Zero Calculator

ZeroBasic User Manual (v2.27.1 dated 12.29.2025)

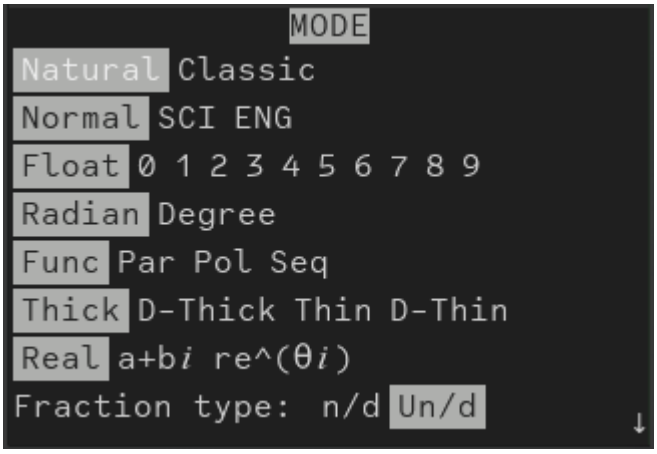
ZeroBasic is a programming language used for working with Zero Calculator. Zero Calculator supports three modes for working with the ZeroBasic language: the expression mode, the main screen expression mode, and the script file mode. Each mode has its own characteristics in processing the ZeroBasic language.

I. Lexical structure of the ZeroBasic language

The lexical structure of the ZeroBasic language describes the methods and rules for extracting a set of lexemes from the source code, which are then transformed into tokens. For example, the lexemes `14`, `-8.5`, `6E-1` are converted into the `Number` token. The lexemes `[A]`, `FooBar`, `Ans`, `L₀` are converted into the `Identifier` token.

1.1. Processing mathematically formatted expressions

Zero Calculator has two modes for displaying expressions: **Classic** and **Natural**. You can switch modes in **MODE** window.



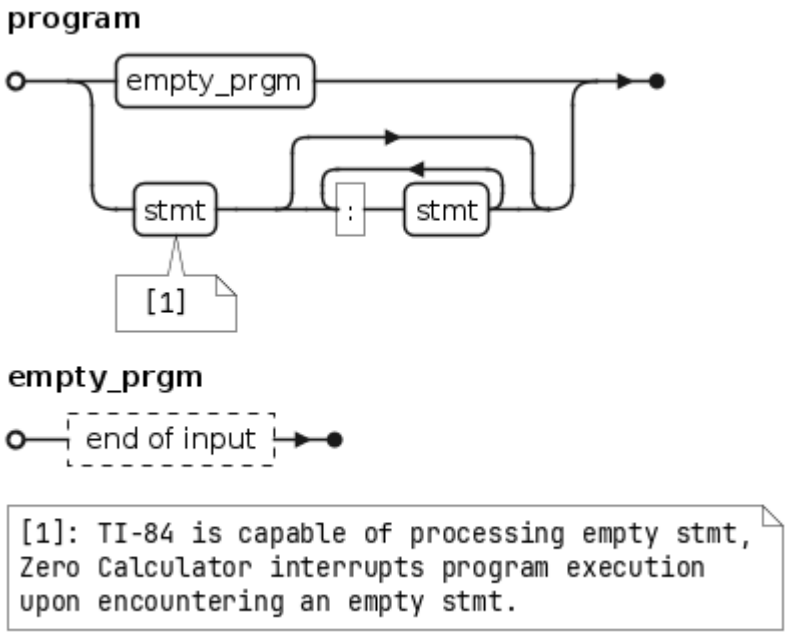
The **Classic** mode implies inputting expressions according to the lexical structure of the ZeroBasic language. The **Natural** mode allows mathematical formatting for certain expressions.

Name	Natural	Classic
Matrix	$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$	`[[A,B]\n[C,D]]`
Common Fraction	$\frac{A}{B}$	`((A)⁄(B))` ('⁄' \x9D)
Mixed Fraction	$1\frac{2}{3}$	`(1␣2⁄3)` ('␣' \xA0) ('⁄' \x9D)
Absolute Value	$\lvert A \rvert$	`abs(A)`
Square Root	$\sqrt{A}$	`√(A)` ('√' \x7F)

Name	Natural	Classic
Root with Power	$\sqrt[B]{A}$	`ⁿ√(A,B)` ('ⁿ√' \xA1)
Exponentiation	5^{A}	`5^(A)`
Logarithm	$\log_{A}(B)$	`logBASE(B,A)`
Derivative	$\frac{d}{dA}(B)\|_{A=C}$	`nDeriv(B,A,C)`
Integral	$\int_{A}^{B}(C)dD$	`fnInt(C,D,A,B)`
Sum	$\sum_{A=B}^{C}(D)$	`Σ(D,A,B,C)` ('Σ' \xA2)
Permutations	${}_{n}P_{r}$	`nPr(n,r)`
Combinations	${}_{n}C_{r}$	`nCr(n,r)`

1.2. Lexical structure of Main screen expressions

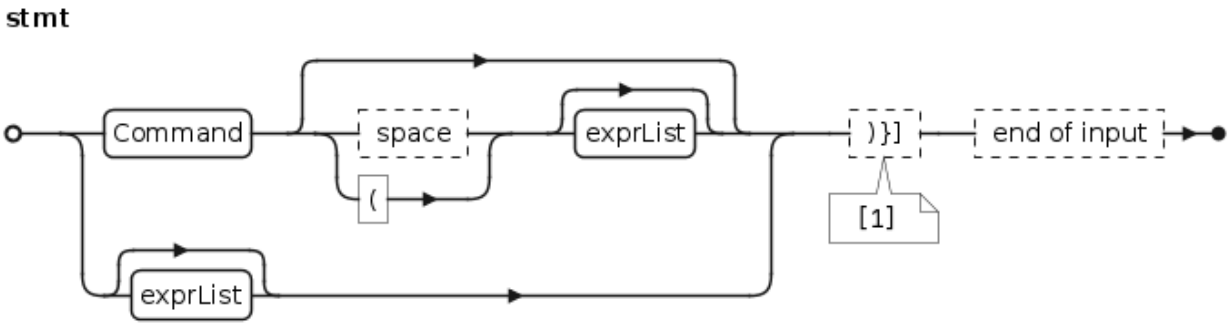
The lexical scheme for main screen expressions is shown below (text description of the structure in EBNF format is available at the end of the document):

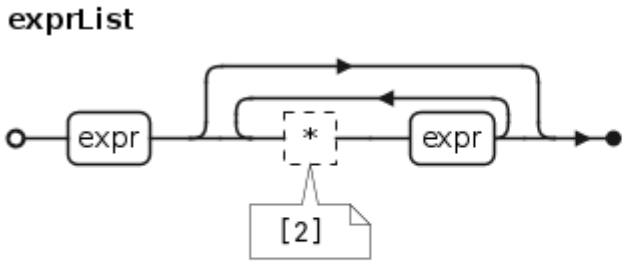


The lexical scheme of an expression (`stmt`) is discussed further in the section Lexical structure of expressions.

1.3. Lexical structure of an expression

In addition to the Main screen, Zero Calculator has many screens and modes where you can enter a single expression. A part of the lexical scheme of single expression is presented below (text description of the structure in EBNF format is available at the end of the document). The other part of the scheme is common to both expressions and the script file (the name of common tokens begins with a capital letter) and is discussed in the section The structure of lexemes common to a file and an expression.

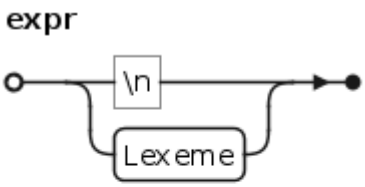


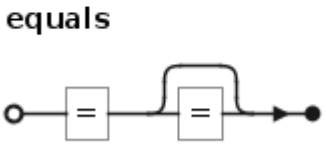



```
[1]: After lexical analysis, the list of lexemes is divided into blocks.
The block boundaries defined by [→]. At the end of each block, tokens
for any missing closing brackets ( [)], []], [}] ) are added.
The insertion order is the reverse of the opening brackets.
```

```
[2]: An insertion of the [*] token between two [expr] occurs in
the following cases:
    previous [expr]     next [expr]           example

  - [Number],           [Identifier]         "3X"
  - [Number],           [(]                  "3("
  - [Number],           [{]                  "3{"
  - [Number],           [[]                  "3["

  - [Imag],             [Identifier]         "iX"
  - [Imag],             [(]                  "i("
  - [Imag],             [{]                  "i{"
  - [Imag],             [[]                  "i["
  - [Imag],             [Imag]               "ii"

  - [Identifier],       [Number]             "X5"
  - [Identifier],       [Imag]               "Xi"
  - [Identifier],       [Identifier]         "Xπ"
  - [Identifier],       [{]                  "X{"
```

```
- [Identifier],       [[]                  "X["

- [)],                [()]                 ")("
- [)],                [{]                  "){"
- [)],                [[]                  ")["
- [)],                [Number]             ")3"
- [)],                [Imag]               ")i"
- [)],                [Identifier]         ")X"
- [)],                [UnaryMinus]         ")-"

- [}],                [Number]             "}3"
- [}],                [Imag]               "}i"
- [}],                [Identifier]         "}X"
- [}],                [()]                 "}("
- [}],                [{]                  "}{"

- [Identifier],       [()    provided that [Identifier] is not
                             a FunctionIdentifier,
                             a CustomListIdentifier,
                             a StandardListIdentifier,
                             a MatrixIdentifier,
```





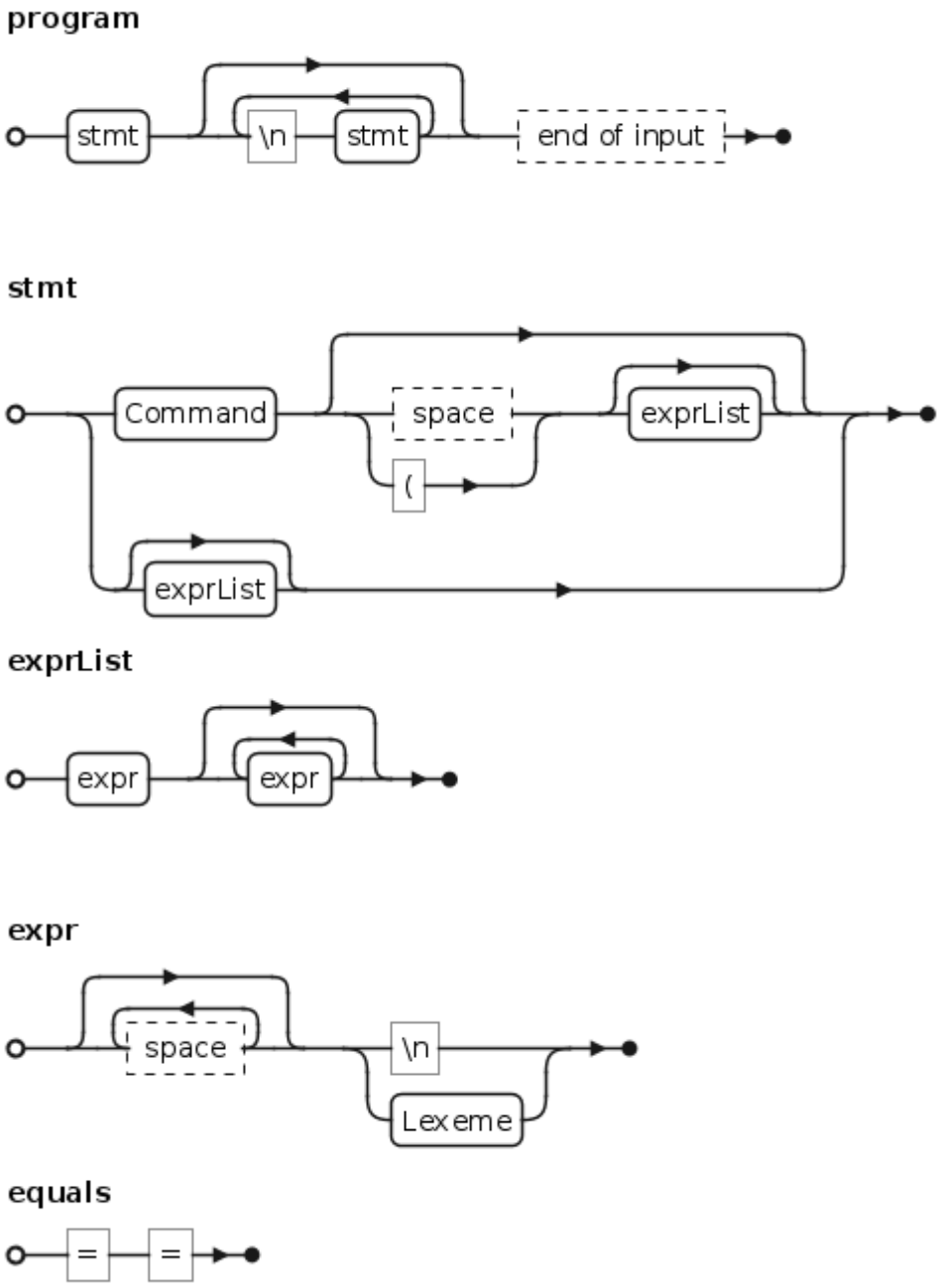
1.4. Lexical structure of a script file (zcb)

Using the language construct `call` (Calling the script file) allows you to interpret the contents of `.zcb` file in the global ZeroBasic context. The lexical structure of the file has some differences from the structure of a single expression:

	Expression	**File**
Command processing	One command at the beginning of an expression	One command at the beginning of a line, (or several commands which ending with `()`)
Spaces	A regular space (\x20) will cause a syntax error (except when processing a command lexeme)	`expr` can contain an unlimited number of spaces
Equality	`=` or `==`	`==`

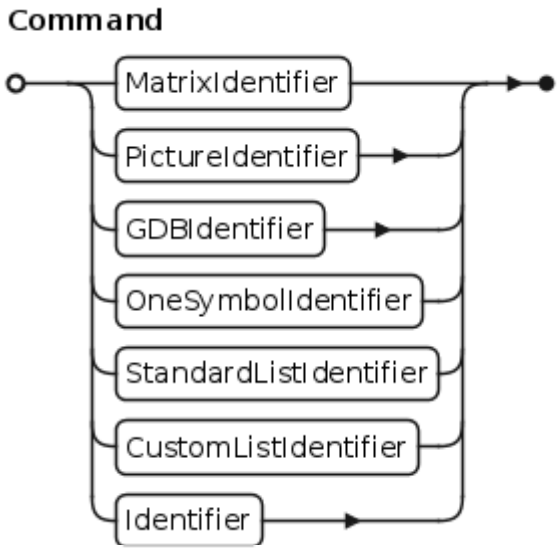
	Expression	File
Insert `*`	Inserting a `*` between two `expr` in some cases	
Insert `)`, `}`, `]`	At the end of each block (the block boundary is `→`), the lexemes of the missing closing brackets are added. The insertion order is the reverse of the order of the opening brackets	

Part of the lexical scheme of a script file is presented below (text description of the structure in EBNF format is available at the end of the document). The other part of the schema is common to both expressions and a script file (the name of common tokens begins with a capital letter) and is discussed further in the section The structure of lexemes common to a file and an expression.

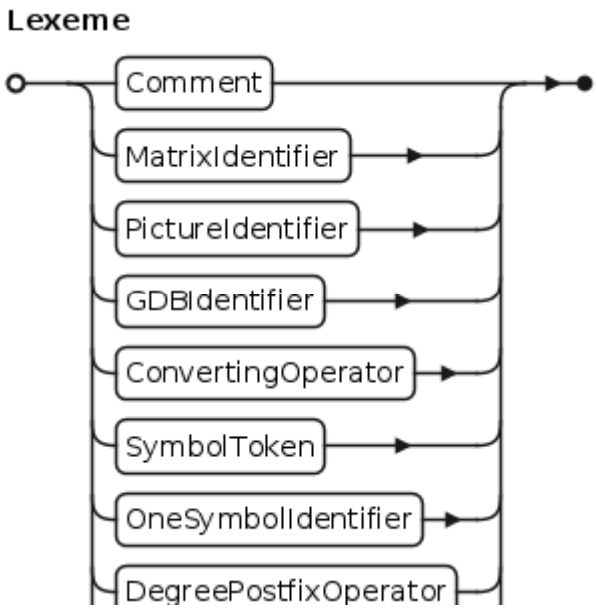


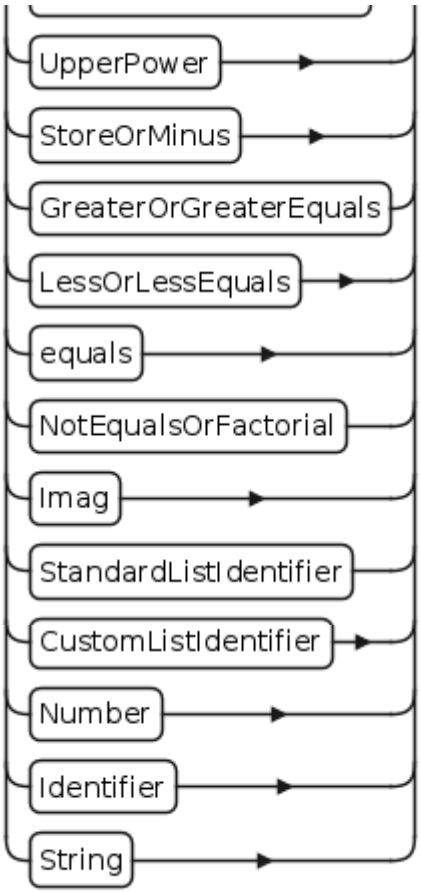
1.5. The structure of lexemes common to a file and an expression

Part of the lexical schema is common to both expressions and a script file (the name of common lexemes begins with a capital letter) is presented below (text description of the structure in EBNF format is available at the end of the document).

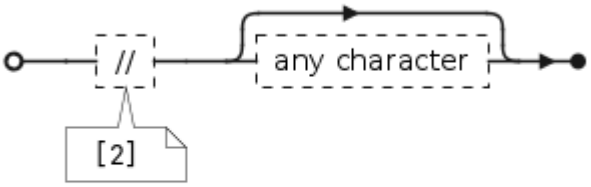



```
The command lexeme is formed only when
the textual content of the identifier
matches the command name (a complete list
is available in the Commands section)
or the [Call] token.
```



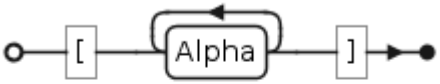


Comment

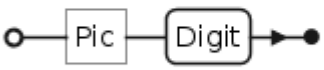



```
[2]: A comment is treated as
"\n" character or end of input
```

MatrixIdentifier

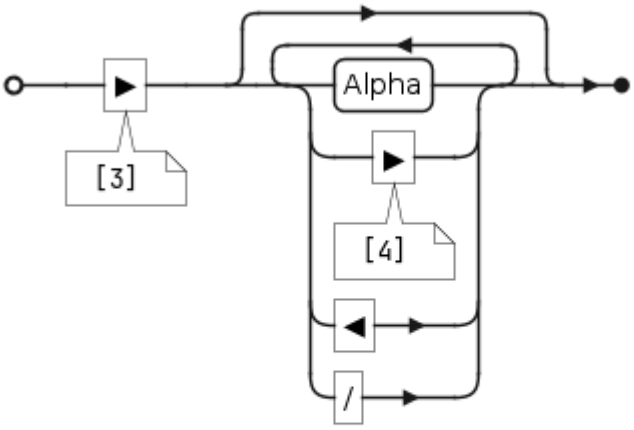


PictureIdentifier



GDBIdentifier

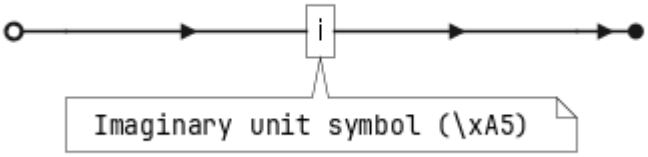
ConvertingOperator



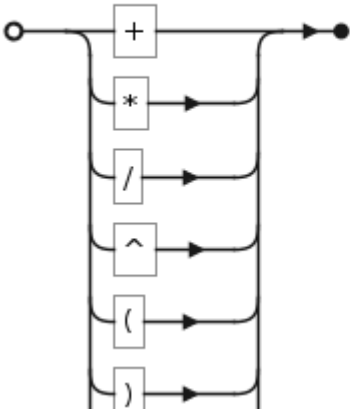
[3]: '▶' Conversion symbol (\xDA)

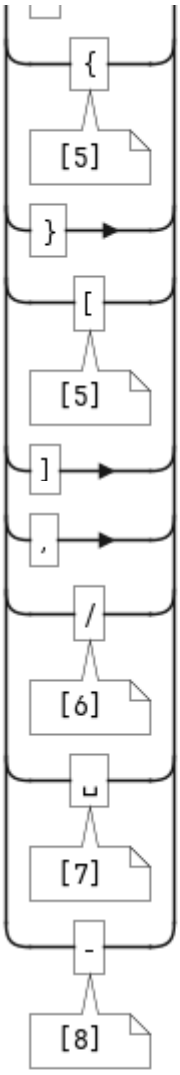
[4]: '▶' Right triangle arrow symbol (\x9E)

Imag



SymbolToken



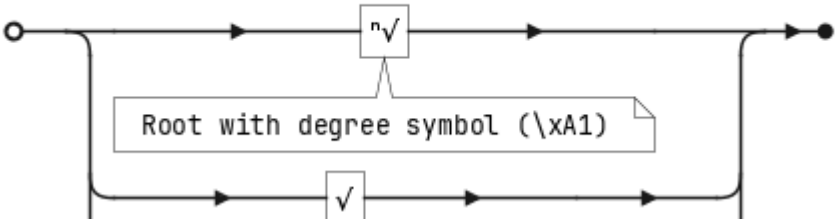


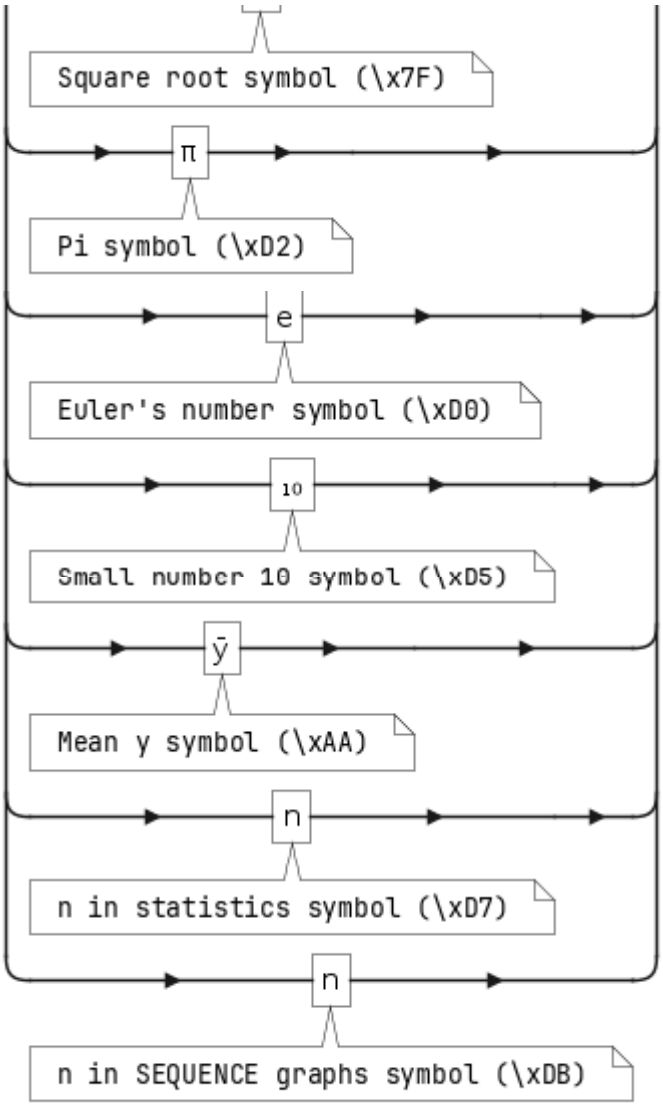
[5]: The next character must not be a closing bracket

[6]: '/' Common fraction symbol (\x9D)

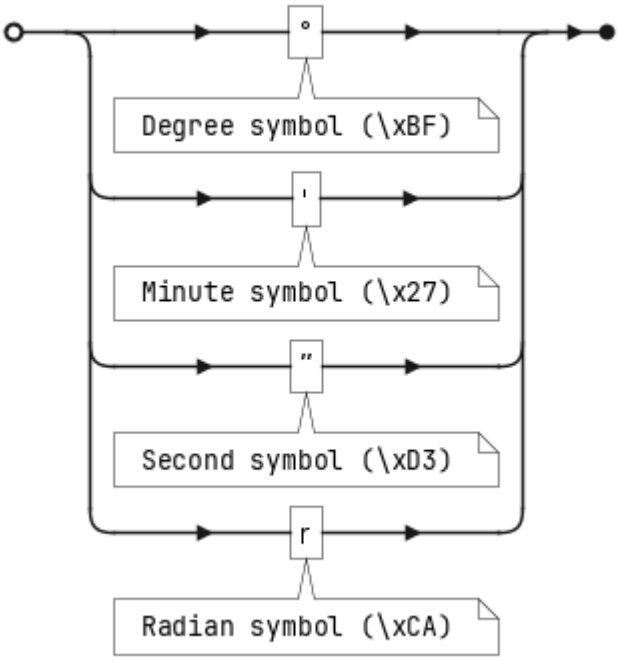
[7]: '␣' Mixed number fraction symbol (\xA0)

OneSymbolIdentifier

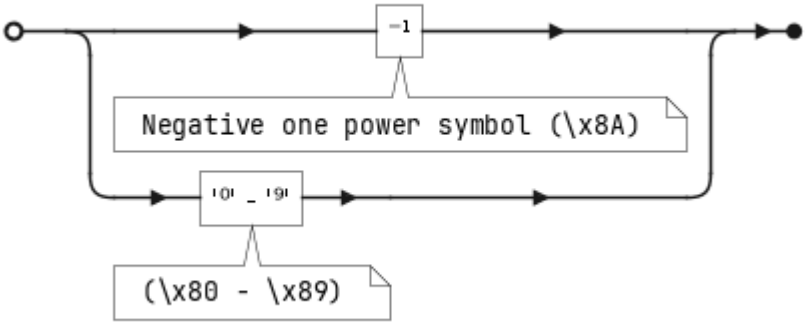



Square root symbol (\x7F)
π
Pi symbol (\xD2)
e
Euler's number symbol (\xD0)
10
Small number 10 symbol (\xD5)
ȳ
Mean y symbol (\xAA)
n
n in statistics symbol (\xD7)
n
n in SEQUENCE graphs symbol (\xDB)

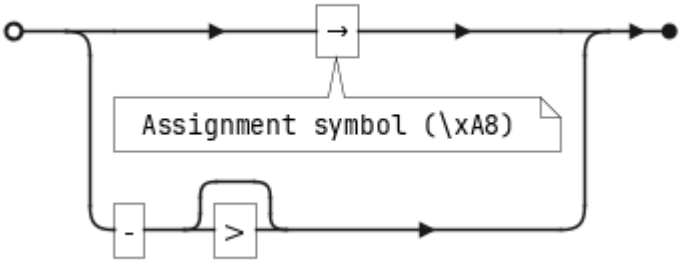
DegreePostfixOperator



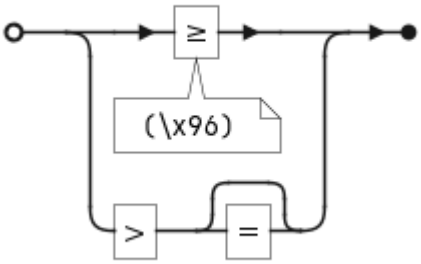
UpperPower



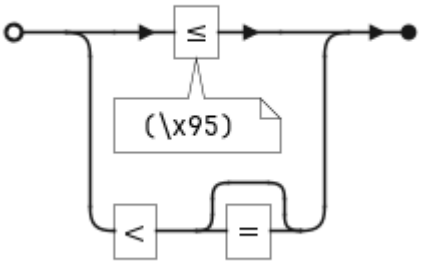
StoreOrMinus



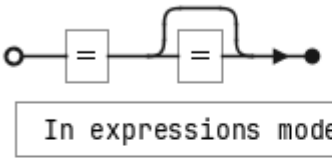
GreaterOrGreaterEquals



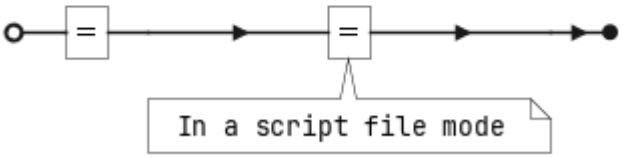
LessOrLessEquals



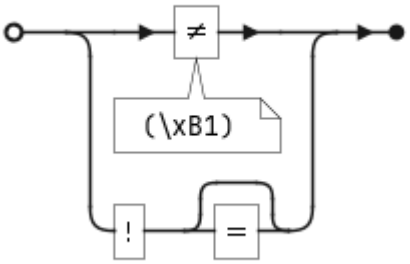
equals



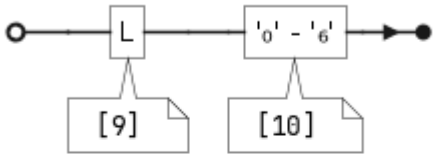
equals



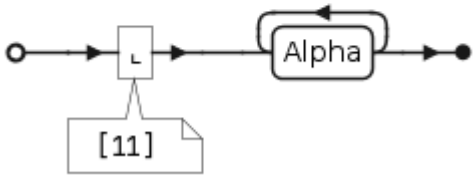
FactorialOrNotEquals



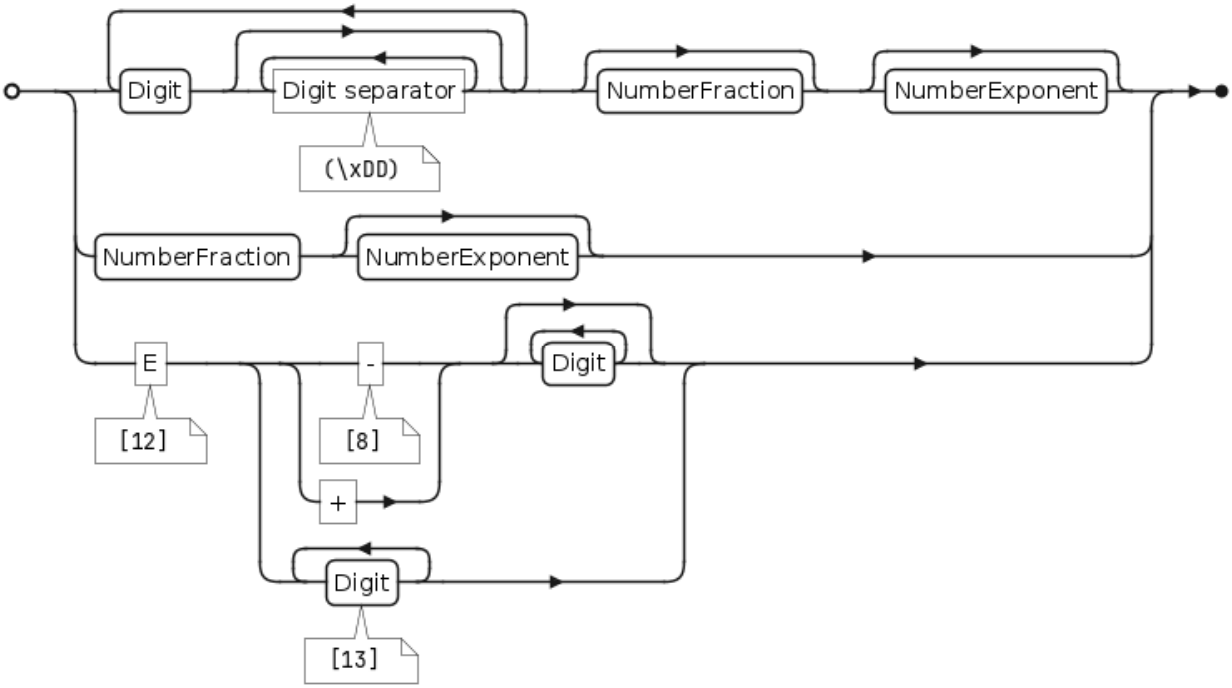
StandardListIdentifier



CustomListIdentifier

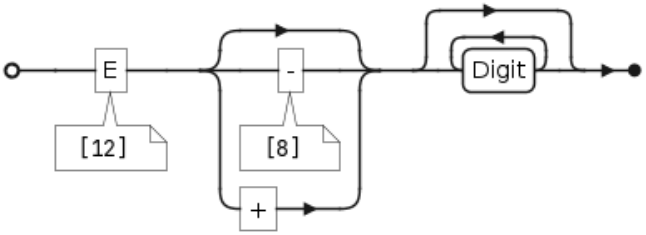


Number

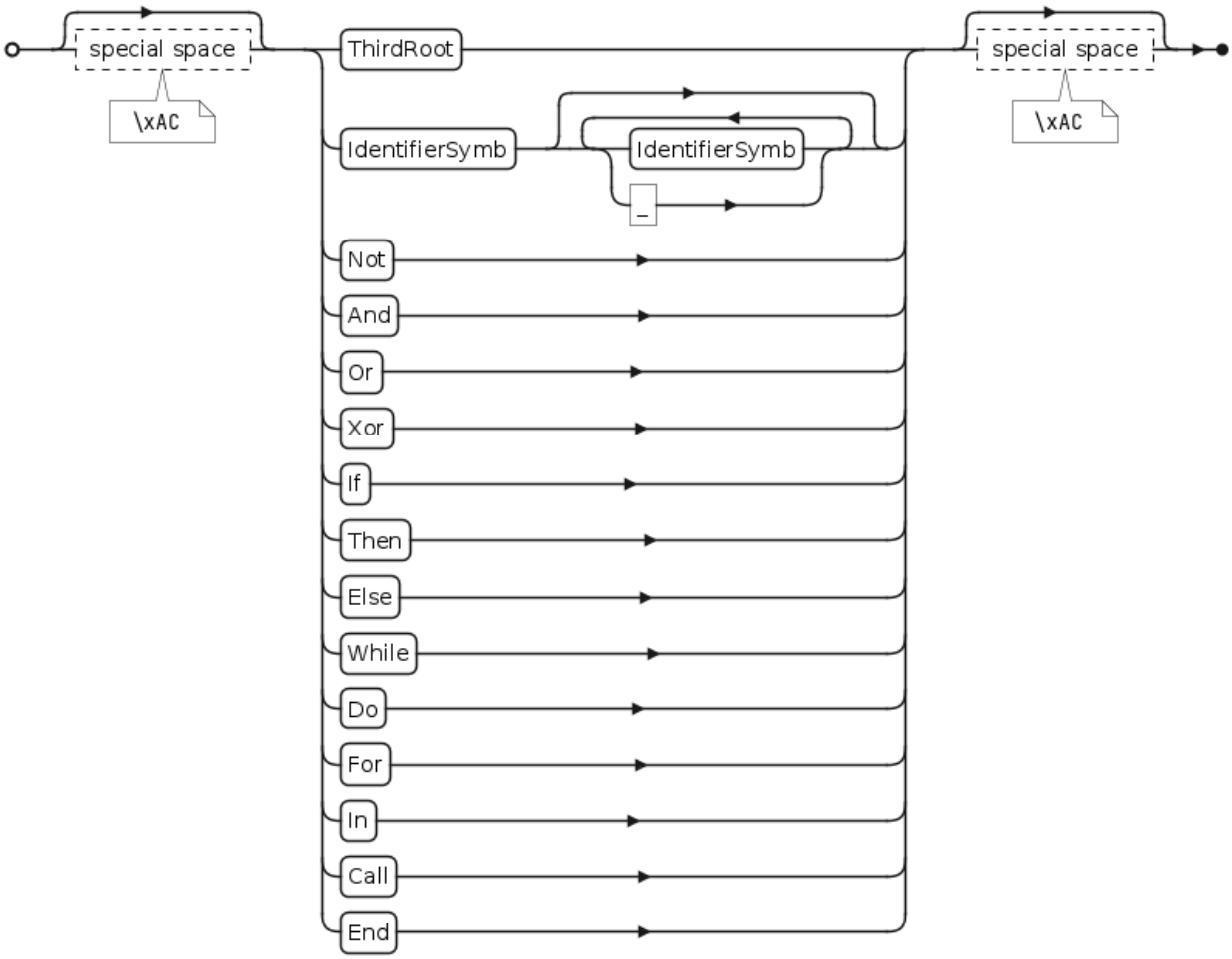


NumberFraction

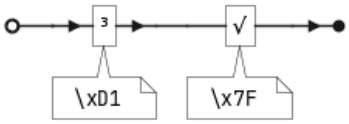
NumberExponent



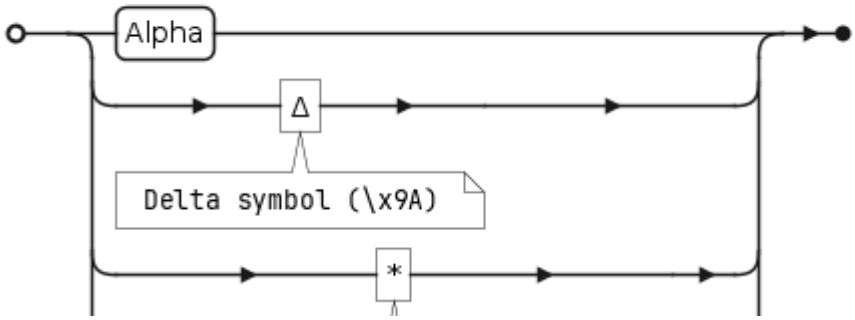
Identifier

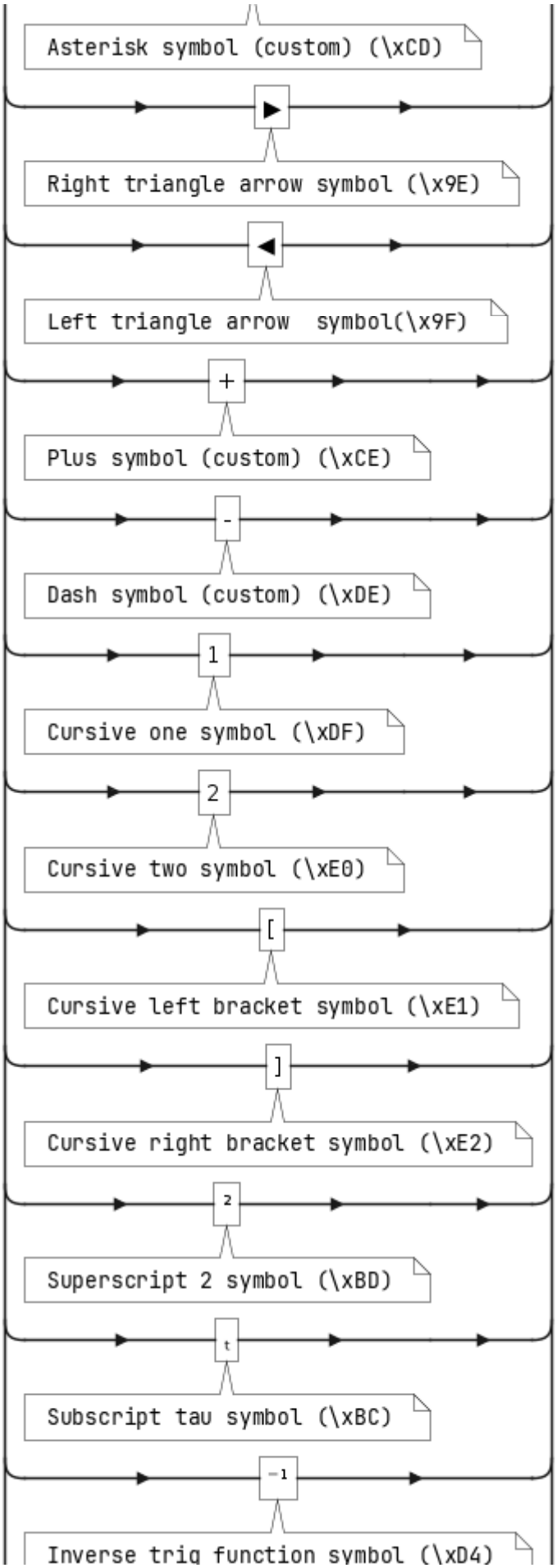


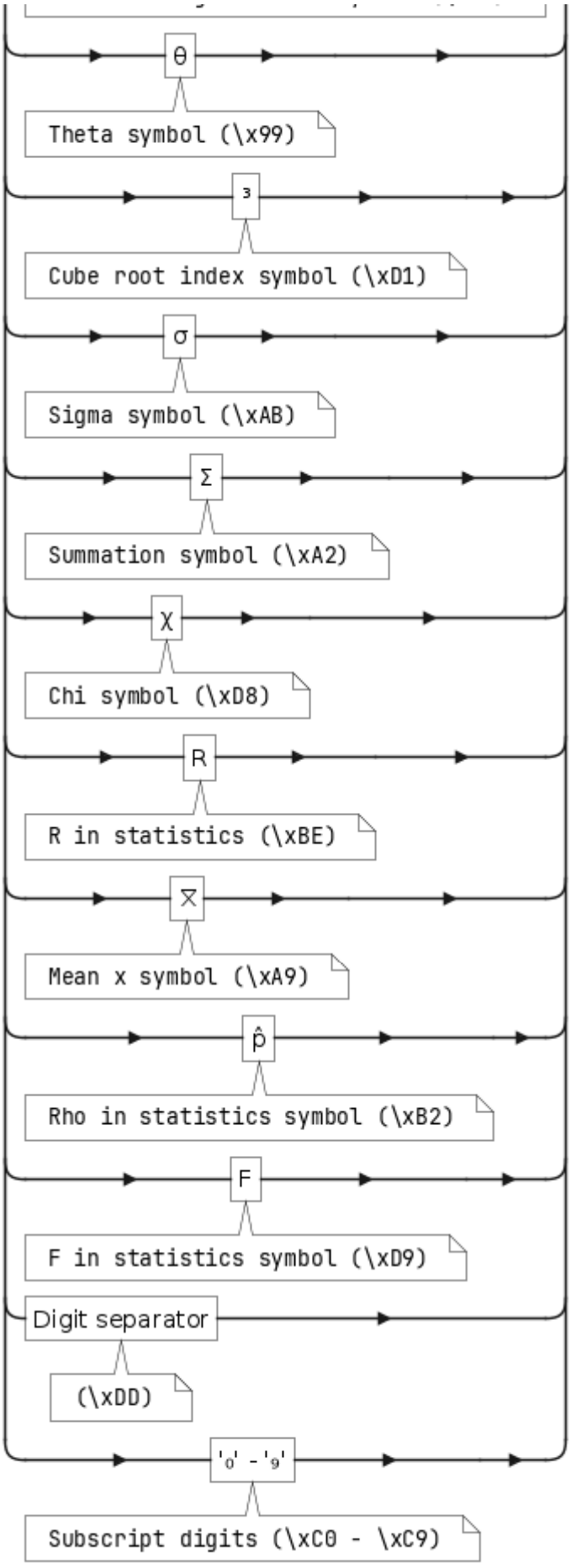
ThirdRoot



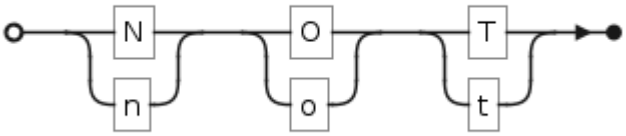
IdentifierSymb



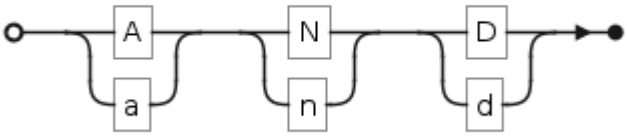

Asterisk symbol (custom) (\xCD)
▶
Right triangle arrow symbol (\x9E)
◀
Left triangle arrow symbol(\x9F)
+
Plus symbol (custom) (\xCE)
-
Dash symbol (custom) (\xDE)
1
Cursive one symbol (\xDF)
2
Cursive two symbol (\xE0)
[
Cursive left bracket symbol (\xE1)
]
Cursive right bracket symbol (\xE2)
2
Superscript 2 symbol (\xBD)
t
Subscript tau symbol (\xBC)
-1
Inverse trig function symbol (\xD4)


θ
Theta symbol (\x99)
³
Cube root index symbol (\xD1)
σ
Sigma symbol (\xAB)
Σ
Summation symbol (\xA2)
χ
Chi symbol (\xD8)
R
R in statistics (\xBE)
x̄
Mean x symbol (\xA9)
p̂
Rho in statistics symbol (\xB2)
F
F in statistics symbol (\xD9)
Digit separator
(\xDD)
'₀' - '₉'
Subscript digits (\xC0 - \xC9)

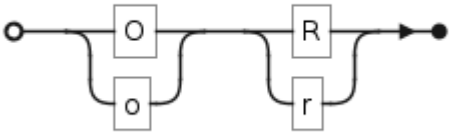
Not



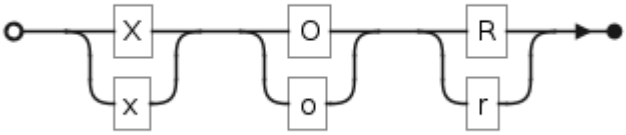
And



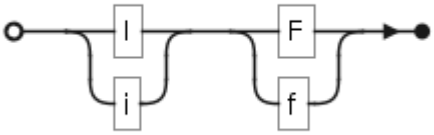
Or



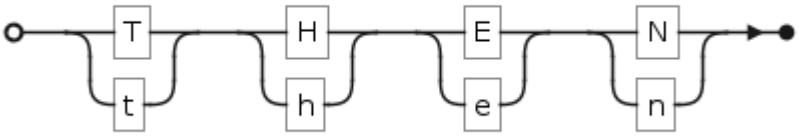
Xor



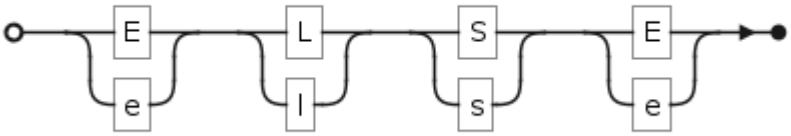
If



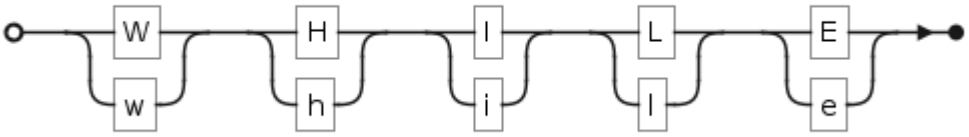
Then



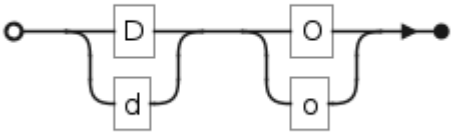
Else



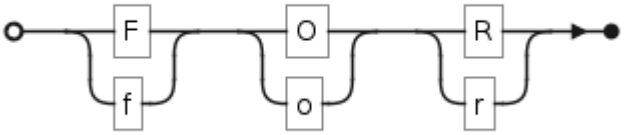
While



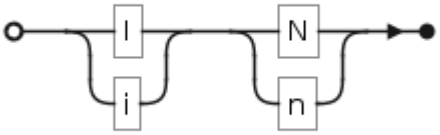
Do



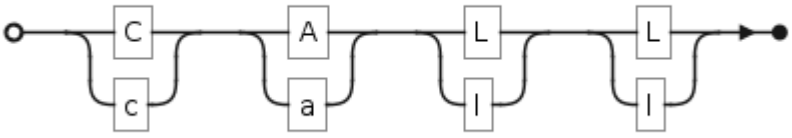
For



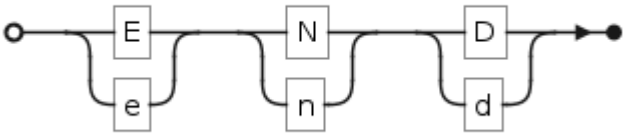
In



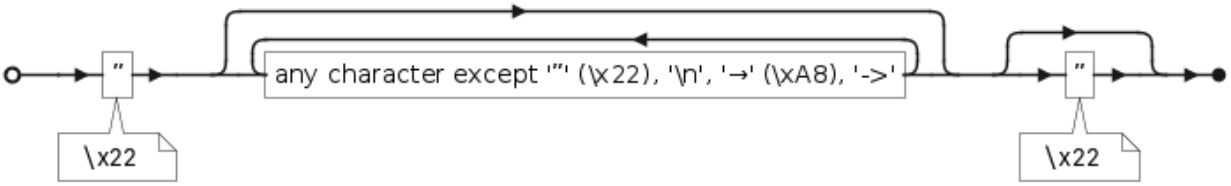
Call



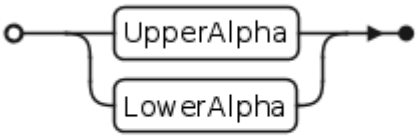
End



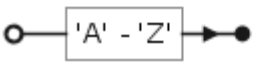
String



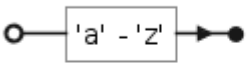
Alpha



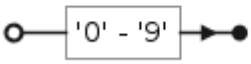
UpperAlpha



LowerAlpha



Digit




```
[7]: '-' Unary minus symbol (\x98)
```

```
[8]: 'L' List symbol (\xA6)
```

```
[9]: '₀' - '₆' (\xC0 - \xC6)
```

```
[10]: '꜀' List symbol (\xA7)
```

```
[11]: 'E' Decimal exponent symbol (\xD6)
```

```
[12]: The third branch of a number also constitutes
the exponent part (NumberExponent), but in the first
```

```
two cases the order may be omitted due to mantissa
presence ('5E'). The third branch requires either
a sign of exponent or explicit exponent value
('E-', 'E6', 'E+7')
```

II. Syntactic structure of the ZeroBasic language

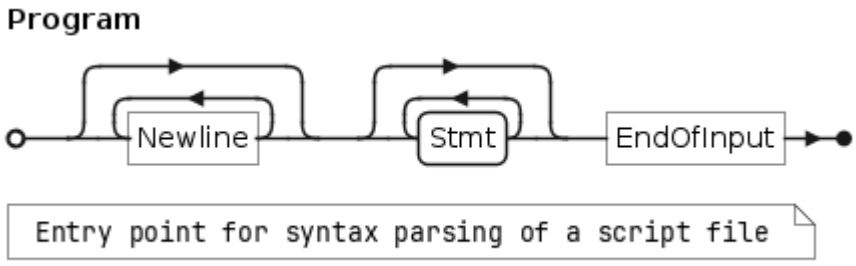
The syntactic structure of the ZeroBasic language defines the rules for composing nodes of the Abstract Syntax Tree (AST) from a set of tokens. The appropriate parsing method for a token set is selected top-down according to the syntactic structure scheme. If the token set does not conform to the structure of the currently analyzed node, parsing proceeds to the next node (if specified that a mismatch results in an error, the token set analysis will return a syntax error).
The table below shows the mapping between lexical scheme nodes and the resulting tokens:

Token	Lexical node	
String	String	
Identifier	OneSymbolIdentifier	
	MatrixIdentifier	
	PictureIdentifier	
	GDBIdentifier	
	StandardListIdentifier	
	CustomListIdentifier	
	Identifier	
	Imag	Imag
Number	Number	
UND	SymbolToken (␣ \xA0)	
ND	SymbolToken (/ \x9D)	
Minute	DegreePostfixOperator (' \x27)	
Degree	DegreePostfixOperator (° \xBF)	
Factorial	FactorialOrNotEquals (!)	
Radian	DegreePostfixOperator (r \xCA)	
Second	DegreePostfixOperator (" \xD3)	
UpperPower	UpperPower	
Power	SymbolToken (^)	
Not	Not	

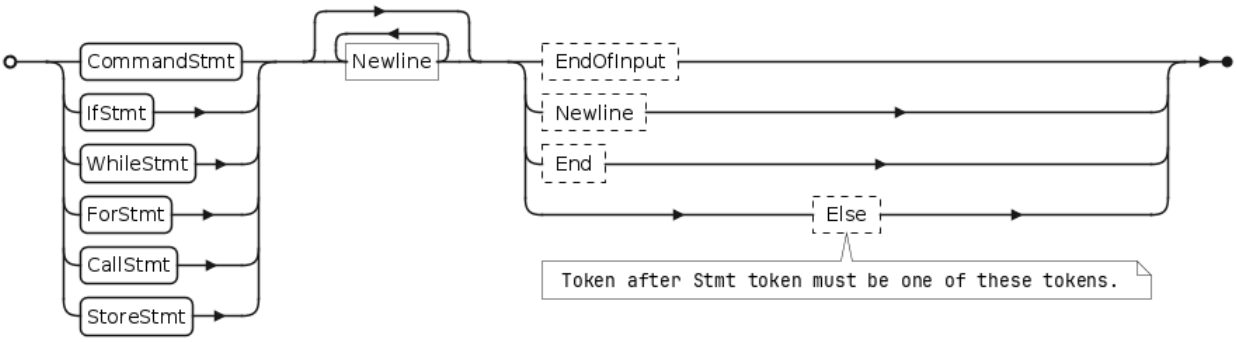
Token	Lexical node
UnaryMinus	SymbolToken (`-` \x98)
Divide	SymbolToken (`/`)
Mult	SymbolToken (`*`)
	Automatic insertion during lexical analysis of an expression
Minus	SymbolToken (`-`)
	StoreOrMinus (`-`)
Plus	SymbolToken (`+`)
LessEquals	LessOrLessEquals
Less	LessOrLessEquals (`<`)
GreaterEquals	GreaterOrGreaterEquals
Greater	GreaterOrGreaterEquals (`>`)
NotEquals	FactorialOrNotEquals
Equals	equals
Xor	Xor
Or	Or
And	And
ConvertingOp	ConvertingOperator
Store	StoreOrMinus
While	While
End	End
If	If
Then	Then
For	For
In	In
Do	Do
Call	Call
Command	Command
Stmt	Stmt (for an expression)
	Stmt (for a script file)

Token	Lexical node
]	SymbolToken (`]`)
	Automatic insertion during lexical analysis of an expression
[	SymbolToken (`[`)
}	SymbolToken (`}`)
	Automatic insertion during lexical analysis of an expression
{	SymbolToken (`{`)
)	SymbolToken (`)`)
	Automatic insertion during lexical analysis of an expression
(	SymbolToken (`(`)
	stmt (`(`) (for an expression and for a script file)
,	SymbolToken (`,`)
Newline	expr (for an expression and for a script file)
	program (`\n`) (for a script file)
EndOfInput	stmt (`end of input`) (for an expression)
	program (`end of input`) (for a script file)

The syntactic scheme of the ZeroBasic language structure is presented below. (text description of the structure in EBNF format is available at the end of the document):

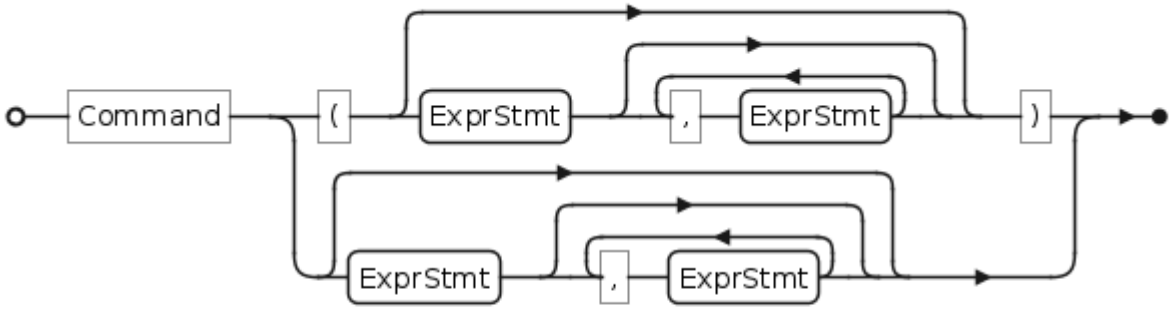


Stmt



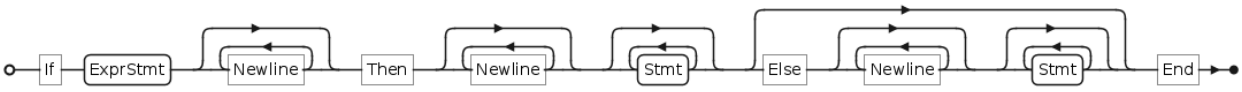
Entry point for syntax parsing of an expression
(The main screen independently splits the entered
command into expressions)

CommandStmt



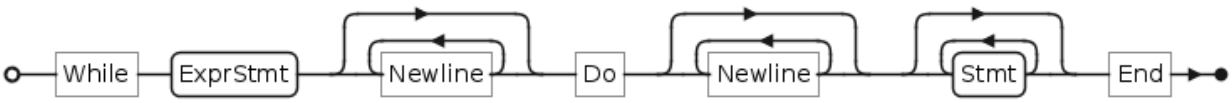
The marker for analyzing CommandStmt is the [Command]
token. Mismatch in the following syntax will lead to
an error.

IfStmt



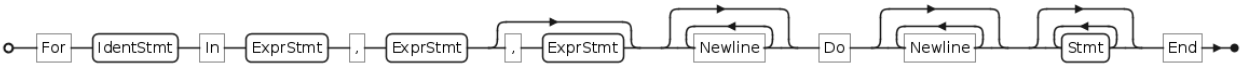
The marker for analyzing IfStmt is the [If] token.
Mismatch in the following syntax will lead to an error.

WhileStmt



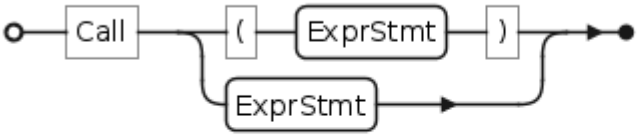
The marker for analyzing WhileStmt is the [While] token.
Mismatch in the following syntax will lead to an error.

ForStmt



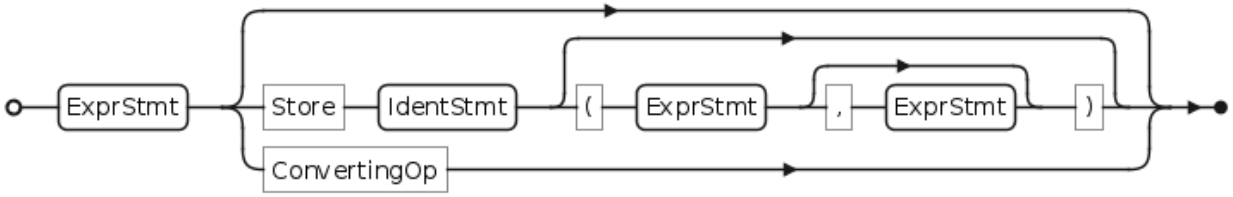

```
The marker for analyzing ForStmt is the [For] token.
Mismatch in the following syntax will lead to an error.
```

CallStmt




```
The marker for analyzing CallStmt is the [Call] token.
Mismatch in the following syntax will lead to an error.
```

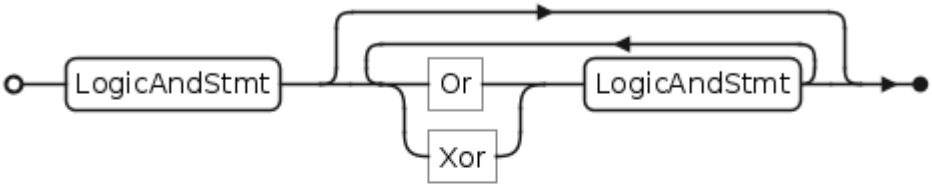
StoreStmt



IdentStmt

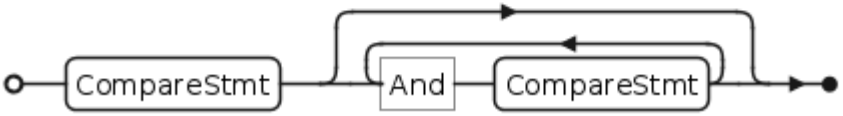
ExprStmt

LogicOrStmt

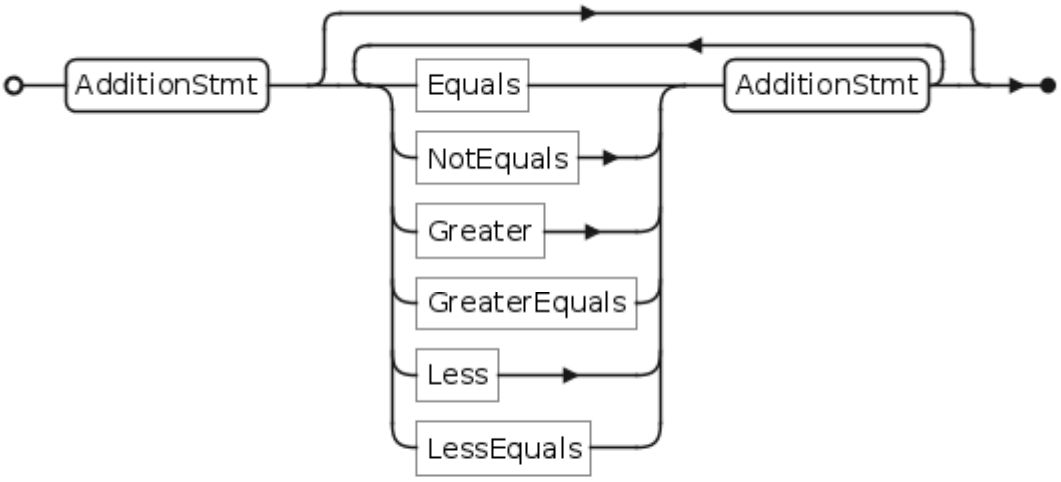



```
Execution order of logical operations is not specified.
Operations are executed sequentially.
```

LogicAndStmt

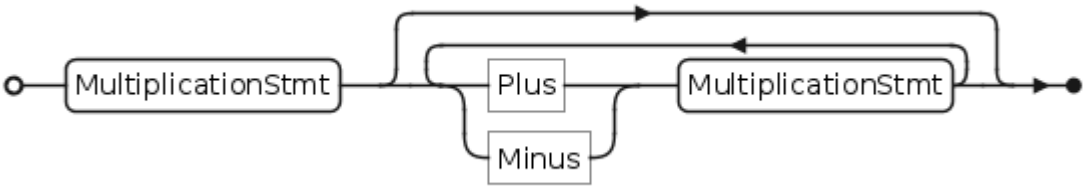


CompareStmt

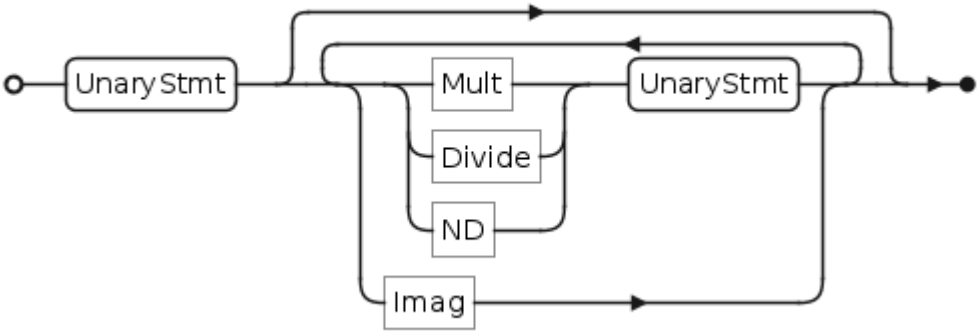



```
Execution order of comparison operations is not specified.
Operations are executed sequentially.
```

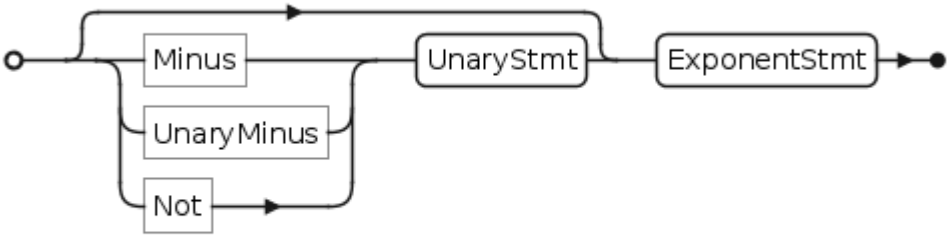
AdditionStmt



MultiplicationStmt

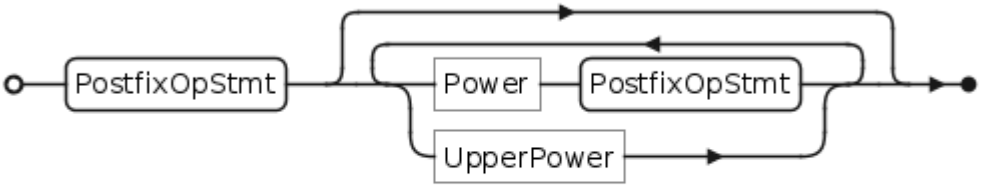


UnaryStmt




```
The [UnaryMinus] token is replaced by the [Minus] token
```

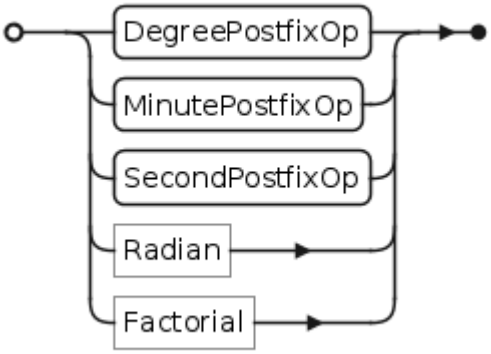
ExponentStmt



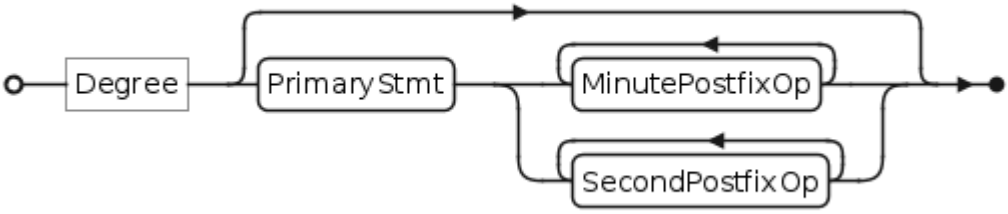

```
The [UpperPower] token contains only a single
superscript digit
```

PostfixOpStmt

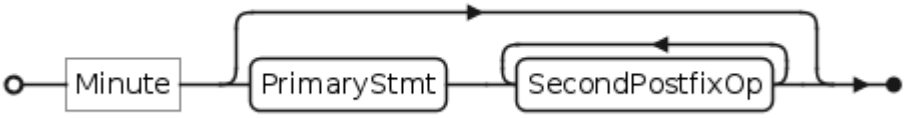
PostfixOp



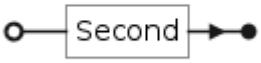
DegreePostfixOp



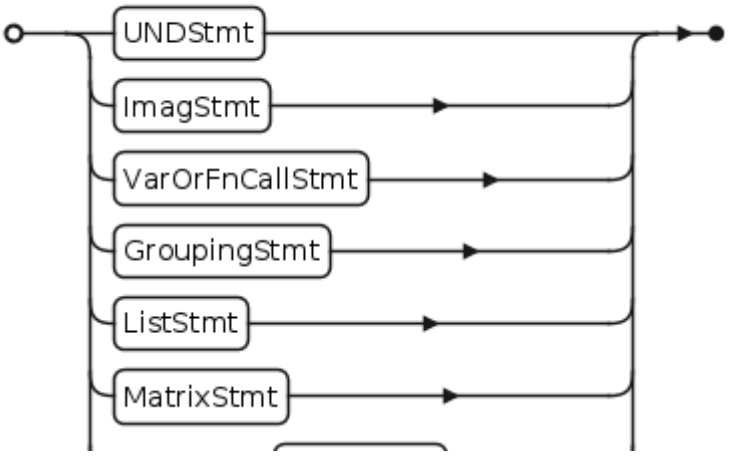
MinutePostfixOp

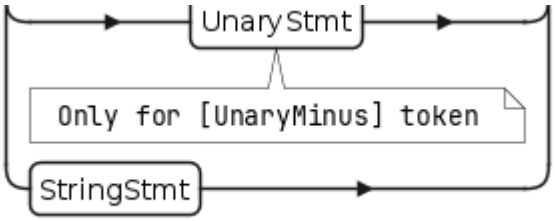


SecondPostfixOp

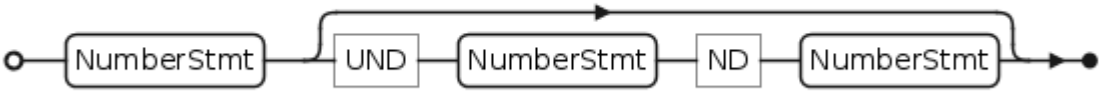


PrimaryStmt

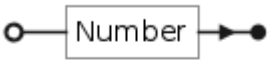




UNDStmt



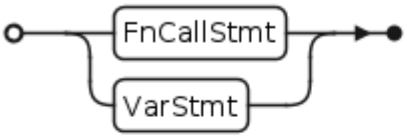
NumberStmt



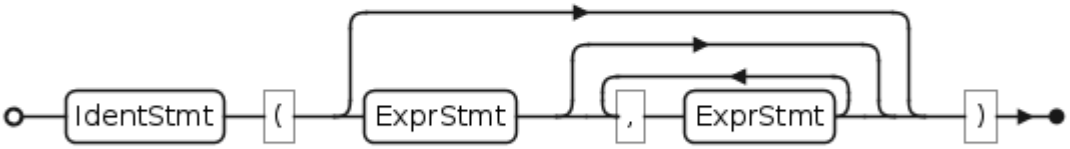
ImagStmt

StringStmt

VarOrFnCallStmt

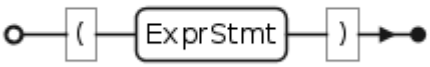


FnCallStmt

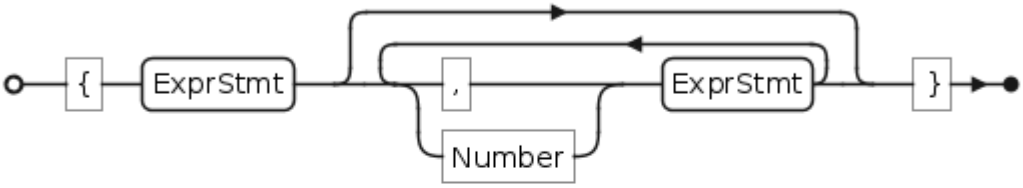


VarStmt

GroupingStmt

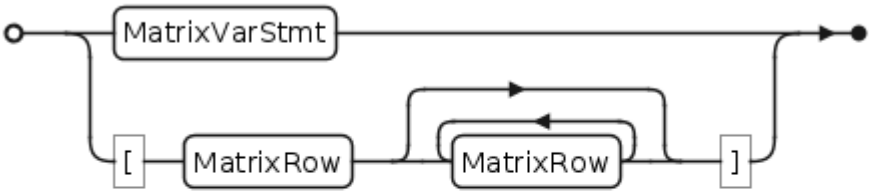


ListStmt

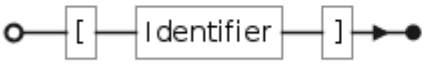



```
The [Number] token is included in the subsequent
ExprStmt node. This analysis structure allows for
the following syntax in the script file:
{1,2,3}
{1 2 3}
```

MatrixStmt

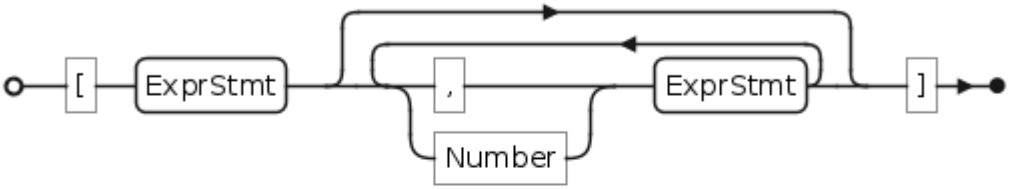


MatrixVarStmt




```
 The [Identifier] token must consist only of letters
```

MatrixRow




```
The [Number] token is included in the subsequent
ExprStmt node. This analysis structure allows for
the following syntax in the script file:
[[1,2,3]]
[[1 2 3]]
```

III. Interpretation of the ZeroBasic language

3.1. Data types

The ZeroBasic language has two fundamental data types:

- Decimal number (*Dec*), whose mantissa can contain up to 40 decimal digits. The decimal exponent of the number is within the range of -32767 to 32767. Corresponds to the `NumberStmt` node. The lexical structure of a number is described in the section The structure of lexemes common to a file and an expression, under the Number block.
- Character string (*Str*). Corresponds to the `StringStmt` node. The lexical structure of a string is described in the section The structure of lexemes common to a file and an expression, under the String block.

The language also includes three derived data types:

- Complex number (*Imag*), whose imaginary and real parts are represented by decimal numbers. Corresponds to the `ImagStmt` node.
- List (array) (*List*) contains an unlimited number (limited only by available RAM) of decimal and/or complex numbers. Zero-length lists are not supported. Corresponds to the `ListStmt` node.
- Matrix (*Matr*) contains an unlimited number of lists (limited only by available RAM), which represent the rows of the matrix and consist of decimal and/or complex numbers. The number of elements in each list must be the same (otherwise, an *INVALID DIMENSION* error will occur). Zero-dimension matrices are not supported. Corresponds to the `MatrixStmt` node.

3.2. Fractions

The evaluation of a fraction is performed with the highest priority (taking into account grouping parentheses). Corresponds to the `UNDStmt` node. The symbol for a fraction with an integer part (`␣`) has the hexadecimal representation \xA0. The symbol for a proper fraction (`⁄`) has the hexadecimal representation \x9D. This structure allows obtaining the result of an expression in the form of a proper fraction. The appearance of this fraction depends on the **Fraction type** and **Answers** modes (changing is accessible via the **MODE** window). The *Dec* data type supports non-negative float numbers as operands. If the `<Denominator>` is equal to 0, it will result in a *DIVIDE BY 0* error.

> `<Integer part:Dec>` `␣` `<Numerator:Dec>` `⁄` `<Denominator:Dec>` `-> Dec`
>
> Equivalent to the expression `<Integer part> + ( <Numerator> / <Denominator> )`.

3.3. Variables

A variable corresponds to the `IdentStmt` node. The lexical structure of a variable is described in the section The structure of lexemes common to a file and an expression in the blocks MatrixIdentifier, PictureIdentifier, GDBIdentifier, OneSymbolIdentifier, StandardListIdentifier, CustomListIdentifier, Identifier. The **maximum length** of a variable name is **16 characters**. Using a name longer than this will result in a *NAME LENGTH* error. Variables are in the super-global scope, meaning any variable created in expression mode (or in other

modes) is accessible in other screens and modes, as well as during the execution of the script file. A variable can be deleted in the **MEMORY** management window, as well as using variable deletion commands (`delVar`, `setUpEditor`). Initially, a basic set of variables is available to the user. This set can be expanded later through assignment. Creating a variable with a name that matches the name of a function or a command will be generated in an error *FUNCTION NAME*. Some screens and applications (including settings screens) provide the user with access to their settings through modification of corresponding variables. It is not possible to change the data type of standard variables via assignment. The new value will either be ignored or a *DATA TYPE* or *DOMAIN* error will be generated. If a variable is not defined, its value will be 0 (except in certain cases, as described later). Variables with the lexical structure MatrixIdentifier always have and must have the data type *Matr*. If such a variable is not defined, an *UNDEFINED* error will be generated. Variables with the lexical structure StandardListIdentifier and CustomListIdentifier always have and must have the data type *List*. If such a variable is not defined, an *UNDEFINED* error will be generated.

3.3.1. Predefined constants

This type of variable remains unchanged throughout the operation of the calculator. The values of these variables are automatically initialized. An attempt to modify or delete it will result in an error *READ ONLY VAR*.

Variable	Type	Value
`pi`	Dec	3.14159265358979323846264338327950288 4
`π` (\xD2)	Dec	3.14159265358979323846264338327950288 4
`e` (\xD0)	Dec	2.71828182845904523536028747135266249 8
`10` (\xD5)	Dec	10
`LEFT`	Dec	1
`CENTER`	Dec	2
`RIGHT`	Dec	3
`BLUE`	Dec	10
`RED`	Dec	11
`BLACK`	Dec	12
`MAGENTA`	Dec	13
`GREEN`	Dec	14
`ORANGE`	Dec	15
`BROWN`	Dec	16
`NAVY`	Dec	17
`LTBLUE`	Dec	18
`YELLOW`	Dec	19
`WHITE`	Dec	20
`LTGRAY`	Dec	21

Variable	Type	Value
`MEDGRAY`	Dec	22
`GRAY`	Dec	23
`DARKGRAY`	Dec	24
`DARK`	Dec	25
`CYAN`	Dec	26

3.3.2. Readonly variables

This type of variable cannot be modified by the user. Attempting to modify or delete it will result in a *READ ONLY VAR* error. All of the variables listed below have the type *Dec*. These readonly variables are set by statistical commands. Input these variables is possible through the **VARS** > **Statistics…** menu.

Variable	Value
Tab **XY**	
`n` (\xD7)	number of data points
`x̄` (\xA9)	mean of x values
`Sx`	sample standard deviation of x
`σx` (\xABx)	population standard deviation of x
`ȳ` (\xAA)	mean of y values
`Sy`	sample standard deviation of y
`σy` (\xABy)	population standard deviation of y
`minX`	minimum of x values
`maxX`	maximum of x values
`minY`	minimum of y values
`maxY`	maximum of y values
Tab **Σ**	
`Σx` (\xA2x)	sum of x values
`Σx²` (\xA2x\xBD)	sum of x² values
`Σy` (\xA2y)	sum of y values
`Σy²` (\xA2y\xBD)	sum of y² values
`Σxy` (\xA2xy)	sum of x*y
Tab **EQ**	
`RegEQ`	regression equation (**not read only**)
`a`, `b`	regression/fit coefficients
`a` - `e`	polynomial, `Logistic`, and `SinReg` coefficients

Variable	Value
`r`	correlation coefficient
`r²` (r\xBD), `R²` (\xBE\xBD)	coefficient of determination
Tab **TEST**	
`p`	p-value
`z`	test statistics
`t`	
`χ²` (\xD8\xBD)	
`F` (\xD9)	
`df`	degrees of freedom
`df₂` (df\xC2)	
`p^` (\xB2)	estimated sample proportion
`p^₁` (\xB2\xC1)	estimated sample proportion for population 1
`p^₂` (\xB2\xC2)	estimated sample proportion for population 2
`s`	standard error about the line
`x̄₁` (\xA9\xC1)	sample mean of x values for sample 1 and sample 2
`x̄₂` (\xA9\xC2)	
`Sx₁` (Sx\xC1)	sample standard deviation of x for sample 1 and sample 2
`Sx₂` (Sx\xC2)	
`Sxp`	pooled standard deviation
`n₁` (\xD7\xC1)	number of data points for sample 1 and sample 2
`n₂` (\xD7\xC2)	
`lower`	confidence interval pair
`upper`	
`F₁` (F\xC1)	used in `2-SampFTest`
`F₂` (F\xC2)	
Tab **PTS**	
`x₁` (x\xC1)	summary points (`Med-Med` only)
`y₁` (y\xC1)	
`x₂` (x\xC2)	
`y₂` (y\xC2)	
`x₃` (x\xC3)	
`y₃` (y\xC3)	
`Q₁` (Q\xC1)	1st quartile (the median of points between `minX` and `Med`)

Variable	Value
`Med`	median
`Q3` (Q\xC3)	3rd quartile (the median of points between `Med` and `maxX`)

3.3.3. Y-functions variables

This type of variable is used in Y-function calculations. The variable responsible for the function value cannot be modified. Its value is computed based on the assigned expression and the argument value (the argument value is taken from the global context and is 0 by default). If the expression of a Y-function contains a direct (`Y0=Y0+1`) or indirect (`Y0=Y1`; `Y1=Y0`) self-reference, an error *Yn RECURSION* will be generated. If the expression of the function is not defined, an *INVALID* error will be generated. Using argument variables for storing user-defined values is not recommended, since these variables may change due to being involved in function value computations in other windows or applications.
Below is a table of function variables (interactions marked with * are not possible as described):

Number	Function	Argument
Functions, mode: Func		
0	`Y0` (Y\xC0)	`X`
...		
9	`Y9` (Y\xC9)	
Parametric function, mode: Par		
0	`X0t` (X\xC0\xBC)	`T`
1	`Y0t` (Y\xC0\xBC)	
...		
8	`X4t` (X\xC4\xBC)	
9	`Y4t` (Y\xC4\xBC)	
Polar functions, mode: Pol		
0	`r0` (r\xC0)	`θ` (\x99)
...		
9	`r9` (r\xC9)	
Sequences, mode: Seq		
0	`u`	`n` (\xDB)
1*	`u(nMin)`	`nMin` (\xDBMin)
2*	`u(nMin+1)`	`nMin+1` (\xDBMin+1)
3	`v`	`n` (\xDB)
4*	`v(nMin)`	`nMin` (\xDBMin)
5*	`v(nMin+1)`	`nMin+1` (\xDBMin+1)
6	`w`	`n` (\xDB)

Number	Function	Argument
7*	`w(nMin)`	`nMin` (\xDBMin)
8*	`w(nMin+1)`	`nMin+1` (\xDBMin+1)

Deleting argument variables is the same as deleting other variables. Deleting a function variable will clear the corresponding expression.

3.3.4. Strings

The calculator includes predefined zero-length string variables: `Str0` to `Str9` (Str\xC0 - Str\xC9). The type of predefined string variables cannot be changed. Deleting these variables will clear their contents.

3.3.5. Lists

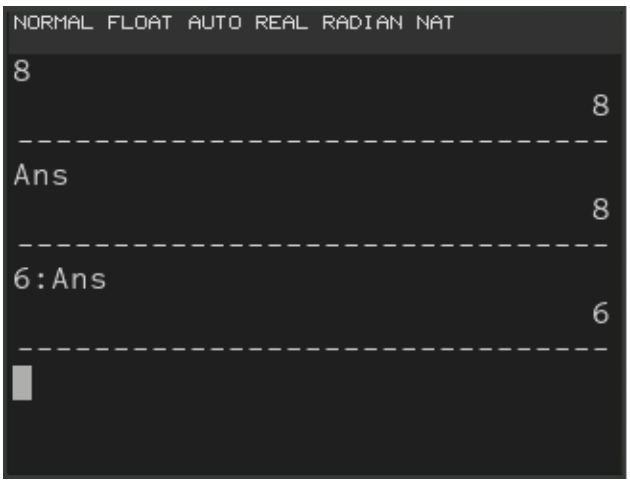
The calculator includes predefined zero-length list variables: `L0` to `L6` (\xA6\xC0 - \xA6\xC6). Using zero-length lists will result in an *INVALID DIMENSION* error. User-defined lists follow the CustomListIdentifier lexical structure and have a maximum name length of 8 characters (including special L characters). Accessing an undefined variable with the StandardListIdentifier or CustomListIdentifier lexical structure will result in an *UNDEFINED* error.

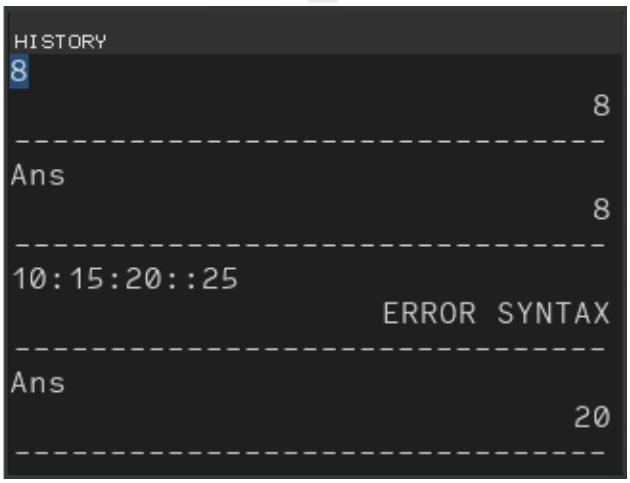
3.3.6. Matrices

The calculator has predefined zero-dimensional matrix variables: `[A]` - `[J]`. Using zero-dimensional matrices will result in an *INVALID DIMENSION* error. User-defined matrices must conform to the lexical structure of MatrixIdentifier, with a maximum name length of 8 characters (including square brackets). Using overly long names will trigger a *NAME LENGTH* error. Accessing an undefined variable matching the lexical structure of MatrixIdentifier will result in an *UNDEFINED* error.

3.3.7. The Ans variable

The `Ans` variable contains a copy of the result of the previous valid expression (excluding *Done*) entered on the home screen. Note that expressions on the home screen can be separated by the `:` character.





Manual assignment of a variable `Ans` (including attempts to modify list or matrix elements) will result in an *READ ONLY VAR* error. Assigning a list value to the variable `Ans`, as well as using the construct `<Size:Dec>→dim(Ans)`, will lead to the creation or modification of the variable `ⱠAns`. Assigning a matrix value to the variable `Ans`, as well as using the construct `<Height, Width:List>→dim(Ans)`, will lead to the creation or modification of the variable `[Ans]`.

3.4. Assignment

Assigning a new value to constants, the Ans variable, readonly variables will result in a *READ ONLY VAR* error, except in cases of assigning a list or a matrix. Creating a variable with a name that matches the name of a function or a command will be generated in an error *FUNCTION NAME*. Assigning a data type other than the one expected to predefined variables (strings, lists, matrices) will result in a *DATA TYPE* or *DOMAIN* error. Assigning incompatible data types to variables with lexical structures StandardListIdentifier, CustomListIdentifier and MatrixIdentifier (`[[1]]→ⱠB`, `{1}→[C]`) will result in a *DATA TYPE* error.

Assignment corresponds to the `StoreStmt` node. It allows placing the result of an expression into a variable.

> `<Value:Dec|Imag>` `→` `var`
>
> This construct allows assigning a numerical value to a variable, screen parameter, or application parameter. The length of the variable name is limited to 16 characters.

> `<Value:Str>` `→` `var`
>
> This construct allows assigning a string value to a variable, screen parameter, or application parameter. The length of the variable name is limited to 16 characters. If the variable name matches a function variable, the string contents will replace the corresponding function expression.

> `<Value:List>` `→` `L₀`
>
> This construct allows assigning a list value to the variable `L₀`.
> If the list variable does not conform to the syntax of StandardListIdentifier or CustomListIdentifier, an attempt will be made to convert it to a correct syntax (`A`: `ⱠA`, `1`: error, `[A]`: error, `aBCdeFG`: `ⱠaBCdeFG`, `aBCdeFGh`: name length error). The exception is the `TblInput` variable.

> `<Value:Dec|Imag>` `→` `L₀` `(` `<Index:Dec>` `)`
>
> This construct allows assigning a value to a list element. The list variable (`L₀`) must conform to the lexical structure StandardListIdentifier or CustomListIdentifier. The exception is the `TblInput` variable. The value `<Index>` must be an integer greater than 0 but not exceeding the size of the list. If `<Index>` is 1 greater than the size of the list, `<Value>` will be appended to the end of the list.

`<Size:Dec>` → `dim(` `⌞ist` `)`

This construct allows changing the size of the list `⌞ist`. If the list `⌞ist` does not exist, this construct will create a new list with size `<Size>`. The value `<Size>` must be an integer between 0 and 999. New list elements are initialized to 0.
If the list variable does not conform to the syntax of StandardListIdentifier or CustomListIdentifier, an attempt will be made to convert it to a correct syntax (`A`: `⌞A`, `1`: error, `[A]`: error, `aBCdeFG`: `⌞aBCdeFG`, `aBCdeFGh`: name length error).

The exception is the `TblInput` variable. Using the construction `<Size:Dec>→dim(TblInput)` will lead to the modification elements of the list `TblInput`. The list will be filled with values starting from the value `TblStart` with the step value `ΔTbl` (\x9ATbl).

`<Value:Matr>` → `[A]`

This construct allows assigning a matrix value to the variable `[A]`.
If the matrix variable does not conform to the syntax of MatrixIdentifier, an attempt will be made to convert it to correct syntax (`A`: `[A]`, `[A`: error, `A]`: error, `1`: error, `⌞A`: error, `aBCdeF`: `[aBCdeF]`, `aBCdeFGh`: name length error).

`<Value:Dec|Imag>` → `[A]` `(` `<Row:Dec>`, `<Column:Dec>` `)`

This construct allows assigning a value to a matrix element. The matrix variable (`[A]`) must conform to the lexical structure MatrixIdentifier. The value `<Row>` must be an integer greater than 0 but not exceeding the matrix height. The value `<Column>` must be an integer greater than 0 but not exceeding the matrix width.

`<Height, Width:List>` → `dim(` `[A]` `)`

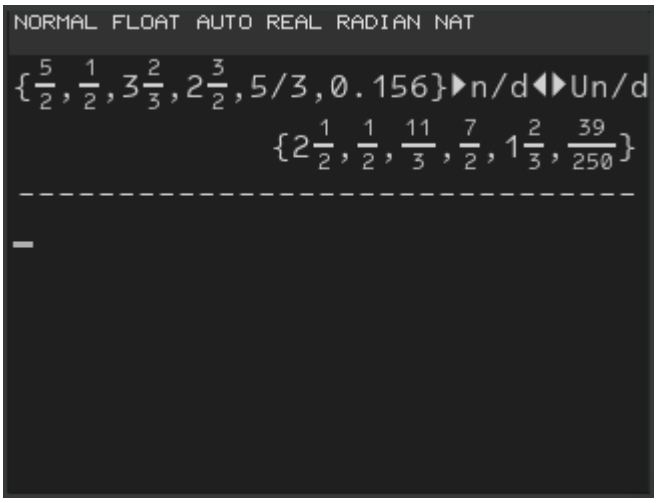
This construct allows changing the size of the matrix `[A]`. The list `<Height, Width>` must consist of two integer elements, which are within the range from 0 to 99. The first element is the new height of the matrix, and the second element is the new width of the matrix. Using only one zero dimension will result in the *INVALID DIMENSION* error. If the matrix `[A]` does not exist, this construct will create a new matrix with the sizes specified in `<Height, Width>`. New matrix elements are initialized to 0.
If the matrix variable does not conform to the syntax of MatrixIdentifier, an attempt will be made to convert it to correct syntax (`A`: `[A]`, `[A`: error, `A]`: error, `1`: error, `⌞A`: error, `aBCdeF`: `[aBCdeF]`, `aBCdeFGh`: name length error).

3.5. Conversion

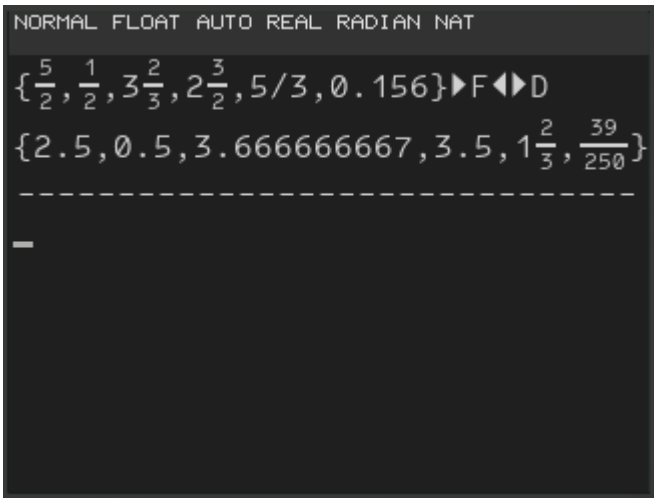
Value conversion is part of the `StoreStmt` node and corresponds to the ConvertingOp token. All conversion commands start with the conversion character ▶, whose hexadecimal representation is \xDA. The command may also include the characters ▶ and ◀, with hexadecimal representations \x9E and \x9F, respectively.
The following operations are available:

> `<Value:Dec|Imag|List|Matr>` `▶Frac`
>
> Enables representation of numeric values as common fractions. The format of the fraction (mixed or improper) depends on the current **Fraction type** mode.

> `<Value:Dec|Imag|List|Matr>` `▶Dec`
>
> Enables representation of numeric values as decimal fractions with a dot.

> `<Value:Dec|Imag|List|Matr>` `▶n/d◀▶Un/d`
>
> Converts numbers represented as mixed fractions to improper fractions, improper fractions to mixed fractions, and all other numbers to fractions.
>


> `<Value:Dec|Imag|List|Matr>` `▶F◀▶D`
>
> Converts numbers represented as common fractions to decimal fractions with a point, and numbers represented as decimal fractions to common fractions.



`<Value:Dec|Imag|List|Matr>` `▶Polar`

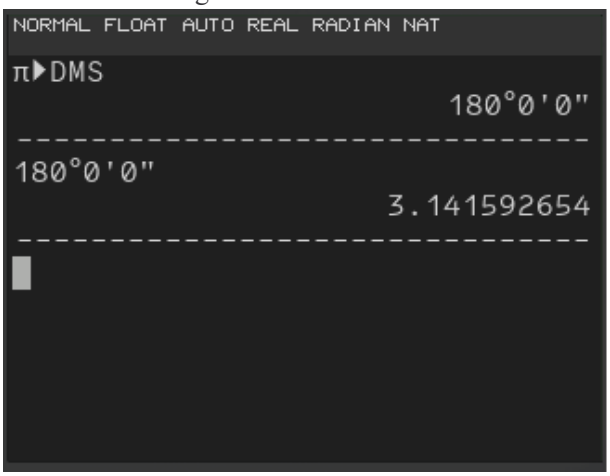
Converts a complex number in algebraic form to exponential form `z=re^(iφ)`. The angle φ depends on the current angle unit mode (**Radian** or **Degree**).

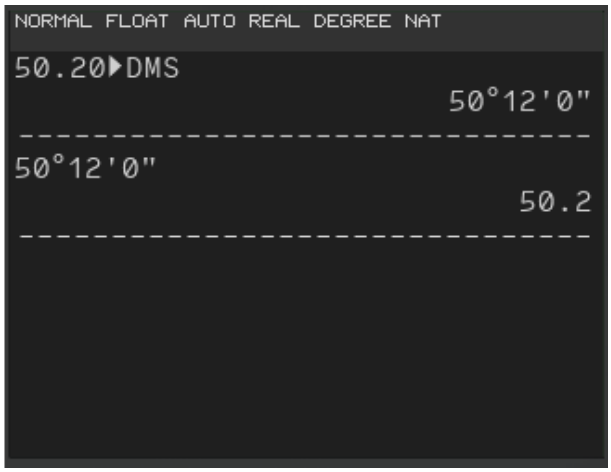
`<Value:Dec|Imag|List|Matr>` `▶Rect`

Converts a complex number in exponential form to algebraic form `z=a+bi`. Conversion is not possible in **Degree** mode.

`<Value:Dec>` `▶DMS`

Converts a value to its string representation in [degree measure]. Takes the current angle mode into account. If the mode is **Radian**, the `<Value>` is first converted to degrees (`<Value> * 180 / π`) and then formatted as a degree measure.





3.6. Commands

Corresponds to the `CommandStmt` node. A command is a special type of function with a different syntax (it may use a space instead of parentheses) and can be used only once in an expression. Commands are case-sensitive. The result of executing a command is the *Done*. Commands generate the following errors (unless otherwise specified):

- *SYNTAX* if any `<Ident>` parameter does not match the `IdentStmt` node.
- *DATA TYPE*
  - if a parameter type is incorrect.
  - if a variable name does not match the variable type. See the Variables section for details on variable types and naming.
- *ARGUMENT* if the number of arguments is less or more than expected.
- *NAME LENGTH* if the length of a variable name exceeds the defined range. See the Variables section for details on ranges.
- *eval break* if the `On` button is pressed.

3.6.1. Lists-related commands

clrAllLists

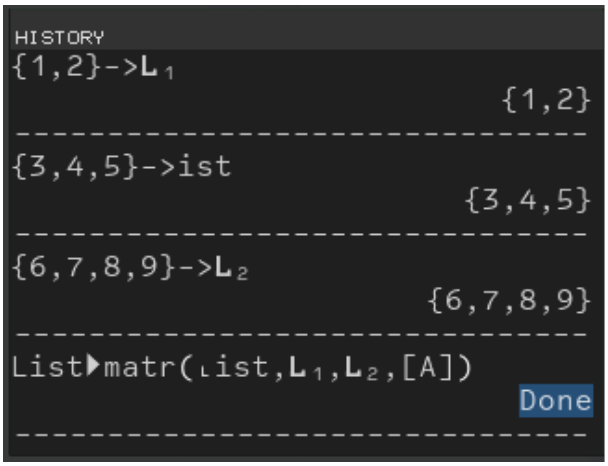
> `clrAllLists`
>
> Sets all lists to zero length.

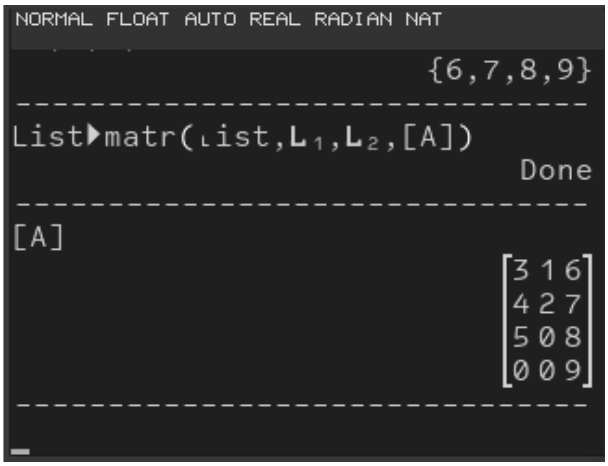
ClrList

> `ClrList` `<Identₙ:List>`, `...`
>
> Sets all provided lists to zero length.
>
> ERRORS:
>
> - *UNDEFINED* if any of the variables are not found.

List▶matr

> (List\x9Ematr) `List▶matr` `<Ident Listₙ:List>`, `...`, `<Ident Matrix:Matr>`
>
> Constructs the matrix `<Ident Matrix>` from the values of the lists `<Ident Listₙ>`. The list values are arranged vertically — the first list occupies the leftmost column of the matrix, the last list occupies the rightmost column. The first element of each list is placed at the top of the matrix, the last element at the bottom. The resulting matrix has a size of `m x n`, where `m` is the maximum length among the lists, and `n` is the number of lists provided. If a list has fewer than `m` elements, missing values are replaced with 0.





ERRORS:

- *UNDEFINED* if any of the `<Ident Listn>` variables are not found.
- *INVALID DIMENSION* if any of `<Ident Listn>` lists is zero-length.

Matr▶list

(Matr\x9Elist) `Matr▶list` `<Matrix:Matr>`, `<Column:Dec>`, `<Ident List:List>`

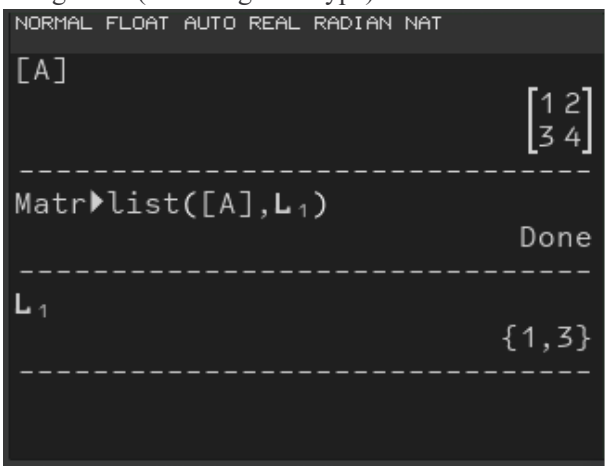
Extracts the values of column `<Column>` from the matrix `<Matrix>` and places them into the list `<Ident List>`. The value of `<Column>` must be an integer within the range from 1 to the number of columns in the matrix `<Matrix>`.

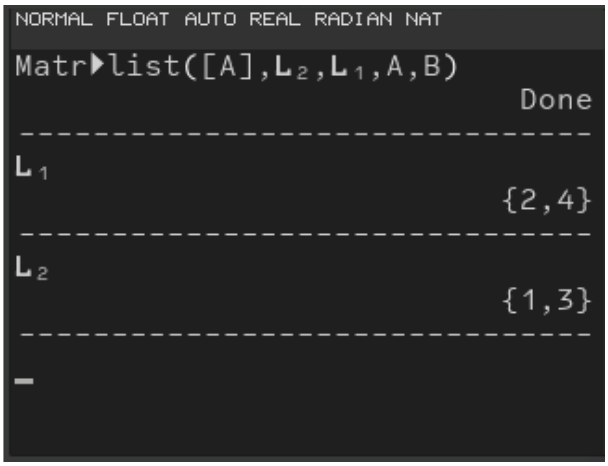
ERRORS:

- *INVALID DIMENSION* if the value of `<Column>` is out of bounds.

(Matr\x9Elist) `Matr▶list` `<Matrix:Matr>`, `<Ident Listn:List>`, `...`

Sequentially (left to right) extracts the column values from the matrix `<Matrix>` and places them into the corresponding lists `<Ident Listn>`. If the number of lists is less than the number of columns, extraction stops. If the number of columns is less than the number of list variables provided, the remaining variables are ignored (including their type).





ERRORS:

- *UNDEFINED* if any of the variables `<Ident Listₙ>` are not found.

resize

`resize` `<Ident:List>`, `<Size:Dec>`

Sets the size of list `<Ident>` to `<Size>`. The value of `<Size>` must be an integer in the range from 0 to 999. If `<Size>` is greater than the current list length, the new elements are initialized with 0. This command is equivalent to `<Size>→dim(<Ident>)` (see section Assignment), except it does not create a new list.

ERRORS:

- *UNDEFINED* if the list `<Ident>` is not found.
- *INVALID DIMENSION* if the value of `<Size>` is out of range.

`resize` `<Ident:Matr>`, `<Height:Dec>`, `<Width:Dec>`

Changes the size of the matrix `<Ident>`. `<Height>` is the new height, `<Width>` is the new width. The value of `<Height>` must be in the range 0 to 99. The value of `<Width>` must also be in the range 0 to 99. If the new size is greater than the current size, new elements are initialized with 0. This command is equivalent to `{<Height>,<Width>}→dim(<Ident>)` (see section Assignment), except it does not create a new matrix.

ERRORS:

- *UNDEFINED* if the matrix `<Ident>` is not found.
- *INVALID DIMENSION*
  - if `<Height>` is out of bounds.
  - if `<Width>` is out of bounds.
  - if only one of the values `<Height>` or `<Width>` is equal to 0.

setUpEditor

`setUpEditor`

Deletes all lists, then recreates lists `L₀` to `L₆` with zero length. Saves changes to calculator memory.

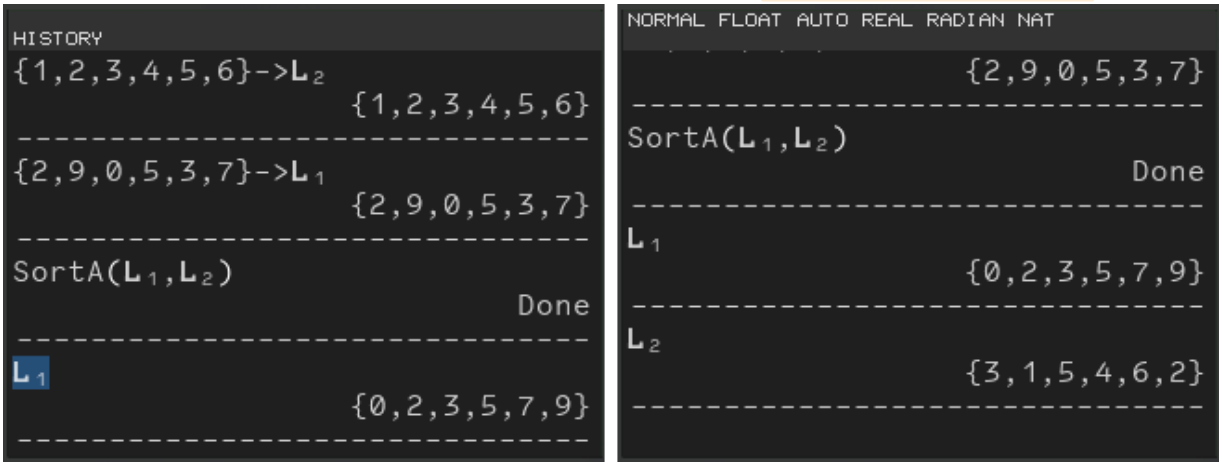
`setUpEditor` `<Identₙ>`, `...`

Attempts to create lists with the names `<Identₙ>`. Lists with names that cannot be converted to valid list names (see section Lists) will not be created. Saves changes to calculator memory.

SortA

> `SortA` `<Ident MainList:List>`
>
> Performs sorting of values in the list `<Ident MainList>` in ascending order. The comparison of all numbers is done based on their **absolute value**.

> `SortA` `<Ident MainList:List>`, `<Identn:List>`, `...`
>
> Additional lists `<Identn>` must have the same size as `<Ident MainList>`. The order of elements in the additional lists is changed according to the order of elements in `<Ident MainList:List>`.



> This command is relatively quiet and does not alert the user about errors (the only check is that all arguments are nodes of `IdentStmt`).

SortD

> `SortD` `<Ident MainList:List>`
>
> `SortD` `<Ident MainList:List>`, `<Identn:List>`, `...`
>
> This command works similarly to the `SortA` command, with the difference that the sorting is done in descending order.

3.6.2. Time-related Commands

setTmFmt

> `setTmFmt` `<Format:Dec>`
>
> Sets the clock format. `<Format>` should be either 12 or 24.
>
> ERRORS:
>
> - *DOMAIN* if the value `<Format>` is not 12 or 24.

setDtFmt

> `setDtFmt` `<Format:Dec>`
>
> Sets the date display format.
>
> - `<Format>` = 1 - **M/D/Y**
> - `<Format>` = 2 - **D/M/Y**
> - `<Format>` = 3 - **Y/M/D**
>
> ERRORS:
>
> - *DOMAIN* if the value `<Format>` is not 1, 2, or 3.

3.6.3. Input/Output-related commands

Disp

> `Disp`
>
> Opens the **home** window. Outputs empty string. If the number of output lines exceeds 100, the older lines will be removed.
>
> ERRORS:
>
> - *INVALID* if the command is not called within a script file.

> `Disp` `<Valueₙ:Any>`, `...`
>
> Opens the **home** window. Outputs the formatted value of the variables `<Valueₙ>` to the main screen in **Classic** format (see the section Processing mathematically formatted expressions for more details). Each value is output on a new line. If the number of output lines exceeds 100, the older lines will be removed.
>
> ERRORS:
>
> - *INVALID* if the command is not called within a script file.

DispGraph

> `DispGraph`
>
> Opens or updates the **graph** window.
>
> ERRORS:
>
> - *INVALID* if the command is not called within a script file.

DispTable

> `DispTable`

Opens or updates the **table** window.

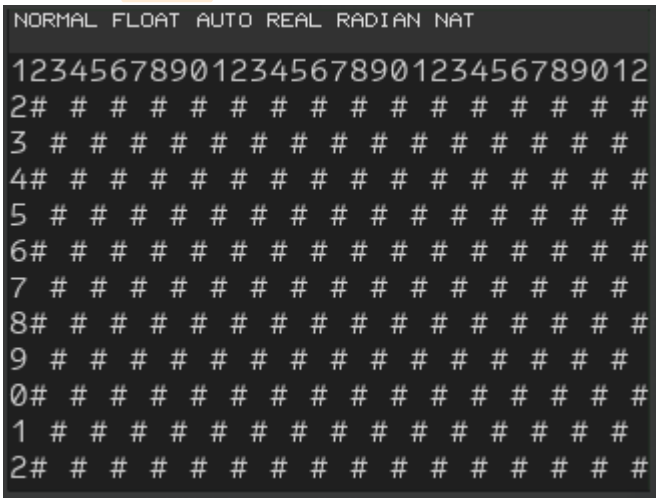
ERRORS:

- *INVALID* if the command is not called within a script file.

Output

`Output` `<Row:Dec>` `<Col:Dec>` `<Value:Any>`

Outputs the formatted value of the variable `<Value>` to the current screen in **Classic** format (see the section Processing mathematically formatted expressions) at the specified location. The coordinates of the point are given by the integer values `<Row>` and `<Col>`. The value `<Row>` must be between 1 and 12. The value `<Col>` must be between 1 and 32.



The result of the program will be erased by subsequent screen updates. It is recommended to use together with the commands `DispGraph` and `Pause`.

ERRORS:

- *INVALID* if the command is not called within a script file.
- *DATA TYPE* if the value of `<Row>` or `<Col>` is not an integer.
- *INVALID DIMENSION* if the values of `<Row>` or `<Col>` are out of the specified range.

Prompt

`Prompt` `<Identₙ:Any>`, `...`

The function allows sequentially assigning values to variables `<Identₙ>` during script file execution using user input. Unlike assignment, this function does not attempt to convert the variable name to the correct syntax. It is similar to the `Input(` function.

ERRORS:

- *INVALID*
  - if nothing was entered (empty input).
  - if used call of script file
  - if used some command

    - if the command is not called within a script file.
- *DATA TYPE* if the result of the user input expression has a different type than the variable `<Ident Var>`.
- Various errors related to assignment.

Pause

`Pause`

Pauses the program execution until the user presses the `Enter` key on the keyboard.

ERRORS:

- *INVALID* if the command is not called within a script file.

`Pause` `<Value:Any>`

Outputs the value `<Value>` using the `Disp` command and then passes control to the `Pause` command without parameters.

ERRORS:

- *INVALID* if the command is not called within a script file.

`Pause` `<Value:Any>`, `<Delay:Dec>`

Outputs the value `<Value>` using the `Disp`. Pauses the program execution until the user presses the `Enter` key or the specified delay time elapses. The delay time is specified in `<Delay>` seconds. The value of `<Delay>` must be an integer and greater than 0.

ERRORS:

- *INVALID* if the command is not called within a script file.
- *DATA TYPE* if the value of `<Delay>` is out of the specified range.

Wait

`Wait` `<Delay:Dec>`

Pauses the program execution until the specified delay time elapses. The delay time is specified in `<Delay>` seconds. The value of `<Delay>` must be greater than 0.001. The value `<Delay>` is rounded down to the thousandth place (0.00298 -> 0.002). The maximum delay time is 100 seconds and does not depend on the value `<Delay>`.

ERRORS:

- *INVALID* if the command is not called within a script file.
- *DOMAIN* if the value of `<Delay>` is out of the specified range.

Select

> `Select` `<Ident ListX:List>`, `<Ident ListY:List>`
>
> This command works only with linear () and scatter () statistical graphs. This command opens a window where the user selects a statistical graph using the ▼ and ▲ buttons. Then, using the ◄, ► and `Enter` buttons, the left boundary of the range is selected. Then, similarly, the right boundary of the range is selected. As a result of the selection, the X coordinates of the points within the selected range are placed into the list `<Ident ListX>`, and the corresponding Y coordinates are placed into the list `<Ident ListY>`. The command tries to convert the variables `<Ident ListX>` and `<Ident ListY>` to the lexical structure of list variables (`X` -> `⌞X`). The lists used by the selected graph will be replaced by `<Ident ListX>` and `<Ident ListY>`.
>
> ERRORS:
>
> - *INVALID* if the command cannot be executed.

3.6.4. Y-functions related commands

FnOff

> `FnOff`
>
> Disables the display of all Y-functions according to the current mode (**Func**, **Par**, **Pol**, **Seq**).

> `FnOff` `<Funcn:Dec>`, `...`
>
> Sequentially disables the display of Y-functions with the corresponding indexes `<Funcn>`. In **Par** mode, it disables a couple of functions X_{nt}, Y_{nt}. In **Seq** mode, disable functions 0, 3 and 6 will disable the corresponding three functions (0,1,2; 3,4,5; 6,7,8). `<Funcn>` must be an integer and located in the range from 0 to 9.
>
> ERRORS:
>
> - *DOMAIN* if the value of `<Funcn>` exceeds the specified limits.

FnOn

> `FnOn`
>
> Enables the display of all Y-functions according to the current mode (**Func**, **Par**, **Pol**, **Seq**).

> `FnOn` `<Funcn:Dec>`, `...`
>
> Sequentially enables the display of Y-functions with corresponding indices `<Funcn>`. In **Par** mode, it enables a pair of functions X_{nt}, Y_{nt}. `<Funcn>` must be an integer and within the range from 0 to 9.

> ERRORS:
>
> • *DOMAIN* if the value of `<Funcn>` is out of the specified range.

PlotsOff

> `PlotsOff`
>
> Disables the display of all statistical functions.

> `PlotsOff` `<StatPlotn:Dec>`, `...`
>
> Sequentially disables the display of statistical functions with corresponding indices `<StatPlotn>`. `<StatPlotn>` must be an integer and within the range from 1 to 3.
>
> ERRORS:
>
> • *DOMAIN* if the value of `<StatPlotn>` is out of the specified range.

PlotsOn

> `PlotsOn`
>
> Enables the display of all statistical functions.

> `PlotsOn` `<StatPlotn:Dec>`, `...`
>
> Sequentially enables the display of statistical functions with corresponding indices `<StatPlotn>`. `<StatPlotn>` must be an integer and within the range from 1 to 3.
>
> ERRORS:
>
> • *DOMAIN* if the value of `<StatPlotn>` is out of the specified range.

3.6.5. Statistical Commands

These commands operate with statistical variables. By default, these commands use the first list in the **MEMORY** > **Mem Management/Delete…** > **List** as the list `<Ident X>`, the second list in **MEMORY** > **Mem Management/Delete…** > **List** as the list `<Ident Y>`, and the list of the same length as the number of values, filled with values of 1, as the list `<Ident Freq>`. The list `<Ident Freq>` indicates how many times the corresponding point appears in the analyzed dataset.

1-VarStats

> (\xDF\xDEVarStats) `1-VarStats`
>
> `1-VarStats`

`1-VarStats` `<Ident X:List>`

`1-VarStats` `<Ident X:List>`, `<Ident Freq:List>`

`1-VarStats` (one-variable statistics) analyzes data with one measured variable.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `x̄`, `Sx`, `σx`, `Σx`, `Σx²`, `n`, `Q1`, `Q3`, `Med`, `minX`, `maxX`.

ERRORS:

- *UNKNOWN* if the command cannot be executed.
- *INVALID DIMENSION* if the list `<Ident X>` has a zero length.
- *DIM MISMATCH* if the length of the list `<Ident Freq>` does not match the length of the list `<Ident X>`.

2-VarStats

(\xE0\xDEVarStats) `2-VarStats`

`2-VarStats`

`2-VarStats` `<Ident X:List>`

`2-VarStats` `<Ident X:List>`, `<Ident Y:List>`

`2-VarStats` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`2-VarStats` (two-variable statistics) analyzes paired data. `<Ident X>` is the independent variable. `<Ident Y>` is the dependent variable.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `n`, `Σxy`, `x̄`, `Sx`, `σx`, `Σx`, `Σx²`, `minX`, `maxX`, `ȳ`, `Sy`, `σy`, `Σy`, `Σy²`, `minY`, `maxY`.

ERRORS:

- *UNKNOWN* if the command cannot be executed.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>`, and `<Ident Freq>` do not match.

Med-Med

(Med\xDEMed) `Med-Med`

`Med-Med`

`Med-Med` `<Ident X:List>`

`Med-Med` `<Ident X:List>`, `<Ident Y:List>`

`Med-Med` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`

`Med-Med` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`Med-Med` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`

`Med-Med` (median-median) fits the model equation `y=ax+b` to the data using the median-median line (resistant line) technique, calculating the summary points x1, y1, x2, y2, x3 and y3. `Med-Med` calculates values for `a` (slope) and `b` (y-intercept). If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `aX+b` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`.

ERRORS:

- *UNKNOWN* if the command cannot be executed.
- *STATISTICAL* if the command cannot be executed.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` do not match.

LinReg[ax+b]

(LinReg\xE1ax\xCEb\xE2) `LinReg[ax+b]`

`LinReg[ax+b]`

`LinReg[ax+b]` `<Ident X:List>`

`LinReg[ax+b]` `<Ident X:List>`, `<Ident Y:List>`

`LinReg[ax+b]` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`

`LinReg[ax+b]` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`LinReg[ax+b]` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`

`LinReg[ax+b]` (linear regression) fits the model equation `y=ax+b` to the data using a least-squares fit. It calculates values for `a` (slope) and `b` (y-intercept). When mode **Stat diagnostics** is **On** , it also calculates values for r^2 and `r`. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `aX+b` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
Sets the following statistical variables: `a`, `b`, [r^2, `r`].

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.

- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

LinReg[a+bx]

(LinReg\xE1a\xCEbx\xE2) `LinReg[a+bx]`

`LinReg[a+bx]`

`LinReg[a+bx]` `<Ident X:List>`

`LinReg[a+bx]` `<Ident X:List>`, `<Ident Y:List>`

`LinReg[a+bx]` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`

`LinReg[a+bx]` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`LinReg[a+bx]` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`

`LinReg[a+bx]` (linear regression) fits the model equation `y=a+bx` to the data using a least-squares fit. It calculates values for `a` (y-intercept) and `b` (slope). When mode **Stat diagnostics** is **On**, it also calculates values for `r²` and `r`. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `a+bX` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
Sets the following statistical variables: `a`, `b`, [`r²`, `r`].

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

QuadReg

`QuadReg`

`QuadReg` `<Ident X:List>`

`QuadReg` `<Ident X:List>`, `<Ident Y:List>`

`QuadReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`

`QuadReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`QuadReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`

`QuadReg` (quadratic regression) fits the second-degree polynomial `y=ax²+bx+c` to the data. It calculates values for `a`, `b` and `c`. When mode **Stat diagnostics** is **On**, it also calculates values for `R²`. For three data points, the equation is a polynomial fit. For four or more, it is a polynomial regression. At least three data points are required. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `aX²+bX+c` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`, `c`, [`R²`].

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

CubicReg

`CubicReg`

`CubicReg <Ident X:List>`

`CubicReg <Ident X:List>, <Ident Y:List>`

`CubicReg <Ident X:List>, <Ident Y:List>, <Ident Yfunc:Func>`

`CubicReg <Ident X:List>, <Ident Y:List>, <Ident Freq:List>`

`CubicReg <Ident X:List>, <Ident Y:List>, <Ident Freq:List>, <Ident Yfunc:Func>`

`CubicReg` (cubic regression) fits the third-degree polynomial `y=ax³+bx²+cx+d` to the data. It calculates values for `a`, `b`, `c` and `d`. When mode **Stat diagnostics** is **On**, it also calculates values for `R²`. For four points, the equation is a polynomial fit. For five or more, it is a polynomial regression. At least four points are required. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `aX³+bX²+cX+d` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`, `c`, `d`, [`R²`].

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

QuartReg

> `QuartReg`
>
> `QuartReg` `<Ident X:List>`
>
> `QuartReg` `<Ident X:List>`, `<Ident Y:List>`
>
> `QuartReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`
>
> `QuartReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`
>
> `QuartReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`
>
> `QuartReg` (quartic regression) fits the fourth-degree polynomial $y=ax^4+bx^3+cx^2+dx+e$ to the data. It calculates values for `a`, `b`, `c`, `d` and `e`. When mode **Stat diagnostics** is **On**, it also calculates values for R^2. For five points, the equation is a polynomial fit. For six or more, it is a polynomial regression. At least five points are required. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression $aX^4+bX^3+cX^2+dX+e$ with the set coefficients.
>
> If the command was not called within a script file, a screen with the measurement results will open.
> It sets the following statistical variables: `a`, `b`, `c`, `d`, `e`, [R^2].
>
> ERRORS:
>
> - *UNKNOWN* if the command execution is impossible.
> - *STATISTICAL* if the command execution is impossible.
> - *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
> - *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

LnReg

> `LnReg`
>
> `LnReg` `<Ident X:List>`
>
> `LnReg` `<Ident X:List>`, `<Ident Y:List>`
>
> `LnReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`
>
> `LnReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`
>
> `LnReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`
>
> `LnReg` (logarithmic regression) fits the model equation `y=a+b*ln(x)` to the data using a least-squares fit and transformed values ln(x) and y. It calculates values for `a` and `b`. When mode **Stat diagnostics** is **On**, it also calculates values for r^2 and `r`. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `a+b*ln(X)` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`, [`r`, `r²`].

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.
- *DOMAIN* if the list `<Ident X>` has a negative value.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

ExpReg

`ExpReg`

`ExpReg <Ident X:List>`

`ExpReg <Ident X:List>, <Ident Y:List>`

`ExpReg <Ident X:List>, <Ident Y:List>, <Ident Yfunc:Func>`

`ExpReg <Ident X:List>, <Ident Y:List>, <Ident Freq:List>`

`ExpReg <Ident X:List>, <Ident Y:List>, <Ident Freq:List>, <Ident Yfunc:Func>`

`ExpReg` (exponential regression) fits the model equation $y=ab^x$ *to the data using a least-squares fit and transformed values x and ln(y). It calculates values for* `a` *and* `b`*. When mode Stat diagnostics is On**, it also calculates values for `r²` and `r`. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `a*b^X` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`, [`r`, `r²`].

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.
- *DOMAIN* if the lists `<Ident Y>` or `<Ident X>` have a negative value.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

PwrReg

`PwrReg`

`PwrReg <Ident X:List>`

`PwrReg <Ident X:List>, <Ident Y:List>`

`PwrReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`

`PwrReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`PwrReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`

`PwrReg` (exponential regression) fits the model equation $y=ax^b$ *to the data using a least-squares fit and transformed values ln(x) and ln(y). It calculates values for* `a` *and* `b`. *When mode Stat diagnostics is On**, it also calculates values for r^2 and `r`. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `aX^b` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`, [`r`, r^2].

ERRORS:

- *UNKNOWN* if the command execution fails.
- *STATISTICAL* if the command execution fails.
- *DOMAIN* if the lists `<Ident Y>` or `<Ident X>` contain negative values.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>`, and `<Ident Freq>` are not equal.

Logistic

`Logistic`

`Logistic` `<Ident X:List>`

`Logistic` `<Ident X:List>`, `<Ident Y:List>`

`Logistic` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`

`Logistic` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`Logistic` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`

`Logistic` fits the model equation $y=c/(1+a*e^{-b*x})$ to the data using an iterative least-squares fit. It calculates values for `a`, `b` and `c`. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `c/(1+ae^(-(bX)))` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`, `c`.

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.

- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

SinReg

`SinReg`

`SinReg` `<Ident X:List>`

`SinReg` `<Ident X:List>`, `<Ident Y:List>`

`SinReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Yfunc:Func>`

`SinReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`

`SinReg` `<Ident X:List>`, `<Ident Y:List>`, `<Ident Freq:List>`, `<Ident Yfunc:Func>`

`SinReg` (sinusoidal regression) fits the model equation `y=a*sin(b*x+c)+d` to the data using an iterative least-squares fit. It calculates values for `a`, `b`, `c` and `d`.At least four data points are required. At least two data points per cycle are required in order to avoid aliased frequency estimates. The output of `SinReg` is always in radians, regardless of the **Radian**/**Degree** mode setting. If the argument `<Ident Yfunc>` is set, which is a Y-function variable, the command will assign the function the expression `a sin(bX+c)+d` with the set coefficients.

If the command was not called within a script file, a screen with the measurement results will open.
It sets the following statistical variables: `a`, `b`, `c`, `d`.

ERRORS:

- *UNKNOWN* if the command execution is impossible.
- *STATISTICAL* if the command execution is impossible.
- *INVALID DIMENSION* if the lists `<Ident X>` or `<Ident Y>` have a zero size.
- *DIM MISMATCH* if the lengths of the lists `<Ident X>`, `<Ident Y>` and `<Ident Freq>` are not equal.

Manual-Fit

(Manual\xDEFit) `Manual-Fit`

`Manual-Fit`

`Manual-Fit` `<Ident Yfunc:Func>`

By default, the first Y-function is used for storage. Opens the **graph** window and allows the user to select 2 points on the screen, through which a straight line will be drawn. The function expression `Y=mX+b` defining the specified line will be stored in `<Ident Yfunc>`.

ERRORS:

- *mX+b func calculate error* if the function calculation fails.

ANOVA

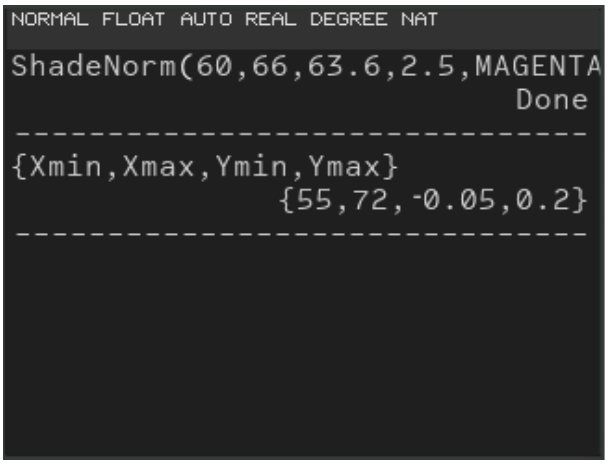
> `ANOVA` `<Ident Population:List>`, `<Ident Populationn:List>`, ...
>
> `ANOVA` computes a one-way analysis of variance for comparing the means of two or more populations. The `ANOVA` procedure for comparing these means involves analysis of the variation in the sample data. The null hypothesis H_0: $\mu1=\mu2=...=\mu k$ is tested against the alternative Ha: not all $\mu1...\mu k$ are equal.
>
> If the command was not called within a script file, a screen with the measurement results will open. `SS` is sum of squares and `MS` is mean square.
> It sets the following statistical variables: `F`, `p`, `Sxp`.

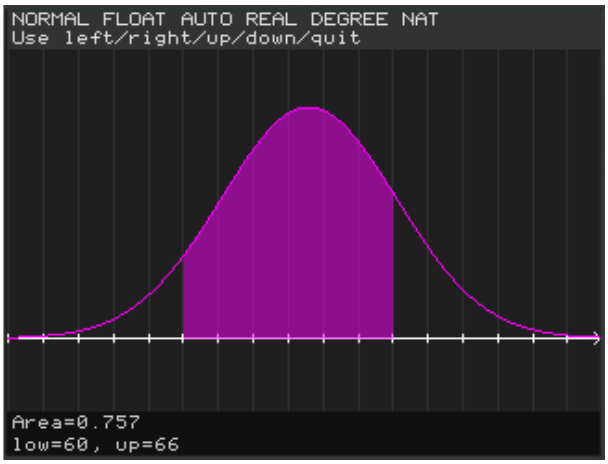
3.6.6. Distribution draw commands

These commands include the argument `<Color:Dec>`, which defines the color of graph. Default color is *BLUE* (10). If the value of `<Color>` is not an integer or does not match a valid color, a *DOMAIN* error will be generated.

ShadeNorm

> `ShadeNorm` `<lower:Dec>`, `<upper:Dec>`
>
> `ShadeNorm` `<lower:Dec>`, `<upper:Dec>`, `<Mu:Dec>`
>
> `ShadeNorm` `<lower:Dec>`, `<upper:Dec>`, `<Mu:Dec>`, `<Sigma:Dec>`
>
> `ShadeNorm` `<lower:Dec>`, `<upper:Dec>`, `<Mu:Dec>`, `<Sigma:Dec>`, `<Color:Dec>`
>
> `ShadeNorm` draws the normal density function specified by mean `<Mu>` and standard deviation `<Sigma>` and shades the area between `<lower>` and `<upper>`. The defaults are `<Mu>` = 0, `<Sigma>` = 1.





> ERRORS:
>
> - *DOMAIN* if the value `<Sigma>` is less or equal 0.

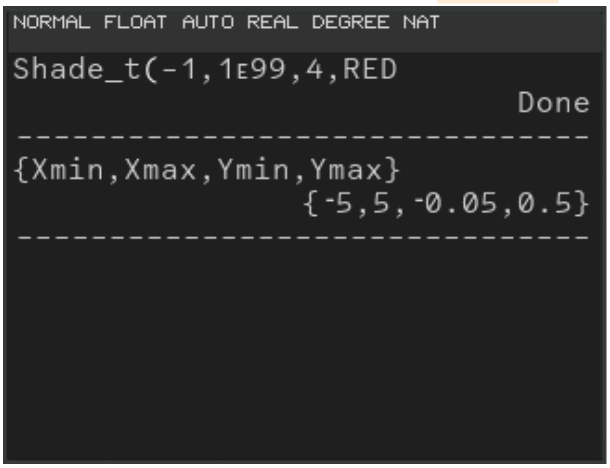
Shade_t

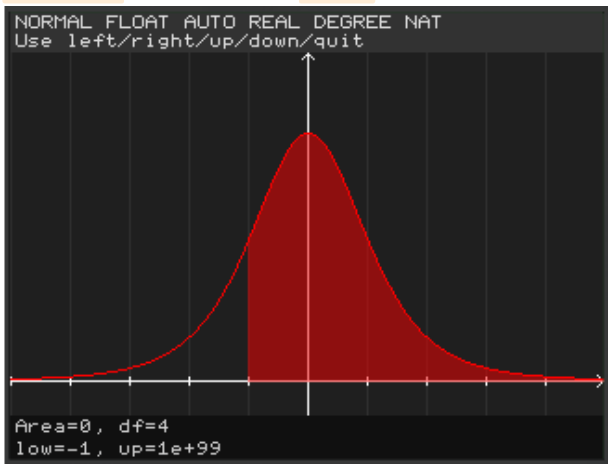
`Shade_t` `<lower:Dec>`, `<upper:Dec>`

`Shade_t` `<lower:Dec>`, `<upper:Dec>`, `<df:Dec>`

`Shade_t` `<lower:Dec>`, `<upper:Dec>`, `<df:Dec>`, `<Color:Dec>`

`Shade_t` draws the density function for the Student-t distribution specified by `<df>` (degrees of freedom) and shades the area between `<lower>` and `<upper>`. The default is `<df>` = 1.





ERRORS:

- *DOMAIN* if the value `<df>` is less or equal 0.

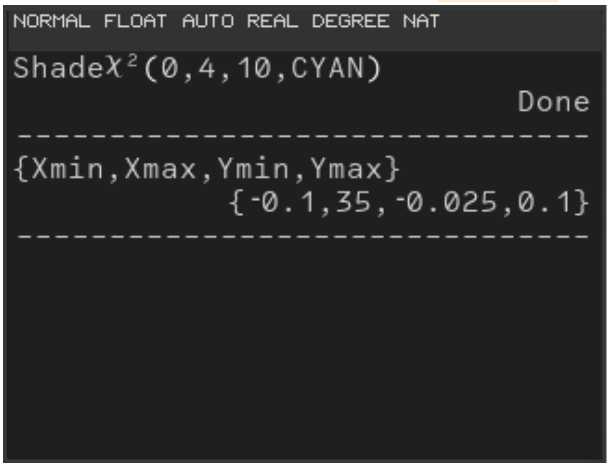
Shadeχ²

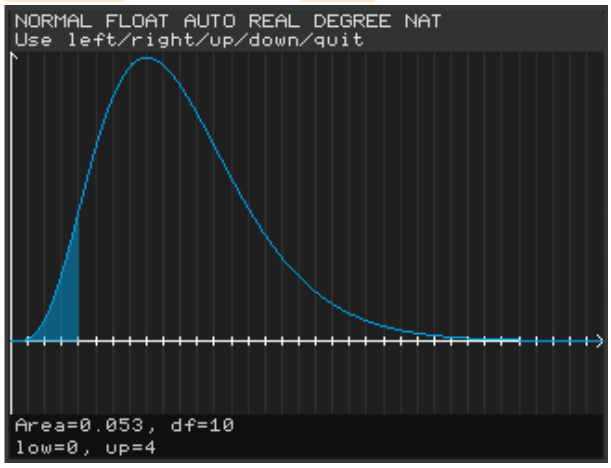
(Shade\xD8\xBD) `Shadeχ²` `<lower:Dec>`, `<upper:Dec>`

`Shadeχ²` `<lower:Dec>`, `<upper:Dec>`, `<df:Dec>`

`Shadeχ²` `<lower:Dec>`, `<upper:Dec>`, `<df:Dec>`, `<Color:Dec>`

`Shadeχ²` draws the density function for the χ² (chi-square) distribution specified by `<df>` (degrees of freedom) and shades the area between `<lower>` and `<upper>`. The default is `<df>` = 1.





ERRORS:

- *DOMAIN* if the value `<df>` is less or equal 0.

ShadeF

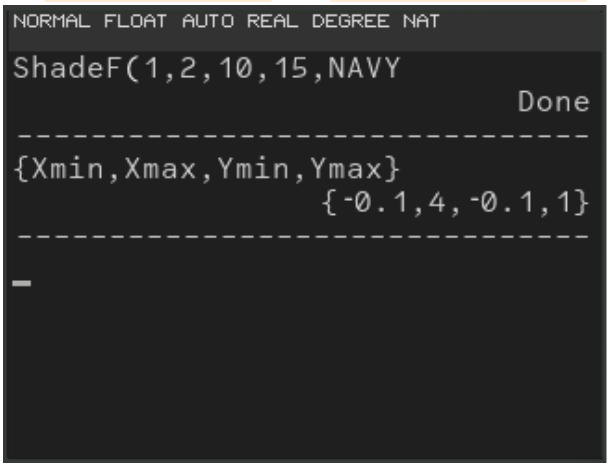
(Shade\xD8\xBD) `ShadeF` `<lower:Dec>`, `<upper:Dec>`

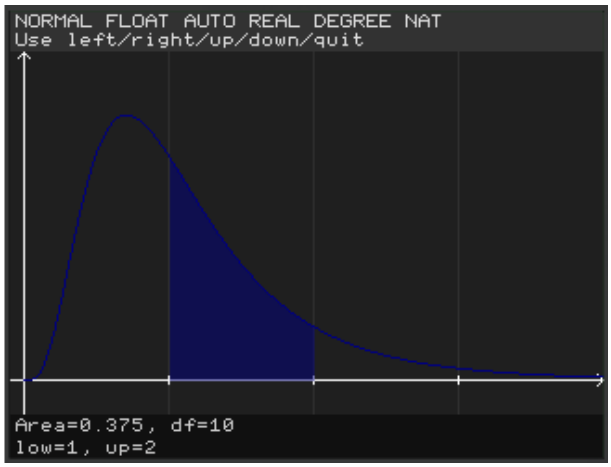
`ShadeF` `<lower:Dec>`, `<upper:Dec>`, `<Numerator df:Dec>`

`ShadeF` `<lower:Dec>`, `<upper:Dec>`, `<Numerator df:Dec>`, `<Denominator df:Dec>`

`ShadeF` `<lower:Dec>`, `<upper:Dec>`, `<Numerator df:Dec>`, `<Denominator df:Dec>`, `<Color:Dec>`

`ShadeF` draws the density function for the **F** distribution specified by `<Numerator df>` (degrees of freedom) and `<Denominator df>` and shades the area between `<lower>` and `<upper>`. The defaults are `<Numerator df>` = 1, `<Denominator df>` = 1.





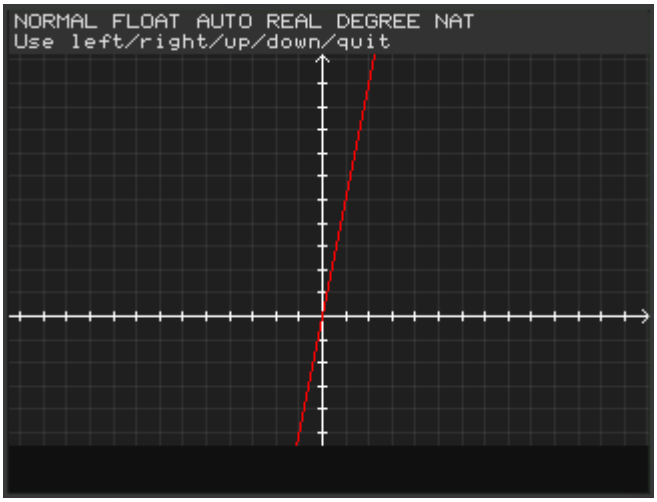
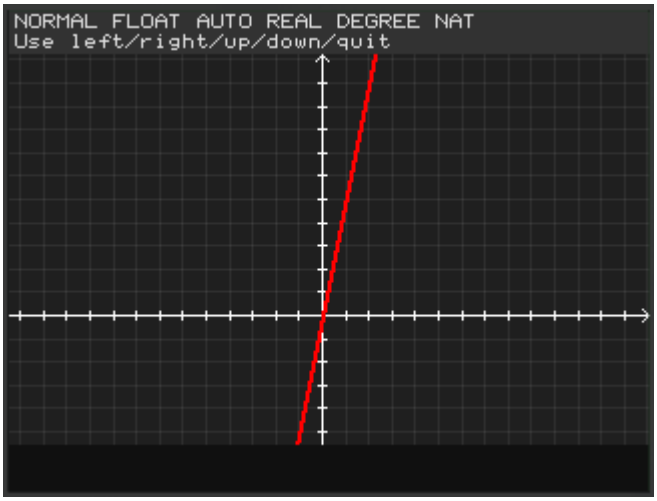
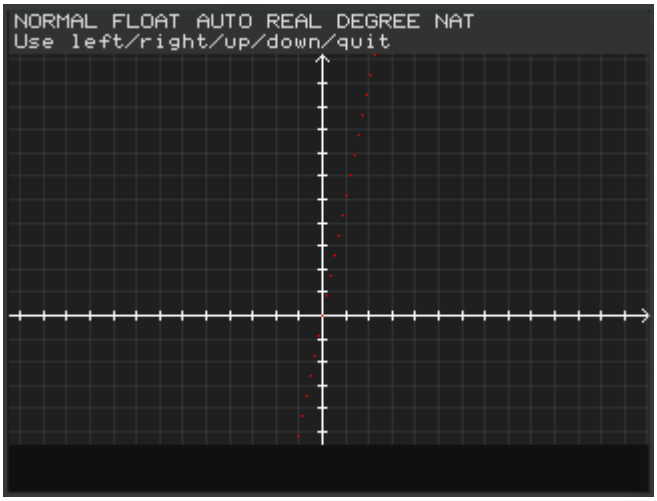
ERRORS:

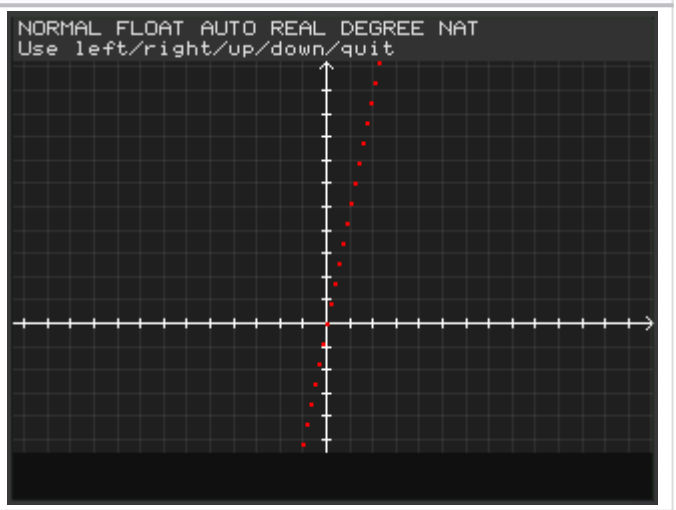
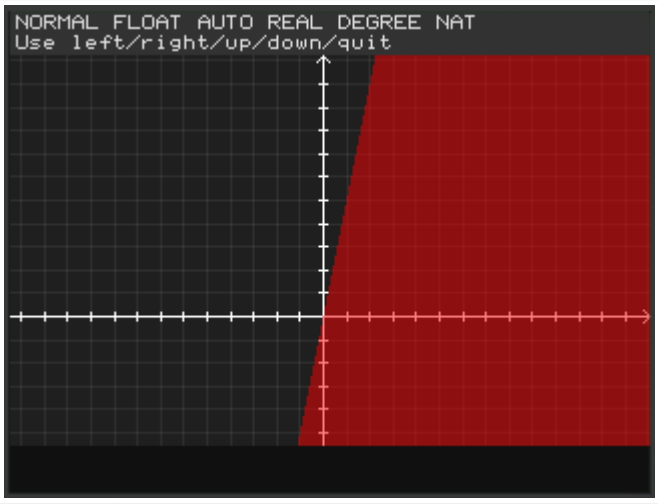
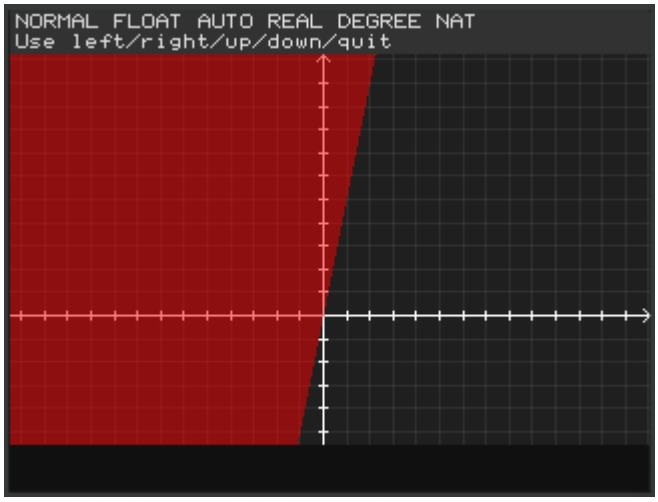
- *DOMAIN*
  - if the value `<Numerator df>` is less or equal 0.
  - if the value `<Denominator df>` is less or equal 0.

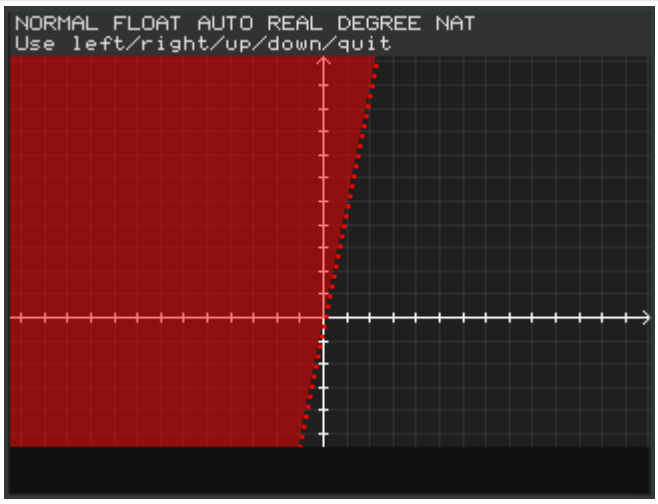
3.6.7. Draw commands

These commands may include the argument `<Color:Dec>`, which defines the color of graph. Default color is *BLUE* (10). If the value of `<Color>` is not an integer or does not match a valid color, a *DOMAIN* error will be generated.

These commands may also include the argument `<Style:Dec>`, which defines the line style. Defualt line style is *Thin Line* (0). `<Style>` can take the following values (unless otherwise specified). Using an invalid value will result in a *DOMAIN* error.

Name	Value	Icon	Example
Thin Line	0		
Bold Line	1		
Thin Dots	2		

Name	Value	Icon	Example
Bold Dots	3		
Draw Point	4		
Draw Point Line	5		
Fill Below	6		
Fill Above	7		

Name	Value	Icon	Example
Uneqal Below	8		
Uneqal Above	9		

ClrDraw

> `ClrDraw`
>
> Clears the **graph** window of elements drawn by commands (does not affect the rendering of active Y-functions). If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`.

Line

> `Line` `<X1:Dec>`, `<Y1:Dec>`, `<X2:Dec>`, `<Y2:Dec>`
>
> `Line` `<X1:Dec>`, `<Y1:Dec>`, `<X2:Dec>`, `<Y2:Dec>`, `<Color:Dec>`
>
> `Line` `<X1:Dec>`, `<Y1:Dec>`, `<X2:Dec>`, `<Y2:Dec>`, `<Color:Dec>`, `<Style:Dec>`
>
> Adds a line segment drawing to the **graph** window. Opens the **graph** window if the command is called outside of a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The values `<X1>` and `<Y1>` correspond to the coordinates of the first end. The values `<X2>` and `<Y2>` correspond to the coordinates of the second end.

> Usable styles: `Thin Line`, `Bold Line`, `Fill Below`, `Fill Above`.

Horizontal

> `Horizontal` `<Y:Dec>`
>
> `Horizontal` `<Y:Dec>`, `<Color:Dec>`
>
> `Horizontal` `<Y:Dec>`, `<Color:Dec>`, `<Style:Dec>`
>
> Adds a horizontal straight line drawing to the **graph** window. Opens the **graph** window if the command is called outside of a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The value `<Y>` corresponds to the coordinate through which the horizontal line passes.
>
> Usable styles: `Thin Line`, `Bold Line`, `Fill Below`, `Fill Above`.

Vertical

> `Vertical` `<X:Dec>`
>
> `Vertical` `<X:Dec>`, `<Color:Dec>`
>
> `Vertical` `<X:Dec>`, `<Color:Dec>`, `<Style:Dec>`
>
> Adds a vertical straight line drawing to the **graph** window. Opens the **graph** window if the command is called outside of a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The value `<X>` corresponds to the coordinate through which the vertical line passes.
>
> Usable styles: `Thin Line`, `Bold Line`.

Tangent

> `Tangent` `<Func>`, `<X/θ/T:Dec>`
>
> `Tangent` `<Func>`, `<X/θ/T:Dec>`, `<Color:Dec>`
>
> `Tangent` `<Func>`, `<X/θ/T:Dec>`, `<Color:Dec>`, `<Style:Dec>`
>
> Adds the drawing of the function `<Func>` and the tangent line to the graph of the function at the point `<X/θ/T>` on the **graph** window. Opens the **graph** window if the command is called outside of a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The command works only in **Func** and **Pol** modes. `<Func>` can be an expression (`1+X^2`), a string (`"sin(θ)"`), a Y-function variable (`Y₁`), or a variable containing a string with an expression (`Str₂`).
>
> ERRORS:
>
> - Various calculation errors related to the computation of values.
> - *NOT IMPLEMENT* if the current mode is **Seq**.

- *DOMAIN*
  - if the value `<X/θ/T>` is less than `θmin` (\x99min) or greater than `θmax` (\x99max) (screen **Window**) in **Pol** mode.
  - if the value `<X/θ/T>` is less than `Xmin` or greater than `Xmax` (screen **Window**) in **Func** mode.
  - if the value `<X/θ/T>` is less than `Tmin` or greater than `Tmax` (screen **Window**) in **Par** mode.
- *NONREAL ANS* if it is impossible to calculate the function or derivative value at the point `<X/θ/T>` or the calculated value is complex.
- *FUNC MODE MISMATCH* if the current display mode differs from the display mode in which the command was originally invoked.

DrawF

`DrawF` `<Func>`

`DrawF` `<Func>`, `<Color:Dec>`

Adds the drawing of the function `<Func>` on the **graph** window. Opens the **graph** window if the command is called outside of a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. `<Func>` can be an expression (`1+X^2`), a string (`"sin(θ)"`), a Y-function variable (`Y₁`), or a variable containing a string with an expression (`Str₂`).

ERRORS:

- Various calculation errors related to the computation of values.

Shade

`Shade` `<Lower Func>`, `<Upper Func>`

`Shade` `<Lower Func>`, `<Upper Func>`, `<X Left:Dec>`

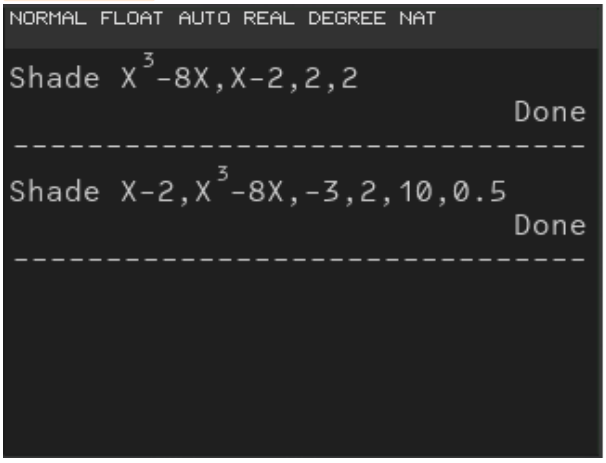
`Shade` `<Lower Func>`, `<Upper Func>`, `<X Left:Dec>`, `<X Right:Dec>`

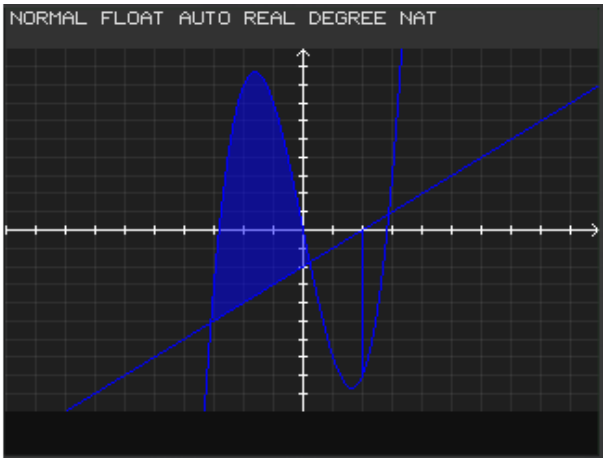
`Shade` `<Lower Func>`, `<Upper Func>`, `<X Left:Dec>`, `<X Right:Dec>`, `<Color:Dec>`

`Shade` `<Lower Func>`, `<Upper Func>`, `<X Left:Dec>`, `<X Right:Dec>`, `<Color:Dec>`, `<Opacity:Dec>`

Adds the drawing of two functions `<Lower Func>` and `<Upper Func>`, as well as the shaded area with the geometric locus of points with coordinates `<Lower Func>` ≤ Y ≤ `<Upper Func>`, `<X Left>` ≤ X ≤ `<X Right>`, on the **graph** window. Opens the **graph** window if the command is called outside of a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. `<Lower Func>` and `<Upper Func>` can be an expression (`1+X^2`), a string (`"sin(θ)"`), a Y-function variable (`Y₁`), or a variable containing a string with an expression (`Str₂`). If `<X Right>` is less than `<X Left>`, the area will not be shaded. `<Opacity>` is the transparency value of the shaded area and ranges from 0 to 1.

The defaults are `<X Left>` = `Xmin` (window **Window**), `<X Right>` = `Xmax` (window **Window**), `<Opacity>` = 1.





ERRORS:

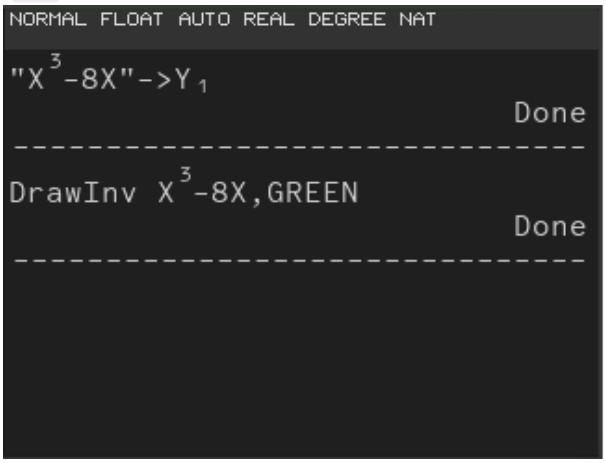
- Various calculation errors related to the computation of values.
- *DOMAIN* if the value `<Opacity>` exceeds the specified limits.

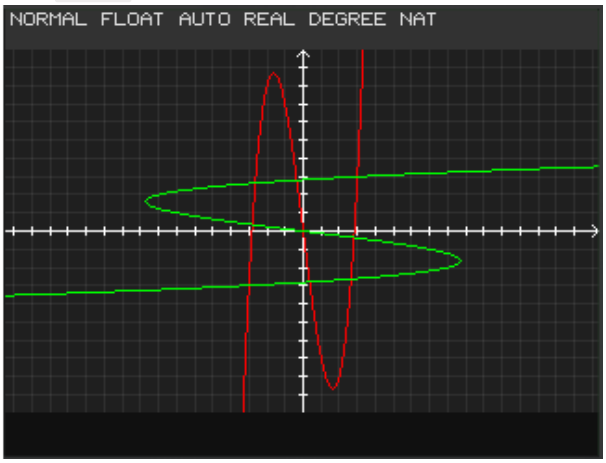
DrawInv

`DrawInv` `<Func>`

`DrawInv` `<Func>`, `<Color:Dec>`

Adds the drawing of the function, the mirror function `<Func>` with respect to `Y=X` (X values are projected onto the Y-axis, Y values are projected onto the X-axis), on the **graph** window. Opens the **graph** window if the command is called outside of a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. `<Func>` can be an expression (`1+X^2`), a string (`"sin(θ)"`), a Y-function variable (`Y1`), or a variable containing a string with an expression (`Str2`).





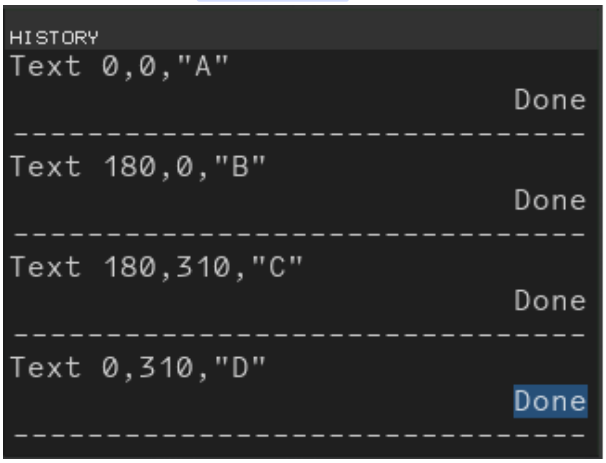
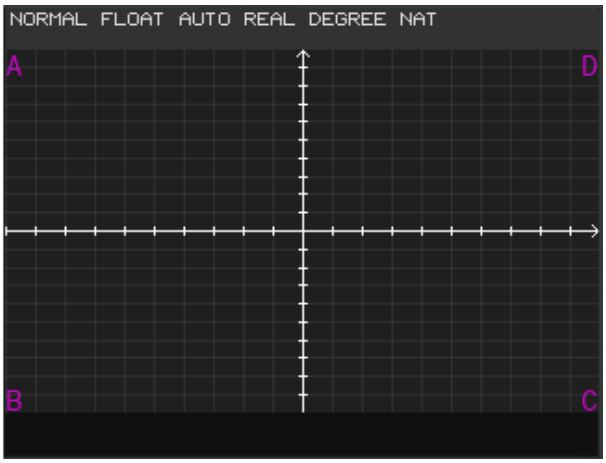
ERRORS:

- Various calculation errors related to the computation of values.

Circle

> `Circle` `<X:Dec>`, `<Y:Dec>`, `<Radius:Dec>`
>
> `Circle` `<X:Dec>`, `<Y:Dec>`, `<Radius:Dec>`, `<Color:Dec>`
>
> `Circle` `<X:Dec>`, `<Y:Dec>`, `<Radius:Dec>`, `<Color:Dec>`, `<Style:Dec>`
>
> Adds the drawing of a circle to the **graph** window. Opens the **graph** window if the command is called outside of a script file. The values `<X>` and `<Y>` correspond to the center of the circle. The value `<Radius>` corresponds to the radius of the circle and must not be less than 0.
>
> Usable styles: `Thin Line`.
>
> ERRORS:
>
> - *DOMAIN* if the value `<Radius>` is less than 0.

Text

> `Text` `<Y:Dec>`, `<X:Dec>`, `<Valuen:Any>`, ...
>
> Adds the drawing of text to the **graph** window. Opens the **graph** window if the command is called outside of a script file. The size of each character is **18 pixels in height and 10 pixels in width**. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The values `<X>` and `<Y>` correspond to the coordinates of the upper-left pixel of the first character. The drawing area is **195 pixels in height and 320 pixels in width** (it is possible to place part of the first character under the left and top border of the frame). The origin is located in the upper-left corner (0, 0), Y values increase from top to bottom, and X values increase from left to right. The values `<Valuen>` are converted to a string representation and displayed sequentially. The text color is set by the command `TextColor`.
>
> 
>
> 
>
>
> ERRORS:
>
> - *DOMAIN*
>   - if the value of `<X>` or `<Y>` is not an integer.
>   - if the value of `<X>` is less than -9 or greater than 319.
>   - if the value of `<Y>` is less than -17 or greater than 194.

TextColor

> `TextColor`
>
> Returns the name of the current text color (`Text`).

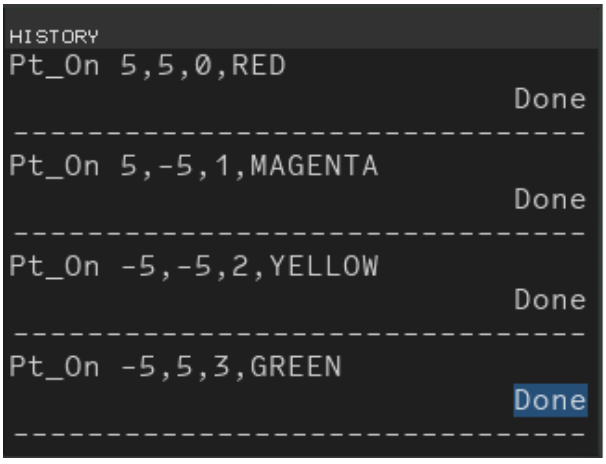
> `TextColor` `<Color:Dec>`
>
> Sets the current text color (`Text`). This value persists even after the calculator is rebooted.

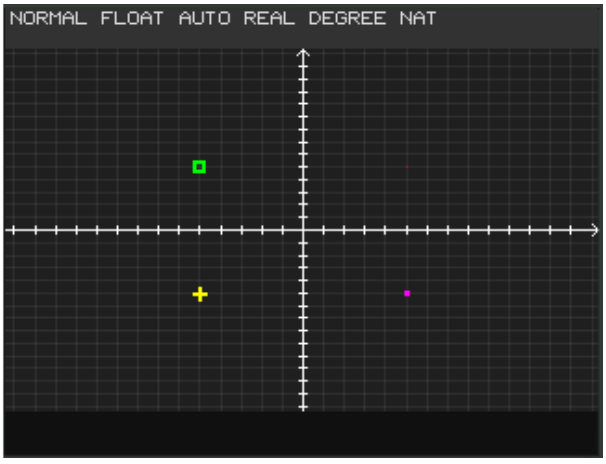
> `TextColor` `<Color:Dec>` , `<Y:Dec>` , `<X:Dec>` , `<Valuen:Any>` , ...
>
> Similar to the `Text` command, but also allows specifying the text color. **Does not set the current color**.

Pt_On

> `Pt_On` `<X:Dec>` , `<Y:Dec>`
>
> `Pt_On` `<X:Dec>` , `<Y:Dec>` , `<Point style:Dec>`
>
> `Pt_On` `<X:Dec>` , `<Y:Dec>` , `<Point style:Dec>` , `<Color:Dec>`
>
> Adds the drawing of a point on the **graph** window. Opens the **graph** window if the command is called not from a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The value `<Point style>` determines the style of the displayed point:

Name	Value	Icon
Thin Point	0	.
Bold Point	1	▪
Plus	2	+
Box	3	◻





ERRORS:

- *DOMAIN* if the value `<Point style>` is less than 0 or greater than 3.

Pt_Off

`Pt_Off` `<X:Dec>`, `<Y:Dec>`

`Pt_Off` `<X:Dec>`, `<Y:Dec>`, `<Point style:Dec>`

Adds the drawing of a point with the background color on the **graph** window. Opens the **graph** window if the command is called not from a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The value `<Point Style>` determines the style of the displayed point, which is described in detail in the command `Pt_On`.

ERRORS:

- *DOMAIN* if the value `<Point style>` is less than 0 or greater than 3.

Pt_Change

`Pt_Change` `<X:Dec>`, `<Y:Dec>`

`Pt_Change` `<X:Dec>`, `<Y:Dec>`, `<Point style:Dec>`

`Pt_Change` `<X:Dec>`, `<Y:Dec>`, `<Point style:Dec>`, `<Color:Dec>`

Adds the drawing of an inverse point to the **graph** window. Opens the **graph** window if the command is not called from a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. Inversion means that if the color of the pixel under the point matches the point's color, the pixel color will be replaced with the background color. Otherwise, it will be set to the point's color. The value `<Point style>` determines the style of the displayed point, which is described in detail in the command `Pt_On`.

ERRORS:

- *DOMAIN* if the value of `<Point style>` is less than 0 or greater than 3.

Pxl_On

`Pxl_On` `<X:Dec>`, `<Y:Dec>`

`Pxl_On` `<X:Dec>`, `<Y:Dec>`, `<Color:Dec>`

Adds the drawing of a pixel to the **graph** window. Opens the **graph** window if the command is not called from a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The values of `<X>` and `<Y>` correspond to the pixel coordinates. The rendering area size is **195 pixels in height and 320 pixels in width**. The origin is located in the top-left corner (0, 0); Y-axis values increase from top to bottom, and X-axis values increase from left to right.

ERRORS:

- *DOMAIN*
  - if the value of `<X>` or `<Y>` is not an integer.
  - if the value of `<X>` is less than 0 or greater than 319.
  - if the value of `<Y>` is less than 0 or greater than 194.

Pxl_Off

`Pxl_Off` `<X:Dec>`, `<Y:Dec>`

Adds the drawing of a pixel with background color to the **graph** window. Opens the **graph** window if the command is not called from a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. The values of `<X>` and `<Y>` correspond to the pixel coordinates. The rendering area size is **195 pixels in height and 320 pixels in width**. The origin is located in the top-left corner (0, 0); Y-axis values increase from top to bottom, and X-axis values increase from left to right.

ERRORS:

- *DOMAIN*
  - if the value of `<X>` or `<Y>` is not an integer.
  - if the value of `<X>` is less than 0 or greater than 319.
  - if the value of `<Y>` is less than 0 or greater than 194.

Pxl_Change

`Pxl_Change` `<X:Dec>`, `<Y:Dec>`

`Pxl_Change` `<X:Dec>`, `<Y:Dec>`, `<Color:Dec>`

Adds the drawing of an inverse pixel to the **graph** window. Opens the **graph** window if the command is not called from a script file. If the command is called from a script file, the **graph** window will not be updated. To update the **graph** window, you need to call the command `DispGraph`. Inversion means that if the color of the pixel under the point matches the point's color, the pixel's color will be replaced with the background color. Otherwise, the point's color will be set. The values of `<X>` and `<Y>` correspond to the pixel coordinates. The drawing area size is **195 pixels in height and 320 pixels in width**. The origin is located in the top-left corner (0, 0); Y-axis values increase from top to bottom, and X-axis values increase from left to right.

ERRORS:

- *DOMAIN*
  - if the value of `<X>` or `<Y>` is not an integer.
  - if the value of `<X>` is less than 0 or greater than 319.
  - if the value of `<Y>` is less than 0 or greater than 194.

StorePic

> `StorePic` `<Index:Dec>`
>
> Saves all displayed function graphs and graphic elements added using commands to the variable `Pic0` to `Pic9`. Opens the **graph** window if the command is not called from a script file. The value of `<Index>` specifies the number of the `Pic` variable and must be in the range from 0 to 9.
>
> The command will not be executed if an error is generated while rendering other commands or functions.
>
> ERRORS:
>
> - *DOMAIN*
> - if the value of `<Index>` is not an integer.
> - if the value of `<Index>` is less than 0 or greater than 9.

RecallPic

> `RecallPic` `<Index:Dec>`
>
> Restores all saved function graphs and graphic elements added using commands from the variable `Pic0` to `Pic9`. Clears current list of commands. Opens the **graph** window if the command is not called from a script file. The value of `<Index>` specifies the number of the `Pic` variable and must be in the range from 0 to 9.
>
> ERRORS:
>
> - *UNDEFINED* if the saved data could not be retrieved.
> - *FUNC MODE MISMATCH* if the data was saved in another function mode.
> - *DOMAIN*
> - if the value of `<Index>` is not an integer.
> - if the value of `<Index>` is less than 0 or greater than 9.

StoreGDB

> `StoreGDB` `<Index:Dec>`
>
> Saves the parameters of the **graph** window (**WINDOW** window), including zoom settings, format settings (**FORMAT** window), function graph style mode (**Thin**, **Thick**, **D-Thin**, **D-Thick**), function mode (**Func**, **Par**, **Pol**, **Seq**), and all functions into a variable `GDB0` to `GDB9`. The value of `<Index>` specifies the number of the `GDB` variable and must be in the range from 0 to 9.
>
> ERRORS:
>
> - *DOMAIN*
> - if the value of `<Index>` is not an integer.
> - if the value of `<Index>` is less than 0 or greater than 9.

RecallGDB

> `RecallGDB` `<Index:Dec>`
>
> Restores the parameters of the **graph** window (**WINDOW** window), including zoom settings, format settings (**FORMAT** window), function graph style mode (**Thin**, **Thick**, **D-Thin**, **D-Thick**), function mode (**Func**, **Par**, **Pol**, **Seq**), and all functions from the variable `GDB0` to `GDB9`. The value of `<Index>` specifies the number of the `GDB` variable and must be in the range from 0 to 9.
>
> ERRORS:
>
> - *UNDEFINED* if the saved data could not be retrieved.
> - *DOMAIN*
>   - if the value of `<Index>` is not an integer.
>   - if the value of `<Index>` is less than 0 or greater than 9.

3.6.8. Various commands

Equ▶String

> (Equ\x9EString) `Equ▶String` `<Ident Yfunc:Func>`, `<Ident Var:Str>`
>
> Saves the expression of the Y-function `<Ident Yfunc>` to the string variable `<Ident Var>`.

String▶Equ

> (String\x9EEqu) `String▶Equ` `<Var:Str>`, `<Ident Yfunc:Func>`
>
> Replaces the expression of the Y-function `<Ident Yfunc>` with the expression in the string variable `<Var>`.

ClearEntries

> `ClearEntries`
>
> Clears the contents of the main screen history.

clrHome

> `clrHome`
>
> Clears the main screen from the content. Fills the screen with the default color (depends on the mode **Dark mode**).

delPrgm

> `delPrgm` `<Programn:Str>`, `...`

> Deletes ZeroBasic programs from the file system. Since the data type of the argument `<Programₙ>` is *Str*, it is not possible to work with programs whose names contain double quotes (`"`). `<Programₙ>` corresponds to a file named `.zcb`. The search for programs is performed in the `/exchange/` path, excluding subdirectories, within the calculator's file system.
>
> ERRORS:
>
> - *no such program* if one of the files is not found (previous files will be deleted, subsequent ones will not).

delVar

> `delVar` `<Identₙ>`, `...`
>
> Deletes and clears the value of the passed variables from the calculator's memory. More details in the section Variables. The `TblInput` list will be cleared, not removed.
>
> ERRORS:
>
> - *READ ONLY VAR* if one of the parameters constant, or a readonly variable (previous parameters will be processed, subsequent ones will not).

Fill

> `Fill` `<Value:Dec|Imag>`, `<Ident:List|Matr>`
>
> Replaces the values of the matrix or list `<Ident>` with the argument `<Value>`.
>
> ERRORS:
>
> - *UNDEFINED* if the matrix or list with the name `<Ident>` is not found.

3.7. Functions

Corresponds to the `FnCallStmt` node. Functions are case-sensitive. The result of a function execution can have a different type and can be used within expressions.
Functions generate the following errors (unless stated otherwise):

- *SYNTAX* if one of the parameters `<Ident>` does not correspond to the `IdentStmt` node.
- *DATA TYPE*
  - if the type of one of the parameters is incorrect.
  - if the variable name does not match the variable type. More details about variable types and naming conventions can be found in the Variables section.
- *ARGUMENT* if the number of arguments is less than or greater than the expected number.
- *NAME LENGTH* if the length of the used variables exceeds the specified range. More details about the ranges can be found in the Variables section.
- *INVALID DIMENSION*
  - if the list argument contains no elements.

    - if the matrix argument has a zero size.
- *eval break* if the `On` button is pressed.
- *UNDEFINED*
    - if the list argument does not exist.
    - if the matrix argument does not exist.

3.7.1. List indexing

Indexing a list refers to retrieving the value of a specified element from the list. The syntax of this operation is similar to a function call, provided that the identifier (`IdentStmt`) matches the lexical structure of StandardListIdentifier or CustomListIdentifier. The exception is the `TblInput` variable.

> `␣ist(` `<Index:Dec>` `)` `->` *`Dec|Imag`*
>
> ERRORS:
>
> - *UNDEFINED* if the list does not exist.
> - *DATA TYPE* if the value of `<Index>` is not an integer.
> - *INVALID DIMENSION* if the value of `<Index>` is less than 1 or greater than the list length.

3.7.2. Matrix indexing

Matrix indexing refers to retrieving the value of a specified element from a matrix. The syntax of this operation is similar to a function call, provided that the identifier (`IdentStmt`) matches the lexical structure of MatrixIdentifier.

> `[M](` `<Row:Dec>` `<Column:Dec>` `)` `->` *`Dec|Imag`*
>
> ERRORS:
>
> - *UNDEFINED* if the matrix does not exist.
> - *DATA TYPE* if the value of `<Row>` or `<Column>` is not an integer.
> - *INVALID DIMENSION*
>   - if the value of `<Row>` is less than 1 or greater than the height of the matrix.
>   - if the value of `<Column>` is less than 1 or greater than the width of the matrix.

3.7.3. Ans indexing

The `Ans` variable is characterized by the absence of strict typing. Therefore, indexing `Ans` depends on the type of its content. The syntax of this operation is similar to a function call. If `Ans` does not exist in the current context, an *UNDEFINED* error will be generated. Indexing `Ans` when it contains a string will result in a *DATA TYPE* error.

Dec, Imag types

> `Ans(` `<Value:Dec|Imag|List|Matr>` `)` `->` *`Dec|Imag|List|Matr`*

> Similar to the multiplication operation between a number and the `<Value>` variable.

List type

The operation is similar to list indexing.

> `Ans(` `<Index:Dec>` `)` `-> Dec|Imag`
>
> ERRORS:
>
> - *DATA TYPE* if the value of `<Index>` is not an integer.
> - *INVALID DIMENSION* if the value of `<Index>` is less than 1 or greater than the list length.

Matr Type

The operation is similar to matrix indexing.

> `Ans(` `<Index Row:Dec>` `<Index Column:Dec>` `)` `-> Dec|Imag`
>
> ERRORS:
>
> - *DATA TYPE* if the value of `<Index Row>` or `<Index Column>` is not an integer.
> - *INVALID DIMENSION*
>   - if the value of `<Index Row>` is less than 1 or greater than the height of the matrix.
>   - if the value of `<Index Column>` is less than 1 or greater than the width of the matrix.

3.7.4. Y-functions

> `Y₀(` `<X:Dec|Imag>` `)` `-> Dec|Imag`

> `X₀ₜ(` `<T:Dec|Imag>` `)` `-> Dec|Imag`

> `r₀(` `<θ:Dec|Imag>` `)` `-> Dec|Imag`

> `u(` `<n:Dec|Imag>` `)` `-> Dec|Imag`

This type of functions evaluates the expression of the corresponding Y-function. A list of Y-functions and their arguments is provided in the table in the section Y-function variables. Before evaluation, the value of the argument `<X>`, `<T>`, `<θ>`, `<n>` replaces the value of the variable corresponding to the argument of the Y-function. If the Y-function expression (except for sequences, **Seq**) contains a direct (`Y₀=Y₀+1`) or indirect (`Y₀=Y₁` ; `Y₁=Y₀`) reference to its own value, an error *Yn RECURSION* will be generated. If the expression of the used function is not defined, an error *INVALID* will be generated.

3.7.5. Math Functions

sqrt, √

Synonym: (\x7F) `√(` ...

> `sqrt(` `<Value:Dec>` `)` -> *Dec*|*Imag*
>
> Returns the result of extracting the square root of the number `<Value>`
>
> ERRORS:
>
> - *NONREAL ANS* if the current mode is **Real** and the result is a complex number.

> `sqrt(` `<Value:Imag>` `)` -> *Imag*
>
> Returns the result of extracting the square root of the complex number `<Value>`

> `sqrt(` `<Value:List>` `)` -> *List*
>
> Returns a list composed of elements to which the specified operation was applied.

> `sqrt(` `<Value:Matr>` `)` -> *Matr*
>
> Returns a matrix composed of elements to which the specified operation was applied.

³√ (cube root)

> (\xD1\x7F) `³√(` `<Value:Dec>` `)` -> *Dec*
>
> (\xD1\x7F) `³√(` `<Value:Imag>` `)` -> *Imag*
>
> (\xD1\x7F) `³√(` `<Value:List>` `)` -> *List*
>
> (\xD1\x7F) `³√(` `<Value:Matr>` `)` -> *Matr*
>
> Returns the result of extracting the cube root from the number(s) `<Value>`. The function is analogous to the expression `root(` `<Value>` `, 3)`.

root, ⁿ√

Synonym: (\xA1) `ⁿ√(` ...

> `root(` `<Value:Dec>` , `<Index:Dec>` `)` -> *Dec*|*Imag*
>
> Returns the result of extracting the root of degree `<Index>` from the number `<Value>`. This is analogous to the expression `<Value>` `^(1/` `<Index>` `)`.

ERRORS:

- *DOMAIN* if the value of `<Index>` is 0.
- *NONREAL ANS* if the current mode is **Real**, and the result is a complex number.

`root( <Value:Imag> , <Index:Dec|Imag> )` -> *Imag*

ERRORS:

- *DOMAIN* if the value of `<Index>` is 0.

`root( <Value:List> , <Index:Dec|Imag> )` -> *List*

Returns a list formed from the elements over which the specified operation is performed.

ERRORS:

- *DOMAIN* if the value of `<Index>` is 0.

`root( <Value:Matr> , <Index:Dec|Imag> )` -> *Matr*

Returns a matrix formed from the elements over which the specified operation is performed.

ERRORS:

- *DOMAIN* if the value of `<Index>` is 0.

fMin

`fMin( <Func:Func> , <Ident Var:Dec> , <Lower:Dec> , <Upper:Dec> )` -> *Dec*

`fMin(` return the value at which the local minimum value of `<Func>` with respect to `<Ident Var>` occurs, between `<Lower>` and `<Upper>` values for `<Ident Var>`. The accuracy is `1e-6`.

`<Func>` can be an expression (`1+X^2`) or a variable of a Y-function (`Y₁`).

ERRORS:

- Various calculation errors related to computing values.
- *bound* if the value of `<Lower>` is greater than the value of `<Upper>`

fMax

`fMax( <Func:Func> , <Ident Var:Dec> , <Lower:Dec> , <Upper:Dec> )` -> *Dec*

`fMax(` return the value at which the local maximum value of `<Func>` with respect to `<Ident Var>` occurs, between `<Lower>` and `<Upper>` values for `<Ident Var>`. The accuracy is `1e-6`.

`<Func>` can be an expression (`1+X^2`) or a variable of a Y-function (`Y1`).

ERRORS:

- Various calculation errors related to computing values.
- *bound* if the value of `<Lower>` is greater than the value of `<Upper>`

nDeriv

`nDeriv(` `<Func:`*`Func`*`>`, `<Ident Var:`*`Dec`*`>`, `<Value:`*`Dec`*`>` `)` -> *Dec*

`nDeriv(` `<Func:`*`Func`*`>`, `<Ident Var:`*`Dec`*`>`, `<Value:`*`Dec`*`>`, `<Tolerance:`*`Dec`*`>` `)` -> *Dec*

`nDeriv(` returns an approximate derivative of `<Func>` with respect to `<Ident Var>`, given the `<Value>` at which to calculate the derivative and `<Tolerance>`. By default, the value of `<Tolerance>` is `1e-3`. `nDeriv(` uses the symmetric difference quotient method, which approximates the numerical derivative value as the slope of the secant line through these points.

$$f'(x) = \frac{f(x + \varepsilon) - f(x - \varepsilon)}{2\varepsilon}$$

As `<Tolerance>` becomes smaller, the approximation usually becomes more accurate. `<Func>` can be an expression (`1+X^2`) or a variable of a Y-function (`Y1`).

ERRORS:

- Various calculation errors related to computing values.

fnInt

`fnInt(` `<Func:`*`Func`*`>`, `<Ident Var:`*`Dec`*`>`, `<Lower:`*`Dec`*`>`, `<Upper:`*`Dec`*`>` `)` -> *Dec*

`fnInt(` returns the numerical integral (Gauss–Legendre quadrature method) of `<Func>` with respect to `<Ident Var>`, given `<Lower>` limit, `<Upper>` limit. The calculation uses `abs(<Upper> - <Lower>)*5` partitions, but no more than 500. `<Func>` can be an expression (`1+X^2`) or a variable of a Y-function (`Y1`).

ERRORS:

- Various calculation errors related to computing values.

summ, Σ

Synonym: (\xA2) `Σ(` ...

`summ(` `<Func:`*`Func`*`>`, `<Ident Var:`*`Dec`*`>`, `<Lower:`*`Dec`*`>`, `<Upper:`*`Dec`*`>` `)` -> *Dec*

`summ(` returns the sum of the results of calculating `<Func>` for `<Ident Var>`. `<Ident Var>` takes values from `<Lower>` to `<Upper>` with a step of 1. `<Func>` can be an expression (`1+X^2`) or a variable of a Y-function (`Y1`).

ERRORS:

- Various calculation errors related to the computation of values.
- *INCREMENT* if the value of `<Lower>` is greater than the value of `<Upper>`.

exp

`exp( <Value:Dec|Imag> ) -> Dec|Imag`

Returns the result of raising the number `e` to the power of `<Value>`. Equivalent to the expression `e^ <Value>`.

`exp( <Value:List> ) -> List`

Returns a list formed from elements to which the specified operation was applied.

ln

`ln( <Value:Dec|Imag> ) -> Dec|Imag`

Returns the result of calculating the natural logarithm of the number `<Value>`.

ERRORS:

- *DOMAIN* if the value of `<Value>` is equal to 0.

`ln( <Value:List> ) -> List`

Returns a list formed from elements to which the specified operation was applied.

log

`log( <Value:Dec|Imag> ) -> Dec|Imag`

Returns the result of calculating the common (base-10) logarithm of the number `<Value>`.

ERRORS:

- *DOMAIN* if the value of `<Value>` is equal to 0.

`log( <Value:List> ) -> List`

Returns a list formed from elements to which the specified operation was applied.

logBASE, log

Synonym: `log(`...

> `logBASE(` `<Value:`*`Dec|Imag`*`>`, `<Index:`*`Dec|Imag`*`>` `)` -> *`Dec|Imag`*
>
> Returns the result of calculating the logarithm of the number `<Value>` with base `<Index>`. Equivalent to the expression `log10(<Value>) / log10(<Index>)`.
>
> ERRORS:
>
> - *DOMAIN*
> - if the value of `<Value>` is equal to 0.
> - if the value of `<Index>` is equal to 0.
> - *DIVIDE BY 0* if the value of `<Index>` is equal to 1

> `logBASE(` `<Value:`*`List`*`>`, `<Index:`*`Dec|Imag`*`>` `)` -> *`List`*
>
> Returns a list formed from elements to which the specified operation was applied.

3.7.6. Numeric functions

abs

> `abs(` `<Value:`*`Dec|Imag`*`>` `)` -> *`Dec`*
>
> Returns the absolute value of the number `<Value>`.

> `abs(` `<Value:`*`List`*`>` `)` -> *`List`*
>
> Returns a list formed from elements to which the specified operation was applied.

sign

> `sign(` `<Value:`*`Dec`*`>` `)` -> *`Dec`*
>
> Returns:
>
> - -1 if `<Value>` is less than 0.
> - 0 if `<Value>` is equal to 0.
> - 1 if `<Value>` is greater than 0.

> `sign(` `<Value:`*`List`*`>` `)` -> *`List`*
>
> Returns a list formed from elements to which the specified operation was applied.

round

> `round( <Value:Dec|Imag> )` -> *Dec|Imag*
>
> `round( <Value:Dec|Imag>, <Digits:Dec> )` -> *Dec|Imag*
>
> Rounds the value `<Value>` (both imaginary and real parts separately) to `<Digits>` digits after the decimal point. If `<Digits>` is 0, the value `<Value>` is rounded to the nearest integer. If `<Digits>` is less than 0, the value `<Value>` is rounded to the corresponding decimal digit (including it). By default, `<Digits>` is 0.
>
> ERRORS:
>
> - *DATA TYPE* if the value of `<Digits>` is not an integer.

> `round( <Value:List> )` -> *List*
>
> `round( <Value:List>, <Digits:Dec> )` -> *List*
>
> Returns a list formed from elements to which the specified operation was applied.

> `round( <Value:Matr> )` -> *Matr*
>
> `round( <Value:Matr>, <Digits:Dec> )` -> *Matr*
>
> Returns a matrix formed from elements to which the specified operation was applied.

ceil

> `ceil( <Value:Dec|Imag> )` -> *Dec|Imag*
>
> Rounds the value (both imaginary and real parts separately) `<Value>` up to the smallest integer such that `ceil(x) ≥ x`.

> `ceil( <Value:List> )` -> *List*
>
> Returns a list formed from elements to which the specified operation was applied.

floor, int

> Synonym: `int(`...
>
> `floor( <Value:Dec|Imag> )` -> *Dec|Imag*

> Rounds the value (both imaginary and real parts separately) `<Value>` down to the largest integer such that `floor(x) ≤ x`.
> The synonym `int(` may be misleading, as it suggests that this function returns the integer part of a number. This is not the case. The function that returns the integer part is `iPart(`.

> `floor( <Value:List> ) -> List`
>
> Returns a list formed from elements to which the specified operation was applied.

iPart

> `iPart( <Value:Dec|Imag> ) -> Dec|Imag`
>
> Returns the integer part of the value `<Value>` (real and imaginary parts separately).

> `iPart( <Value:List> ) -> List`
>
> Returns a list formed from elements to which the specified operation was applied.

> `iPart( <Value:Matr> ) -> Matr`
>
> Returns a matrix formed from elements to which the specified operation was applied.

fPart

> `fPart( <Value:Dec|Imag> ) -> Dec|Imag`
>
> Returns the fractional part of the value `<Value>` (both imaginary and real parts separately).

> `fPart( <Value:List> ) -> List`
>
> Returns a list composed of elements to which the specified operation was applied.

> `fPart( <Value:Matr> ) -> Matr`
>
> Returns a matrix composed of elements to which the specified operation was applied.

min

> `min( <A:Dec>, <B:Dec> ) -> Dec`
>
> Returns the smaller number between the values `<A>` and `<B>`.

`min( <Numbers:List> ) -> Dec`

Returns the smaller number from the values in the list `<Numbers>`.

`min( <Value:Dec>, <Numbers:List> ) -> List`

`min( <Numbers:List>, <Value:Dec> ) -> List`

Returns a list formed from the results of applying the specified operation between the number `<Value>` and the corresponding element in the list `<Numbers>`.

ERRORS:

- *DATA TYPE* if the list `<Numbers>` contains a complex number.

`min( <A:List>, <B:List> ) -> List`

Returns a list formed from the results of applying the specified operation between the corresponding elements of the lists `<A>` and `<B>`.

ERRORS:

- *DATA TYPE*
  - if the list `<A>` contains a complex number.
  - if the list `<B>` contains a complex number.
- *DIM MISMATCH* if the lengths of lists `<A>` and `<B>` are different.

max

`max( <A:Dec>, <B:Dec> ) -> Dec`

Returns the largest number between the values `<A>` and `<B>`.

`max( <Numbers:List> ) -> Dec`

Returns the largest number from the values in the list `<Numbers>`.

`max( <Value:Dec>, <Numbers:List> ) -> List`

`max( <Numbers:List>, <Value:Dec> ) -> List`

Returns a list formed from the results of applying the specified operation between the number `<Value>` and the corresponding element in the list `<Numbers>`.

ERRORS:

- *DATA TYPE* if the list `<Numbers>` contains a complex number.

`max( <A:List>, <B:List> ) -> List`

Returns a list formed from the results of applying the specified operation between the corresponding elements of the lists `<A>` and `<B>`.

ERRORS:

- *DATA TYPE*
  - if the list `<A>` contains a complex number.
  - if the list `<B>` contains a complex number.
- *DIM MISMATCH* if the lengths of lists `<A>` and `<B>` are different.

lcm

`lcm( <A:Dec>, <B:Dec> ) -> Dec`

`lcm(` returns the least common multiple of `<A>` and `<B>`. `<A>` and `<B>` must be nonnegative integers.

ERRORS:

- *DOMAIN*
  - if `<A>` or `<B>` is not an integer.
  - if `<A>` or `<B>` is less than 0.

`lcm( <Value:Dec>, <Numbers:List> ) -> List`

`lcm( <Numbers:List>, <Value:Dec> ) -> List`

Returns a list formed from the results of applying the specified operation between the number `<Value>` and the corresponding element in the list `<Numbers>`.

ERRORS:

- *DATA TYPE* if the list `<Numbers>` contains a complex number.

`lcm( <A:List>, <B:List> ) -> List`

Returns a list formed from the results of applying the specified operation between the corresponding elements of the lists `<A>` and `<B>`.

ERRORS:

- *DATA TYPE*
  - if the list `<A>` contains a complex number.
  - if the list `<B>` contains a complex number.
- *DIM MISMATCH* if the lengths of lists `<A>` and `<B>` are different.

gcd

`gcd( <A:Dec>, <B:Dec> ) -> Dec`

`gcd(` returns the greatest common divisor of `<A>` and `<B>`. `<A>` and `<A>` must be nonnegative integers.

ERRORS:

- *DOMAIN*
  - if `<A>` or `<B>` is not an integer.
  - if `<A>` or `<B>` is less than 0.

`gcd( <Value:Dec>, <Numbers:List> ) -> List`

`gcd( <Numbers:List>, <Value:Dec> ) -> List`

Returns a list formed from the results of applying the specified operation between the number `<Value>` and the corresponding element in the list `<Numbers>`.

ERRORS:

- *DATA TYPE* if the list `<Numbers>` contains a complex number.

`gcd( <A:List>, <B:List> ) -> List`

Returns a list formed from the results of applying the specified operation between the corresponding elements of the lists `<A>` and `<B>`.

ERRORS:

- *DATA TYPE*
  - if the list `<A>` contains a complex number.
  - if the list `<B>` contains a complex number.
- *DIM MISMATCH* if the lengths of lists `<A>` and `<B>` are different.

remainder, rem

Synonym: `rem(` ...

> `remainder( <Dividend:Dec>, <Divisor:Dec> ) -> Dec`
>
> Returns the remainder of dividing `<Dividend>` by `<Divisor>`.
>
> ERRORS:
>
> - *DIVIDE BY 0* if the value of `<Divisor>` is 0.

3.7.7. Trigonometric and hyperbolic functions

sin

> `sin( <Value:Dec> ) -> Dec`
>
> Returns the sinus of `<Value>`. The result depends on the current **Radian** or **Degree** mode.

> `sin( <Value:Imag> ) -> Imag`
>
> Returns the sinus of `<Value>`.

> `sin( <Value:List> ) -> List`
>
> Returns a list formed from the elements after applying the specified operation.

asin, sin⁻¹

Synonym: (sin\xD4) `sin⁻¹(`...

> `asin( <Value:Dec> ) -> Dec`
>
> Returns the inverse sinus of `<Value>`. The value of `<Value>` must be located in the range from -1 to 1. The result depends on the current **Radian** or **Degree** mode.
>
> ERRORS:
>
> - *DOMAIN* if the value of `<Value>` is less than -1 or greater than 1.

> `asin( <Value:Imag> ) -> Imag`
>
> Returns the inverse sinus of `<Value>`.

> `asin( <Value:List> ) -> List`
>
> Returns a list formed from the elements after applying the specified operation.

sinh

> `sinh( <Value:Dec|Imag> )` -> *Dec|Imag*
>
> Returns the hyperbolic sinus of `<Value>`.

> `sinh( <Value:List> )` -> *List*
>
> Returns a list formed from the elements after applying the specified operation.

arsinh, sinh⁻¹

Synonym: (sinh\xD4) `sinh⁻¹(`...

> `arsinh( <Value:Dec|Imag> )` -> *Dec|Imag*
>
> Returns the inverse hyperbolic sinus of `<Value>`.

> `arsinh( <Value:List> )` -> *List*
>
> Returns a list formed from the elements after applying the specified operation.

cos

> `cos( <Value:Dec> )` -> *Dec*
>
> Returns the cosinus of `<Value>`. The result depends on the current **Radian** or **Degree** mode.

> `cos( <Value:Imag> )` -> *Imag*
>
> Returns the cosinus of `<Value>`.

> `cos( <Value:List> )` -> *List*
>
> Returns a list formed from the elements after applying the specified operation.

acos, cos⁻¹

Synonym: (cos\xD4) `cos⁻¹(`...

> `acos( <Value:Dec> )` -> *Dec*
>
> Returns the inverse cosinus of `<Value>`. The value of `<Value>` must be located in the range from -1 to 1. The result depends on the current **Radian** or **Degree** mode.

ERRORS:

- *DOMAIN* if the value of `<Value>` is less than -1 or greater than 1.

`acos( <Value:Imag> ) -> Imag`

Returns the inverse cosinus of `<Value>`.

`acos( <Value:List> ) -> List`

Returns a list formed from the elements after applying the specified operation.

cosh

`cosh( <Value:Dec|Imag> ) -> Dec|Imag`

Returns the hyperbolic cosinus of `<Value>`.

`cosh( <Value:List> ) -> List`

Returns a list formed from the elements after applying the specified operation.

arcosh, cosh⁻¹

Synonym: (cosh\xD4) `cosh⁻¹(` ...

`arcosh( <Value:Dec> ) -> Dec`

Returns the inverse hyperbolic cosinus of `<Value>`. The value of `<Value>` must be greater than or equal to 1.

ERRORS:

- *DOMAIN* if the value of `<Value>` is less than 1.

`arcosh( <Value:Imag> ) -> Imag`

Returns the inverse hyperbolic cosinus of `<Value>`.

`arcos( <Value:List> ) -> List`

Returns a list formed from the elements after applying the specified operation.

tan

> `tan( <Value:Dec> )` -> *Dec*
>
> Returns the tangent of `<Value>`. The result depends on the current **Radian** or **Degree** mode.

> `tan( <Value:Imag> )` -> *Imag*
>
> Returns the tangent of `<Value>`.

> `tan( <Value:List> )` -> *List*
>
> Returns a list formed from the elements after applying the specified operation.

atan, tan⁻¹

Synonym: (tan\xD4) `tan⁻¹(`...

> `atan( <Value:Dec> )` -> *Dec*
>
> Returns the inverse tangent of `<Value>`. The result depends on the current **Radian** or **Degree** mode.

> `atan( <Value:Imag> )` -> *Imag*
>
> Returns the inverse tangent of `<Value>`.

> `atan( <Value:List> )` -> *List*
>
> Returns a list formed from the elements after applying the specified operation.

tanh

> `tanh( <Value:Dec|Imag> )` -> *Dec|Imag*
>
> Returns the hyperbolic tangent of `<Value>`.

> `tanh( <Value:List> )` -> *List*
>
> Returns a list formed from the elements after applying the specified operation.

artanh, tanh⁻¹

Synonym: (tanh\xD4) `tanh⁻¹(`...

> `artanh( <Value:Dec> ) -> Dec`
>
> Returns the inverse hyperbolic tangent of `<Value>`. The value of `<Value>` must be greater than -1 and less than 1.
>
> ERRORS:
>
> - *DOMAIN*
>   - if the value of `<Value>` less than or equal to -1.
>   - if the value of `<Value>` greater than or equal to -1.

> `artanh( <Value:Imag> ) -> Imag`
>
> Returns the inverse hyperbolic tangent of `<Value>`.

> `artan( <Value:List> ) -> List`
>
> Returns a list formed from the elements after applying the specified operation.

3.7.8. Complex numbers related functions

conj

> `conj( <Value:Dec> ) -> Dec`
>
> Returns the number `<Value>`.

> `conj( <Value:Imag> ) -> Imag`
>
> Returns the complex conjugate of `<Value>`. Equivalent to the expression `real( <Value> ) - imag( <Value> )`.

> `conj( <Value:List> ) -> List`
>
> Returns a list formed from the elements after applying the specified operation.

> `conj( <Value:Matr> ) -> Matr`
>
> Returns a matrix formed from the elements after applying the specified operation.

real, Re

Synonym: `Re(`...

> `real(` `<Value:Dec>` `)` -> *Dec*
>
> Returns the number `<Value>`.

> `real(` `<Value:Imag>` `)` -> *Dec*
>
> Returns the real part of the complex number `<Value>`.

> `real(` `<Value:List>` `)` -> *List*
>
> Returns a list composed of elements to which the specified operation was applied.

> `real(` `<Value:Matr>` `)` -> *Matr*
>
> Returns a matrix composed of elements to which the specified operation was applied.

imag, Im

Synonym: `Im(`...

> `imag(` `<Value:Dec>` `)` -> *Dec*
>
> Returns 0.

> `imag(` `<Value:Imag>` `)` -> *Dec*
>
> Returns the imaginary part of the complex number `<Value>`.

> `imag(` `<Value:List>` `)` -> *List*
>
> Returns a list composed of elements to which the specified operation was applied.

> `imag(` `<Value:Matr>` `)` -> *Matr*
>
> Returns a matrix composed of elements to which the specified operation was applied.

angle, Arg

Synonym: `Arg(`...

> `angle(` `<Value:`*`Imag`*`>` `)` -> *`Imag`*
>
> Returns the angle (also known as the polar angle) between the radius vector of the corresponding point and the positive real axis. If `<Value>` is 0, it **returns 0** (TI-84 compatibility). The result depends on the current mode. In **Radian** mode, the function will return the angle in radians. In **Degree** mode, the function will return the angle in degrees.

> `angle(` `<Value:`*`List`*`>` `)` -> *`List`*
>
> Returns a list composed of elements to which the specified operation was applied.

cmplx_polar

> `cmplx_polar(` `<Value:`*`Imag`*`>` `)` -> *`Imag`*
>
> Converts the algebraic representation of the complex number `<Value>` to the exponential form `z=re^(iφ)`, where the value of φ depends on the current angle mode (**Radian** or **Degree**). Equivalent to conversion.

> `cmplx_polar(` `<Value:`*`List`*`>` `)` -> *`List`*
>
> Returns a list composed of elements to which the specified operation was applied.

> `cmplx_polar(` `<Value:`*`Matr`*`>` `)` -> *`Matr`*
>
> Returns a matrix composed of elements to which the specified operation was applied.

3.7.9. Probability functions

rand

> `rand(` `)` -> *`Dec`*
>
> Returns a pseudo-random decimal number in the range from 0 (inclusive) to 1 (exclusive), generated using the Mersenne Twister (MT19937). Calling the function initializes the PRNG (sets the seed) based on the number of milliseconds elapsed since the calculator was powered on.

> `rand(` `<Count:`*`Dec`*`>` `)` -> *`List`*

Returns a list composed of values generated by the `rand(` function. Unlike multiple invocations of `rand(`, the PRNG is initialized **only once**. The length of the resulting list is equal to `<Count>`.

ERRORS:

- *DATA TYPE* if `<Count>` is not an integer.
- *DOMAIN* if `<Count>` is less than 1.

randInt

`randInt(` `<Lower:Dec>`, `<Upper:Dec>` `)` -> *Dec*

Returns a pseudo-random integer in the range from `<Lower>` (inclusive) to `<Upper>` (inclusive), generated using the Mersenne Twister (MT19937). The function automatically swaps `<Upper>` and `<Lower>` if `<Upper>` is less than `<Lower>`. Calling the function initializes the PRNG (sets the seed) based on the number of milliseconds elapsed since the calculator was powered on.

ERRORS:

- *DOMAIN* if `<Lower>` or `<Upper>` is not an integer.

`randInt(` `<Lower:Dec>`, `<Upper:Dec>`, `<Count:Dec>` `)` -> *List*

Returns a list composed of values generated by the function `randInt(` `<Lower>`, `<Upper>`. Unlike multiple invocations of `randInt(`, the PRNG is initialized **only once**. The length of the resulting list is equal to `<Count>`.

ERRORS:

- *DOMAIN*
  - if `<Count>` is not an integer.
  - if `<Count>` is less than 1.

randIntNoRep

`randIntNoRep(` `<Lower:Dec>`, `<Upper:Dec>` `)` -> *Dec*

The function is equivalent to `randInt(` `<Lower>`, `<Upper>`.

`randIntNoRep(` `<Lower:Dec>`, `<Upper:Dec>`, `<Count:Dec>` `)` -> *List*

Returns a list of non-repeating values generated by the `randInt(` `<Lower>`, `<Upper>`. Unlike multiple calls to the `randInt(` function, in this case the PRNG is initialized **only once**. The resulting list length is equal to `<Count>`. The function only works if the difference between `<Upper>` and `<Lower>` (the function automatically swaps `<Upper>` and `<Lower>` if `<Upper>` is less than `<Lower>`) is greater than or equal to `<Count>` `- 1`.

ERRORS:

- *DOMAIN*
  - if the `<Count>` value is not an integer.
  - if the `<Count>` value is less than 1.
  - if the difference between `<Upper>` and `<Lower>` is less than `<Count> - 1`.

randBin

`randBin( <Trials count:Dec>, <Probability:Dec> ) -> Dec`

Function returns a random integer from a specified Binomial distribution. The value of `<Trials count>` must be an integer greater than or equal to one. The value of `<Probability>` (probability of success) must be in the range from 0 (inclusive) to 1 (inclusive).

`randBin( <Trials count:Dec>, <Probability:Dec>, <Count:Dec> ) -> List`

Returns a list of elements generated by the function `randBin( <Trials count>, <Probability>`. Unlike multiple calls to the `randBin(` function, the PRNG is initialized **only once**. The resulting list length is equal to `<Count>`.

ERRORS:

- *DOMAIN*
  - if the `<Trials count>` value is not an integer.
  - if the `<Trials count>` value is less than 1.
  - if the `<Probability>` value is less than 0 or greater than 1.
  - if the `<Count>` value is not an integer.
  - if the `<Count>` value is less than 1.

randNorm

`randNorm( <Mu:Dec>, <Sigma:Dec> ) -> Dec`

Function random real number from a specified Normal distribution. Each generated value could be any real number, but most will be within the interval [`<Mu> - 3( <Sigma> )`, `<Mu> + 3( <Sigma> )`].

`randNorm( <Mu:Dec>, <Sigma:Dec>, <Count:Dec> ) -> List`

Returns a list formed from the elements generated by the function `randNorm( <Mu>, <Sigma>`. Unlike calling the function `randNorm(` multiple times, in this case the RNG is **initialized once**. The length of the resulting list is equal to `<Count>`.

ERRORS:

- *DOMAIN*
  - if the value of `<Count>` is not an integer.
  - if the value of `<Count>` is less than 1.

nCr

`nCr( <N:Dec>, <R:Dec> ) -> Dec`

`nCr(` returns the number of combinations of `<N>` taken `<R>` at a time. `<N>` and `<R>` must be nonnegative integers.

ERRORS:

- *DOMAIN*
  - if the value of `<N>` or `<R>` is not an integer.
  - if the value of `<N>` or `<R>` is less than 0.
  - if the value of `<N>` is less than `<R>`.

`nCr( <N:Dec>, <Rs:List> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the number `<N>` and the corresponding element of the list `<Rs>`.

ERRORS:

- *DATA TYPE* if the list `<Rs>` contains a complex number.

`nCr( <Ns:List>, <R:Dec> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the corresponding element of the list `<Ns>` and the number `<R>`.

ERRORS:

- *DATA TYPE* if the list `<Ns>` contains a complex number.

`nCr( <Ns:List>, <Rs:List> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the corresponding elements of the lists `<Ns>` and `<Rs>`.

ERRORS:

- *DATA TYPE*
  - if the list `<Ns>` contains a complex number.
  - if the list `<Rs>` contains a complex number.
- *DIM MISMATCH* if the lengths of the lists `<Ns>` and `<Rs>` are different.

nPr

`nPr( <N:Dec>, <R:Dec> ) -> Dec`

`nPr(` returns the number of permutations of `<N>` taken `<R>` at a time. `<N>` and `<R>` must be nonnegative integers.

ERRORS:

- *DOMAIN*
  - if the value of `<N>` or `<R>` is not an integer.
  - if the value of `<N>` or `<R>` is less than 0.
  - if the value of `<N>` is less than `<R>`.

`nPr( <N:Dec>, <Rs:List> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the number `<N>` and the corresponding element of the list `<Rs>`.

ERRORS:

- *DATA TYPE* if the list `<Rs>` contains a complex number.

`nPr( <Ns:List>, <R:Dec> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the corresponding element of the list `<Ns>` and the number `<R>`.

ERRORS:

- *DATA TYPE* if the list `<Ns>` contains a complex number.

`nPr( <Ns:List>, <Rs:List> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the corresponding elements of the lists `<Ns>` and `<Rs>`.

> ERRORS:
>
> - *DATA TYPE*
>   - if the list `<Ns>` contains a complex number.
>   - if the list `<Rs>` contains a complex number.
> - *DIM MISMATCH* if the lengths of the lists `<Ns>` and `<Rs>` are different.

3.7.10. Coordinate conversion functions

These functions take two arguments. If the function has the following syntax: `func( <A:Dec>, <B:List> ) -> List`, then the function returns a list composed of the results of sequentially applying the specified operation between the number `<A>` and the corresponding element of the list `<B>`.

If the function has the following syntax: `func( <A:List>, <B:Dec> ) -> List`, then the function returns a list composed of the results of sequentially applying the specified operation between each element of the list `<A>` and the number `<B>`.

If the function has the following syntax: `func( <A:List>, <B:List> ) -> List`, then the function returns a list composed of the results of sequentially applying the specified operation between corresponding elements of the lists `<A>` and `<B>`.

The result of these functions depends on the current angle unit mode, as the calculations use trigonometric functions.

ERRORS:

- *DATA TYPE* if the list `<A>` or `<B>` contains a complex number.
- *DIM MISMATCH* if the lengths of the lists `<A>` and `<B>` differ.

P▶Rx

> (P\x9ERx) `P▶Rx( <r:Dec|List>, <θ:Dec|List> ) -> Dec|List`
>
> Converts polar coordinates to rectangular coordinates and returns the `X` coordinate. Equivalent to the expression `<r> * cos( <θ> )`.

P▶Ry

> (P\x9ERy) `P▶Ry( <r:Dec|List>, <θ:Dec|List> ) -> Dec|List`
>
> Converts polar coordinates to rectangular coordinates and returns the `Y` coordinate. Equivalent to the expression `<r> * sin( <θ> )`.

R▶Pr

> (R\x9EPr) `R▶Pr( <X:Dec|List>, <Y:Dec|List> ) -> Dec|List`

> Converts rectangular coordinates to polar coordinates and returns the vector magnitude `r`. Equivalent to the expression `sqrt( <X> ^2 + <Y> ^2)`.

R▶Pθ

> (R\x9EP\x99) `R▶Pθ( <X:Dec|List>, <Y:Dec|List> )` -> *Dec|List*
>
> Converts rectangular coordinates to polar coordinates and returns the angular component `θ` of the vector. Equivalent to the expression `atan( <Y> / <X> )`. For the point with coordinates (0, 0), the angular component `θ` is 0.

3.7.11. Lists (and matrices) related functions

dim

> `dim( <Value:List> )` -> *Dec*
>
> Returns the length of the passed list `<Value>`.

> `dim( <Value:Matr> )` -> *List*
>
> Returns a list of two elements. The first element is the height of the matrix `<Value>`. The second element is the width of the matrix `<Value>`.

seq

> `seq( <Func:Func>, <Ident Var:Dec>, <From:Dec>, <To:Dec> )` -> *List*
>
> `seq( <Func:Func>, <Ident Var:Dec>, <From:Dec>, <To:Dec>, <Step:Dec> )` -> *List*
>
> Returns a list in which each element is the result of evaluating `<Func>` with respect to `<Ident Var>` for values ranging from `<From>` to `<To>` with a step of `<Step>`. By default, `<Step>` is 1. `<Step>` can be a negative number, in which case `<From>` must be greater than `<To>`. `<Func>` can be an expression (`1+X^2`) or a Y-function variable (`Y₁`).
>
> ERRORS:
>
> - Various calculation errors related to value evaluation.
> - *INCREMENT*
>   - if the value of `<Step>` is 0.
>   - if the value of `<Step>` is greater than 0 and the value of `<From>` is greater than the value of `<To>`.
>   - if the value of `<Step>` is less than 0 and the value of `<From>` is less than the value of `<To>`.

cumSum

> `cumSum( <Value:List> )` -> *List*
>
> Returns a list composed of elements that are the sum of all previous elements and the current element of the list `<Value>`.

> `cumSum( <Value:Matr> )` -> *Matr*
>
> Returns a new matrix with the same dimensions as the matrix `<Value>`. The function transforms each column from top to bottom separately. Each column is sequentially composed of elements that are the sum of all previous elements and the current element of the corresponding column in the matrix `<Value>`.

ΔList

> (\x9AList) `ΔList( <Value:List> )` -> *List*
>
> Returns a list containing the differences between consecutive elements in `<Value>`. `ΔList(` subtracts the first element in `<Value>` from the second element, subtracts the second element from the third, and so on. The result list of differences is always one element shorter than the original list `<Value>`.
>
> ERRORS:
>
> - *INVALID DIMENSION* if the list `<Value>` contains fewer than two elements.

> (\x9AList) `ΔList( <Value:Matr> )` -> *Matr*
>
> Returns a new matrix with the same height as the matrix `<Value>`. The width of the new matrix is one element less. The rows of the new matrix are the result of transforming the corresponding rows of the matrix `<Value>` using the `ΔList(` function.
>
> ERRORS:
>
> - *INVALID DIMENSION* if the width of the matrix `<Value>` is less than two.

augment

> `augment( <A:List>, <B:List> )` -> *List*
>
> Returns a list formed by the sequence of elements of list `<A>`, followed by the sequence of elements of list `<B>`. (concatenates lists `<A>` and `<B>`).

> `augment( <A:Matr>, <B:Matr> )` -> *Matr*

Returns a matrix whose rows are formed by the sequence of elements of the corresponding rows of matrix `<A>`, followed by the sequence of elements of the corresponding rows of matrix `<B>`. (concatenates matrices `<A>` and `<B>` horizontally).

ERRORS:

- *DIM MISMATCH* if the heights of matrices `<A>` and `<B>` are not equal.

mean

`mean( <Value:List> ) -> Dec`

Returns the arithmetic mean of the elements in the `<Value>` list. The elements must be of type *Dec*.

ERRORS:

- *DATA TYPE* if the `<Value>` list contains a complex number.

`mean( <Value:List>, <Freq:List> ) -> Dec`

Returns the arithmetic mean of the elements in the `<Value>` list, weighted by the frequencies specified in the `<Freq>` list. Each element in the `<Value>` list corresponds to a number in the `<Freq>` list indicating the number of occurrences of the corresponding element. Equivalent to the sum of the products of the corresponding elements of the `<Value>` and `<Freq>` lists, divided by the sum of elements in the `<Freq>` list.

ERRORS:

- *DATA TYPE* if the `<Value>` or `<Freq>` list contains a complex number.
- *DIM MISMATCH* if the lengths of the `<Value>` and `<Freq>` lists are not equal.
- *STATISTICAL* if the `<Freq>` list contains a negative number.

median

`median( <Value:List> ) -> Dec`

Returns the median value of the elements in the `<Value>` list. The elements must be of type *Dec*.

ERRORS:

- *DATA TYPE* if the `<Value>` list contains a complex number.

`median( <Value:List>, <Freq:List> ) -> Dec`

Returns the median value of the elements in the `<Value>` list, weighted by the frequencies specified in the `<Freq>` list. Each element in the `<Value>` list corresponds to a number in the `<Freq>` list indicating the number of occurrences of the corresponding element.

ERRORS:

- *DATA TYPE* if the `<Value>` or `<Freq>` list contains a complex number.
- *DIM MISMATCH* if the lengths of the `<Value>` and `<Freq>` lists are not equal.
- *DOMAIN*
  - if the `<Freq>` list contains a non-integer element.
  - if the `<Freq>` list contains a negative number.

variance

`variance( <Value:List> )` -> *Dec*

Returns the variance of the elements in `<Value>`.

ERRORS:

- *DATA TYPE* if the list `<Value>` contains a complex number.

stdDev

`stdDev( <Value:List> )` -> *Dec*

Returns standard deviation of the elements in `<Value>`.

ERRORS:

- *DATA TYPE* if the list `<Value>` contains a complex number.

sum

`sum( <Value:List> )` -> *Dec|Imag*

`sum( <Value:List>, <From:Dec> )` -> *Dec|Imag*

`sum( <Value:List>, <From:Dec>, <To:Dec> )` -> *Dec|Imag*

Returns the sum of the elements of the list `<Value>`, starting from the element number `<From>` and ending with the element number `<To>`. Element numbering starts at 1. By default, `<From>` is 1, and `<To>` is the length of the list `<Value>`.

ERRORS:

- *INVALID DIMENSION* if the list `<Value>` has zero length.
- *DOMAIN*
  - if the value of `<From>` or `<To>` is not an integer.
  - if the value of `<From>` or `<To>` is less than 1.

prod

`prod( <Value:List> )` -> *Dec|Imag*

> `prod( <Value:List>, <From:Dec> )` -> *Dec|Imag*
>
> `prod( <Value:List>, <From:Dec>, <To:Dec> )` -> *Dec|Imag*
>
> Returns the product of the elements of the list `<Value>`, starting from the element number `<From>` and ending with the element number `<To>`. Element numbering starts at 1. By default, `<From>` is 1, and `<To>` is the length of the list `<Value>`.
>
> ERRORS:
>
> - *INVALID DIMENSION* if the list `<Value>` has zero length.
> - *DOMAIN*
>   - if the value of `<From>` or `<To>` is not an integer.
>   - if the value of `<From>` or `<To>` is less than 1.

3.7.12. Matrices related functions

det

> `det( <Value:Matr> )` -> *Dec*
>
> Returns the determinant of a square matrix `<Value>`.
>
> ERRORS:
>
> - *INVALID DIMENSION* if the matrix `<Value>` is not square.

transpose

> `transpose( <Value:Matr> )` -> *Matr*
>
> Returns a matrix in which each element (row, column) is swapped with the corresponding element (column, row) of `<Value>`.

identity

> `identity( <Dimension:Dec> )` -> *Matr*
>
> Returns the identity matrix of `<Dimension>` rows x `<Dimension>` columns.
>
> ERRORS:
>
> - *INVALID DIMENSION*
>   - if the value `<Dimension>` is not an integer.
>   - if the value `<Dimension>` is less than 1.

inverse

> `inverse( <Value:Matr> )` -> *Matr*

> Returns the inverse matrix for the matrix `<Value>`. Equivalent to the expression `<Value>^-1`.
>
> ERRORS:
>
> - *SINGULAR MATR* if the matrix `<Value>` is singular.

randM

> `randM( <Rows:Dec>, <Columns:Dec> ) -> Matr`
>
> Returns a matrix with `<Rows>` rows and `<Columns>` columns, filled with random integers from -9 (inclusive) to 9 (inclusive).
>
> ERRORS:
>
> - *INVALID DIMENSION*
>   - if the value `<Rows>` or if the value `<Columns>` is not an integer.
>   - if the value `<Rows>` is less than 1.
>   - if the value `<Columns>` is less than 1.

ref

> `ref( <Value:Matr> ) -> Matr`
>
> Returns the row-echelon form of a real matrix `<Value>`.
>
> ERRORS:
>
> - *DATA TYPE* if the matrix `<Value>` contains a complex number.
> - *INVALID DIMENSION* if the height of the matrix `<Value>` is greater than its width.

rref

> `rref( <Value:Matr> ) -> Matr`
>
> Returns the reduced row-echelon form of a real matrix `<Value>`.
>
> ERRORS:
>
> - *DATA TYPE* if the matrix `<Value>` contains a complex number.
> - *INVALID DIMENSION* if the height of the matrix `<Value>` is greater than its width.

rowSwap

> `rowSwap( <Value:Matr>, <Row 1:Dec>, <Row 2:Dec> ) -> Matr`
>
> Returns a matrix `<Value>`, where rows numbered `<Row 1>` and `<Row 2>` are swapped.
>
> ERRORS:
>
> - *DATA TYPE* if the value of `<Row 1>` or `<Row 2>` is not an integer.

- *INVALID DIMENSION*
  - if the value of `<Row 1>` or `<Row 2>` is less than 1.
  - if the value of `<Row 1>` or `<Row 2>` exceeds the height of the matrix `<Value>`.

row+

(row\xCE) `row+( <Value:Matr> , <Additional row:Dec> , <Row:Dec> ) -> Matr`

Returns a matrix `<Value>`, where the elements of the row numbered `<Row>` are added to the corresponding elements of the row numbered `<Additional row>`.

ERRORS:

- *DATA TYPE* if the value of `<Additional row>` or `<Row>` is not an integer.
- *INVALID DIMENSION*
  - if the value of `<Additional row>` or `<Row>` is less than 1.
  - if the value of `<Additional row>` or `<Row>` exceeds the height of the matrix `<Value>`.

***row**

(\xCDrow) `*row( <Coefficient:Dec|Imag> , <Value:Matr> , <Row:Dec> ) -> Matr`

Returns a matrix `<Value>`, where the elements of the row numbered `<Row>` are multiplied by the value `<Coefficient>`.

ERRORS:

- *DATA TYPE* if the value of `<Row>` is not an integer.
- *INVALID DIMENSION*
  - if the value of `<Row>` is less than 1.
  - if the value of `<Row>` exceeds the height of the matrix `<Value>`.

***row+**

(\xCDrow\xCE) `*row+( <Coefficient:Dec|Imag> , <Value:Matr> , <Row 1:Dec> , <Row 2:Dec> ) -> Matr`

Returns a matrix `<Value>`, where the elements of the row numbered `<Row 2>` are added to the corresponding elements of the row numbered `<Row 1>`, multiplied by the value `<Coefficient>`.

ERRORS:

- *DATA TYPE* if the value of `<Row 1>` or `<Row 2>` is not an integer.
- *INVALID DIMENSION*
  - if the value of `<Row 1>` or `<Row 2>` is less than 1.
  - if the value of `<Row 1>` or `<Row 2>` exceeds the height of the matrix `<Value>`.

3.7.13. Distribution functions

normalpdf

> `normalpdf( <X:Dec> )` -> *Dec*
>
> `normalpdf( <X:Dec>, <Mu:Dec> )` -> *Dec*
>
> `normalpdf( <X:Dec>, <Mu:Dec>, <Sigma:Dec> )` -> *Dec*
>
> `normalpdf(` computes the probability density function (pdf) for the normal distribution by mean `<Mu>` and standard deviation `<Sigma>` at a specified `<X>` value. The defaults are `<Mu>` = 0, `<Sigma>` = 1. The probability density function (pdf) is:
>
> $$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$
>
> ERRORS:
>
> - *DOMAIN* if the value of `<Sigma>` is less than or equal to 0.

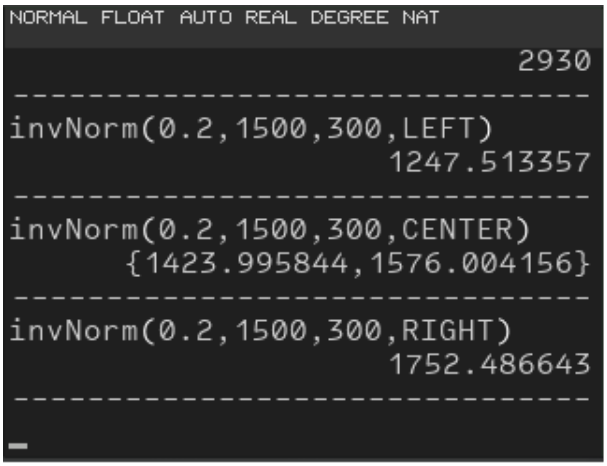
normalcdf

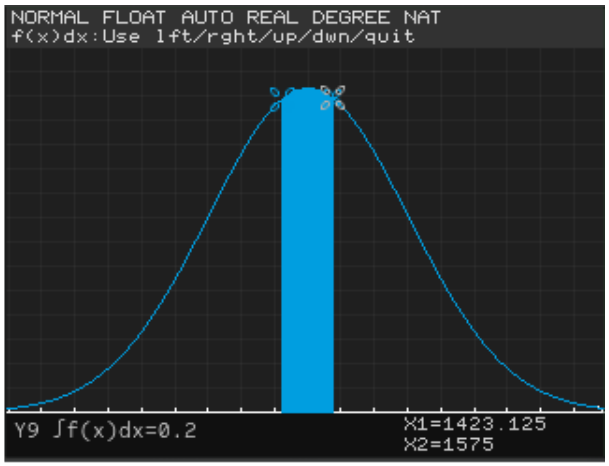
> `normalcdf( <Lower:Dec>, <Upper:Dec> )` -> *Dec*
>
> `normalcdf( <Lower:Dec>, <Upper:Dec>, <Mu:Dec> )` -> *Dec*
>
> `normalcdf( <Lower:Dec>, <Upper:Dec>, <Mu:Dec>, <Sigma:Dec> )` -> *Dec*
>
> `normalcdf(` computes the normal distribution probability between `<Lower>` and `<Upper>` for the specified mean `<Mu>` and standard deviation `<Sigma>`. The defaults are `<Mu>` = 0, `<Sigma>` = 1.
>
> ERRORS:
>
> - *DOMAIN* if the value of `<Sigma>` is less than or equal to 0.

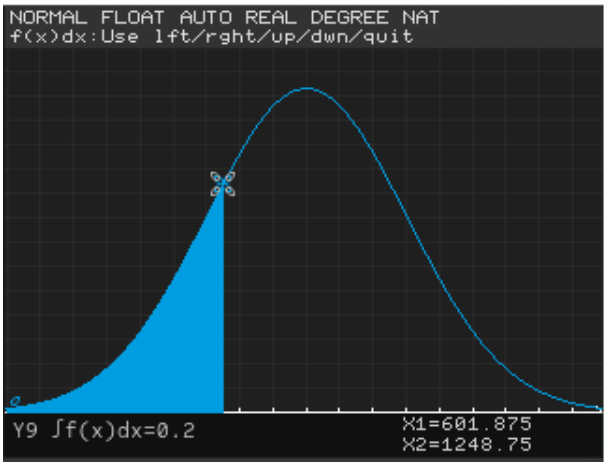
invNorm

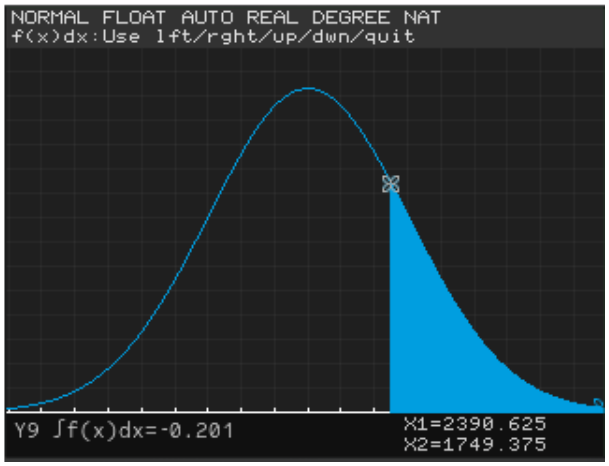
> `invNorm( <Area:Dec> )` -> *Dec*
>
> `invNorm( <Area:Dec>, <Tail:Dec> )` -> *Dec|List*
>
> `invNorm( <Area:Dec>, <Mu:Dec>, <Sigma:Dec> )` -> *Dec*
>
> `invNorm( <Area:Dec>, <Mu:Dec>, <Sigma:Dec>, <Tail:Dec> )` -> *Dec|List*

`invNorm(` computes the inverse cumulative normal distribution function for a given `<Area>` under the normal distribution curve specified by mean `<Mu>` and standard deviation `<Sigma>`. The value of `<Tail>` determines the region used for the calculation: 1 (`LEFT`) - the left area 2 (`CENTER`) - the central area 3 (`RIGHT`) - the right area. If `<Tail>` is 2, the function will return a list consisting of two numbers. The first number is the left boundary of the area, and the second is the right boundary of the area. The defaults are `<Mu>` = 0, `<Sigma>` = 1, `<Tail>` = 1 (`LEFT`).









ERRORS:

- *DOMAIN*
  - if the value of `<Sigma>` is less than or equal to 0.
  - if the value of `<Area>` is less than 0 or greater than 1.

invT

`invT(` `<Area:Dec>` , `<df:Dec>` `)` -> *Dec*

`invT(` computes the inverse cumulative Student-t probability function specified by Degree of Freedom, `<df>` for a given `<Area>` under the curve. *WRONG WORKING*.

ERRORS:

- *DOMAIN*
  - if the value of `<df>` is less than or equal to 0.
  - if the value of `<Area>` is less than 0 or greater than 1.

tpdf

> `tpdf(` `<X:Dec>` , `<df:Dec>` `)` -> *Dec*
>
> `tpdf(` computes the probability density function (pdf) for the Student-t distribution at a specified `<X>` value. `<df>` is degrees of freedom.
>
> ERRORS:
>
> - *DOMAIN* if the value of `<df>` is less than or equal to 0.

tcdf

> `tcdf(` `<Lower:Dec>` , `<Upper:Dec>` , `<df:Dec>` `)` -> *Dec*
>
> `tcdf(` computes the Student-t distribution probability between `<Lower>` and `<Upper>` for the specified `<df>` (degrees of freedom).
>
> ERRORS:
>
> - *DOMAIN* if the value of `<df>` is less than or equal to 0.

χ²pdf, pdftw

Synonym: `pdftw(` ...

> (\xD8\xBDpdf) `χ²pdf(` `<X:Dec>` , `<df:Dec>` `)` -> *Dec*
>
> `χ²pdf(` computes the probability density function (pdf) for the χ² (chi-square) distribution at a specified `<X>` value. `<df>` (degrees of freedom) must be an integer > 0.
>
> ERRORS:
>
> - *DOMAIN*
> - if the value of `<df>` is less than or equal to 0.
> - if the value of `<df>` is not an integer.

> (\xD8\xBDpdf) `χ²pdf(` `<X:Dec>` , `<df:List>` `)` -> *List*
>
> Returns a list formed from the results of sequential operations between the number `<X>` and the corresponding element of the list `<df>` .
>
> ERRORS:
>
> - *DATA TYPE* if the list `<df>` contains a complex number.

> (\xD8\xBDpdf) `χ²pdf(` `<X:List>` , `<df:Dec>` `)` -> *List*

> Returns a list formed from the results of sequential operations between the number `<df>` and the corresponding element of the list `<X>`.
>
> ERRORS:
>
> - *DATA TYPE* if the list `<X>` contains a complex number.

> (\xD8\xBDpdf) `χ²pdf(` `<X:List>`, `<df:List>` `)` `-> List`
>
> Returns a list formed from the results of sequential operations between the corresponding elements of the lists `<X>` and `<df>`.
>
> ERRORS:
>
> - *DATA TYPE*
>   - if the list `<X>` contains a complex number.
>   - if the list `<df>` contains a complex number.
> - *DIM MISMATCH* if the lengths of the lists `<X>` and `<df>` are different.

χ²cdf, cdftw

Synonym: `cdftw(`...

> (\xD8\xBDcdf) `χ²cdf(` `<Lower:Dec>`, `<Upper:Dec>`, `<df:Dec>` `)` `-> Dec`
>
> `χ²cdf(` computes the χ^2 (chi-square) distribution probability between `<Lower>` and `<Upper>` for the specified `<df>` (degrees of freedom).
>
> ERRORS:
>
> - *DOMAIN*
>   - if the value of `<df>` is less than or equal to 0.
>   - if the value of `<df>` is not an integer.

Fpdf

> (\xD9pdf) `Fpdf(` `<X:Dec>`, `<Numerator df:Dec>`, `<Denominator df:Dec>` `)` `-> Dec`
>
> `Fpdf(` computes the probability density function (pdf) for the F distribution at a specified `<X>` value. `<Numerator df>` (degrees of freedom) and `<Denominator df>` must be integers > 0.
>
> ERRORS:
>
> - *DOMAIN*
>   - if the value of `<Numerator df>` or `<Denominator df>` is less than or equal to 0.
>   - if the value of `<df>` or `<Denominator df>` is not an integer.

Fcdf

(\xD9cdf) `Fcdf( <Lower:Dec>, <Upper:Dec>, <Numerator df:Dec>, <Denominator df:Dec> ) -> Dec`

`Fcdf(` computes the F distribution probability between `<Lower>` and `<Upper>` for the specified `<Numerator df>` (degrees of freedom) and `<Denominator df>`. `<Numerator df>` and `<Denominator df>` must be integers > 0.

ERRORS:

- *DOMAIN*
  - if the value of `<Numerator df>` or `<Denominator df>` is less than or equal to 0.
  - if the value of `<df>` or `<Denominator df>` is not an integer.

binompdf

`binompdf( <Trials count:Dec>, <Probability:Dec> ) -> List`

`binompdf(` computes a probability at `X` (list of probabilities from 0 to `<Trials count>`) for the discrete binomial distribution with the specified `<Trials count>` and `<Probability>` of success (p) on each trial. The probability density function (pdf) is:

$$f(x) = \binom{n}{r} p^{x} (1-p)^{n-x}, \quad x = 0, 1, \ldots, n$$

ERRORS:

- *DOMAIN*
  - if the value of `<Trials count>` is not an integer.
  - if the value of `<Trials count>` is less than 1.
  - if the value of `<Probability>` is less than 0 or greater than 1.

`binompdf( <Trials count:Dec>, <Probability:Dec>, <X:Dec> ) -> Dec`

`binompdf(` computes a probability at `<X>` for the discrete binomial distribution with the specified `<Trials count>` and `<Probability>` of success (p) on each trial. If value of `<X>` is less than 0 or more than `<Trials count>`, the function returns 0.

ERRORS:

- *DOMAIN*
  - if the value of `<Trials count>` is not an integer.
  - if the value of `<Trials count>` is less than 1.
  - if the value of `<Probability>` is less than 0 or greater than 1.
  - if the value of `<X>` is not an integer.

`binompdf(` `<Trials count:Dec>`, `<Probability:Dec>`, `<X:List>` `)` -> *List*

`binompdf(` computes a probability at each element from `<X>` list for the discrete binomial distribution with the specified `<Trials count>` and `<Probability>` of success (p) on each trial. If the value of an element from the `<X>` list is less than 0 or more than `<Trials count>`, the function returns 0.

ERRORS:

- *DATA TYPE* if the list `<X>` contains a complex number.
- *DOMAIN*
  - if the value of `<Trials count>` is not an integer.
  - if the value of `<Trials count>` is less than 1.
  - if the value of `<Probability>` is less than 0 or greater than 1.
  - if the list `<X>` contains a non-integer element.

binomcdf

`binomcdf(` `<Trials count:Dec>`, `<Probability:Dec>` `)` -> *List*

`binomcdf(` computes a cumulative probability at `X` (list of probabilities from 0 to `<Trials count>`) for the discrete binomial distribution with the specified `<Trials count>` and `<Probability>` of success (p) on each trial.

ERRORS:

- *DOMAIN*
  - if the value of `<Trials count>` is not an integer.
  - if the value of `<Trials count>` is less than 1.
  - if the value of `<Probability>` is less than 0 or greater than 1.

`binomcdf(` `<Trials count:Dec>`, `<Probability:Dec>`, `<X:Dec>` `)` -> *Dec*

`binomcdf(` computes a cumulative probability at `<X>` for the discrete binomial distribution with the specified `<Trials count>` and `<Probability>` of success (p) on each trial. If the value of `<X>` is less than 0, the function returns 0. If the value of `<X>` is greater than `<Trials count>`, the function returns 1.

ERRORS:

- *DOMAIN*
  - if the value of `<Trials count>` is not an integer.
  - if the value of `<Trials count>` is less than 1.
  - if the value of `<Probability>` is less than 0 or greater than 1.
  - if the value of `<X>` is not an integer.

`binomcdf( <Trials count:Dec>, <Probability:Dec>, <X:List> ) -> List`

`binomcdf(` computes a cumulative probability at each element from the `<X>` list for the discrete binomial distribution with the specified `<Trials count>` and `<Probability>` of success (p) on each trial. If the value of `<X>` is less than 0, the function returns 0. If the value of `<X>` is greater than `<Trials count>`, the function returns 1.

ERRORS:

- *DATA TYPE* if the list `<X>` contains a complex number.
- *DOMAIN*
  - if the value of `<Trials count>` is not an integer.
  - if the value of `<Trials count>` is less than 1.
  - if the value of `<Probability>` is less than 0 or greater than 1.
  - if the list `<X>` contains a non-integer element.

invBinom

`invBinom( <Area:Dec>, <Trials count:Dec>, <Probability:Dec> ) -> Dec`

`invBinom(` is the inverse binomial distribution function with the specified `<Trials count>` and `<Probability>` of success (p) on each trial. It returns the smallest number of successful trials for which the given probability `<Area>` in the binomial distribution is reached or exceeded.

ERRORS:

- *DOMAIN*
  - if the value of `<Trials count>` is not an integer.
  - if the value of `<Trials count>` is less than 1.
  - if the value of `<Probability>` is less than 0 or greater than 1.
  - if the value of `<Area>` is less than 0 or greater than 1.

poissonpdf

`poissonpdf( <Mu:Dec>, <X:Dec|List> ) -> Dec|List`

`poissonpdf(` computes a probability at `<X>` for the discrete Poisson distribution with the specified mean `<Mu>`. If `<X>` is list, function return list of results. Value of `<X>` (or elements) should be integer. If value of `<X>` (or elements) less than 0, function return 0. The probability density function (pdf) is:

$$f(x) = \frac{e^{-\mu}\,\mu^{x}}{x!}$$

ERRORS:

- *DATA TYPE* if the list `<X>` contains a complex number.

- *DOMAIN*
  - if the value or element of the list `<X>` is not an integer.
  - if the value of `<Mu>` is less than or equal to 0.

poissoncdf

`poissoncdf(` `<Mu:`*Dec*`>`, `<X:`*Dec*`|`*List*`>` `)` -> *Dec*`|`*List*

`poissoncdf(` computes a cumulative probability at `<X>` for the discrete Poisson distribution with the specified mean `<Mu>`. If `<X>` is list, function return list of results. If value of `<X>` (or elements) less than 0, function return 0. Note: this function can take the fractional value of the argument `<X>` (or elements), but calculation is performed only for integer part of value (TI-84 compatibility).

ERRORS:

- *DATA TYPE* if the list `<X>` contains a complex number.
- *DOMAIN* if the value of `<Mu>` is less than or equal to 0.

geometpdf

`geometpdf(` `<Probability:`*Dec*`>`, `<X:`*Dec*`|`*List*`>` `)` -> *Dec*`|`*List*

`geometpdf(` computes a probability at `<X>`, the number of the trial on which the first success occurs, for the discrete geometric distribution with the specified `<Probability>` of success. If `<X>` is list, function return list of results. Value of `<X>` (or elements) should be integer. If value of `<X>` (or elements) less than 1, function return 0. The probability density function (pdf) is:

$$f(x) = p(1-p)^{x-1}$$

ERRORS:

- *DATA TYPE* if the list `<X>` contains a complex number.
- *DOMAIN*
  - if the value or an element of the list `<X>` is not an integer.
  - if the value of `<Probability>` is less than 0 or greater than 1.

geometcdf

`geometcdf(` `<Probability:`*Dec*`>`, `<X:`*Dec*`|`*List*`>` `)` -> *Dec*`|`*List*

`geometcdf(` computes a cumulative probability at `<X>`, the number of the trial on which the first success occurs, for the discrete geometric distribution with the specified `<Probability>` of success. If `<X>` is list, function return list of results. Value of `<X>` (or elements) should be integer. If value of `<X>` (or elements) less than 1, function return 0.

ERRORS:

- *DATA TYPE* if the list `<X>` contains a complex number.
- *DOMAIN*
  - if the value or an element of the list `<X>` is not an integer.
  - if the value of `<Probability>` is less than 0 or greater than 1.

3.7.14. Time functions

startTmr

`startTmr( )` -> *Dec*

Returns the number of seconds elapsed since the device was powered on. This function is intended to be used together with the `checkTmr(` function.

checkTmr

`checkTmr( <Seconds:Dec> )` -> *Dec*

Returns the difference between the number of seconds elapsed since the device was powered on and the number `<Seconds>`. Used in combination with the `startTmr(` function to measure elapsed time.

ERRORS:

- *DOMAIN* if the value of `<Seconds>` is less than 0.

getTime

`getTime( )` -> *List*

Returns the current time in 24-hour format as a list, where the first number is the number of hours, the second is the number of minutes, and the third is the number of seconds. The current time can be set via the **MODE** screen in the **Set clock** menu or using the `setTime(` function.

setTime

`setTime( <Hours:Dec>, <Minutes:Dec>, <Seconds:Dec> )` -> *List*

Sets the current platform time in 24-hour format. On success, returns the current time as a list (same format as the `getTime(` function). (Some platforms may not allow changing the current time.)

ERRORS:

- *DOMAIN*
  - if the value of `<Hours>`, `<Minutes>`, or `<Seconds>` is not an integer.
  - if the value of `<Hours>` is less than 0 or greater than 23.
  - if the value of `<Minutes>` is less than 0 or greater than 59.
  - if the value of `<Seconds>` is less than 0 or greater than 59.

getTmFmt

> `getTmFmt( )` -> *Dec*
>
> Returns a number representing the current time format (12 or 24). The time format can be set in the **MODE** screen under the **Set clock** menu or via the `setTmFmt(` command.

getTmStr

> `getTmStr( )` -> *Str*
>
> `getTmStr( <Format:Dec> )` -> *Str*
>
> Returns a string containing the current time. If `<Format>` is 12, the resulting time format will be `09:00 PM`. If `<Format>` is 24, the resulting time format will be `21:00`. By default, the value of `<Format>` is the result of the `getTmFmt(` function.
>
> ERRORS:
>
> - *DOMAIN* if the value of `<Format>` is not 12 or 24.

getDate

> `getDate( )` -> *List*
>
> Returns the current date as a list, where the first number is the year, the second is the month, and the third is the day. The current date can be set via the **MODE** screen in the **Set clock** menu or using the `setDate(` function.

setDate

> `setDate( <Year:Dec>, <Month:Dec>, <Day:Dec> )` -> *List*
>
> Sets the current platform date. On success, returns the current date as a list (same format as the `getDate(` function) (some platforms may not allow changing the current date).
>
> ERRORS:
>
> - *DOMAIN*
>   - if the value of `<Year>`, `<Month>`, or `<Day>` is not an integer.
>   - if the value of `<Year>` is less than 2000 or greater than 2150.
>   - if the value of `<Month>` is less than 1 or greater than 12.
>   - if the value of `<Day>` is less than 1 or greater than 31.

getDtFmt

> `getDtFmt( )` -> *Dec*

Returns a number representing the current date format:

- `<Format>` = 1 - **M/D/Y**
- `<Format>` = 2 - **D/M/Y**
- `<Format>` = 3 - **Y/M/D**

The current date format can be set in the **MODE** screen under the **Set clock** menu or via the `setDtFmt(` command.

getDtStr

`getDtStr( )` -> *Str*

`getDtStr( <Format:Dec> )` -> *Str*

Returns a string containing the current date, formatted according to the specified format (all numbers in the string will be two digits).

- `<Format>` = 1 - **M/D/Y**
- `<Format>` = 2 - **D/M/Y**
- `<Format>` = 3 - **Y/M/D**

By default, the value of `<Format>` is the result of the `getDtFmt(` function.

ERRORS:

- *DOMAIN* if the value of `<Format>` is not 1, 2, or 3.

timeCnv

`timeCnv( <Seconds:Dec> )` -> *List*

Converts the number of seconds `<Seconds>` into a list, where the first number is the number of days, the second is the number of remaining hours, the third is the number of remaining minutes, and the fourth is the number of remaining seconds.

ERRORS:

- *DOMAIN* if the value of `<Seconds>` is not an integer.

dayOfWk

`dayOfWk( <Year:Dec>, <Month:Dec>, <Day:Dec> )` -> *Dec*

Returns the weekday of the specified date. Sunday is the first day of the week, Monday is the second.

ERRORS:

- *DOMAIN*
  - if the value of `<Year>`, `<Month>`, or `<Day>` is not an integer.
  - if the value of `<Year>` is less than 1900.

- if the value of `<Month>` is less than 1 or greater than 12.
- if the value of `<Day>` is less than 1 or greater than 31.
- if the specified date does not exist (e.g., February 30).

dbd

`dbd( <Date 1:Dec>, <Date 2:Dec> ) -> Dec`

Returns the difference in days between the dates `<Date 2>` and `<Date 1>`. Date formats are as follows: `MM.DDYY` or `DDMM.YY`. This format imposes restrictions on the years used (from 1901 to 2000). For example, 2.2889 represents February 28, 1989, and 3112 represents December 31, 2000.

ERRORS:

- *DOMAIN*
  - if the value of `<Date 1>` or `<Date 2>` is less than 0.
  - if the value of `<Date 1>` or `<Date 2>` cannot be converted into an existing date (between 1901 and 2000).

`dbd( <Date 1:Dec>, <Date 2:List> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the number `<Date 1>` and the corresponding element of the list `<Date 2>`.

ERRORS:

- *DATA TYPE* if the list `<Date 2>` contains a complex number.

`dbd( <Date 1:List>, <Date 2:Dec> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the corresponding element of the list `<Date 1>` and the number `<Date 2>`.

ERRORS:

- *DATA TYPE* if the list `<Date 1>` contains a complex number.

`dbd( <Date 1:List>, <Date 2:List> ) -> List`

Returns a list formed from the results of sequentially performing the specified operation between the corresponding elements of the lists `<Date 1>` and `<Date 2>`.

ERRORS:

- *DATA TYPE*
  - if the list `<Date 1>` contains a complex number.

    - if the list `<Date 2>` contains a complex number.
- *DIM MISMATCH* if the lengths of the lists `<Date 1>` and `<Date 2>` are different.

3.7.15. Various functions

existPrgm

`existPrgm( <Program:Str> )` -> *Dec*

Returns 1 if a ZeroBasic program named `.zcb` exists in the file system; otherwise, returns 0. Since the data type of the argument `<Program>` is *Str*, it is not possible to work with programs whose names contain double quotes (`"`). `<Program>` corresponds to a file named `.zcb`. The search for programs is performed in the `/exchange/` path, excluding subdirectories, within the calculator's file system.

Pxl_Test

`Pxl_Test( <X:Dec>, <Y:Dec> )` -> *Dec*

Returns 0 if the pixel color in the **graph window** matches the background color. Otherwise, the function returns 1. The values `<X>` and `<Y>` correspond to the pixel coordinates. The drawing area size is **195 pixels in height and 320 pixels in width**. The origin is at the top-left corner (0, 0), the Y-axis values increase from top to bottom, and the X-axis values increase from left to right.

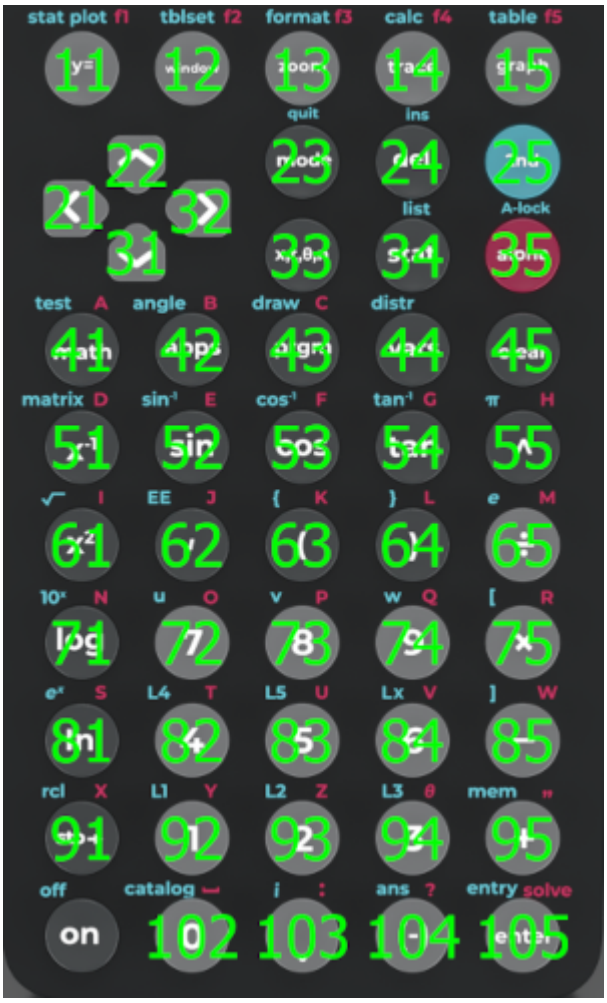
ERRORS:

- *DOMAIN*
    - if the value of `<X>` or `<Y>` is not an integer.
    - if the value of `<X>` is less than 0 or greater than 319.
    - if the value of `<Y>` is less than 0 or greater than 194.

getKey

`getKey( )` -> *Dec*

Returns the number of the pressed key. If no key was pressed, the function returns 0. The function waits for a key press for about 50 ms, so it is preferred to use this function within a loop that waits for a key press.



ERRORS:

- *INVALID* if the command is not called inside a script file.

`getKey( <Delay:Dec> ) -> Dec`

Returns the number of the pressed key. If no key was pressed, the function returns 0. The function waits for a button press for seconds. The value of `<Delay>` must be greater than 0.001. The value `<Delay>` is rounded down to the thousandth place (0.00298 -> 0.002). The maximum delay time is 100 seconds and does not depend on the value `<Delay>`.

ERRORS:

- *INVALID* if the command is not called inside a script file.
- *DOMAIN* if the value of `<Delay>` is out of the specified range.

Input

`Input( ) -> Done`

Opens a modal window of the **graph** and activates the pointer for point selection. Pointer control is done using the buttons ▼, ▲, ◄, ►. After selecting a point using the `Enter` button, the function places the pointer coordinates into the `X` and `Y` variables.

ERRORS:

- *INVALID* if the command is not called inside a script file.

`Input( <Ident Var:Any> )` -> *Any*

`Input( <Prompt:Str>, <Ident Var:Any> )` -> *Any*

This function allows assigning a value to a variable `<Ident Var>` during the execution of the script file via user input. Unlike assignment, this function does not attempt to convert the variable name to valid syntax. If executed successfully, it returns the value assigned to the variable. The `<Prompt>` variable contains a string that is displayed before the user input. By default, `<Prompt>` is "?". It is similar to the `Prompt` command.

ERRORS:

- *INVALID*
  - if nothing was entered (empty input).
  - if entered call of script file
  - if entered call of some `command`
  - if the command is not called inside a script file.
- *DATA TYPE* if the result of the user input expression has a type different from the variable `<Ident Var>` type.
- Various errors related to assignment.

expr

`expr( <Expression:Str> )` -> *Any*

Executes the expression `<Expression>` in the current context, and returns the result of the expression. Parsing is performed in the expression mode.

inString

`inString( <Value:Str>, <Substr:Str> )` -> *Dec*

`inString( <Value:Str>, <Substr:Str>, <Start:Dec> )` -> *Dec*

This function searches for the substring `<Substr>` within the string `<Value>`, starting from position `<Start>`. By default, `<Start>` is 1 (the search starts from the first character, left to right). The search is case-sensitive. The function returns the position of the first character of the found substring. If no match is found, the function returns 0.

ERRORS:

- *DOMAIN*
  - if the value of `<Start>` is not an integer.
  - if the value of `<Start>` is less than 1.

length

`length( <Value:Str> ) -> Dec`

Returns the length of the string `<Value>`.

sub

`sub( <Value:Str>, <Start:Dec> ) -> Str`

`sub( <Value:Str>, <Start:Dec>, <Length:Dec> ) -> Str`

This function returns a substring of `<Value>` starting from the character at position `<Start>`, with a length of `<Length>`. By default, `<Length>` is the number of characters from `<Start>` to the end of the string (it cuts the string to the end).

ERRORS:

- *DOMAIN*
  - if the value of `<Start>` or `<Length>` is not an integer.
  - if the value of `<Start>` is less than 1.
  - if the value of `<Length>` is less than 0.
- *INVALID DIMENSION* if the sum of the values of `<Start>` and `<Length>` minus 1 is greater than the length of the string `<Value>`.

toString, eval

Synonym: `eval(`...

`toString( <Value:Any> ) -> Str`

Returns the value of `<Value>` as a formatted string.

3.8. Conditional statement

Corresponds to the `IfStmt` node. The conditional statement allows implementing branching of expressions or scripts.

`if <Condition:Dec|Imag> then <True Expressions> [ else <False Expressions> ] end`

> If `<Condition>` is not equal to 0 (including the imaginary part), the `<True Expressions>` are executed sequentially; otherwise, `<False Expressions>` are executed (if this block is provided). `<True Expressions>` and `<False Expressions>` are multiple `Stmt` nodes, so they can also contain conditional and other structures. The lexical and syntactical structure of the node implies its use in script files (using spaces as separators, inputting multiline expressions). The node can also be used in expressions with some limitations (it is necessary to separate control structures from other expressions). Using the `:` separator on the main screen will also not yield a result, as expressions separated by it are executed independently of each other (but within the same variable context). The result of executing the construction is the result of the last executed expression.

3.9. Conditional loop

Corresponds to the `WhileStmt` node. A conditional loop allows repeating the execution of child expressions.

> `while` `<Condition:Dec|Imag>` `do` `<Expressions>` `end`
>
> If `<Condition>` is not equal to 0 (including the imaginary part), the `<Expressions>` are executed sequentially, then the value of `<Condition>` is recalculated and the loop repeats. If `<Condition>` is equal to 0 (including the imaginary part), the loop is terminated. `<Expressions>` is composed of several `Stmt` nodes, so they can also contain nested loops and other structures. The lexical and syntactical structure of the node implies its use in script files (using spaces as separators, inputting multiline expressions). The node can also be used in expressions with some limitations (it is necessary to separate control structures from other expressions). Using the `:` separator on the main screen will also not yield a result, as expressions separated by it are executed independently of each other (but within the same variable context). The result of executing the construction is 0. **Interrupt the loop** is possible by pressing the `On` button.

3.10. Iterative loop

Corresponds to the `ForStmt` node. The iterative loop allows repeating the execution of child expressions.

> `for` `<Ident Var:Dec>` `in` `<Start:Dec>` `,` `<Finish:Dec>` `[,` `<Step:Dec>` `]` `do` `<Expressions>` `end`
>
> Calculates the values of `<Start>`, `<Finish>`, and `<Step>`. By default, the value of `<Step>` is 1.
> The variable `<Ident Var>` is assigned the value of `<Start>`. Then, if the value of the variable `<Ident Var>` is less than `<Finish>`, the `<Expressions>` are executed sequentially; otherwise, the loop terminates. After executing the `<Expressions>` nodes, the value of the variable `<Ident Var>` is increased by the value of `<Step>`, and the loop repeats. Note that the comparison of the value of `<Ident Var>` and `<Finish>` is strict, and the loop will not run if they are equal. After the loop completes, the value of the variable `<Ident Var>` will be greater than or equal to `<Finish>`.
> `<Expressions>` consists of several `Stmt` nodes, so they can also contain nested loops and other structures. The lexical and syntactical structure of the node implies its use in script files (using spaces as separators, inputting multiline expressions). The node can also be used in expressions with some limitations (it is necessary to separate control structures from other expressions). Using the `:` separator on the main

> screen will also not yield a result, as expressions separated by it are executed independently of each other (but within the same variable context). The result of executing the construction is the value of the variable `<Ident Var>` after the loop completes. **Interrupt the loop** is possible by pressing the `On` button.

3.11. Calling a script file

Corresponds to the `CallStmt` node. This construct is intended for executing script files in the global context (ZeroBasic has only a global execution context). Using script files is similar to using subprograms or functions (with some exceptions) in other programming languages. Interrupt **the execution of the program** or **the calculation of expressions** are possible by pressing the `On` button.

> `call` `<Program:Str>`
>
> Since the data type of the argument `<Program>` is *Str*, it is not possible to work with programs whose names contain double quotes (`"`). `<Program>` corresponds to a file named `.zcb`. The search for programs is performed in the `/exchange/` path, excluding subdirectories, within the calculator's file system. The ZeroBasic application allows managing script files. Lexical analysis of a script file differs from lexical analysis of expressions, for more details see the sections Lexical structure of expressions and Lexical structure of a script file (zcb). The syntactical structure of a script file corresponds to the `Program` node, which can contain several `Stmt` nodes, so they can also contain calls to script files and other constructs. Expressions (nodes `Stmt`) are executed sequentially. The result of executing the construction is the result of the last executed expression. Some functions take into account that they are called from a script file.
>
> ERRORS:
>
> - *No such program* if the file is not found.
> - *eval break* if the `On` button is pressed.
> - *SYNTAX* if the file contains syntax errors.
> - Various errors related to program execution.

3.12. Operations

Below is a list of mathematical operations in the order of their execution priority. Equivalent operations are grouped together. Expressions enclosed in parentheses (parentheses groups) have priority in the execution order. All operations have **left associativity** (evaluated left to right), unless stated otherwise. Using incorrect data types will result in a *DATA TYPE* error.

3.12.1. Postfix operations

Corresponds to the `PostfixOp` node. These constructions are operations, so they can be combined and executed sequentially, which may not be immediately obvious. Expression `10!°"'r` is a valid expression.

Factorial

Corresponds to the Factorial token.

> `<Base:Dec>` `!` -> *Dec*
>
> Factorial of 0 is 1. The result is limited to the value `7.257415615e+306` (`<Base>` ≤ 170).
>
> ERRORS:
>
> - *DOMAIN*
>   - if the value of `<Base>` is not an integer.
>   - if the value of `<Base>` is negative.
> - *OVERFLOW* if the result overflows.

Conversion to radians

Corresponds to the Radian token.

> `<Value:Dec>` (\xCA) `r` -> *Dec*
>
> The expression is equivalent to `<Value> / <Angle Measurement>`. The value of `<Angle Measurement>` is `π/180` (≈ 0.0175) in **Degree** mode, and `1` in **Radian** mode.

Conversion to seconds

Corresponds to the `SecondPostfixOp` node.

> `<Seconds:Dec>` (\xD3) `"` -> *Dec*
>
> The expression is equivalent to `<Seconds> / 3600`.

Conversion to minutes

Corresponds to the `MinutePostfixOp` node. This node will be connected to the following second node if a second node exists.

> `<Minutes:Dec>` (\x27) `'` [`<Seconds:Dec>` (\xD3) `"`] -> *Dec*
>
> The expression is equivalent to `<Minutes> / 60 + <Seconds> / 3600`. By default, the value of `<Seconds>` is 0.

The expression `30'25""` is equivalent to `30'(25")"`. The expression `30'25°"` is equivalent to `30'((25°)°)"`.

Conversion to degrees

Corresponds to the `DegreePostfixOp` node. This node will be connected to the following minute or second node if they exist.

> `<Degrees:Dec>` (\xBF) `°` [`<Minutes:Dec>` (\x27) `'`] [`<Seconds:Dec>` (\xD3) `"`] -> *Dec*

> The expression is equivalent to `(<Degrees> <Minutes> / 60 + <Seconds> / 3600) * <Angle Measurement>`. By default, the value of `<Minutes>` is 0, the value of `<Seconds>` is 0. The value of `<Angle Measurement>` is `1` in **Degree** mode, and `π/180` (≈ 0.0175) in **Radian** mode.

The expression `30°°30''30""` is equivalent to `(30°)°(30')'(30")"`.

3.12.2. Exponentiation

Corresponds to the `ExponentStmt` node. Raising 0 to the power of 0 will cause an error.

> `<Base:Dec>` `^` `<Exponent:Dec>` `->` *`Dec`*
>
> The expression corresponds to the equivalent mathematical operation. Raising 0 to the power of 0 will cause an error.
>
> ERRORS:
>
> - *DIVIDE BY 0* if the value of `<Base>` is negative and the value of `<Exponent>` is fractional.
> - *DOMAIN* if both `<Base>` and `<Exponent>` are 0.

> `<Base:Imag>` `^` `<Exponent:Imag>` `->` *`Imag`*
>
> ERRORS:
>
> - *DOMAIN* if the **Degree** mode is active.

> `<Base:List>` `^` `<Exponent:Dec|Imag>` `->` *`List`*
>
> The resulting list will be formed from the elements on which the specified operation is performed.

> `<Base:Matr>` `^` `<Exponent:Dec>` `->` *`Matr`*
>
> Returns a matrix of the same dimension.
>
> - If the value of `<Exponent>` is 0, the operation returns the identity matrix.
> - If the value of `<Exponent>` is -1, the operation returns the matrix inverse of the matrix `<Base>`. Generates the error *SINGULAR MATR* if the original matrix is singular.
> - If the value of `<Exponent>` is greater than 0, the operation performs matrix multiplication of the matrix `<Base>` by itself `<Exponent>` times.
>
> ERRORS:
>
> - *INVALID DIMENSION* if the matrix `<Base>` is not square or has zero size.
> - *DOMAIN*
>   - if the value of `<Exponent>` is not an integer.

◦ if the value of `<Exponent>` is less than 0.

3.12.3. Unary operations

Corresponds to the `UnaryStmt` node.

Unary minus

> `-` `<Operand:Dec>` `->` *`Dec`*
>
> Returns the opposite number.

> `-` `<Operand:Imag>` `->` *`Imag`*
>
> Returns the opposite complex number.

> `-` `<Operand:List>` `->` *`List`*
>
> The resulting list will be formed from the elements on which the specified operation is performed.

> `-` `<Operand:Matr>` `->` *`Matr`*
>
> The resulting matrix will be formed from the elements on which the specified operation is performed.

Logical negation

The use of a space in this construction is only allowed in script file mode. To use the operator in expressions, `<Operand>` should be separated by parentheses or special space (\xAC) (see more in the section Lexical structure common for file and expression). The operator `not` is case-insensitive (`not` and `NOT` are equivalent).

> `not` `<Operand:Dec|Imag>` `->` *`Dec`*
>
> If the value of `<Operand>` is 0, it returns 1, otherwise it returns 0.

> `not` `<Operand:List>` `->` *`List`*
>
> The resulting list will be formed from the elements on which the specified operation is performed.

3.12.4. Multiplication and division

Corresponds to the `MultiplicationStmt` node.

Multiplication

> `<Multiplicand:Dec>` `*` `<Multiplier:Dec>` -> *Dec*
>
> The expression corresponds to the equivalent mathematical operation.

> `<Multiplicand:Imag>` `*` `<Multiplier:Imag>` -> *Dec|Imag*
>
> The expression corresponds to the equivalent mathematical operation for complex numbers.

> `<Multiplicand:List>` `*` `<Multiplier:Dec|Imag>` -> *List*
>
> `<Multiplicand:Dec|Imag>` `*` `<Multiplier:List>` -> *List*
>
> The resulting list will be formed from the products of the elements with the number.

> `<Multiplicand:List>` `*` `<Multiplier:List>` -> *List*
>
> The resulting list will be formed from the products of corresponding list elements. The sizes of the lists must match, otherwise the error *DIM MISMATCH* will be generated.

> `<Multiplicand:Matr>` `*` `<Multiplier:Dec|Imag>` -> *Matr*
>
> `<Multiplicand:Dec|Imag>` `*` `<Multiplier:Matr>` -> *Matr*
>
> The resulting matrix will be formed from the products of elements with the number.

> `<Multiplicand:Matr>` `*` `<Multiplier:Matr>` -> *Matr*
>
> The operation performs matrix multiplication of matrix `<Multiplicand>` by `<Multiplier>`. The width of `<Multiplicand>` must equal the height of `<Multiplier>`, otherwise the error *DIM MISMATCH* will be generated. The height of the resulting matrix will be equal to the height of `<Multiplicand>`, and the width will be equal to the width of `<Multiplier>`.

Division

The natural fraction symbol (`⁄` \x9D) in this case is interpreted as the division operator with subsequent conversion of the result to a natural fraction. If `<Divisor>` is 0, this will result in an error *DIVIDE BY 0*.

> `<Dividend:Dec>` `/` `<Divisor:Dec>` -> *Dec*
>
> The expression corresponds to the equivalent mathematical operation.

> `<Dividend:Imag>` `/` `<Divisor:Imag>` `->` *`Dec|Imag`*
>
> The expression corresponds to the equivalent mathematical operation for complex numbers.

> `<Dividend:List>` `/` `<Divisor:Dec|Imag>` `->` *`List`*
>
> The resulting list will be formed from the elements of the list `<Dividend>` divided by `<Divisor>`.

> `<Dividend:Dec|Imag>` `/` `<Divisor:List>` `->` *`List`*
>
> The resulting list will be formed from the elements of `<Dividend>`, divided by the elements of the list `<Divisor>`.

> `<Dividend:List>` `/` `<Divisor:List>` `->` *`List`*
>
> The resulting list will be formed from the ratios of the elements of the list `<Dividend>` to the corresponding elements of the list `<Divisor>`. The sizes of the lists must match, otherwise an error *DIM MISMATCH* will be generated.

3.12.5. Addition and subtraction

Corresponds to the `AdditionStmt` node.

Addition

> `<Addend 1:Dec>` `+` `<Addend 2:Dec>` `->` *`Dec`*
>
> The expression corresponds to the same mathematical operation.

> `<Addend 1:Imag>` `+` `<Addend 2:Imag>` `->` *`Dec|Imag`*
>
> The expression corresponds to the same mathematical operation for complex numbers.

> `<Addend 1:List>` `+` `<Addend 2:Dec|Imag>` `->` *`List`*
>
> `<Addend 1:Dec|Imag>` `+` `<Addend 2:List>` `->` *`List`*
>
> The resulting list will be formed from the sums of elements with the number.

> `<Addend 1:List>` `+` `<Addend 2:List>` `->` *`List`*

> The resulting list will be formed from the sums of corresponding elements of the lists. The list sizes must match, otherwise an error *DIM MISMATCH* will be generated.

> `<Addend 1:Matr>` `+` `<Addend 2:Matr>` `-> Matr`
>
> The resulting matrix will be formed from the sums of corresponding elements of the matrices. The matrix sizes must match, otherwise an error *DIM MISMATCH* will be generated.

> `<Addend 1:Str>` `+` `<Addend 2:Str>` `-> Str`
>
> Concatenates two strings.

Subtraction

> `<Minuend:Dec>` `-` `<Subtrahend:Dec>` `-> Dec`
>
> The expression corresponds to the same mathematical operation.

> `<Minuend:Imag>` `-` `<Subtrahend:Imag>` `-> Dec|Imag`
>
> The expression corresponds to the same mathematical operation for complex numbers.

> `<Minuend:List>` `-` `<Subtrahend:Dec|Imag>` `-> List`
>
> The resulting list will be formed from the elements of the list `<Minuend>`, from which `<Subtrahend>` is subtracted.

> `<Minuend:Dec|Imag>` `-` `<Subtrahend:List>` `-> List`
>
> The resulting list will be formed from the set of differences `<Minuend>` and the elements `<Subtrahend>`.

> `<Minuend:List>` `-` `<Subtrahend:List>` `-> List`
>
> The resulting list will be formed from the differences of corresponding elements of the lists. The list sizes must match, otherwise an error *DIM MISMATCH* will be generated.

> `<Minuend:Matr>` `-` `<Subtrahend:Matr>` `-> Matr`

> The resulting matrix will be formed from the differences of corresponding elements of the matrices. The matrix sizes must match, otherwise an error *DIM MISMATCH* will be generated.

3.12.6. Comparison operations

Corresponds to the `CompareStmt` node.

Equality

The *Equality* operation is represented by `=`, but it can also appear as `==` in expressions.

> `<Value 1:Dec|Imag>` `=` `<Value 2:Dec|Imag>` `-> Dec`
>
> Returns 1 if the difference between `<Value 1>` and `<Value 2>` is less than or equal to 0, otherwise returns 0.

> `<Value 1:List>` `=` `<Value 2:Dec|Imag>` `-> List`
>
> `<Value 1:Dec|Imag>` `=` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from elements that have undergone the *Equality* operation with the specified number.

> `<Value 1:List>` `=` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the results of the specified operation between corresponding elements of the lists. The list sizes must match; otherwise, a *DIM MISMATCH* error will be generated.

> `<Value 1:Matr>` `=` `<Value 2:Matr>` `-> Dec`
>
> Performs the *Equality* operation on corresponding elements of the matrices. If all corresponding elements are equal, it returns 1; otherwise, it returns 0. The matrix sizes must match; otherwise, a *DIM MISMATCH* error will be generated.

> `<Value 1:Str>` `=` `<Value 2:Str>` `-> Dec`
>
> Returns 1 if the strings are equal character by character, otherwise returns 0.

Inequality

The *Inequality* operation is represented by `!=` or `≠` (\xB1). This operation is similar to the *Equality* operation, except that the result is additionally subject to a Logical Negation. The expression `<Value 1>` `≠` `<Value 2>` is equivalent to the expression `not (<Value 1>` `=` `<Value 2>)`.

Greater

> `<Value 1:Dec>` `>` `<Value 2:Dec>` -> *Dec*
>
> Returns 1 if the difference between `<Value 1>` and `<Value 2>` is greater than 0, otherwise returns 0.

> `<Value 1:List>` `>` `<Value 2:Dec>` -> *List*
>
> The resulting list will be formed from the elements of the list `<Value 1>`, where the *Greater* operation was applied with `<Value 2>`.

> `<Value 1:Dec>` `>` `<Value 2:List>` -> *List*
>
> The resulting list will be formed from the set of results of *Greater* operations between `<Value 1>` and the elements of the list `<Value 2>`.

> `<Value 1:List>` `>` `<Value 2:List>` -> *List*
>
> The resulting list will be formed from the results of *Greater* operations between corresponding elements of the list `<Value 1>` and `<Value 2>`. The list sizes must match, otherwise a *DIM MISMATCH* error will be generated.

ERRORS:

- *DATA TYPE* if the list `<Value 1>` or `<Value 2>` contains a complex number.

Greater or equal

The *Greater or equal* operation is represented by `>=` or `≥` (\x96).

> `<Value 1:Dec>` `≥` `<Value 2:Dec>` -> *Dec*
>
> Returns 1 if the difference between `<Value 1>` and `<Value 2>` is greater than or equal to 0, otherwise returns 0.

> `<Value 1:List>` `≥` `<Value 2:Dec>` -> *List*
>
> The resulting list will be formed from the elements of the list `<Value 1>`, where the *Greater or equal* operation was applied with `<Value 2>`.

> `<Value 1:Dec>` `≥` `<Value 2:List>` -> *List*

> The resulting list will be formed from the set of results of *Greater or equal* operations between `<Value 1>` and the elements of the list `<Value 2>`.

> `<Value 1:List>` `≥` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the results of *Greater or equal* operations between corresponding elements of the list `<Value 1>` and `<Value 2>`. The list sizes must match, otherwise a *DIM MISMATCH* error will be generated.

ERRORS:

- *DATA TYPE* if the list `<Value 1>` or `<Value 2>` contains a complex number.

Less

> `<Value 1:Dec>` `<` `<Value 2:Dec>` `-> Dec`
>
> Returns 1 if the difference between `<Value 1>` and `<Value 2>` is less than 0, otherwise returns 0.

> `<Value 1:List>` `<` `<Value 2:Dec>` `-> List`
>
> The resulting list will be formed from the elements of the list `<Value 1>`, where the *Less* operation was applied with `<Value 2>`.

> `<Value 1:Dec>` `<` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the set of results of *Less* operations between `<Value 1>` and the elements of the list `<Value 2>`.

> `<Value 1:List>` `<` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the results of *Less Than* operations between corresponding elements of the list `<Value 1>` and `<Value 2>`. The list sizes must match, otherwise a *DIM MISMATCH* error will be generated.

ERRORS:

- *DATA TYPE* if the list `<Value 1>` or `<Value 2>` contains a complex number.

Less or equal

The *Less or equal* operation is represented by `<=` or `≤` (\x95).

> `<Value 1:Dec>` `≤` `<Value 2:Dec>` `-> Dec`

> Returns 1 if the difference between `<Value 1>` and `<Value 2>` is less than or equal to 0, otherwise returns 0.

> `<Value 1:List>` `≤` `<Value 2:Dec>` `-> List`
>
> The resulting list will be formed from the elements of the list `<Value 1>`, where the *Less or equal* operation was applied with `<Value 2>`.

> `<Value 1:Dec>` `≤` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the set of results of *Less or equal* operations between `<Value 1>` and the elements of the list `<Value 2>`.

> `<Value 1:List>` `≤` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the results of *Less or equal* operations between corresponding elements of the list `<Value 1>` and `<Value 2>`. The list sizes must match, otherwise a *DIM MISMATCH* error will be generated.

ERRORS:

- *DATA TYPE* if the list `<Value 1>` or `<Value 2>` contains a complex number.

3.12.7. Logical operations

Corresponds to the `LogicOrStmt` and `LogicAndStmt` nodes. The *Logical AND* operation executed earlier than the *Logical OR* and *Exclusive OR* operations.

Logical AND

The use of space in this construction is allowed only in script file mode. To use the operation in expressions `<Value 1>` and `<Value 2>`, separate them with parentheses or special space (\xAC) (more details in the section The structure of lexemes common to a file and an expression). The operator `and` is case-insensitive (`and` and `AND` are equivalent).

> `<Value 1:Dec>` `and` `<Value 2:Dec>` `-> Dec`
>
> Returns 1 if `<Value 1>` and `<Value 2>` are not equal to 0, otherwise 0.

> `<Value 1:List>` `and` `<Value 2:Dec>` `-> List`
>
> `<Value 1:Dec>` `and` `<Value 2:List>` `-> List`

> The resulting list will be formed from the results of the specified operation between the list elements and the number.

> `<Value 1:List>` `and` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the results of the specified operation between corresponding elements of list `<Value 1>` and `<Value 2>`. The lists must have the same size, otherwise an error *DIM MISMATCH* will be generated.

ERRORS:

- *DATA TYPE* if the list `<Value 1>` or `<Value 2>` contains a complex number.

Logical OR

The use of space in this construction is allowed only in script file mode. To use the operation in expressions `<Value 1>` and `<Value 2>`, separate them with parentheses or special space (\xAC) (more details in the section The structure of lexemes common to a file and an expression). The operator `or` is case-insensitive (`or` and `OR` are equivalent).

> `<Value 1:Dec>` `or` `<Value 2:Dec>` `-> Dec`
>
> Returns 0 if `<Value 1>` and `<Value 2>` are both equal to 0, otherwise 1.

> `<Value 1:List>` `or` `<Value 2:Dec>` `-> List`
>
> `<Value 1:Dec>` `or` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the results of the specified operation between the list elements and the number.

> `<Value 1:List>` `or` `<Value 2:List>` `-> List`
>
> The resulting list will be formed from the results of the specified operation between corresponding elements of list `<Value 1>` and `<Value 2>`. The lists must have the same size, otherwise an error *DIM MISMATCH* will be generated.

ERRORS:

- *DATA TYPE* if the list `<Value 1>` or `<Value 2>` contains a complex number.

Exclusive OR

The use of space in this construction is allowed only in script file mode. To use the operation in expressions `<Value 1>` and `<Value 2>`, separate them with parentheses or special space (\xAC) (more details in the section The structure of lexemes common to a file and an expression). The operator `xor` is case-insensitive (`xor` and `XOR` are equivalent).

> `<Value 1:Dec>` `xor` `<Value 2:Dec>` `->` *`Dec`*
>
> Returns 0 if `<Value 1>` and `<Value 2>` are both 0 or `<Value 1>` and `<Value 2>` are both non-zero, otherwise 1.

> `<Value 1:List>` `xor` `<Value 2:Dec>` `->` *`List`*
>
> `<Value 1:Dec>` `xor` `<Value 2:List>` `->` *`List`*
>
> The resulting list will be formed from the results of the specified operation between the list elements and the number.

> `<Value 1:List>` `xor` `<Value 2:List>` `->` *`List`*
>
> The resulting list will be formed from the results of the specified operation between corresponding elements of list `<Value 1>` and `<Value 2>`. The lists must have the same size, otherwise an error *DIM MISMATCH* will be generated.

ERRORS:

- *DATA TYPE* if the list `<Value 1>` or `<Value 2>` contains a complex number.

IV. Text description of lexical structure of Main screen expressions

```
@startebnf
program = empty_prgm | ( stmt (* [1] *), {":", stmt} );
empty_prgm = ? end of input ?;
(* [1]: TI-84 is capable of processing empty stmt,
Zero Calculator interrupts program execution
upon encountering an empty stmt. *)
@endebnf
```

V. Text description of lexical structure of an expression

```
@startebnf
page 1x1
stmt = (
           Command, [
               ( ? space ? | "(" ),
               [ exprList ]
           ] |
           [ exprList ]
       ),
       ? )}] ? (* [1] *),
       ? end of input ?;
@endebnf

@startebnf
page 1x5
exprList = expr, { ? * ? (* [2] *), expr };

(* [1]: After lexical analysis, the list of lexemes is divided into blocks.
The block boundaries defined by [→]. At the end of each block, tokens
for any missing closing brackets ( [)], [[], [}] ) are added.
The insertion order is the reverse of the opening brackets.
*)
(* [2]: An insertion of the [*] token between two [expr] occurs in
the following cases:
    previous [expr]     next [expr]           example

  - [Number],           [Identifier]         "3X"
  - [Number],           [(]                  "3("
  - [Number],           [{]                  "3{"
  - [Number],           [[]                  "3["

  - [Imag],             [Identifier]         "iX"
  - [Imag],             [(]                  "i("
  - [Imag],             [{]                  "i{"
  - [Imag],             [[]                  "i["
  - [Imag],             [Imag]               "ii"

  - [Identifier],       [Number]             "X5"
  - [Identifier],       [Imag]               "Xi"
  - [Identifier],       [Identifier]         "Xπ"
  - [Identifier],       [{]                  "X{"
  - [Identifier],       [[]                  "X["

  - [)],                [(]                  ")("
  - [)],                [{]                  "){"
  - [)],                [[]                  ")["
  - [)],                [Number]             ")3"
  - [)],                [Imag]               ")i"
  - [)],                [Identifier]         ")X"
  - [)],                [UnaryMinus]         ")-"

  - [}],                [Number]             "}3"
  - [}],                [Imag]               "}i"
  - [}],                [Identifier]         "}X"
  - [}],                [(]                  "}("
  - [}],                [{]                  "}{"
```

```
  - [Identifier],       [()    provided that [Identifier] is not
                               a FunctionIdentifier,
                               a CustomListIdentifier,
                               a StandardListIdentifier,
                               a MatrixIdentifier,
                               'Ans', 'TblInput'
*)
@endebnf

@startebnf
page 1x1
expr = "\n" | Lexeme;

equals = "=", ["="];
@endebnf
```

VI. Text description of lexical structure of a script file

```
@startebnf
page 1x1
program = stmt, { "\n", stmt }, ? end of input ?;
@endebnf

@startebnf
page 1x2
stmt = Command, [
           ( ? space ? | "(" ),
           [ exprList ]
       ] |
       [ exprList ];

exprList = expr, { expr };
@endebnf

@startebnf
page 1x1
expr = { ? space ? }, ( "\n" | Lexeme );
equals = "=", "=";
@endebnf
```

VII. Text description of the structure of lexemes common to a file and an expression

```
@startebnf
page 1x3
Command = MatrixIdentifier |
          PictureIdentifier |
          GDBIdentifier |
          OneSymbolIdentifier |
          StandardListIdentifier |
          CustomListIdentifier |
          Identifier;
(* The command lexeme is formed only when
the textual content of the identifier
matches the command name (a complete list
is available in the Commands section)
or the [Call] token. *)
@endebnf

@startebnf
page 1x5
Lexeme = Comment |
       MatrixIdentifier |
       PictureIdentifier |
       GDBIdentifier |
       ConvertingOperator |
       SymbolToken |
       OneSymbolIdentifier |
       DegreePostfixOperator |
       UpperPower |
       StoreOrMinus |
       GreaterOrGreaterEquals |
       LessOrLessEquals |
       equals |
       NotEqualsOrFactorial |
       Imag |
       StandardListIdentifier |
       CustomListIdentifier |
       Number |
       Identifier |
       String;
@endebnf

@startebnf
page 1x1
Comment = ? // ? (* [2] *), [? any character ?];
(* [2]: A comment is treated as
"\n" character or end of input *)
@endebnf

@startebnf
page 1x1
MatrixIdentifier = "[", {Alpha}-, "]";
PictureIdentifier = "Pic", Digit;
GDBIdentifier = "GDB", Digit;
@endebnf

@startebnf
```

```
page 1x1
ConvertingOperator = "▶" (* [3] *), { Alpha | "▶" (* [4] *) | "◀" | "/" };
(* [3]: '▶' Conversion symbol (\xDA) *)
(* [4]: '▶' Right triangle arrow symbol (\x9E) *)
@endebnf

@startebnf
page 1x1
Imag = "i" (* Imaginary unit symbol (\xA5) *);
@endebnf

@startebnf
page 1x4
SymbolToken = "+" | "*" | "/" | "^" |
              "(" | ")" | "{" (* [5] *) | "}" |
              "[" (* [5] *) | "]" | "," |
              "⁄" (* [6] *) | "␣" (* [7] *) | "-" (* [8] *);
(* [5]: The next character must not be
a closing bracket *)
(* [6]: '⁄' Common fraction symbol (\x9D) *)
(* [7]: '␣' Mixed number fraction symbol (\xA0) *)
@endebnf

@startebnf
page 1x5
OneSymbolIdentifier = "ⁿ√" (* Root with degree symbol (\xA1) *) |
                      "√"  (* Square root symbol (\x7F) *) |
                      "π"  (* Pi symbol (\xD2) *) |
                      "e"  (* Euler's number symbol (\xD0) *) |
                      "₁₀" (* Small number 10 symbol (\xD5) *) |
                      "ȳ"  (* Mean y symbol (\xAA) *) |
                      "n"  (* n in statistics symbol (\xD7) *) |
                      "n"  (* n in SEQUENCE graphs symbol (\xDB) *);
@endebnf

@startebnf
page 1x1
DegreePostfixOperator = "°" (* Degree symbol (\xBF) *) |
                        "'" (* Minute symbol (\x27) *) |
                        "″" (* Second symbol (\xD3) *) |
                        "r" (* Radian symbol (\xCA) *);
@endebnf

@startebnf
page 1x1
UpperPower = "⁻¹" (* Negative one power symbol (\x8A) *) |
             "'0' - '9'" (* (\x80 - \x89) *);
@endebnf

@startebnf
page 1x1
StoreOrMinus = "→" (* Assignment symbol (\xA8) *) |
             ( "-", [">"]);
@endebnf

@startebnf
page 1x1
GreaterOrGreaterEquals = "≥" (* (\x96) *) |
                         ( ">", ["="]);
@endebnf
```

```
@startebnf
page 1x1
LessOrLessEquals = "≤" (* (\x95) *) |
                   ( "<", ["="]);
@endebnf

@startebnf
page 1x1
equals = "=", ["="] (* In expressions mode *);
equals = "=", "=" (* In a script file mode *);
@endebnf

@startebnf
page 1x1
FactorialOrNotEquals = "≠" (* (\xB1) *) |
                       ( "!", ["="]);
@endebnf

@startebnf
page 1x1
StandardListIdentifier = "L" (* [9] *), "'₀' - '₆'" (* [10] *);
CustomListIdentifier = "ʟ" (* [11] *), { Alpha }-;

@endebnf

@startebnf
page 1x3
Number = (
           {Digit, {"Digit separator" (* (\xDD) *)}}-,
           [NumberFraction],
           [NumberExponent]
         ) |
         (
           NumberFraction,
           [NumberExponent]
         ) |
         (
           "E" (* [12] *),
           (
             ( "-" (* [8] *) | "+" ), {Digit} |
             {Digit (* [13] *)}-
           )
         );
NumberFraction = ".", {Digit}-;
NumberExponent = "E" (* [12] *), [ "-" (* [8] *) | "+" ], {Digit};
@endebnf

@startebnf
page 1x5
Identifier = [ ? special space ? (* \xAC *) ], (
               ThirdRoot |
               IdentifierSymb, { IdentifierSymb | "_" } |
               Not | And | Or | Xor | If | Then | Else | While | Do | For | In | Call | End
             ), [ ? special space ? (* \xAC *) ];
ThirdRoot = "³" (* \xD1 *), "√" (* \x7F *);
@endebnf

@startebnf
page 1x11
IdentifierSymb = Alpha |
                 "Δ" (* Delta symbol (\x9A) *) |
```

```
                  "*" (* Asterisk symbol (custom) (\xCD) *) |
                  "▶" (* Right triangle arrow symbol (\x9E) *) |
                  "◀" (* Left triangle arrow  symbol(\x9F) *) |
                  "+" (* Plus symbol (custom) (\xCE) *) |
                  "-" (* Dash symbol (custom) (\xDE) *) |
                  "1" (* Cursive one symbol (\xDF) *) |
                  "2" (* Cursive two symbol (\xE0) *) |
                  "[" (* Cursive left bracket symbol (\xE1) *) |
                  "]" (* Cursive right bracket symbol (\xE2) *) |
                  "²" (* Superscript 2 symbol (\xBD) *) |
                  "ₜ" (* Subscript tau symbol (\xBC) *) |
                  "⁻¹" (* Inverse trig function symbol (\xD4) *) |
                  "θ" (* Theta symbol (\x99) *) |
                  "³" (* Cube root index symbol (\xD1) *) |
                  "σ" (* Sigma symbol (\xAB) *) |
                  "Σ" (* Summation symbol (\xA2) *) |
                  "χ" (* Chi symbol (\xD8) *) |
                  "R" (* R in statistics (\xBE) *) |
                  "X̄" (* Mean x symbol (\xA9) *) |
                  "pˆ" (* Rho in statistics symbol (\xB2) *) |
                  "F" (* F in statistics symbol (\xD9) *) |
                  "Digit separator" (* (\xDD) *) |
                  "'₀' - '₉'" (* Subscript digits (\xC0 - \xC9) *);
@endebnf

@startebnf
page 1x1
Not = ("N" | "n"), ("O" | "o"), ("T" | "t");
@endebnf

@startebnf
page 1x1
And = ("A" | "a"), ("N" | "n"), ("D" | "d");
@endebnf

@startebnf
page 1x1
Or = ("O" | "o"), ("R" | "r");
@endebnf

@startebnf
page 1x1
Xor = ("X" | "x"), ("O" | "o"), ("R" | "r");
@endebnf

@startebnf
page 1x1
If = ("I" | "i"), ("F" | "f");
@endebnf

@startebnf
page 1x1
Then = ("T" | "t"), ("H" | "h"), ("E" | "e"), ("N" | "n");
@endebnf

@startebnf
page 1x1
Else = ("E" | "e"), ("L" | "l"), ("S" | "s"), ("E" | "e");
@endebnf

@startebnf
```

```
page 1x1
While = ("W" | "w"), ("H" | "h"), ("I" | "i"), ("L" | "l"), ("E" | "e");
@endebnf

@startebnf
page 1x1
Do = ("D" | "d"), ("O" | "o");
@endebnf

@startebnf
page 1x1
For = ("F" | "f"), ("O" | "o"), ("R" | "r");
@endebnf

@startebnf
page 1x1
In = ("I" | "i"), ("N" | "n");
@endebnf

@startebnf
page 1x1
Call = ("C" | "c"), ("A" | "a"), ("L" | "l"), ("L" | "l");
@endebnf

@startebnf
page 1x1
End = ("E" | "e"), ("N" | "n"), ("D" | "d");
@endebnf

@startebnf
page 1x1
String = "”" (* \x22 *),
         { "any character except '”' (\x22), '\n', '→' (\xA8), '->'" },
         [ "”" (* \x22 *) ];
@endebnf

@startebnf
page 1x1
Alpha = UpperAlpha | LowerAlpha;
UpperAlpha = "'A' - 'Z'";
LowerAlpha = "'a' - 'z'";
Digit = "'0' - '9'";
@endebnf

@startebnf
page 1x4
(* [7]: '-' Unary minus symbol (\x98) *)
(* [8]: 'L' List symbol (\xA6) *)
(* [9]: '₀' - '₆' (\xC0 - \xC6) *)
(* [10]: '⌞' List symbol (\xA7) *)

(* [11]: 'E' Decimal exponent symbol (\xD6) *)
(* [12]: The third branch of a number also constitutes
the exponent part (NumberExponent), but in the first
two cases the order may be omitted due to mantissa
presence ('5E'). The third branch requires either
a sign of exponent or explicit exponent value
('E-', 'E6', 'E+7') *)

@endebnf
```

VIII. Text description of syntactic structure of the ZeroBasic language

```
@startebnf
page 1x1
Program = { "Newline" }, { Stmt }, "EndOfInput";
(* Entry point for syntax parsing of a script file *)
@endebnf

@startebnf
page 1x1
Stmt = ( CommandStmt | IfStmt | WhileStmt | ForStmt | CallStmt | StoreStmt ),
       { "Newline" },
       (
           ? EndOfInput ? | ? Newline ? | ? End ? | ? Else ?
           (* Token after Stmt token must be one of these tokens. *)
       );
(* Entry point for syntax parsing of an expression
(The main screen independently splits the entered
command into expressions) *)
@endebnf

@startebnf
page 1x1
CommandStmt = "Command",
              (
                "(", [ ExprStmt, { ",", ExprStmt } ], ")" |
                [ ExprStmt, { ",", ExprStmt } ]
              );
(* The marker for analyzing CommandStmt is the [Command]
token. Mismatch in the following syntax will lead to
an error. *)
@endebnf

@startebnf
page 1x1
IfStmt = "If", ExprStmt, { "Newline" }, "Then", { "Newline" }, { Stmt },
         [ "Else", { "Newline" }, { Stmt } ], "End";
(* The marker for analyzing IfStmt is the [If] token.
Mismatch in the following syntax will lead to an error. *)
@endebnf

@startebnf
page 1x1
WhileStmt = "While", ExprStmt, { "Newline" }, "Do", { "Newline" }, { Stmt }, "End";
(* The marker for analyzing WhileStmt is the [While] token.
Mismatch in the following syntax will lead to an error. *)
@endebnf

@startebnf
page 1x1
ForStmt = "For", IdentStmt, "In", ExprStmt, ",", ExprStmt, [ ",", ExprStmt ],
          { "Newline" }, "Do", { "Newline" }, { Stmt }, "End";
(* The marker for analyzing ForStmt is the [For] token.
Mismatch in the following syntax will lead to an error. *)
@endebnf

@startebnf
page 1x1
CallStmt = "Call",
```

```
           (
             "(", ExprStmt, ")" |
             ExprStmt
           );
(* The marker for analyzing CallStmt is the [Call] token.
Mismatch in the following syntax will lead to an error. *)
@endebnf

@startebnf
page 1x1
StoreStmt = ExprStmt, [
              "Store", IdentStmt, [ "(", ExprStmt, [ ",", ExprStmt ], ")" ] |
              "ConvertingOp"
            ];
@endebnf

@startebnf
page 1x1
IdentStmt = "Identifier";
@endebnf

@startebnf
page 1x1
ExprStmt = LogicOrStmt;
@endebnf

@startebnf
page 1x1
LogicOrStmt = LogicAndStmt, { ( "Or" | "Xor" ), LogicAndStmt };
(* Execution order of logical operations is not specified.
Operations are executed sequentially. *)

LogicAndStmt = CompareStmt, { "And", CompareStmt };
@endebnf

@startebnf
page 1x2
CompareStmt = AdditionStmt, {
                             ( "Equals" |
                               "NotEquals" |
                               "Greater" |
                               "GreaterEquals" |
                               "Less" |
                               "LessEquals" ),
                             AdditionStmt
                           };
(* Execution order of comparison operations is not specified.
Operations are executed sequentially. *)
@endebnf

@startebnf
page 1x1
AdditionStmt = MultiplicationStmt, { ( "Plus" | "Minus" ), MultiplicationStmt };
@endebnf

@startebnf
page 1x1
MultiplicationStmt = UnaryStmt, { ( "Mult" | "Divide" | "ND" ), UnaryStmt |
                                  "Imag" };
@endebnf

@startebnf
```

```
page 1x1
UnaryStmt = [ ( "Minus" | "UnaryMinus" | "Not" ), UnaryStmt ], ExponentStmt;
(* The [UnaryMinus] token is replaced by the [Minus] token *)
@endebnf

@startebnf
page 1x1
ExponentStmt = PostfixOpStmt, { "Power" , PostfixOpStmt | "UpperPower" };
(* The [UpperPower] token contains only a single
superscript digit *)
@endebnf

@startebnf
page 1x1
PostfixOpStmt = PrimaryStmt, { PostfixOp };
@endebnf

@startebnf
page 1x1
PostfixOp = DegreePostfixOp |
            MinutePostfixOp |
            SecondPostfixOp |
            "Radian" |
            "Factorial";
@endebnf

@startebnf
page 1x1
DegreePostfixOp = "Degree", [ PrimaryStmt, ( {MinutePostfixOp}- | {SecondPostfixOp}- )];
@endebnf

@startebnf
page 1x1
MinutePostfixOp = "Minute", [ PrimaryStmt, {SecondPostfixOp}- ];
@endebnf

@startebnf
page 1x1
SecondPostfixOp = "Second";
@endebnf

@startebnf
page 1x3
PrimaryStmt = UNDStmt |
              ImagStmt |
              VarOrFnCallStmt |
              GroupingStmt |
              ListStmt |
              MatrixStmt |
              UnaryStmt (* Only for [UnaryMinus] token *) |
              StringStmt;
@endebnf

@startebnf
page 1x1
UNDStmt = NumberStmt, [ "UND", NumberStmt, "ND", NumberStmt ];
@endebnf

@startebnf
page 1x1
NumberStmt = "Number";
ImagStmt = "Imag";
```

```
StringStmt = "String";
@endebnf

@startebnf
page 1x1
VarOrFnCallStmt = FnCallStmt | VarStmt;
FnCallStmt = IdentStmt, "(", [ ExprStmt, { ",", ExprStmt } ], ")";
VarStmt = IdentStmt;
@endebnf

@startebnf
page 1x1
GroupingStmt = "(", ExprStmt, ")";
@endebnf

@startebnf
page 1x1
ListStmt = "{", ExprStmt, { ("," | "Number"), ExprStmt }, "}";
(* The [Number] token is included in the subsequent
ExprStmt node. This analysis structure allows for
the following syntax in the script file:
{1,2,3}
{1 2 3} *)
@endebnf

@startebnf
page 1x2
MatrixStmt = MatrixVarStmt | "[", MatrixRow, { MatrixRow }, "]";
MatrixVarStmt =  "[", "Identifier", "]";
(* The [Identifier] token must consist only of letters *)
MatrixRow = "[", ExprStmt, { ("," | "Number"), ExprStmt }, "]";
(* The [Number] token is included in the subsequent
ExprStmt node. This analysis structure allows for
the following syntax in the script file:
[[1,2,3]]
[[1 2 3]] *)
@endebnf
```

IX. Documentation changelog

v2.27.1 (2025-12-29)

- Added information about errors in commands RecallPic and StorePic.
- Fixed Tangent command working in Par functions mode.

v2.27.0 (2025-12-15)

- Command delVar can remove variables Ans and TblInput.
- Added information about using construction ->dim(on variable Ans.
- Added information about modification list or matrix elements of Ans variable.
- Construction ->dim(modify elements of TblInput variable.
- Prohibited creation variables whose names match with commands and functions.
- Changed lexing special vars (Ans, TblInput) to same as another identifiers.
- Added information about returned errors for Input and Prompt commands.

v2.26.0 (2025-10-16)

- Increased the limitation of the result of performing the factorial operation.
- Added check of negative values for the factorial operation.
- Increased the limitation of the result of performing the nCr(function.
- Added interruption the execution of the program or the calculation of expressions by On button.
- Changed names of commands 1-VarStats, 2-VarStats, Med-Med, LinReg[ax+b], LinReg[a+bx], Manual-Fit.
- Added several symbols to Identifier lexeme.
- Changed parsing Newline tokens in syntax processing of Stmt, IfStmt, WhileStmt and ForStmt.
- Changed lexical analysis for string lexeme (cancel of consuming newline character).
- Changed lexical analysis for stmt, Command and Call lexemes.
- Changed parsing Command and Call tokens.
- Changed data type for command delPrgm(.
- Added command existPrgm(.
- Changed data type of argument of 'Call' construction.
- Mixed fraction (UNDStmt) moved into PrimaryStmt.
- Changed syntax scheme - PrimaryStmt process UnaryStmt only for UnaryMinus token.
- Fixed order of execution of logical operators (and, or, xor).

v2.25.0 (2025-09-11)

- Change description of Disp command (now command Disp can't interrupt Zerobasic execution).
- Added changelog documentation generating.
- Changed style for time functions and commands.
- Changed description of command ClrDraw (not update graph window).
- Added note that draw commands (ClrDraw, Line, Horizontal, Vertical, Tangent, DrawF, Shade, DrawInv, Circle, Text, Pt_On, Pt_Off, Pt_Change, Pxl_On, Pxl_Off, Pxl_Change) not update graph window if the command is called from a script file.
- The command Wait can now handle millisecond delays.
- The function getKey can now wait for a key press for the specified time.
- Added out-of-bounds errors for commands Text, TextColor, Pxl_On, Pxl_Off, Pxl_change.
- Fixed height of letters in Text command from 16 to 18.
- Added string token clipping by ->.
- Removed assignment "Done" to Ans after commands completion.
- Added inverse hyperbolic functions (arsinh, arcosh, artanh).
- Fix trigonometry functions description.
- Fixed broken links from statistical variables to Readonly variables section.