This document contains a description of the units LazExprDraw and LazExprMake.

Units LazExprDraw and LazExprMake are cross platform (Windows+Linux) versions for the Lazarus of units ExprDraw and ExprMake for drawing mathematical formulas on TCanvas, written by Anton Grigoriev (grigorievab@mail.ru) in Delphi 5 and published in 2002 on delphikingdom.com (http://www.delphikingdom.com/asp/viewitem.asp?catalogID=718). Most of the description is taken from the description of the original ExprDraw and ExprMake, adjusted for changes and additions.

In the original, GDI / Windows API functions and the proprietary font Times New Roman were used, which excluded / made it difficult to use the code in Linux. It also uses the standard features of TCanvas and the XITS font, distributed under a free license SIL Open Font License (https://opensource.org/licenses/OFL-1.1).

XITS fonts are available at github: https://github.com/alif-type/xits. If only standard functions of units are needed and no special text formatting is required, then only two files with Regular and Italic styles are needed:

XITS-Regular.otf (https://github.com/alif-type/xits/blob/master/XITS-Regular.otf)

XITS-Italic.otf (https://github.com/alif-type/xits/blob/master/XITS-Italic.otf)

LazExprDraw unit contains classes used to draw mathematical formulas.

LazExprMake unit is used to create classes based on a symbolic formula entry.

Description language for symbolic formula entry in the second part of this paper, as well as resident in the LazExprGuide application.

Tested in Windows 7, Windows 10, MX Linux 18, Kubuntu 18.10.

Considering the wishes of the author of the original code, the LazExprDraw and LazExprMake units are offered under the free license MIT (https://opensource.org/licenses/MIT)

Оглавление

1. LazExprDraw unit.

The LazExprDraw unit contains classes to draw almost any mathematical formula. The names of all classes are prefixed `TExpr`. To draw an expression, you must build an expression tree whose nodes and leaves are `TExprXXXX` classes. In the simplest case (number, variable) the tree consists of one class. The specific class for each node of the tree is chosen based on the semantic load that the node carries in the formula. For example, to draw a simple fraction, we use the `TExprRatio` class, which has two branches: for the numerator and denominator. The branches themselves can also be composite and the depth of the tree is limited only by computer resources. Thus, it is possible to construct an expression of almost any complexity.

This description of the library LazExprDraw not exhaustive and only outlines the basic principles on the use. In particular, many secondary functions are not described. This is quite enough for the full use of the library, but its improvement will require a more detailed acquaintance with the code.

1.1. About output device resolution

The horizontal and vertical resolution values of the output device are required to calculate the dimensions of the drawn formula elements. In the original ExprDraw their definition is used a function of Windows API `GetDeviceCaps`. The LazExprDraw unit has procedure:

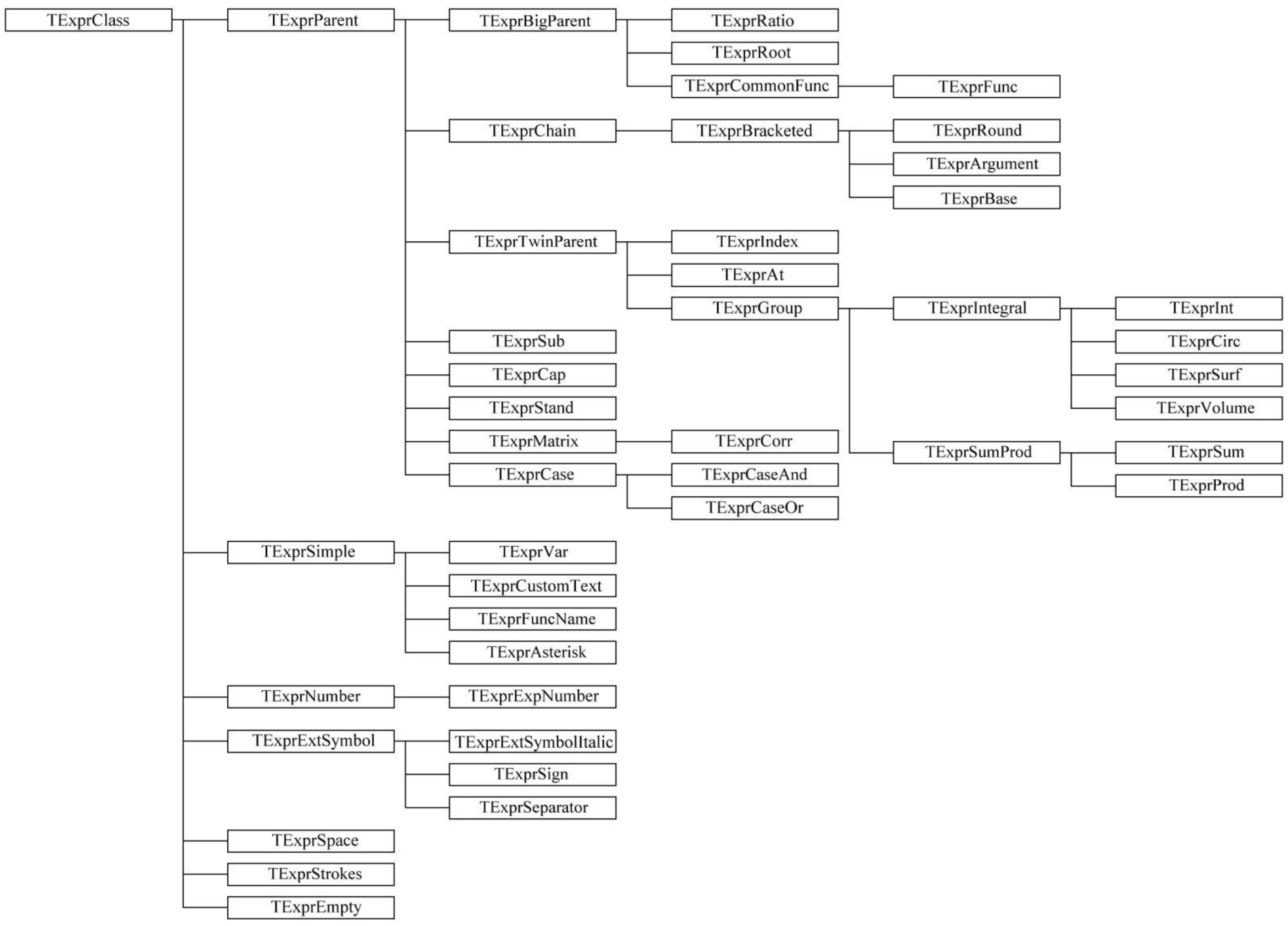
```
SetOutputDPI(const AHorzDPI: Integer = 0; const AVertDPI: Integer = 0);
```

The input parameters are the vertical and horizontal resolutions of the output device. Procedure must be called prior to the appointment of the canvas to the output device. When called without parameters, the screen resolution values are set from `Graphics.ScreenInfo.PixelsPerInchX` and `Graphics.ScreenInfo.PixelsPerInchY`.

To send the formulas on the printer, you need to call the procedure with the parameters `Printers.Printer.XDPI` and `Printers.Printer.YDPI`.

During the initialization of LazExprDraw, the screen resolutions are set, and if it is intended to display formulas only on the screen, you do not need call this procedure.

1.2. LazExprDraw inheritance tree



1.3. TExprClass class

The base class for all classes of the LazExprDraw module is `TExprClass`. It supports the following functionality: it reports its geometrical dimensions (`Width` and `Height` properties), parameters necessary to draw a node (`MidLineUp, MidLineDown, PowerXPos, PowerYPos, IndexXPos, IndexYPos, CapDXRight, CapDXLeft, CapDY` properties). If the node is not the root, then the `FParent` field contains a pointer to the parent node. The `Next` property is used to build class chains. Changing the font or canvas (`Font` and `Canvas` properties) is transmitted further along the chain. The default font is XITS, the use of other fonts is undesirable, since all sizes of characters, symbols, indents, etc. calculated for this font. Changing font styles does not make sense, since each node itself sets its own styles, determined by its meaning. The only parameter of the font that makes sense to manage is its size. The font size is set only for the root node of the tree, then the settings are transferred to all branches (branch nodes do not necessarily get the same font size as the root: in some cases, changing the font sizes of the child nodes is provided, for example, when adding superscript or subscript). A canvas is also assigned to the root node and transferred to all other nodes. Geometrical dimensions and parameters cannot be calculated before the font and canvas are assigned to the tree. Parameters have the following meanings:

`MidLineUp, MidLineDown` - the distance (in pixels) from the middle line to the top and bottom edges of the expression. The middle line is the line on which the "`-`" sign should be, if you put it before the expression.

`PowerXPos, PowerYPos` - if the superscript is added to the expression, then these parameters are used when calculating its position.

`IndexXPos, IndexYPos` – similar parameters for subscript.

`CapDXLeft, CapDXRight, CapDY` - parameters used to position the accent mark (vector, tilde, etc.), if it is placed over the expression.

The calculation of the size and parameters of the expression is performed once using functions with the prefix `Calc`, then it is stored in a variable for optimization purposes. The `FToChange` field holds `tcXXXX` flags that indicate which parameters have already been calculated and which ones have not. If changes occur that affect the values of the parameters (for example, the font size or the canvas changes), the corresponding flags in `FToChange` are set to indicate that the parameters need to be re-calculated, rather than using previously saved values.

The expression is drawn using the `Draw` function with input parameters: canvas coordinates (`X` and `Y`), horizontal (`TExprHorAlign`) and vertical (`TExprVertAlign`) alignment values. You can align with the left edge, right edge and center (horizontally) and top edge, bottom edge and middle line (vertical). The expression mapping is implemented by the `Paint` virtual function. This function is called by the `Draw` function after the corresponding coordinate transformations.

The `FTType` function returns a combination of `efXXXX` flags that are used in the LazExprMake unit when multiplying expressions using the "*" symbol. These flags indicate whether the expression can be multiplied without a sign to the left, to the right, whether it is a number, etc.

The `NeedBrackets` function returns `True` if the appearance of a given expression in a chain of expressions makes it necessary to enclose the given chain in brackets when multiplying it by another expression.

The `ArgNeedBrackets` function returns `True` if the argument must be enclosed in brackets when using this expression as the name of a function (see class `TExprCommonFunc`).

The `Color` property determines the color by which the expression will be drawn. If the color is `clNone`, the color of the parent node is used. If `clNone` is set for the root node, black is used.

The `FWX, FWY, FRWX` and `FRWY` fields are used in calculating the dimensions of gaps, lines, etc. `FWX` is approximately equal to the thickness of the vertical line of the sign "+", and `FWY` – horizontal line. `FRWX` and `FRWY` store values that are linearly dependent on font size. These values are used to draw expressions.

1.4. TExprParent class

The `TExprParent` is the base class for all parent classes. The `Son` property of this class stores a pointer to the child node of the expression tree. Creating `TExprParent` instances does not make sense, since all the functionality of this class comes down to managing fonts and the canvas of a child expression.

The child expression and the expression referenced by the `Next` property vary greatly in meaning. If a class is included in a chain organized with the help of the `Next` property, this does not affect its size or the type of expression displayed with it. The value of the `Next` property is not used in any way by the class itself, except for the fact that the canvas, color and font are transmitted further along the chain. The fact that classes are chained can be used by the first parent node in the chain. On the other hand, the expression defined by the `Son` property is an integral part of the expression defined by the class itself. As an example, a class that implements a square root. `Son` points to a radical expression. When you need to calculate the size of an expression, this class gets `Son.Width` and `Son.Height` and adds the dimensions of the root sign to the values obtained. When calling the `Draw` function, this class draws a square root character, and then calls `Son.Draw` to display the radical. Thus, the `Next` property is used to organize class chains, and the fact that a class is included in such a chain is not used by it. And the `Son` property is used to specify an expression that is part of the expression that is defined by the class.

1.5. TExprBigParent class

The `TExprBigParent` is inherited from the `TExprParent`. It added another child node - the `Daughter` property. This class is the base for the implementation of expressions in which there are two components. An example of such an expression is a simple fraction, in which one of the child expressions defines the numerator, and the other denominator.

1.6. TExprRatio class

The `TExprRatio` is inherited from the `TExprBigParent` and implements a simple fraction. `Son` points to numerator, `Daughter` points to denominator.

1.7. TExprRoot class

The `TExprRoot` is inherited from the `TExprBigParent` and implements a root extraction. `Son` points to radical expression, `Daughter` points to a root exponent. For `Daughter`, `nil` is allowed. In this case, the exponent is not put before the root (square root).

1.8. TExprCommonFunc class

The `TExprCommonFunc` is inherited from the `TExprBigParent`. It implements a “generic” function, as which any expression can be used as a “name”. `Son` points to the "name" of the function, `Daughter` points to its argument.

1.9. TExprFunc class

The class `TExprFunc` is inherited from `TExprCommonFunc`. It implements the traditional function, the name of which is the combination of letters. The constructor itself creates the class of the desired type to display the name of the function. If the name has a length of 1 character, then the `TExprVar` class is used; if it is longer, the `TExprFuncName` class. In the first case, the function name is written in italics, and the argument is always enclosed in brackets. In the second case, the function name is written in regular, and the argument is enclosed in brackets only when necessary.

1.10. TExprChain class

The `TExprChain` is inherited from `TExprParent`. This class is used to draw a chain of expressions. The first drawn in the chain is `Son`, then `Son.Next`, then `Son.Next.Next`, and so on, until the end of the chain is reached.

1.11. TExprBracketed class

The class `TExprBracketed` is inherited from `TExprChain`. It draws a chain of expressions enclosed in brackets. Types of brackets are given by type

```
TExprBracketStyle = (ebNone,ebRound,ebSquare,ebFigure,ebModule,ebNorm,ebAngle,ebFloor,ebCeil);
```

Value	Left bracket	Right bracket
`ebNone`	no bracket	no bracket
`ebRound`	(	)
`ebSquare`	[	]
`ebFigure`	{	}
`ebModule`	|	|
`ebNorm`	‖	‖
`ebAngle`	⟨	⟩
`ebFloor`	⌊	⌋
`ebCeil`	⌈	⌉

The brackets can be unpaired (for example, the opening bracket is round and the closing bracket is square). It is also possible that the bracket is only on one side of the expression, and on the other `ebNone`. The expression is enclosed in brackets only if the `IsBracketed` function returns `True`.

1.12. TExprRound class

The class `TExprRound` is inherited from `TExprBracketed`. It overrides the `FTType` function. This class is used by the LazExprMake unit when reducing extra brackets.

1.13. TExprArgument class

The `TExprArgument` is inherited from `TExprBracketed`. Its constructor is redefined so that the brackets are always round. In addition, the `IsBracketed` function has been overridden so that it returns `True` only if at least one of the chain nodes pointed to `Son` needs brackets (that is, its `NeedBrackets` function returns `True`). In brackets need, for example, nodes that implement the signs "+" and "-". Thus, if the chain of expressions is the sum of several terms, it is enclosed in brackets, but if this product of several factors is not. This is the same as the way the function arguments are written (for example, `cos 2x`, but `cos(x+y)`). The same class is used to implement individual factors: if the multiplier includes addition or subtraction operations, it is enclosed in brackets, but if not includes this operations then it is not enclosed in brackets.

The `SetBrackets` function has also been added to the class, which causes the brackets to be drawn regardless of whether there are nodes in the chain that need brackets.

If the TExprArgument class is used as a function argument (in this case, its parent node is an instance of the TExprCommonFunc class), then the corresponding function of the parent class is called to determine if the argument is needed brackets.

When building a tree manually for a particular expression, it is always known in advance whether it is necessary to enclose some of its parts in brackets or not. In this case, it is better to use the classes TExprBracketed or TExprChain. But with automated tree building (for example, using the LazExprMake unit), this class can be very useful.

1.14. TExprBase class

The TExprBase is inherited from TExprBracketed. It paints an expression enclosed in brackets if the chain pointed to Son contains more than one node. Otherwise, the expression in brackets is not enclosed. This coincides with the way the base should be displayed during the exponentiation (for example, x^2, but $(2x)^2$). The TExprBase class, like TExprArgument, is intended only for automating tree construction.

1.15. TExprTwinParent class

The TExprTwinParent is inherited from TExprParent. It added pointers to two twin nodes. An example of an expression for which such a class may be needed is an indexed expression. Son refers to the expression, to which indices are added, and Twin1, Twin2 to the subscript and superscript.

1.16. TExprIndex class

The TExprIndex is inherited from TExprTwinParent. It used to paint expressions with superscript or subscript. Son points to the expression to which indices are added, Twin1 points to subscript, Twin2 points to superscript. Twin1 or Twin2 can be nil - in this case only one index is added to the expression. The superscript is also used as an exponent.

1.17. TExprAt class

The TExprAt is inherited from TExprTwinParent. It implements "value provided" or value "from ... to ...". This is displayed as follows: to the right of the Son expression a vertical line is drawn, to the right of the vertical line on top is drawn the Twin1 expression, and below the Twin2 expression. The value of Twin2 may be nil, then it is not drawn.

1.18. TExprGroup class

The TExprGroup is inherited from TExprTwinParent. This is the base class for all sum type or integral type expressions. All these expressions are constructed in the same way: Son points to the expression after the sum or integral sign, Twin1 - under the sign, Twin2 - above it. Twin1 and Twin2 can be nil simultaneously or one by one. In this case, nothing is written under the sign and / or above it.

1.19. TExprIntegral class

The TExprIntegral is inherited from TExprGroup. This is the base class for all types of integrals.

1.20. TExprInt class

The TExprInt is inherited from TExprIntegral. It implements a simple integral. The Mult constructor parameter specifies the multiplicity of the integral. If Mult <= 0, the integral symbol of indefinite multiplicity is drawn (two integral symbols, ellipsis, another integral symbol).

1.21. TExprCirc class

The TExprCirc is inherited from TExprIntegral. It implements a closed circuit integral.

1.22. TExprSurf class

The TExprSurf is inherited from TExprIntegral. It implements a closed surface integral.

1.23. TExprVolume class

The TExprVolume is inherited from TExprIntegral. It implements a closed volume integral.

1.24. TExprSumProd class

The TExprSumProd is inherited from TExprGroup class. This is the base class for sum and product expressions.

1.25. TExprSum class

The TExprSum is inherited from TExprSumProd. It implements a sum (using the Greek letter sigma).

1.26. TExprProd class

The TExprProd is inherited from TExprSumProd. It implements a product (using the Greek letter pi).

1.27. TExprSub class

The TExprSub is inherited from TExprParent. This class implements a function like limit (lim): paints the name of the function FuncName and below it the Son expression. The class is intended for sharing with TExprCommonFunc. TExprCommonFunc.Son points to a TExprSub instance, and TExprCommonFunc.Daughter points to an expression for which a function is being evaluated.

1.28. TExprCap class

The TExprCap is inherited from TExprParent. It draws over the Son expression to which Son points an accent mark. Types of marks are given by type

```
TExprCapStyle = (ecPoints,ecVector,ecCap,ecTilde,ecLine);
```

Value	Mark
ecPoints	··
ecVector	→
ecCap	^
ecTilde	~
ecLine	–

The class constructor contains the Count parameter, which specifies the number of points for the ecPoints style. For other styles, this parameter is ignored.

1.29. TExprStand class

The TExprStand is inherited from TExprParent. It paints several expressions in the form of a column. The first is the expression that Son points to, Son.Next is below it, Son.Next.Next is even lower, and so on to the end of the chain. The HorAlign constructor parameter set expressions alignment to the left, right, or center.

1.30. TExprMatrix class

The TExprMatrix is inherited from TExprParent. It paints the matrix. The dimensions of the matrix are determined by the constructor parameters HorSize and VertSize. The upper left element of the matrix is given by the Son expression, the second from the left in the top line is Son.Next, and so on from left to right from top to bottom. If the chain is longer than HorSize*VertSize, the extra elements are ignored when the matrix is painted (however, they are not ignored when calculating the dimensions of the matrix cell, which can lead to a distortion of the matrix picture). If the chain contains fewer elements than HorSize*VertSize, the cells for which there are not enough nodes will be empty.

1.31. TExprCorr class

The TExprCorr is inherited from TExprMatrix. It is intended to paint a correspondence table of two columns separated by a vertical line. The first element of the table (to the left of the line) is defined by the Son expression, its corresponding value (to the right of the line) is Son.Next, the second element is Son.Next.Next, its correspondence is Son.Next.Next.Next, and so on.

1.32. TExprCase class

The TExprCase is inherited from TExprParent. This is the base class for drawing the variant construction: several expressions are enclosed in the left bracket, after each a condition is drawn. The first element is Son expression, the condition of its applicability is Son.Next, the second element is Son.Next.Next expression, the condition of its applicability is Son.Next.Next.Next, etc. If the chain contains an odd number of nodes, the last element remains without a condition.

1.33. TExprCaseAnd class

The TExprCaseAnd is inherited from TExprCase. It paints a variant construction with a figure bracket.

1.34. TExprCaseOr class

The TExprCaseOr is inherited from TExprCase. It paints a variant construction with a square bracket.

1.35. TExprSimple class

The TExprSimple class is used to write plain text.

1.36. TExprVar class

The TExprVar is inherited from TExprSimple. The main difference is that the text is drawn in italics, as is customary when displaying variables in expressions.

1.37. TExprCustomText class

The `TExprCustomText` is inherited from `TExprSimple`. With it, you can write text with any font settings.

1.38. TExprFuncName class

The `TExprFuncName` is inherited from `TExprSimple`. It implements the name of the function in the case when it is written in plain type. Override function `ArgNeedBrackets` so that the argument is not enclosed in brackets, if this is not necessary. Argument must be of type `TExprArgument` to use the result returned by the `TExprFuncName.ArgNeedBrackets`.

1.39. TExprAsterisk class

The `TExprAsterisk` is inherited from `TExprSimple`. It paints a symbol * shifted vertically downwards. This symbol is used as a superscript (for example, to denote a conjugate value).

1.40. TExprNumber class

The `TExprNumber` class is designed to draw numbers in ordinary or exponential form. If the `ExpForm` constructor parameter is `True`, the exponential form is used; otherwise, it is ordinary.

1.41. TExprExpNumber class

The class `TExprExpNumber` is inherited from `TExprNumber`. It is used to draw numbers with the ability to flexibly control the number formatting.

1.42. TExprExtSymbol class

The `TExprExtSymbol` class is used to draw single symbols defined by a decimal code in UTF-16BE encoding, written in plain type.

1.43. Symbol code constants

For ease of use of the `TExprExtSymbol` class and its heirs, symbol code constants are defined:

Constant	Symbol	Constant	Symbol	Constant	Symbol
`esApproxLess`	≲	`esLessOrEqual`	≤	`esMuchLess`	≪
`esApproxGreater`	≳	`esGreaterOrEqual`	≥	`esMuchGreater`	≫
`esLess`	<	`esPlus`	+	`esPlusMinus`	±
`esGreater`	>	`esMinus`	−	`esMinusPlus`	∓
`esEqual`	=	`esIdentical`	≡	`esApproxEqual`	≅
`esNotEqual`	≠	`esNotIdentical`	≢	`esAlmostEqual`	≈
`esMultiply`	·	`esSlash`	/	`esBackSlash`	\
`esCrossMultiply`	×	`esDotsDivide`	÷	`esParallel`	∥
`esSemicolon`	;	`esDot`	.	`esNotParallel`	∦
`esComma`	,	`esTilde`	~	`esPerpendicular`	⊥
`esExists`	∃	`esBelongs`	∈	`esColon`	:
`esNotExists`	∄	`esNotBelongs`	∉	`esIntersection`	∩
`esSubSet`	⊂	`esSuperSet`	⊃	`esUnion`	∪
`esNotSubSet`	⊄	`esNotSuperSet`	⊅	`esAnd`	⋀
`esProportional`	∝	`esSum`	∑	`esOr`	⋁
`esInfinity`	∞	`esProd`	∏	`esNot`	¬
`esNatural`	ℕ	`esStroke`	'	`esXor`	⊻
`esReal`	ℝ	`esPartDiff`	∂	`esForAll`	∀
`esRational`	ℚ	`esAngle`	∠	`esArc`	⌣
`esEntire`	ℤ	`esNabla`	∇	`esEmptySet`	∅
`esComplex`	ℂ	`esPlanck`	ħ	`esEllipsis`	…
`esQuaternion`	ℍ	`esLambdaSpec`	ƛ	`esEllipsVert`	⋮
`esProjective`	ℙ	`esAsterisk`	∗	`esEllipsHoriz`	⋯

esArrowLeft	←	esDoubleArrowLeft	⇐	esEllipsDiagUp	⋰
esArrowRight	→	esDoubleArrowRight	⇒	esEllipsDiagDown	⋱
esArrowUp	↑	esDoubleArrowUp	⇑	esLeftAngleBracket	〈
esArrowDown	↓	esDoubleArrowDown	⇓	esRightAngleBracket	〉
esArrowLeftRight	↔	esDoubleArrowLeftRight	⇔	esLeftFloorBracket	⌊
esArrowUpDown	↕	esDoubleArrowUpDown	⇕	esRightFloorBracket	⌋
esArrowLeftUp	↖	esDoubleArrowLeftUp	⇖	esLeftCeilBracket	⌈
esArrowRightUp	↗	esDoubleArrowRightUp	⇗	esRightCeilBracket	⌉
esArrowRightDown	↘	esDoubleArrowRightDown	⇘	esDivide	∣
esArrowLeftDown	↙	esDoubleArrowLeftDown	⇙	esNotDivide	∤
esArrowLeftNot	↚	esDoubleArrowLeftNot	⇍	esBarArrowLeft	↤
esArrowRightNot	↛	esDoubleArrowRightNot	⇏	esBarArrowRight	↦
esArrowLeftRightNot	↮	esDoubleArrowLeftRightNot	⇎	esBarArrowUp	↥
esInt	∫	esSurf	∯	esBarArrowDown	↧
esCirc	∮	esVolume	∰	esTriangle	△
esAlphaBig	Α	esAlphaSmall	α	esQuadrate	□
esBetaBig	Β	esBetaSmall	β	esRectangle	▭
esGammaBig	Γ	esGammaSmall	γ	esCircle	○
esDeltaBig	Δ	esDeltaSmall	δ	esParallelogram	▱
esEpsilonBig	Ε	esEpsilonSmall	ε	esRhomb	◇
esZetaBig	Ζ	esZetaSmall	ζ	esBegin	◀
esEtaBig	Η	esEtaSmall	η	esEnd	▶
esThetaBig	Θ	esThetaSmall	θ	esSimilar	∾
esIotaBig	Ι	esIotaSmall	ι	esDegree	°
esKappaBig	Κ	esKappaSmall	κ	esPlusCircle	⊕
esLambdaBig	Λ	esLambdaSmall	λ	esMinusCircle	⊖
esMuBig	Μ	esMuSmall	μ	esCrossCircle	⊗
esNuBig	Ν	esNuSmall	ν	esSlashCircle	⊘
esXiBig	Ξ	esXiSmall	ξ	esDotCircle	⊙
esOmicronBig	Ο	esOmicronSmall	ο	esCircleCircle	⊚
esPiBig	Π	esPiSmall	π	esAsteriskCircle	⊛
esRhoBig	Ρ	esRhoSmall	ρ	esEqualCircle	⊜
esSigmaBig	Σ	esSigmaSmall	σ	esHarpArrowsLeftRight	⇋
esTauBig	Τ	esTauSmall	τ	esHarpArrowsRightLeft	⇌
esUpsilonBig	ϒ	esUpsilonSmall	υ	esTwoArrowsRightLeft	⇄
esPhiBig	Φ	esPhiSmall	φ	esTwoArrowsLeftRight	⇆
esChiBig	Χ	esChiSmall	χ	esTwoArrowsUpDown	⇅
esPsiBig	Ψ	esPsiSmall	ψ	esTwoArrowsDownUp	⇵
esOmegaBig	Ω	esOmegaSmall	ω	esTwoArrowsLeft	⇇
esThetaOther	ϑ	esSigmaOther	ς	esTwoArrowsRight	⇉
esDotEqual	≐	esDotsEqualLeftRight	≒	esTwoArrowsUp	⇈

esDotsEqualCenter	≑	esDotsEqualRightLeft	≓	esTwoArrowsDown	⇊
esColonEqual	≔	esEqualColon	≕	esThreeArrows	⇶

1.44. TExprExtSymbolItalic class

The class `TExprExtSymbolItalic` is inherited from `TExprExtSymbol`. It paints symbols in italics.

1.45. TExprSign class

The class `TExprSign` is inherited from `TExprExtSymbol`. It overrides the `NeedBrackets` function, which helps to put brackets correctly, so when automatically building expressions even for simple characters (for example, "+", "-", etc.), we recommend using the `TExprSign` class, and not `TExprExtSymbol` or `TExprSimple`.

1.46. TExprSeparator class

The `TExprSeparator` is inherited from `TExprExtSymbol`. It paints separator character - a comma or a semicolon. The function `NeedBrackets` has been overridden so that expressions containing a separator should be enclosed in brackets if necessary.

1.47. TExprSpace class

The `TExprSpace` class inserts empty space into a formula. The class constructor parameter `Count` sets the width of this space in units of width (one unit of width is approximately equal to the width of the vertical line in the "+" symbol).

1.48. TExprStrokes class

The class `TExprStrokes` paints the strokes that denote the derivative. It used in conjunction with the `TExprIndex` class: the function is specified in `TExprIndex.Son`, and `TExprStrokes` is used as a superscript.

1.49. TExprEmpty class

The `TExprEmpty` class is used to paint an expression with a width of zero but a height equal to the height of a plain text.

1.50. Examples

Example 1: $a+b\left(\frac{c}{d}+1\right)$

```
var
  Expr,Expr2: TExprClass;
begin
  Expr:= TExprRatio.Create(TExprVar.Create('c'),TExprVar.Create('d'));
  // Now Expr contains c/d
  Expr.AddNext(TExprSign.Create(esPlus));
  Expr.AddNext(TExprNumber.Create(1));
  // Now Expr is start of a chain with a "+" sign and number 1
  Expr:= TExprBracketed.Create(Expr,ebRound,ebRound);
  // Now Expr contains c/d+1
  Expr2:= TExprVar.Create('a');
  Expr2.AddNext(TExprSign.Create(esPlus));
  Expr2.AddNext(TExprVar.Create('b'));
  Expr2.AddNext(Expr);
  // Now Expr2 contains the entire expression chain that we need.
  Expr:= TExprChain.Create(Expr2);
  // We use TExprChain class to display the chain as a whole.
  // The expression tree is complete.
  Expr.Font.Size:=12; // Font size set
  // To display an expression on a form we need to set its Canvas and call Draw
  Expr.Canvas:= Form1.Canvas;
  Expr.Draw(5,5,ehLeft,evTop);
  // We can send expression to the printer
  SetOutputDPI(Printer.XDPI, Printer.YDPI); // See clause 1.1.
  Printer.BeginDoc;
  Expr.Canvas:= Printer.Canvas;
  Expr.Draw(50,50,ehLeft,evTop);
  Printer.EndDoc;
  // We can change the font and go back to the form
  SetOutputDPI;
  Expr.Canvas:= Form1.Canvas;
  Expr.Font.Height:=24;
  Expr.Draw(Form1.ClientWidth-5,Form1.ClientHeight-5,ehRight,evBottom);
  Expr.Free; // or FreeAndNil(Expr)
  // We need to destroy expression tree after use. Destructor automatically calls destructors
  // for all tree nodes. Expr2.Free does not need to be called in this case, because Expr2 tree
  // has become part of the Expr tree, therefore, when Expr.Free is called, Expr2 will also be
  // destroyed.
```

```
end;
```

In this example, in addition to actually constructing the expression tree, it is shown how to use this tree later, but the simplest method does not always allow achieving the required performance. Each time you change the canvas and font, you have to re-calculate all the sizes of the expression, so you need to change them as little as possible. In addition, even if all sizes have already been calculated, the derivation of the formula also occurs relatively slowly. If the formula is intended for multiple display only on the screen, it is better to build the tree once, display the formula on TBitmap, then release the formula tree, and then each time display this Bitmap.

After calling `TExprClass.Draw`, canvas properties pen, brush and font usually change. If the display of formulas is interspersed with the output of graphic primitives, each time after calling Draw, you should restore the brush, pen, and font again.

Example 2: $\cos^2\varphi + \sin^2\varphi = 1$

```
var
  Expr,Expr2: TExprClass;
begin
  Expr2:= TExprIndex.Create(TExprSimple.Create('cos'),nil,TExprNumber.Create(2,False));
  // Expr2 contains cos²
  Expr:= TExprCommonFunc.Create(Expr2,TExprExtSymbol.Create(esPhiSmall));
  // Expr contains function with name "cos²" with argument "φ"
  Expr.AddNext(TExprSign.Create(esPlus));
  // "+" was added to the chain
  Expr2:= TExprIndex.Create(TExprSimple.Create('sin'),nil,TExprNumber.Create(2,False));
  Expr.AddNext(TExprCommonFunc.Create(Expr2,TExprExtSymbol.Create(esPhiSmall)));
  // "sin²φ" was added to the chain
  Expr.AddNext(TExprSign.Create(esEqual));
  // "=" was added to the chain
  Expr.AddNext(TExprNumber.Create(1,False));
  // "1" was added to the chain
  Expr:= TExprChain.Create(Expr);
  // The tree is complete. There should be a code that displays it
  Expr.Free;
end;
```

Example 3: $y = e^x$

```
var
  Expr: TExprClass;
begin
  Expr:= TExprVar.Create('y');
  Expr.AddNext(TExprSign.Create(esEqual));
  Expr.AddNext(TExprIndex.Create(TExprVar.Create('e'),nil,TExprVar.Create('x')));
  Expr:= TExprChain.Create(Expr);
  // The tree is complete. There should be a code that displays it
  Expr.Free;
end;
```

Example 4: $y = \sqrt[3]{x-1}$

```
var
  Expr,Expr2: TExprClass;
begin
  Expr:= TExprVar.Create('x');
  Expr.AddNext(TExprSign.Create(esMinus));
  Expr.AddNext(TExprNumber.Create(1,False));
  Expr:= TExprChain.Create(Expr);
  Expr2:= TExprVar.Create('y');
  Expr2.AddNext(TExprSign.Create(esEqual));
  Expr2.AddNext(TExprRoot.Create(Expr,TExprNumber.Create(3,False)));
  Expr:= TExprChain.Create(Expr2);
  // The tree is complete. There should be a code that displays it
  Expr.Free;
end;
```

Example 5: $\lim\limits_{x \to 0} \frac{1}{x} = \pm\infty$

```
var
  Expr,Expr2: TExprClass;
begin
  Expr2:= TExprVar.Create('x');
  Expr2.AddNext(TExprSign.Create(esArrowRight));
  Expr2.AddNext(TExprNumber.Create(0,False));
  Expr2:= TExprChain.Create(Expr2);
  Expr2:= TExprSub.Create(Expr2, 'lim');
  Expr:= TExprRatio.Create(TExprNumber.Create(1,False),TExprVar.Create('x'));
  Expr:= TExprCommonFunc.Create(Expr2,Expr);
  Expr.AddNext(TExprSign.Create(esEqual));
  Expr.AddNext(TExprSign.Create(esPlusMinus));
  Expr.AddNext(TExprExtSymbol.Create(esInfinity));
  Expr:= TExprChain.Create(Expr);
  // The tree is complete. There should be a code that displays it
  Expr.Free;
end;
```

Example 6: $(f(x)g(x))' = f'(x)g(x) + f(x)g'(x)$

```
var
  Expr,Expr2: TExprClass;
begin
  Expr2:= TExprFunc.Create('f',TExprBracketed.Create(TExprVar.Create('x'),ebRound,ebRound));
  Expr2.AddNext(TExprFunc.Create('g',TExprBracketed.Create(TExprVar.Create('x'),ebRound,ebRound)));
  Expr2:= TExprBracketed.Create(Expr2,ebRound,ebRound);
  Expr:= TExprIndex.Create(Expr2,nil,TExprStrokes.Create(1));
  Expr.AddNext(TExprSign.Create(esEqual));
  Expr2:= TExprIndex.Create(TExprVar.Create('f'),nil,TExprStrokes.Create(1));
  Expr2:=TExprCommonFunc.Create(Expr2,TExprBracketed.Create(TExprVar.Create('x'),ebRound,ebRound));
  Expr.AddNext(Expr2);
  Expr.AddNext(TExprFunc.Create('g',TExprBracketed.Create(TExprVar.Create('x'),ebRound,ebRound)));
  Expr.AddNext(TExprSign.Create(esPlus));
  Expr.AddNext(TExprFunc.Create('f',TExprBracketed.Create(TExprVar.Create('x'),ebRound,ebRound)));
  Expr2:= TExprIndex.Create(TExprVar.Create('g'),nil,TExprStrokes.Create(1));
  Expr2:=TExprCommonFunc.Create(Expr2,TExprBracketed.Create(TExprVar.Create('x'),ebRound,ebRound));
  Expr.AddNext(Expr2);
  Expr:= TExprChain.Create(Expr);
  // The tree is complete. There should be a code that displays it
  Expr.Free;
end;
```

Example 7: $\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j$

```
var
  Expr,Expr2: TExprClass;
begin
  Expr:= TExprIndex.Create(TExprVar.Create('a'),TExprVar.Create('ij'),nil);
  Expr.AddNext(TExprIndex.Create(TExprVar.Create('x'),TExprVar.Create('i'),nil));
  Expr.AddNext(TExprIndex.Create(TExprVar.Create('x'),TExprVar.Create('j'),nil));
  Expr:= TExprChain.Create(Expr);
  Expr2:= TExprVar.Create('j');
  Expr2.AddNext(TExprSign.Create(esEqual));
  Expr2.AddNext(TExprNumber.Create(1,False));
  Expr2:= TExprChain.Create(Expr2);
  Expr:= TExprSum.Create(Expr,Expr2,TExprVar.Create('n'));
  Expr2:= TExprVar.Create('i');
  Expr2.AddNext(TExprSign.Create(esEqual));
  Expr2.AddNext(TExprNumber.Create(1,False));
  Expr2:= TExprChain.Create(Expr2);
  Expr:= TExprSum.Create(Expr,Expr2,TExprVar.Create('n'));
  // The tree is complete. There should be a code that displays it
  Expr.Free;
end;
```

An example of incorrect construction of an expression tree

If the expression contains duplicate parts, the same node should not be used in multiple places. Otherwise, there will be problems when changing the font, and when the tree is released, the program will try to free this node twice, and as a result, Access Violation will occur.

We illustrate this with an example expression: $e^{x} + x$

Incorrect example:

```
  Expr2:= TExprVar.Create('x');
  Expr:= TExprIndex.Create(TExprVar.Create('e'),nil,Expr2); // Expr2 used for the first time
  Expr.AddNext(TExprSign.Create(esPlus));
  Expr.AddNext(Expr2); // Expr2 is used a second time
  Expr:= TExprChain.Create(Expr);
```

Correct example: you need to create another node to display "x", and not reuse existing

```
Expr2:= TExprVar.Create('x');
Expr:= TExprIndex.Create(TExprVar.Create('e'),nil,Expr2);
Expr.AddNext(TExprSign.Create(esPlus));
Expr2:= TExprVar.Create('x'); // In order not to reuse the node, create another one of the same
Expr.AddNext(Expr2);
Expr:= TExprChain.Create(Expr);
```

2. LazExprMake unit

LazExprMake unit is designed to automate the construction of formulas using the classes of the LazExprDraw unit.

The unit contains the `TExprBuilder` class, which builds the expression tree from a symbolic formula entry. The syntax of the symbolic formula entry with examples is given in detail below, and is also presented as a reference to the LazExprGuide program.

2.1. TExprBuilder class

`TExprBuilder` class has two functions for constructing expressions: `BuildExpr` and `SafeBuildExpr`. Both of these functions take a string as a parameter and return a pointer to the root of the constructed tree.

If the expression contains errors, an exception `EincorrectExpr` will occur. If an exception occurs during the operation of the `BuildExpr` function, garbage from a partially constructed tree remains in memory, therefore, the `BuildExpr` function can be used only for obviously correct expressions.

`SafeBuildExpr` function first runs the tree idle algorithm. At this stage, the tree is not built, but exceptions with incorrect syntax occur. Only if the first stage is completed successfully, the function proceeds to the next stage: building a tree.

The tree created as a result of the `BuildExpr` and `SafeBuidExpr` functions must be destroyed after use by calling `Free`.

LazExprMake unit also contains global functions `BuildExpr` и `SafeBuildExpr`. They create an instance of the `TExprBuilder` class, call its function of the same name, and then destroy the created `TExprBuilder`. These functions can be used if you need to create only one tree, otherwise it is more optimal to create an instance of the `TExprBuilder` class, call `TExprBuilder.BuildExpr` or `TExprBuilder.SafeBuildExpr` as many times as necessary, and then destroy it.

Examples of creating expressions using LazExprMake (see clause 1.50.) can be rewritten as follows:

Example 1: $a+b\left(\frac{c}{d}+1\right)$

```
Expr:= BuildExpr('a+b*(c/d+1)');
```

Example 2: $\cos^2\varphi+\sin^2\varphi=1$

```
Expr:= BuildExpr('cos(phi)^2+sin(phi)^2=1');
```

Example 3: $y=e^{x}$

```
Expr:= BuildExpr('y=Pow(e,x)');
```

Example 4: $y=\sqrt[3]{x-1}$

```
Expr:= BuildExpr('y=Root(3,x-1)');
```

Example 5: $\lim\limits_{x\to 0}\frac{1}{x}=\pm\infty$

```
Expr:= BuildExpr('lim(x->0,1/x)=+-Inf');
```

Example 6: $(f(x)g(x))'=f'(x)g(x)+f(x)g'(x)$

```
Expr:= BuildExpr('(f(x)*g(x))`=f(x)`*g(x)+f(x)*g(x)`');
```

Example 7: $\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{n}a_{ij}x_ix_j$

```
Expr:= BuildExpr('Sum(Sum(a_(i*j)*x_i*x_j,j=1,n),i=1,n)');
```

2.2. Symbolic formula entry syntax

The string describing the formula consists of expressions. Simple expressions include **identifiers**, **tokens**, and **numeric constants**.

Tokens are reserved words used to denote special characters (see example 1 in table below). Tokens are case insensitive, with the exception of tokens denoting Greek letters. If the first character of such a token has an uppercase, then a capital letter is obtained, if the lower one is lowercase.

Identifiers are displayed as they are written. Identifiers can consist of English and Russian letters and numbers. The identifier must begin with a letter. If the identifier begins with an English letter, it is displayed in italics (example 2), if with Russian - in the regular font (example 3). You can display English text in regular font using the function `String`. If at the end of the identifier are numbers, and `TExprBuilder.VarAutoIndex:=True` (by default), then they are displayed as a subscript (example 4). If the numbers are after the token, then they are also displayed as a subscript (example 5).

Numeric constants are written in the Pascal syntax (example 6). You can use the English "e" or "E" to indicate the exponential part of the number. The optimal format for displaying the number is selected automatically. If this format doesn't suit you for some reason, you can use the “#” symbol or the `Num` function.

Example number	Entry	Result
1	`'Alpha, alpha'`	A, α
2	`'EnglEnc'`	***EnglEnc***
3	`'РУС'`	РУС
4	`'x12'`	x_{12}
5	`'beta0'`	β_0
6	`'1.234,0.7e9,1234678'`	1.234, 700000000, 1234678

Complex expressions consist of operands separated by operation symbols. Operands can be one of the following types:

1. Simple expression i.e. numeric constant (including starting with the symbol "#"), identifier or token.
2. Expression enclosed in round, square or figure brackets.
3. Function.

Example number	Entry	Result
7A	`‘B_(2*k)=((-1)^(n-1)**2*(2*k)!/(2*pi)^(2*k))*sum(1/n^(2*k),n=1,inf)’`	$B_{2k}=\dfrac{(-1)^{n-1}2(2k)!}{(2\pi)^{2k}}\sum_{n=1}^{\infty}\dfrac{1}{n^{2k}}$
7Б	`'line(u_i^2)=(2*planck^2/M/k/Theta)*((T^2/Theta^2)*int(x*diff(x)/(e^x-1),0,T//Theta)+1/4)'`	$\overline{u_i^2}=\dfrac{2\hbar^2}{Mk\Theta}\left(\dfrac{T^2}{\Theta^2}\int_0^{T/\Theta}\dfrac{x\,dx}{e^x-1}+\dfrac{1}{4}\right)$

2.2.1. Operation signs

Sign	Description	Example number	Entry	Result
+	Addition.	8	`'a+b'`	$a+b$
–	Subtraction.	9	`'a-b'`	$a-b$
*	Multiplication. Checks if the factors can be multiplied unsigned. If they can, then the multiplication sign is not used (example 10). The factors can be interchanged to provide multiplication without a character (example 11). If there are several numbers among the factors, then the numbers are combined into one (example 12). If multiplication without a symbol is impossible under any order of factors, the dot symbol is used (example 13).	10	`'5*x'`	$5x$
		11	`'y*2'`	$2y$
		12	`'2*a*3'`	$6a$
		13	`'sin(x)*cos(x)'`	$\sin x\cdot\cos x$
**	Multiplication. The factors are unsigned multiplied regardless of whether such multiplication is permissible. The permutation of the factors is not performed.	14	`'5**x'`	$5x$
		15	`'y**2'`	$y\,2$
		16	`'2**a**3'`	$2a\,3$
		17	`'sin(x)**cos(x)'`	$\sin x\cos x$
*.	Multiplication. The factors are multiplied using the dot sign. The permutation of the factors is not performed.	18	`'5*.x'`	$5\cdot x$
		19	`'y*.2'`	$y\cdot 2$
		20	`'2*.a*.3'`	$2\cdot a\cdot 3$
		21	`'sin(x)*.cos(x)'`	$\sin x\cdot\cos x$
*+	Multiplication. The factors are multiplied using the cross sign. The permutation of the factors is not performed.	22	`'5*+x'`	$5\times x$
		23	`'y*+2'`	$y\times 2$
		24	`'2*+a*+3'`	$2\times a\times 3$
		25	`'sin(x)*+cos(x)'`	$\sin x\times\cos x$
/	Division. The division in the form of a simple fraction is always used. In complex expressions using various multiplication and division symbols in random order, distributes the factors between the numerator and denominator (example 27). To take the factor	26	`'a/b'`	$\dfrac{a}{b}$
		27	`'(x+1)/(x-1)*(x+2)/(x-2)/(x//y)*4'`	$\dfrac{4(x+1)(x+2)}{(x-1)(x-2)x/y}$
		28	`'(1/2)*x'`	$\dfrac{1}{2}x$

	outside the fraction, you need to enclose the fraction in brackets (examples 28, 29 and 30)	29	`'(3/4)*((x+1)/(x-1))'`	$\frac{3}{4}\frac{x+1}{x-1}$
		30	`'(3/4)*(x+1)/(x-1)'`	$\frac{\frac{3}{4}(x+1)}{x-1}$
//	Division. A slash is used. In some cases, it removes the brackets incorrectly (example 32). In this case, it is recommended to use “!()” brackets (example 33)	31	`'a//b'`	a/b
		32	`'x//(2*y)'`	$x/2y$
		33	`'x//!(2*y)'`	$x/(2y)$
/+	Division.	34	`'a/+b'`	$a \div b$
:	Division.	35	`'(1/2):(1/3):(1/4)=12/2=6'`	$\frac{1}{2}:\frac{1}{3}:\frac{1}{4}=\frac{12}{2}=6$
\	Set difference.	36	`'A\B'`	$A \setminus B$
+-	Plus-minus.	37	`'a+-b'`	$a \pm b$
-+	Minus-plus.	38	`'a-+b'`	$a \mp b$
=	Equally.	39	`'a=b'`	$a = b$
==	Identically.	40	`'a==b'`	$a \equiv b$
=~	Equal or same order.	41	`'a=~b'`	$a \cong b$
~	Same order.	42	`'a~b'`	$a \sim b$
~~	Approximately equal.	43	`'a~~b'`	$a \approx b$
<>	Not equal.	44	`'a<>b'`	$a \neq b$
>	More.	45	`'a>b'`	$a > b$
>>	Mach more.	46	`'a>>b'`	$a \gg b$
>~	More or same order.	47	`'a>~b'`	$a \gtrsim b$
>=	More or equally.	48	`'a>=b'`	$a \geq b$
<	Less.	49	`'a<b'`	$a < b$
<<	Mach less.	50	`'a<<b'`	$a \ll b$
<~	Less or same order.	51	`'a<~b'`	$a \lesssim b$
<=	Less or equally.	52	`'a<=b'`	$a \leq b$
->	Tends to.	53	`'a->b'`	$a \to b$

2.2.2. Characters before the operand

The operand may be preceded by the character "_", denoting a vector, and/or the characters "+", "-", "+-", "-+", denoting the corresponding unary operations. For numeric constants, the "#" character can be used.

Symbol	Description	Example number	Entry	Result
"_"	Vector. In some cases, incorrect placement of brackets is possible, then it is recommended to use the function `Vect`.	54	`'_a'`	$\vec{a}$
"+"	Unary plus.	55	`'+a'`	$+a$
"-"	Unary minus.	56	`'-a'`	$-a$
"+-"	Unary plus-minus.	57	`'+-a'`	$\pm a$
"-+"	Unary minus-plus.	58	`'-+a'`	$\mp a$
"#"	Preceded by numeric constants to indicate that they should be output in scientific format.	59	`'0.03*x'`	$0{,}03x$
		60	`'#0.03*x'`	$3\cdot10^{-2}x$

2.2.3. Characters after the operand

If `TExprBuilder.PostSymbols:=True` (by default), the operand may be followed by the characters "_", "^", "!" or "`".

Symbol	Description	Example number	Entry	Result
"_"		61	`'a_b'`	a_b

	Subscript. In some cases, incorrect placement of brackets or multi-stage indexes is possible (example 62). In such cases, it is recommended to use the Ind function (example 63)	62	'a_x_0'	a_{x0}
		63	'Ind(a,Ind(x,0))'	a_{x_0}
"^"	Superscript or exponentiation. In some cases, incorrect placement of brackets or multi-stage indexes is possible. In such cases, it is recommended to use the Pow function. If used together with a subscript (character "_" or the Ind function), then the subscript is indicated first, and then the superscript (example 65), otherwise the indices will not be displayed correctly (example 66).	64	'a^b'	a^b
		65	'Ind(x,a)^2, x_a^2'	x_a^2, x_a^2
		66	'Ind(x^2,a), x^2_a'	$\left(x^2\right)_a, x^{2_a}$
"!"	Factorial.	67	'C_n^k=n!/k!/(n-k)!'	$C_n^k = \frac{n!}{k!(n-k)!}$
"`"	Derivative. It is permissible to use several characters in a row to denote derivatives of higher degrees. In some cases, incorrect placement of brackets is possible. In such cases, it is recommended to use the Strokes function.	68	'f(x)`'	$f'(x)$
		69	'f(x)```'	$f'''(x)$

2.2.4. Expression concatenation

Multiple expressions can be concatenated into one using the "&" symbol (not drawn in the formula image) or the separators "," and ";" (drawn in the image of the formula).

Symbol	Description	Example number	Entry	Result
"&"	Concatenation of two expressions. The character must either not be separated from both expressions by spaces (example 69), or separated from each of them by one space (example 70)	70	'x&y'	xy
		71	'x & y'	$x\,y$
"," ";"	A character that separates multiple consecutive expressions. There can be an arbitrary number of spaces after the character, but this does not affect the spacing between expressions, which is seven units of width (see 1.47). There must be no spaces before the character.	72	'a0,a1, a2, a3'	a_0, a_1, a_2, a_3
		73	'a0;a1; a2; a3'	$a_0; a_1; a_2; a_3$

2.2.5. Brackets

Symbols	Descriprion	Example number	Entry	Result
( )	Parentheses are used to change the order in which actions are performed. They can be removed by the expression builder, if this does not lead to a distortion of the meaning of the expression. Use the '!()' brackets to force parentheses.	74	'(x+1)*(y-2)'	$(x+1)(y-2)$
		75	'(x+1)/(x-1)'	$\frac{x+1}{x-1}$
		76	'a+(b+c)=d*.(e*.f)'	$a+\left(b+c\right)=d\cdot e\cdot f$
		77	'y=(1+1/(1+1/x))'	$y=\left(1+\frac{1}{1+\frac{1}{x}}\right)$
!( )	They are used in the same place as regular parentheses, but are never removed by the expression builder.	78	'!(x+1)*!(y-2)'	$(x+1)(y-2)$
		79	'!(x+1)/!(x-1)'	$\frac{(x+1)}{(x-1)}$
		80	'a+!(b+c)=d*.!(e*.f)'	$a+\left(b+c\right)=d\cdot\left(e\cdot f\right)$
		81	'y=!(1+1/!(1+1/x))'	$y=\left(1+\frac{1}{\left(1+\frac{1}{x}\right)}\right)$
[ ]	Square brackets. Never removed by the expression builder.	82	'[x+1]*[y-2]'	$[x+1][y-2]$
		83	'[x+1]/[x-1]'	$\frac{[x+1]}{[x-1]}$
		84	'a+[b+c]=d*.[e*.f]'	$a+\left[b+c\right]=d\cdot\left[e\cdot f\right]$
		85	'y=[1+1/[1+1/x]]'	$y=\left[1+\frac{1}{\left[1+\frac{1}{x}\right]}\right]$

{ }	Braces. Never removed by the expression builder.	86	'{x+1}*{y-2}'	$\left\{x+1\right\}\left\{y-2\right\}$
		87	'{x+1}/{x-1}'	$\frac{\left\{x+1\right\}}{\left\{x-1\right\}}$
		88	'a+{b+c}=d*.{e*.f}'	$a+\left\{b+c\right\}=d\cdot\left\{e\cdot f\right\}$
			'y={1+1/{1+1/x}}'	$y=\left\{1+\frac{1}{\left\{1+\frac{1}{x}\right\}}\right\}$
| |	Straight brackets. Never removed by the expression builder.	89	'|x+1|*|y-2|'	$\left\lvert x+1\right\rvert\left\lvert y-2\right\rvert$
		90	'|x+1|/|x-1|'	$\frac{\lvert x+1\rvert}{\lvert x-1\rvert}$
		91	'a+|b+c|=d*.|e*.f|'	$a+\left\lvert b+c\right\rvert=d\cdot\left\lvert e\cdot f\right\rvert$
		92	'y=|1+1/|1+1/x||'	$y=\left\lvert 1+\frac{1}{\left\lvert 1+\frac{1}{x}\right\rvert}\right\rvert$

More options for displaying brackets are provided by the function `Brackets`.

2.2.6. Greek character tokens

Token	Character	Token	Character	Token	Character	Token	Character
Alpha	Α	alpha	α	Nu	Ν	nu	ν
Beta	Β	beta	β	Xi	Ξ	xi	ξ
Gamma	Γ	gamma	γ	Omicron	Ο	omicron	ο
Delta	Δ	delta	δ	Pi	Π	pi	π
Epsilon	Ε	epsilon	ε	Rho	Ρ	rho	ρ
Zeta	Ζ	zeta	ζ	Sigma	Σ	sigma	σ
Eta	Η	eta	η			sigmao	ς
Theta	Θ	theta	θ	Tau	Τ	tau	τ
		thetao	ϑ	Upsilon	ϒ	upsilon	υ
Iota	Ι	iota	ι	Phi	Φ	phi	φ
Kappa	Κ	kappa	κ	Chi	Χ	chi	χ
Lambda	Λ	lambda	λ	Psi	Ψ	psi	ψ
Mu	Μ	mu	μ	Omega	Ω	omega	ω

2.2.7. Arrow character tokens

Token	Character	Token	Character	Token	Character
ArrowL	←	DArrowL	⇐	BArrowL	↤
ArrowR	→	DArrowR	⇒	BArrowR	↦
ArrowU	↑	DArrowU	⇑	BArrowU	↥
ArrowD	↓	DArrowD	⇓	BArrowD	↧
ArrowLR	↔	DArrowLR	⇔	TArrowRL	⇄
ArrowUD	↕	DArrowUD	⇕	TArrowLR	⇆
ArrowLU	↖	DArrowLU	⇖	TArrowUD	⇅

ArrowRU	↗	DArrowRU	⇗	TArrowDU	⇵
ArrowRD	↘	DArrowRD	⇘	TArrowL	⇇
ArrowLD	↙	DArrowLD	⇙	TArrowR	⇉
ArrowLN	↚	DArrowLN	⇍	TArrowU	⇈
ArrowRN	↛	DArrowRN	⇏	TwoArrowD	⇊
ArrowLRN	↮	DArrowLRN	⇎	TRArrow	⇶
HArrowLR	⇋	HArrowRL	⇌		

2.2.8. Other tokens

Token	Character	Description
...	…	Ellipsis located at the level of punctuation marks
And	⋀	Logic "AND"
Angle	∠	Angle sign
Arc	⌣	Arc sign
Asterisk	∗	Asterisk - the symbol "*", somewhat lowered down compared to the usual position. Intended for use in superscripts.
AsteriskC	⊛	Asterisk in a circle
Begin	◀	Beginning of proof (calculations)
Belongs	∈	Belongs
BelongsN	∉	Not belongs
Circle	○	Circle
CircleC	⊚	Circle in circle
Colon	:	Colon. In most cases, it can be replaced with the Colon function
Comma	,	Comma. Unlike the "," character, it does not insert spaces after the comma. In most cases, it can be replaced by the "," character or by the Comma function
Complex	ℂ	Set of complex numbers
Const	const	The word "const", denoting an arbitrary constant. Unlike regular identifiers, it is written in roman type, not in italics.
CrossC	⊗	Cross in circle
Degree	°	Degree
Divide	∣	Devides
DivideN	∤	Not Devides
Dot	.	Dot
DotC	⊙	Dot in circle
DotEqual	≐	Approaching the limit
DotEqualC	≑	Geometrically equal
DotEqualLR	≒	Represent
DotEqualRL	≓	Represent
DotsD	⋱	Diagonal dots. Can be used, for example, to indicate missing elements of a matrix
DotsH	⋯	Horizontal dots. Can be used, for example, to indicate missing elements of a matrix
DotsU	⋰	Diagonal dots. Can be used, for example, to indicate missing elements of a matrix
DotsV	⋮	Vertical dots. Can be used, for example, to indicate missing elements of a matrix
Empty		Empty expression. Unlike Nil, it only has a width of zero, and the height is equal to the height of the characters.

`EmptySet`	∅	Empty set
`End`	▶	Ending of proof (calculations). Q.E.D
`Entire`	ℤ	Set of integers
`EqualC`	⊜	Equal in circle
`Exists`	∃	Exists
`ExistsN`	∄	Not exists
`ForAll`	∀	For all
`Ident`	≡	Identical. Sign analogue "=="
`IdentN`	≢	Not identical
`Inf`	∞	Infinity
`Intersection`	∩	Intersection
`Minus`	−	Minus. Designed for use primarily in indexes. Other characters ("+", "=", etc.) can be displayed using the `String` function. But the expression "`String(-)`" will not give a minus, but a hyphen, which is significantly shorter.
`MinusC`	⊖	Minus in circle
`Nabla`	∇	Hamilton operator
`Natural`	ℕ	Set of natural numbers
`Nil`		An empty expression with zero dimensions. Intended for use in functions where, according to the syntax, there should be an expression, but in a specific case it is required that it should not be
`Not`	¬	Logical negation
`Or`	⋁	Logical "OR"
`Parallel`	∥	Parallel
`ParallelN`	∦	Not parallel
`Parallelogram`	▱	Parallelogram
`Perpendicular`	⊥	Perpendicular
`PLambda`	ƛ	Lambda with dash. Used, for example, in quantum mechanics
`Planck`	ħ	Planck's constant
`PlusC`	⊕	Plus in circle
`Projective`	ℙ	Projective space
`Prop`	∝	Proportionally
`Quadrate`	□	Quadrate
`Quaternion`	ℍ	Hamilton's set of quaternions
`Rational`	ℚ	Set of rational numbers
`Real`	ℝ	Set of real numbers
`Rectangle`	▭	Rectangle
`Rhomb`	◇	Rhomb
`Semicolon`	;	Semicolon. Unlike the character "`;`" does not insert spaces after the semicolon. In most cases, it can be replaced by the character "`;`" or the `Semicolon` function.
`Similar`	∾	Similar
`SlashC`	⊘	Slash in circle
`SubSet`	⊂	Subset
`SubSetN`	⊄	Not subset
`SuperSet`	⊃	Superset

`SuperSetN`	$\not\supset$	Not superset
`Triangle`	$\triangle$	Triangle
`Union`	$\cup$	Union
`Xor`	$\underline{\vee}$	Logical "XOR"

2.2.9. Functions

A function is a text followed by one or more arguments in parentheses. There are reserved function names, which are described below.

If the function name is not a reserved word, the result depends on the length of the name. Single-character function names are displayed in italics, and their arguments are always enclosed in parentheses (Example 93). Longer function names are displayed in roman type, and their arguments are enclosed in parentheses only when necessary (example 94). You can force parentheses around an argument by using "`!(`" instead of the opening parenthesis (Example 95).

When raising a function to a power or adding an index to it, the signs of the corresponding operations must appear after the argument (example 96).

Digits at the end of a function name are interpreted as a subscript if `TExprBuilder.FuncAutoIndex:=True` (by defaults; example 97).

As function names you can use tokens denoting Greek letters, as well as tokens `Nabla` and `PLambda`. In this case, the digits at the end are also interpreted as a subscript (example 98).

Example number	Entry	Result
93	`'f(x,y,z)'`	$f\left(x, y, z\right)$
94	`'cos(x), sin(pi/2+x), tg(1/x)'`	$\cos x, \sin\left(\frac{\pi}{2}+x\right), \operatorname{tg}\frac{1}{x}$
95	`'cos!(x)'`	$\cos\left(x\right)$
96	`'f(x)_n,cos(x)^2'`	$f_n\left(x\right), \cos^2 x$
97	`'f0(x)=g1(x)'`	$f_0\left(x\right)=g_1\left(x\right)$
98	`'gamma0(x)'`	$\gamma_0\left(x\right)$

2.2.10. Reserved functions

Reserved function names are case insensitive. When describing reserved functions, the following conventions are used:

`E,E1,E2` etc. – required expressions (if you want the required expression to be empty, you need to insert the `Nil` token instead (examples 138, 167));

`[,E1]` etc. – optional expressions (may not be written depending on the situation);

`m,n,n1,n2` etc. – integer constants;

`R` - real constant.

Width units used in some reserved functions to specify the amount of white space are chosen so that one width unit is approximately equal to the thickness of the vertical line in the "+" symbol.

Reserved function	Description	Example number	Entry	Result
`Abs(E)`	The absolute value of E. An analogue of brackets "`||`", added for compatibility with Pascal syntax	99	`'Abs(x^2)=Abs(x)^2'`	$\left\|x^2\right\|=\left\|x\right\|^2$
`At(E1[,E2[,E3]])`	E1 value under E2 condition (example 100) or E1 value from E2 to E3 (example 101)	100	`'At(DiffRF(f,x),x=0)=1'`	$\left.\frac{df}{dx}\right\|_{x=0}=1$
		101	`'Int(x*Diff(x),a,b)= At(x^2/2,a,b)= (b^2-a^2)/2'`	$\int_a^b x\,dx=\left.\frac{x^2}{2}\right\|_a^b=\frac{b^2-a^2}{2}$
`Brackets(S1S2,E)`	Encloses E in various brackets. S1 can be the symbol "`(`", "`[`", "`{`", "`|`" or "`0`", "`1`", "`2`", "`3`", "`4`". S2 can be the symbol "`)`", "`]`", "`}`", "`|`" or "`0`", "`1`", "`2`", "`3`", "`4`", "`0`" - means no parenthesis on that side,	102	`'Brackets((],0&comma&1)'`	$\left(0,1\right]$
		103	`'Brackets(11, _a&comma(5)&_b)'`	$\left\langle\vec{a}, \vec{b}\right\rangle$
		104	`'Brackets(22,3.4)=3<> Brackets(33,3.4)=4'`	$\left\lfloor 3{,}4\right\rfloor=3\neq\left\lceil 3{,}4\right\rceil=4$
		105	`'Brackets(23,3.4)=3'`	$\left\lfloor 3{,}4\right\rceil=3$

	"1" - angle brackets "⟨" and "⟩" (to denote, for example, the scalar product of vectors), "2" – brackets "⌊" and "⌋", denoting rounding down to the nearest integer, "3" – brackets "⌈" and "⌉", denoting rounding up to the nearest integer, "4" - double straight brackets.	106	`'Brackets(23,3.6)=4'`	$\lfloor 3{,}6 \rceil = 4$
		107	`'Brackets(44,_x)'`	$\left\Vert \vec{x} \right\Vert$
`Cap(E)`	Sign "^" over E	108	`'Cap(x)'`	$\hat{x}$
`CaseAnd(E[,...])`	Selecting one of the possible options with the "AND" condition. Expressions in parentheses are pairs of conditional options.	109	`'|x|=CaseAnd(-x,x<0,0,x=0,x,x>0)'`	$\lvert x \rvert = \begin{cases} -x & x<0 \\ 0 & x=0 \\ x & x>0 \end{cases}$
`CaseOr(E[,...])`	Select one of the possible options with the "OR" condition. Expressions in parentheses are pairs of conditional options.	110	`'f(x)=CaseOr(1//x,x<>0,0,x=0)'`	$f(x) = \left[\begin{array}{ll} 1/x & x \neq 0 \\ 0 & x = 0 \end{array} \right.$
`Circ(E1[,E2[,E3]])`	Closed-loop integral of the expression E1. E2 is placed under the integration sign, E3 is placed above it	111	`'Circ(F*Diff(L),L)'`	$\oint\limits_L F\,dL$
`Colon(n)`	Inserts a colon into the expression followed by a space n width units of width	112	`'x & colon(15) & y'`	$x{:}\quad y$
`Comma(n)`	Inserts a comma into the expression followed by a space n width units of width	113	`'x & comma(15) & y'`	$x,\quad y$
`Corr(E[,...])`	Arranges expressions in a lookup table of two columns separated by a vertical bar. Expressions in parentheses are pairs of corresponding values	113A	`|Corr(u=x, u`=1, v`=sin(x), v=-cos(x))|`	$\left\lvert \begin{array}{l} u = x \\ v' = \sin x \end{array} \quad \middle\lvert \begin{array}{l} u' = 1 \\ v = -\cos x \end{array} \right\rvert$
`Diff(E1[,E2])`	Differential of the expression E1 raised to the power of E2	114	`'Diff(x)'`	dx
		115	`'Diff(x,n)'`	dx^n
`DiffN(E1[,E2])`	Differential with E2 degree of E1 expression	116	`'DiffN(x)'`	dx
		117	`'DiffN(x,n)'`	$d^n x$
`DiffR(E1[,E2])`	Derivative with E2 degree with respect to E1	118	`'DiffR(x)'`	$\frac{d}{dx}$
		119	`'DiffR(x,n)'`	$\frac{d^n}{dx^n}$
		120	`'DiffR(x,2)*f(x)=DiffR(x)*DiffR(x)*f(x)'`	$\frac{d^2}{dx^2} f(x) = \frac{d}{dx} \frac{d}{dx} f(x)$
`DiffRF(E1,E2[,E3])`	Derivative with E3 degree of E1 with respect to E2	121	`'DiffRF(f,x)'`	$\frac{df}{dx}$
		122	`'DiffRF(f(x),x,n)'`	$\frac{d^n f(x)}{dx^n}$
`Dot(n)`	Inserts a dot into the expression followed by a space n width units of width	123	`'x & dot(15) & y'`	$x.\quad y$
`Fact(E)`	Factorial E	124	`'Fact(n)'`	$n!$
		125	`'Fact(k+1)'`	$(k+1)!$
`Func(E1,E2)`	Function with name E1 and argument E2	126	`'Func(PDiffRF(f,x,3),x)'`	$\frac{\partial^3 f}{\partial x^3}(x)$

`FuncSub("Name", E1,E2)`	Function named "Name" of expression E2 on condition E1	127	`'FuncSub("Res",z=z0,f(z))'`	$\operatorname*{Res}_{z=z_0} f(z)$
		128	`'FuncSub("max", x&Belongs&[a,b],f(x))'`	$\max_{x\in[a,b]} f(x)$
`Ind(E1,E2)`	Adding a subscript to E1 as E2. In most cases, it can be replaced by the character "_". When used with the `Pow` function, must be applied before `Pow` (example 130)	129	`'Ind(a,n)'`	a_n
		130	`'Pow(Ind(x,n),2)'`	x_n^2
`Int(E1[,E2[,E3]])`	Integral of expression E1. E2 is placed under the integral sign, E3 is placed above it	131	`'F(x)=Int(f(x)*Diff(x))'`	$F(x)=\int f(x)\,dx$
		132	`'Phi=Int(_H*Diff(_S),S)'`	$\Phi=\int_S \vec{H}\,d\vec{S}$
		133	`'Int(Diff(x),0,1)=1'`	$\int_0^1 dx=1$
		134	`'M=Int(Diff(x)*Int(x*y* Diff(y),0,1-x),0,1)'`	$M=\int_0^1 dx\int_0^{1-x} xy\,dy$
`IntM(n, E1[,E2[,E3]])`	Multiple integrals with fold `n` of the expression E1. E2 is placed under the integral sign, E3 is placed above it. If n=0, an integral with unknown multiplicity is drawn (ellipsis is used)	135	`'IntM(3,f(x,y,z)*Diff(x)* Diff(y)*Diff(z),V)'`	$\iiint_V f(x,y,z)\,dx\,dy\,dz$
		136	`'IntM(0,f(x1,...,x_n)* DiffN(x,n))'`	$\int\!\!\int\ldots\int f(x_1,\ldots,x_n)\,d^n x$
`Lim(E1,E2)`	Limit of the expression E2 with the condition E1	137	`'Lim(x->0,f(x))'`	$\lim_{x\to0} f(x)$
		138	`'Lim(Nil,f)'`	$\lim f$
		139	`'Lim(StandC( x->0,x>0),f(x))=1'`	$\lim_{\substack{x\to0\\x>0}} f(x)=1$
`Line(E)`	Horizontal line over E	140	`'Line(x)'`	$\overline{x}$
`Log(E1,E2)`	Logarithm of E2 to the base of E1	141	`'log(a,x+1)=ln(x+1)/ln(a)'`	$\log_a(x+1)=\frac{\ln(x+1)}{\ln a}$
`Matrix( n,m,E[,...])`	An m by n matrix. The expressions E and those following it are placed in the cells of the matrix. There can be fewer expressions than m*n - in this case, the last cells remain empty. The matrix is not framed by brackets, brackets must be added explicitly	142	`'Matrix(2,3,x,y,x-y, x+y,z,z+y)'`	$\begin{matrix}x&y\\x-y&x+y\\z&z+y\end{matrix}$
		143	`'!(Matrix(2,2,1,2,-3,4))'`	$\begin{pmatrix}1&2\\-3&4\end{pmatrix}$
		144	`'det(A)=|Matrix(3,3,a_11, DotsH,a_(1&n),DotsV,DotsD, DotsV,a_m1,DotsH,a_mn)|'`	$\det A=\begin{vmatrix}a_{11}&\cdots&a_{1n}\\\vdots&\ddots&\vdots\\a_{m1}&\cdots&a_{mn}\end{vmatrix}$
		145	`'[_a,_b]=|Matrix(3,3,_e_x, _e_y,_e_z,x_a,y_a,z_a,x_b, y_b,z_b)|'`	$[\vec{a},\vec{b}]=\begin{vmatrix}\vec{e}_x&\vec{e}_y&\vec{e}_z\\x_a&y_a&z_a\\x_b&y_b&z_b\end{vmatrix}$
`Num( R[,n1[,n2[,n3]]])`	Allows you to control the notation format of the number R. If the order of the number is less than or equal to -n3, the scientific notation is used with an accuracy of n1, if it is greater - the usual notation with the number of digits before the point n1 and the total n2. Default n1=4, n2=4, n3=2	146	`'Num(0.00123456)'`	$1{,}235\cdot10^{-3}$
		147	`'Num(0.00123456, 6)'`	$1{,}23456\cdot10^{-3}$
		148	`'Num(0.00123456, 4, 4, 3)'`	0,0012
		149	`'Num(0.00123456, 4, 6, 3)'`	0,001235
`PDiff(E1[,E2])`	The "partial differential" of the expression E1 raised to the power of E2. From a mathematical point of view, such a "differential" does	150	`'PDiff(x)'`	∂x
		151	`'PDiff(x,n)'`	∂x^n

	not make sense, but the function is very convenient for creating expressions like example 152	152	'PDiffN(f(x,y),3)/PDiff(x)/PDiff(y,2)'	$\frac{\partial^3 f(x,y)}{\partial x\,\partial y^2}$
PDiffN(E1[,E2])	The "partial differential" of the power E2 of the expression E1. From a mathematical point of view, such a "differential" does not make sense, but the function is very convenient for creating expressions like example 152	153	'PDiffN(x)'	∂x
		154	'PDiffN(x,n)'	$\partial^n x$
PDiffR(E1[,E2])	Partial derivative of the degree E2 with respect to E1	155	'PDiffR(x)'	$\frac{\partial}{\partial x}$
		156	'PDiffR(x,n)'	$\frac{\partial^n}{\partial x^n}$
		157	'Nabla=PDiffR(x)*_e_x+ PDiffR(y)*_e_y+ PDiffR(z)*_e_z'	$\nabla = \frac{\partial}{\partial x}\vec{e}_x + \frac{\partial}{\partial y}\vec{e}_y + \frac{\partial}{\partial z}\vec{e}_z$
PDiffRF(E1,E2[,E3])	Partial derivative of the degree E3 of the expression E1 with respect to E2	158	'PDiffRF(f,x)'	$\frac{\partial f}{\partial x}$
		159	'PDiffRF(f(x,y),x,n)'	$\frac{\partial^n f(x,y)}{\partial x^n}$
Points(E[,n])	Dots over E, usually meaning derivative with respect to time	160	'Points(y,2)=y*Points(x)'	$\ddot{y} = y\dot{x}$
Pow(E1,E2)	Raising E1 to the power of E2. When used with a Ind function, must be applied after Ind (Example 162). In most cases, it can be replaced by the "^" symbol	161	'Pow(x+2,2//3)'	$(x+2)^{2/3}$
		162	'Pow(Ind(x,a),3)'	x_a^3
Prod(E1[,E2[,E3]])	Product of expressions E1. E2 is placed under the product sign, E3 is placed above it	163	'Prod(a_i)'	$\prod a_i$
		164	'Prod(a_i,i<>j)'	$\prod\limits_{i \neq j} a_i$
		165	'Prod(a_i,i=0,n)'	$\prod\limits_{i=0}^{n} a_i$
Root(E1,E2)	Extracting the root of the degree E1 from the expression E2	166	'Root(3,x-1)'	$\sqrt[3]{x-1}$
		167	'Root(nil, x-1)'	$\sqrt{x-1}$
Semicolon(n)	Inserts a semicolon into the expression followed by a space n width units of width	168	'x & semicolon(15) & y'	$x;\quad y$
Space(n)	A space of n units of width. Used to separate expressions	169	'y=x & space(7) & z=q'	$y = x\quad z = q$
Sqr(E)	Squaring expression E. Has no advantage over using the "^" character or the Pow function. Added for compatibility with Pascal syntax	170	'Sqr(a+b)= Sqr(a)+2*a*b+Sqr(b)'	$(a+b)^2 = a^2 + 2ab + b^2$
Sqrt(E)	Extracting the square root of E	171	'Sqrt(x^2+y^2)'	$\sqrt{x^2+y^2}$
StandC(E[,...])	Places multiple expressions one below the other, centered	172	'StandC(0<=i<n,i<>j)'	$\begin{array}{c} 0 \le i < n \\ i \neq j \end{array}$
StandL(E[,...])	Places multiple expressions one below the other, left aligned	173	'StandL(0<=i<n,i<>j)'	$\begin{array}{l} 0 \le i < n \\ i \neq j \end{array}$
StandR(E[,...])	Places multiple expressions one below the other, right aligned	174	'StandR(0<=i<n,i<>j)'	$\begin{array}{r} 0 \le i < n \\ i \neq j \end{array}$
String(Текст) String("Текст")	Text displayed in roman type without modification. If the text	175	'String(Произвольный текст)'	Произвольный текст

	contains parentheses, it must be enclosed in double quotes.	176	`'String("Текст (со скобками)")'`	Текст (со скобками)
`Strokes(E[,n])`	Adds strokes to E, usually denoting a derivative	177	`'Strokes(f(x))'`	$f'\left(x\right)$
		178	`'Strokes(y,3)'`	y'''
`Sum(E1[,E2[,E3]])`	The sum of expressions E1. E2 is placed under the sum sign, E3 is placed above it	179	`'Sum(a_i)'`	$\sum a_i$
		180	`'Sum(a_i,i<>j)'`	$\sum\limits_{i \neq j} a_i$
		181	`'Sum(a_i,i=0,n)'`	$\sum\limits_{i=0}^{n} a_i$
`Surf(E1[,E2[,E3]])`	Integral over a closed surface of the expression E1. E2 is placed under the integration sign, E3 is placed above it	182	`'Surf(F*Diff(S),S)'`	$\oiint\limits_S F\,dS$
`Symbol(n)`	Inserts a UTF-16BE encoded character with decimal code n into an expression in roman	183	`'Symbol(198)=1'`	Æ = 1
`SymbolI(n)`	Inserts a UTF-16BE encoded character with decimal code n into an expression in italic	184	`'SymbolI(198)=1'`	*Æ* = 1
`SystemAnd(E[,...])`	Combines expressions into a figure bracket system (with "AND" condition)	185	`'SystemAnd(x+y=5,x*y=6)'`	$\begin{cases} x+y=5 \\ xy=6 \end{cases}$
		186	`'SystemAnd(x+y=5 & Semicolon,x*y=6 & Dot)'`	$\begin{cases} x+y=5; \\ xy=6. \end{cases}$
`SystemOr(E[,...])`	Combines expressions into square bracket system (with "OR" condition)	187	`'x^2=4 & space(7) & DArrowR & space(7) & SystemOr(x=2 & comma, x=-2 & Dot)'`	$x^2=4 \quad \Rightarrow \quad \left[\begin{array}{l} x=2, \\ x=-2. \end{array}\right.$
`Tilde(E)`	Sign "~" over E	188	`'Tilde(x)'`	$\tilde{x}$
`Vect(E)`	Arrow (vector sign) above E. In most cases, can be replaced by the character "_" before E	189	`'Vect(a)'`	$\vec{a}$
`Volume( E1[,E2[,E3]])`	Integral over a closed volume of the expression E1. E2 is placed under the integration sign, E3 is placed above it	190	`'Volume(F*Diff(V),V)'`	$\oiiint\limits_V F\,dV$

2.3. Examples of using

Example 1	
Entry	`'Si!(x)=Int((sin(x)/x)*Diff(x),0,x)=pi/2-Int((sin(x)/x)*Diff(x),x,+Inf)= x-(1/3!)*(x^3/3)+(1/5!)*(x^5/5)-+...'`
Result	$\mathrm{Si}\left(x\right)=\int\limits_0^x \frac{\sin x}{x}\,dx=\frac{\pi}{2}-\int\limits_x^{+\infty}\frac{\sin x}{x}\,dx=x-\frac{1}{3!}\frac{x^3}{3}+\frac{1}{5!}\frac{x^5}{5}\mp\ldots$

Example 2	
Entry	`'FuncSub("Res",z=a,f(z))=lim(z->a,[(1/(m-1)!)*.DiffR(z,m-1)*[(z-a)^m*.f(z)]])'`
Result	$\operatorname*{Res}\limits_{z=a} f\left(z\right)=\lim\limits_{z\to a}\left[\frac{1}{\left(m-1\right)!}\cdot\frac{d^{m-1}}{dz^{m-1}}\left[\left(z-a\right)^m\cdot f\left(z\right)\right]\right]$

Example 3	
Entry	`'P(H_i&Divide&E)=P(H_i&Intersection&E)/P(E)= P(H_i)*P(E&Divide&H_i)/Sum(P(H_i)*P(E&Divide&H_i),i)'`
Result	$P\left(H_i\mid E\right)=\frac{P\left(H_i\cap E\right)}{P\left(E\right)}=\frac{P\left(H_i\right)P\left(E\mid H_i\right)}{\sum\limits_i P\left(H_i\right)P\left(E\mid H_i\right)}$

Example 4	
Entry	`'{StandC(1,1&space(15)&2)}_S&Ident&{StandC(1,2&space(15)&1)}_S&Ident& (G*E_thetao-F*G_u)/(2*(E*G-F^2))'`
Result	$\left\{ {\ \ 1 \atop 1\ \ 2} \right\}_S \equiv \left\{ {\ \ 1 \atop 2\ \ 1} \right\}_S \equiv \frac{GE_\theta - FG_u}{2\left(EG-F^2\right)}$

Example 5	
Entry	`'SystemAnd(a11*x1+a12*x2+...+a_(1&n)*x_n=b1 & Comma, a21*x1+a22*x2+...+a_(2&n)*x_n= b2 & Comma,DotsV,a_(n&1)*x1+a_(n&2)*x2+...+a_nn*x_n=b_n & Semicolon)& String(     или     ) & Sum(a_(i*k)*b_i,k=1,n),!(i=1,2,...,n)'`
Result	$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n; \end{cases}$ или $\sum_{k=1}^{n} a_{ik}b_i,\ \left(i = 1,\ 2,\ \ldots,\ n\right)$

Example 6	
Entry	`'A*.X=B & String(,     где   )&A= !(Matrix(4,4,a11,a12,DotsH,a_(1&n),a21,a22,DotsH,a_(2&n),DotsV,DotsV,DotsH,DotsV,a_(n&1), a_(n&2),DotsH,a_nn))&comma(20)&X= !(StandC(x1,x2,DotsV,x_n))&comma(20)&B=!(StandC(b1,b2,DotsV,b_n)) & Dot'`
Result	$A \cdot X = B,$ где $A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$

Example 7	
Entry	`'Int(Int(x*y*Diff(x)*Diff(y),x^2,sqrt(x)),0,1)= Int(At([(1/2)*x*y^2],x^2,sqrt(x))**Diff(x),0,1)= (1/2)*Int(x*(x-x^4)*Diff(x),0,1)=(1/2)*At(x^3/3-x^6/6,0,1)=1/12'`
Result	$\int_0^1 \int_{x^2}^{\sqrt{x}} xy\,dx\,dy = \int_0^1 \left[\frac{1}{2}xy^2\right]\Bigg\|_{x^2}^{\sqrt{x}} dx = \frac{1}{2}\int_0^1 x\left(x - x^4\right)dx = \frac{1}{2}\left(\frac{x^3}{3} - \frac{x^6}{6}\right)\Bigg\|_0^1 = \frac{1}{12}$