

Rückgabewert: das leere Objekt vom Typ `DOM_NULL`.

Seiteneffekte: Die Funktion reagiert auf die Umgebungsvariable `WRITEPATH`. Hat diese Variable einen Wert, so wird die Datei in dem entsprechenden Verzeichnis angelegt, anderenfalls im „aktuellen Arbeitsverzeichnis“.

Verwandte Funktionen: `fclose`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `WRITEPATH`

Details:

☞ `write` dient zur Speicherung von Informationen aus der momentanen MuPAD-Sitzung in einer Datei. Die Datei enthält die aktuellen Werte von Bezeichnern. Diesen Bezeichnern werden die gespeicherten Werte zugewiesen, wenn diese Datei mittels `read` in eine andere MuPAD-Sitzung eingelesen wird.

☞ Die Datei kann direkt durch ihren Namen spezifiziert werden. Hierbei wird entweder eine neue Datei geöffnet oder eine existierende Datei dieses Namens wird überschrieben. Dabei öffnet und schließt `write` die Datei automatisch.

Wenn `WRITEPATH` keinen Wert hat, interpretiert `write` den Dateinamen als Pfadnamen relativ zum „aktuellen Arbeitsverzeichnis“.

Man beachte, dass die Bedeutung des „aktuellen Arbeitsverzeichnisses“ vom Betriebssystem abhängt. Unter Windows ist es das Verzeichnis, in dem MuPAD installiert ist. Unter UNIX und Linux ist es das Verzeichnis, in dem MuPAD gestartet wurde.

Auf dem Macintosh kann auch ein leerer Name angegeben werden. In diesem Fall wird ein Dialog geöffnet, in dem der Benutzer eine Datei auswählen kann. Weiterhin warnt MacMuPAD den Benutzer vor dem Überschreiben existierender Dateien.

Auch absolute Pfadnamen werden von `write` verarbeitet.

☞ Anstatt eines Dateinamens kann auch der Dateibezeichner einer durch `fopen` geöffneten Datei übergeben werden. Siehe Beispiel ???. Hierbei werden alle durch `write` geschriebenen Daten an das Ende der entsprechenden Datei angehängt. Die Datei wird nicht automatisch von `write` geschlossen und muss durch einen folgenden Aufruf von `fclose` ‚per Hand‘ geschlossen werden.

Man beachte, dass `fopen(filename)` die Datei nur im Lesemodus öffnet, d. h., ohne Schreibberechtigung. Ein folgender `fprint`-Befehl auf diese Datei liefert einen Fehler. Man verwende die *Write*- bzw. *Append*-Option von `fopen` um die Datei im Schreibmodus zu öffnen.

Der Dateibezeichner 0 stellt den Bildschirm dar.

☞ `write` speichert nur den Wert eines Bezeichners, nicht die vollständige Auswertung. Siehe Beispiel ??.



☞ `write` ist eine Funktion des Systemkerns.

Option **<Text>**:

☞ Im ASCII-Format werden Zuweisungen der Form

```
sysassign(Bezeichner, hold(Wert)):
```

in die Datei geschrieben. Siehe Beispiel ??.

Beispiel 1. Die mit dem Wert `b + 1` versehene Variable `a` wird in einer Datei namens „test“ gespeichert:

```
>> a := b + 1: write(Text, "test", a):
```

Der Inhalt dieser Datei wird mittels `ftextinput` angezeigt:

```
>> ftextinput("test")

      "sysassign(a, hold(b + 1)):"
```

Der Wert von `a` wird gelöscht. Durch Einlesen der Datei „test“ mittels `read` wird der Wert wiederhergestellt:

```
>> delete a: read("test"): a

      b + 1

>> delete a:
```

Beispiel 2. Eine Binärdatei „test“ wird im Schreibmodus geöffnet:

```
>> n := fopen("test", Write)
```

18

Diese Zahl ist der Dateibezeichner und kann in `write`-Befehlen benutzt werden:

```
>> a := b + 1: write(n, a):
>> delete a: read("test"): a
```

$b + 1$

Die Datei wird geschlossen und nicht mehr benötigte Bezeichner werden freigegeben:

```
>> fclose(n): delete n, a:
```

Beispiel 3. Der Wert $b + 1$ wird dem Bezeichner a zugewiesen. Nach der Zuweisung des Wertes 2 an b liefert die vollständige Auswertung von a das Ergebnis 3:

```
>> a := b + 1: b := 2: a
```

3

Der Wert von a ist jedoch weiterhin der symbolische Ausdruck $b + 1$. Dieser Wert wird durch einen `write`-Befehl gespeichert:

```
>> write(Text, "test", a): ftextinput("test")
"sysassign(a, hold(b + 1)):"
```

Konsequenterweise wird dieser Wert restauriert, wenn die Datei eingelesen wird:

```
>> delete a, b: read("test"): a
b + 1
```

```
>> delete a:
```

Änderungen:

☞ Keine Änderungen.

zeta – die riemannsche Zetafunktion

`zeta(z)` stellt die riemannsche Zetafunktion $\zeta(z) = \sum_{k=1}^{\infty} k^{-z}$ dar.

Aufruf(e):

☞ `zeta(z)`

Parameter:

`z` — ein arithmetischer Ausdruck

Rückgabewert: ein arithmetischer Ausdruck.

Überladbar durch: `z`

Seiteneffekte: Für Gleitpunktargumente reagiert die Funktion auf die Umgebungsvariable `DIGITS`, welche die aktuelle numerische Rechengenauigkeit festlegt.

Verwandte Funktionen: `bernoulli`

Details:

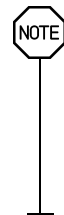
☞ Die Zetafunktion ist in der gesamten komplexen Ebene außer an der einfachen Polstelle $z = 1$ definiert.

☞ Für Gleitpunktargumente wird eine Gleitpunktzahl berechnet.

☞ Die folgenden speziellen Werte sind implementiert: $\zeta(z) = 0$ für gerade ganze Zahlen $z < 0$, $\zeta(z) = -\text{bernoulli}(z)/(1 - z)$ für ungerade ganze Zahlen $z < 0$, $\zeta(0) = -1/2$, $\zeta(z) = (2\pi)^z |\text{bernoulli}(z)|/2/z!$ für gerade ganze Zahlen $z > 0$.

Kann das Argument nicht zu einer der obigen Zahlen evaluiert werden, so liefert `zeta` einen unevaluierten Funktionsaufruf zurück.

☞ Das `float`-Attribut von `zeta` ist eine Kernfunktion, d. h., die numerische Auswertung ist schnell. Hierbei ist die direkte Übergabe eines Gleitpunktarguments `zeta(float(z))` dem Aufruf `float(zeta(z))` vorzuziehen, da für betragsmäßig großes ganzzahliges `z` die Berechnung des exakten Zwischenergebnisses `zeta(z)` zeitaufwendig sein kann.



Beispiel 1. Einige Aufrufe mit exakten bzw. symbolischen Eingabedaten:

```
>> zeta(-6), zeta(-5), zeta(-4), zeta(-3), zeta(-2), zeta(-1)
```

```
0, -1/252, 0, 1/120, 0, -1/12
```

```
>> zeta(0), zeta(2), zeta(3), zeta(4), zeta(5), zeta(6), zeta(7)
```

$$-1/2, \frac{\pi^2}{6}, \zeta(3), \frac{\pi^4}{90}, \zeta(5), \frac{\pi^6}{945}, \zeta(7)$$

```
>> zeta(1/2), zeta(1 + I), zeta(z^2 - I)
```

$$\zeta(1/2), \zeta(1 + I), \zeta(z^2 - I)$$

Für Gleitkommazahlen wird der numerische Wert von zeta berechnet:

```
>> zeta(-1001.0), zeta(12.3), zeta(0.5 + 14.13472514*I)
-1.348590824e1771, 1.000199699,
0.0000000002163160213 - 0.000000001358779595 I
```

Der Punkt $z = 1$ ist eine Polstelle:

```
>> zeta(1)
Error: singularity [zeta]
```

Beispiel 2. Zur Suche nach Nullstellen der Zetafunktion wird die Funktion $f(z) = |\zeta(z)|$ auf der „kritischen Linie“ aller komplexen Zahlen mit Realteil $1/2$ gezeichnet:

```
>> plotfunc2d(Labels = ["", ""], Title = "", Grid = 500,
abs(zeta(1/2 + y*I)), y = 0..30)
```

Die folgende Prozedur ist eine einfache Implementation des üblichen Newton-Verfahrens zur numerischen Nullstellensuche von ζ . Man beachte, dass innerhalb des Newton-Schritts numerisch differenziert wird, da MuPAD keine symbolische Ableitung von zeta zur Verfügung stellt:

```
>> NewtonStep := proc(z)
local h, f, f2, fprime;
begin
z := float(z);
h := 10^(-DIGITS/2.0)*(1 + abs(z));
f := zeta(z);
f2 := zeta(z + h);
fprime := (f2 - f)/h;
return(z - f/fprime)
end_proc;
```

Die durch `z:=NewtonStep(z)` definierte Folge konvergiert gegen eine Nullstelle, falls der Startwert eine hinreichend gute Approximation einer Nullstelle ist:

```
>> z:= 1/2 + 21*I:  z := NewtonStep(z): z, abs(zeta(z))
                                0.5002926366 + 21.0220145 I, 0.0003338475592
>> z := NewtonStep(z): z, abs(zeta(z))
                                0.4999999108 + 21.02203966 I, 0.0000001039451698
>> z := NewtonStep(z): z, abs(zeta(z))
                                0.5 + 21.02203964 I, 1.387291733e-11
>> delete NewtonStep, z:
```

Änderungen:

- ⌘ Explizite Ausdrücke werden nun für alle negativen ganzzahligen sowie alle positiven geraden ganzzahligen Argumente geliefert.

zip – elementweises Verknüpfen von Listen

`zip(Liste1, Liste2, f)` verknüpft zwei Listen mittels einer Funktion `f`. Es wird eine Liste zurückgeliefert, deren i -ter Eintrag `f(Liste1[i], Liste2[i])` ist. Ihre Länge ist das Minimum der Längen der beiden Eingabelisten.

`zip(Liste1, Liste2, f, default)` liefert eine Liste zurück, deren Länge das Maximum der Längen der Eingabelisten ist. Die kürzere Eingabeliste wird dabei mit dem Füllwert `default` aufgefüllt.

Aufruf(e):

- ⌘ `zip(list1, list2, f)`
- ⌘ `zip(list1, list2, f, default)`

Parameter:

- `list1, list2` — Listen beliebiger MuPAD-Objekte
- `f` — ein beliebiges MuPAD-Objekt. Typischerweise eine Funktion zweier Argumente.
- `default` — ein beliebiges MuPAD-Objekt

Rückgabewert: eine Liste.

Überladbar durch: `list1, list2`

Verwandte Funktionen: `map, op, select, split`

Details:

- ⌘ Liefert `f` das leere Objekt vom Typ `DOM_NULL`, so wird das entsprechende Element aus der Ausgabeliste entfernt.
 - ⌘ `zip` ist eine empfehlenswerte Funktion zur schnellen Manipulation von Listen. Sie ist eine Funktion des Systemkerns.
-

Beispiel 1. Der schnellste Weg, zwei Listen elementweise zu addieren, ist die Verknüpfung mittels der Funktion `_plus`:

```
>> zip([a, b, c, d], [1, 2, 3, 4], _plus)

      [a + 1, b + 2, c + 3, d + 4]
```

Haben die Eingabelisten unterschiedliche Länge, so bestimmt die kürzere Liste die Länge der Ausgabeliste:

```
>> zip([a, b, c, d], [1, 2], _plus)

      [a + 1, b + 2]
```

Wird als viertes Argument ein Füllwert für die kürzere Liste übergeben, so bestimmt die längere Eingabeliste die Länge der Ausgabeliste:

```
>> zip([a, b, c, d], [1, 2], _plus, 17)

      [a + 1, b + 2, c + 17, d + 17]
```

Änderungen:

- ⌘ Keine Änderungen.