Lhogho: The Real Logo Compiler

User Documentation

Author: Pavel Boytchev
Contributing authors: Peter Armyanov, Michael Downes

July 2013

Table of Contents

Chapter I. Introduction

1. General information

(a) About Lhogho

Lhogho is a compiler for the Logo language. It supports Logo programs with traditional number/word/list processing, turtle graphics, 3D graphics, OOP, parallel processes, etc. Well, it *will* support all this things when we finish it. Hopefully!

Why is it called *Lhogho*? One of the main goals was to find a name which sounded like Logo, but to be written in a way which no one else has seen before. We did test *Lhogho* with the major search machines (back in 2005) and we got 0 hits. Today (2013) we get 67000 hits. Unfortunately we do not know how to pronounce the name. Honestly. We asked several native English speakers, but they ricocheted back to us the same question. So far we pronounce it the same way as *Logo*, but with a slight double-wink. If you have any suggestions about pronunciation let us know.

Lhogho is developed by a team at Department of Mathematics and Informatics at Sofia Universit, led by Pavel Boytchev, Assoc. Prof., PhD.

(b) License

Lhogho is free software; you can redistribute it and/or modify it under the terms of the *GNU General Public License* as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program - see file `LICENSE.TXT`; if not, write to:

```
Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor,
Boston, MA  02110-1301,
USA
```

2. Quick start

(a) Getting Lhogho

Lhogho web site provides links to prebuilt Lhogho distributions, ready to start without compilation.

The distribution file for Windows is a zip file called `lhogho.zip` (the German version is `lhogho-de.zip`). To unzip the file use either Windows Explorer's unzipper or any 3rd party unzipper.

The distribution file for Linux is discontinued, as it appears that each Linux distro would need a different binary file. The option for Linux users is to download the developer's package (i.e. the Lhogho sources, `lhogho.src.zip`) and to compile Lhogho on their machines.

(b) Using Lhogho

The examples in this section are for Windows console window. For Linux console windows you may need to append `./` before the names of the executable programs.

If you run Lhogho without providing any inputs, it will shows its version, platform, language and production date:

```
D:\lhogho\>lhogho
LHOGHO - The LOGO Compiler [0.0.028, windows-i386(en), Jul
4 2013]
```

Lhogho comes with several sample source programs – `*.lgo` files. These are plain text files. Let us run `hello.lgo`:

```
D:\lhogho>lhogho hello.lgo
Hello world
```

Now let us use `primes.lgo` to print all prime numbers up to 60:

```
D:\lhogho>lhogho primes.lgo 60
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
```

And now up to 1000:

```
D:\lhogho>lhogho primes.lgo 1000
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97 101 103 107 109 113 127 131 137 139
149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281
```

```
283 293 307 311 313 317 331 337 347 349 353 359 367
373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691
701 709 719 727 733 739 743 751 757 761 769 773 787
797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 919 929 937 941 947 953 967 971
977 983 991 997
```

Because Lhogho is a compiler, it can build standalone executable files, which can be run without Lhogho. Here is how to compile `primes.lgo` into `primes.exe`:

```
D:\lhogho>lhogho -x primes.lgo
D:\lhogho>primes 60
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
```

In Windows the name of the executable file is `primes.exe`, while in Linux it is `primes`. Lhogho will remove the file name extension of the source file if it is one of these: `.lgo`, `.log`, `.lg`, `.logo`, `.lho` or `.lhogho`. If the extension is another one, then Lhogho will append `.exe` (for Windows) or `.run` (for Linux) to the name of the compiled program.

Executable files produced by the Lhogho could be fully-functional Lhogho compilers. Let us compile `hello.lgo` into a `hello.exe` and then use it to recompile `primes.lgo`:

```
D:\lhogho>lhogho -xc hello.lgo
D:\lhogho>hello -x primes.lgo
D:\lhogho>primes 60
Hello world
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
```

Note that `hello.exe` is already an executable file that contains a Lhogho compiler. It makes all compiled program to print “Hello World” at the beginning. This is like a custom-made Lhogho compiler.

(c) Non-English Lhogho

Lhogho is distributed in two versions – an English Lhogho and a German Lhogho. The German version has everything translated into German (i.e. primitives, error messages, compiler options, source examples and libraries) except for the user Documentation.

Lhogho outputs text as UTF-8. This is encoding which allows support for characters outside the range of the traditional Latin alphabet. When a non-English version of Lhogho is used in a console window, the text output may not look correct if the console does not support UTF-8.

A Windows XP/7 console can be switched in UTF-8 mode by `chcp.com` (Change codepage) command:

```
D:\lhogho>chcp.com 65001
Active code page: 65001
```

Also, in Windows XP/7 the console window should use a TrueType font. The default raster font can render only the basic Latin characters.

3. Scripting

Lhogho can be used as a scripting engine. The following sections describe various scenarios. Scripting for Windows and Linux is done in two conceptually different, but compatible ways. Thus, it is possible to make a source file executable directly from the command prompts of both Windows and Linux.

(a) Windows console

Lhogho can be defined as a default application for running `.lgo` files. The following two commands executed from the Windows command prompt associates `.lgo` files with `lhogho.exe`:

```
D:\lhogho>assoc .lgo=LhoghoScript
.lgo=LhoghoScript
D:\lhogho>ftype LhoghoScript=D:\lhogho\lhogho.exe
"%1" %*
LhoghoScript=D:\lhogho\lhogho.exe "%1" %*
```

Once the association is done, Lhogho sources can be executed “immediately”:

```
D:\lhogho>hello.lgo
Hello world
```

(b) Linux terminal

If the first line of source code is a shell comment pointing to the compiler, then the source file can be executed “immediately”. Consider a source file `hello.lgo` with the following contents:

```
#! ~/lhogho/lhogho
print [Hello world]
```

In Linux it can be executed in this way:

```
~/lhogho$ hello.lgo
Hello world
```

4. Compiling the compiler

If none of the prebuilt binaries work on your system try to recompile Lhogho and generate binaries. To do this first download the latest source package – it is platform independent and is called `lhogho.src.zip`.

Unpack the distributable and consult the `INSTALL.TXT` file which contains information how to recompile Lhogho, its documentation and how to create new distributable.

5. Additional information

Information about Lhogho as well as various resources can be accessed from its home page at `http://pavel.it.fmi.uni-sofia.bg/projects/lhogho`. To contact the authors, ask questions or share opinions visit the Lhogho Forum at `http://sourceforge.net/apps/phpbb/lhogho`, and the SourceForge portal is at `http://sourceforge.net/projects/lhogho`.

You are encouraged to report bugs in Lhogho, its libraries or documentation at `http://sourceforge.net/p/lhogho/bugs` and to request new features at `http://sourceforge.net/p/lhogho/feature-request`.

Chapter II. Syntax and Tokenization

1. Logo syntax

The source programs that Lhogho understands are ASCII or UNICODE text files that contain Logo instructions. The source text consists of *tokens*, which are specialized units of texts, like words and punctuation marks in sentences.

(a) Overview

The syntax of Logo programs is fairly simple. Usually tokens, that describe instructions, are separated by spaces, but some special punctuation tokens can be written next to each other. A typical syntax of a Logo instruction is:

`command` `param`$_1$ `param`$_2$ `param`$_3$ `…`

where `command` is a token, and `param`$_i$ are expressions made of tokens. Expressions have the same structure as instructions:

`operation` `param`$_1$ `param`$_2$ `param`$_3$ `…`

(b) Data types

Lhogho supports two Logo data types: *words* and *lists*.

Words are used to represent texts. For a token to be evaluated as a word, it must be preceded by double quotes:

```
"word
```

If a word is in a list of data, the double quotes are not necessary.

Numbers are special kind of words which are self-evaluable, i.e. it is not necessary to use double quotes:

```
3.14
"3.14
```

Finally, words are used to denote *Boolean values*, which are results of predicate functions (like `equal?`), or are used in conditional statements (like `if`). The words representing Boolean values are:

```
"true
"false
```

In the German Lhogho the Boolean values are `"wahr` and `"falsch`.

Lists are finite sequences of words and other lists. A sequence of tokens is evaluated as a list if it is enclosed in square brackets `[…]`. The square brackets are not elements of lists.

```
[a list of items]
[a list [of sublists]]
[ ]
```

(c) Parentheses

Logo is descendent of the programming language LISP and thus the parentheses play significant role in Logo programs. When parentheses are used, the opening parenthesis is placed before the first token of an instruction or an expression, and the closing parentheses – after the last token.

```
(command param₁ param₂ param₃ …)
```

This rule also applies to infix operations (i.e. operation with an input before the name of the operation):

```
(param₁ operation param₂)
```

Parentheses can be used for several reasons. One of them is to make source code clearer by visually grouping tokens that form a parameter. Except for beautification, this use of parentheses provides hints for the scope of each expression. The parentheses in the next example are not necessary:

```
print item (count :n) (word "abc :n)
```

Another purpose of parentheses is to change the order of calculations. If they are not used, an expression may still be syntactically valid, but will produce another result. The next expressions will produce the values *(a+5)(b-10)* and *sin(30)+10*. Without parentheses, the values will be *a+5b-10* and *sin(30+10)*.

```
(:a+5)*(:b-10)
(sin 30)+10
```

Finally, parentheses are used to force execution of instructions and calculation of expressions that have number of inputs different from the default one. The next example shows the function `word` which is forced to process four inputs. Without parentheses, `word` will process only two inputs.

```
(word "a "b "c :n)
```

(d) Programming entities

A typical feature of Logo is that data and program are expressed in the same way – by sequences of tokens. Depending on the context, Lhogho decides how to process any particular token or a group of tokens.

Text literals are tokens which first character is double quotes `"`. Such tokens are considered by Lhogho as text literals (the double quotes are excluded from the literal).

```
"sample
```

To define a token with special characters or punctuation see section *Tokenization of data*.

If the first character of a token is colon `:`, then the rest of the token is considered as a name of a variable and Lhogho extracts its value The next one-token expression returns the value of variable called “sample”:

```
:sample
```

In other cases, if the token is not punctuation, then it is considered as name of a command to execute or operation to evaluate.

Logo can group tokens in larger structures called *lists*. The list is a sequence of tokens or other lists framed in square brackets. Semantically, a list can represent a sequence of words as well as a sequence of instructions.

```
[list of tokens]
```

(e) User commands and operations

The tokens `to` and `end` are used to define a new command or operation. The syntax of such definitions starts with a header line describing the name of the command and the names of the formal inputs. In the next example the name of the command is `sample` and there are two inputs called `param` and `state`:

```
to sample :param :state
   :
end
```

The instructions that represent the essence of the command are placed following the header line. The end of the definition is the token `end`.

The token `learn` is a synonym of `to` and can be used together with `end`:

```
learn sample :param :state
   :
End
```

The German equivalents of `to…end` and `learn…end` are called `pr…ende` and `lerne…ende`.

(f) Variable number of inputs

Lhogho allows the definition of local commands and operations. In the following example function `fib` is local to function `fibonacci` and is only accessible within its scope:

```
to fibonacci :x
   to fib :x
      if :x<2 [output :x] [output (fib :x-1)+(fib :x-
2)]
   end
   if :x<0 [(throw "error [Invalid input to
fibonacci])]
   output fib :x
end

print fibonacci 10
print fibonacci -4
```

Lhogho allows the definition of commands and operations with undefined number of inputs. Typically, Lhogho will generate an error message if a command in used with more or less inputs that the defined one. If the list of formal inputs ends with “`...`” then it is possible to provide more or less actual inputs:

```
to average :a ...
   local "sum
   make "sum 0
   repeat inputs [make "sum :sum+input repcount]
   output :sum/inputs
end
print (average 1 2 3 4 5)
print (average -1 1 3 5)
```

Note: For exemplary definitions of functions `input` and `inputs` see the documentation of `_stackframe` and `_stackframeatom`.

(g) Prefix, infix and postfix notations

The order of formal inputs of a user-defined command or operation determines whether it is prefix, infix or postfix. The following example defines a prefix operation for square cube $\sqrt[3]{x}$, infix operation for binomial coefficients $\binom{n}{k}$, and postfix operation for factorial $n!$:

```
to sqrt3 :x
   output power :x 1/3
end
to :n over :k
   output (:n !)/(:k !)/(:n-:k !)
end
to :n !
   if :n<2 [output 1] [output :n*(:n-1 !)]
end


print sqrt3 27
print 5 over 3
print 5 !
```

2. Tokenization of data

Tokenization is the process of splitting Logo source code into tokens. Generally Logo tokenizes sequences of characters in two different ways – *data* and *command tokenization*, depending whether the input is expected to contain data or commands. These two tokenizations occur implicitly - i.e. the Logo implies them automatically.

Tokenization of data is the process of splitting text containing data into tokens. It is weaker than the command tokenization, because `2+3` is considered as one word in data sequences, and three words in command sequences.

Characters can be classified into three categories: *special*, *ordinary* and *whitespaces*. Special characters are those which have special meaning and treatment. Whitespaces are the invisible characters like spaces and tabs. All other characters are ordinary.

The general rules for data tokenization are:

- Whitespaces are delimiters of words.
- New Line character is a delimiter of lines.

- Brackets `[` and `]` are tokens by themselves.

Spaces and tabs are the most common token delimiters. Two or more of them in a row are considered as a single delimiter.

```
print [Spaces    and     Tabs]
Spaces and Tabs
```

Newline characters are also considered as whitespaces in data tokenization (this is not true for command tokenization).

Square brackets `[` and `]` are considered as single-character tokens. They are essential part of the Logo syntax for representing lists.

```
print [ List [ of [    ] words ] ]
List [of [] words]
```

The fragment in the brackets `[ ]` is created as a sublist. Thus the tokens of square brackets do not appear as elements of the list.

(a) Comments

Comments are fragments of the program which are ignored. Lhogho provides two forms of comments: *line* and *shell* comments.

Lhogho uses a semicolon `;` to comment the text till the end of the line excluding the new line character. These are called *line comments*. Some special characters in a comment loose their properties. For example brackets `[ ]` and bars `|` ... `|` are treated as a part of the comment.

```
print "Hello ;world
Hello
```

Backslash `\` and tilde `~` characters keep their specialty in line comments.

```
print "Hello ;world\
print "again
Hello
```

Shell comments are lines starting with `#!` in a Logo program which are intended to be processed by the command shell of the operating system. Lhogho treats these lines as comments.

```
#! /usr/local/bin/logo
(print "Shell "comment)
Shell comment
```

Not all operating systems recognize shell comments, e.g. `#!` does not work under MS DOS and Windows.

(b) Line continuation

A line can be continued onto the next line if its last visible character is tilde `~`. This is often used when a line too long.

Placed at the end of a line a tilde `~` makes it continue into the next line. Whitespaces after the tilde are ignored. If there are other characters between the tilde and the new line, then the tilde it is treated as ordinary character and the whitespaces after it (if any) are not ignored:

```
print "Long~
word
Longword
```

A line with a line comment can still be continued with a tilde `~`:

```
print "Really; Yes!~
long; comment ~
word
Reallylongword
```

(c) Backslashes

In many cases it is needed to include characters in a word which are otherwise treated as special. Lhogho does this with bars `|`...`|` and backslashes `\`.

To include an otherwise delimiting character (including semicolon or tilde) in a word, precede it with backslash `\`. To include a backslash in a word, use `\\`.

Backslashes turn other characters into ordinary ones - spaces, square brackets, bars, semicolons, tildes and other backslashes.

```
print "Back\\slashed\ word
Back\slashed word
print "Bracket\[word
Bracket[word
```

If the last character of a line is a backslash, then the newline character following the backslash will be part of the last word on the line, and the line continues onto the following line.

```
print "Two-line\
word
Two-line
word
```

If the new line character at the end of a comment is backslashed, then it becomes a part of the comment together with the next line.

```
print "one;comment\
print "two
one
print "three
three
```

If a tilde is backslashed in a comment it becomes a part of the comment, and the new line character is not ignored.

```
print "one;comment\~
one
print "two
two
print "three
three
```

(d) Bars

Bars are used when whitespaces or new lines must be included in a word. Inside bars all special characters except the backslash become ordinary characters. To include a bar inside bars use `\|`.

When bars are next to a word, their contents is a part of the word too. The bars themselves are not a part of the tokenized word.

```
print "|bars and spaces|
bars and spaces
print "bar|s and
new lines|
bars and
new lines
```

All special characters except backslash `\` become ordinary when placed in bars. For example, comments and line continuations are not available inside bars as shown in the next case:

```
print |bar;red~
comment|
bar;red~
comment
```

The only way to include a bar inside bars is to backslash it.

```
print [|bars in |..| bars|]
bars in .. bars
print [|bars in \|..\| bars|]
bars in |..| bars
```

3. Tokenization of commands

Tokenization of commands is the process of splitting text containing Logo commands into tokens. This tokenization is differs from tokenization of data because it has additional rules:

- Parentheses are delimiters.
- Mathematical operators are partial delimiters.

(a) Special characters

Special characters like `"` and `:` are not delimiters. For example words containing them are parsed together with them. Lhogho processes `"` and `:` later on, during compilation. Words after `"` are delimited by `[`, `]`, `(`, `)` or whitespace.

```
print "1+2
1+2
```

Words not after `"` are delimited by `[`, `]`, `(`, `)`, whitespace or any of the infix operators `+`, `-`, `*`, `/`, `=`, `<`, `>`, `<=`, `>=`, `<>`. Words starting with `:` fall into this category.

```
print 1+2
3
```

(b) Parentheses

Parentheses are delimiters. They are processed as single-character tokens, but they do not appear in the resulting abstract tree. To include a parenthesis in a word use backslash `\`.

```
print "\(abc\)
(abc)
```

(c) Infix operators

Each infix operator character is a token in itself, except that the two-character sequences `<=`, `>=` and `<>` with no intervening space are recognized as a single token.

Note: Tokenization of infix operators depends on the special character `"`.

(d) Templates

A non-backslashed question mark followed by a number is tokenized into a sequence of four tokens.

```
print runparse [1+?37]
1 + ( ? 37 )
print runparse [1+\?37]
1 + ?37
```

Chapter III. Primitives

1. Numerical operations

Numerical operations are functions and operators which process numbers (either integer or floating-point). Integer numbers can be represented in decimal radix and in hexadecimal radix (with the prefix `0x`, e.g. `0xFF`).

(a) Arithmetics

```
value + value
      + value

value – value
      - value

value * value

value / value
```

Arithmetic operators are used to add, subtract, multiply or divide numbers. They correspond to the basic mathematical operators `+`, `-`, `*` and `/`. The `+` and `–` operators can be binary or unary, while `*` and `/` are only binary.

```
print (1+3)*(7-4)
12
print (1+3)/(5-3)
2
```

Division can easily produce large numbers especially if the second input is close to zero or is zero. When a number becomes too big it is reported as being *infinity* (`INF`).

```
print -3/0
-inf
print 5/inf
0
```

```
sum :value :value                                    DE: summe
(sum :value :value :value … )
```

Function. Outputs the sum of its inputs. Can be called with arbitrary count of arguments.

```
print (sum 1 2 3)
6
```

```
difference :value :value                          DE: differenz
```

Function. Outputs the difference of its inputs.

```
print difference 1 2
-1
```

```
minus :value
```

Function. Outputs the negative of its input.

```
print minus 3
-3
print minus -4
4
```

```
 product :value :value                             DE: produkt
(product :value :value :value … )
```

Function. Outputs the product of its inputs. Can be called with arbitrary count of arguments.

```
print product 4 5
20
print (product 1 2 3)
6
```

```
quotient :value :value
```

Function. Outputs the quotient of its inputs.

```
print quotient (1+3) (5-3)
2
```

Division can easily produce large numbers especially if the second input is close to zero or is zero. When a number becomes too big it is reported as being *infinity* (`INF`).

```
print quotient 3 0
inf
```

```
print quotient -3 0
-inf
```

`remainder :value :value` DE: `rest`

Function. Outputs the remainder on dividing its arguments. Both must be integers and the result is an integer with the same sign as first one.

```
print remainder 2 3
2
print remainder 5 -2
1
print remainder -5 2
-1
```

(b) Rounding

`int :value`

Function. Outputs its input with fractional part removed, i.e. an integer with the same sign as the input, whose absolute value is the largest integer less than or equal to the absolute value of the input.

```
print int 5.5
5
print int -5.3
-5
```

`round :value` DE: `runde`

Function. Outputs the nearest integer to the input.

```
print round 5.3
5
print round 5.5
6
print round -5.5
-6
```

(c) Exponential and logarithmic functions

`sqrt :value` DE: `qw`

Function. Outputs the square root of the input, which must be nonnegative.

```
print sqrt 4
2
print sqrt 5
2.236068
```

`power :value :value` `DE:potenz`

Function. Outputs its first argument to the power of second argument. If first is negative, then second must be an integer.

```
print power 2 2
4
print power 4 0.5
2
```

`exp :value`

Function. Outputs `e=2.718281828…` to the input power.

```
print exp 1
2.718282
print exp -1
0.367879
```

`log10 :value`

Function. Outputs the common logarithm of the input.

```
print log10 100
2
print log10 12345
4.091491
```

`ln :value`

Function. Outputs the natural logarithm of the input.

```
print ln 10
2.302585
```

`abs :value`

Function. Outputs the absolute value of the input.

```
print abs 5
5
```

```
print abs -5
5
```

(d) Trigonometric functions

```
pi
```

Function. Outputs the number π.

```
print pi
3.141593
```

```
sin :value

cos :value
```

Functions. Output the sine or the cosine of their inputs, which are taken in degrees.

```
print sin 45
0.707107
print cos 10
0.984808
```

```
radsin :value

radcos :value
```

Functions. Output the sine or the cosine of their inputs, which are taken in radians.

```
print radsin (pi/4)
0.707107
print radcos (-pi/2)
0
```

```
 arctan :value
(arctan :value :value)
```

Function. Outputs the arctangent, in degrees, of its input. If there are two inputs outputs the arctangent in degrees of `y/x`, where `x` is first argument of function, and `y` is second.

```
print arctan sqrt 2
54.73561
print (arctan 1 1)
45
```

```
 radarctan :value
(radarctan :value :value)
```

Function. Outputs the arctangent, in radians, of its input. If there are two inputs outputs the arctangent in radians of `y/x`, where `x` is first argument of function, and `y` is second.

```
print radarctan sqrt 2
0.955317
print (radarctan 1 1)
0.785398
```

(e) Random numbers

```
 random :max                                  DE: zufallszahl, zz
 random :list
(random :min :max)
```

Function. If called with one argument and argument is a number then outputs a random number between `0` and `max`.

If called with one argument list, outputs a randomly selected element of the list. This functionality is available only when Lhogho is in extended, non-traditional mode.

If called with two arguments, outputs an integer number between `min` and `max` inclusive. Value `min` must be nonnegative and less or equal to `max`.

```
print random 4
0
print random [1 2 3]
3
print (random 4 6)
5
```

```
 rerandom                                    DE: startezufall, sz
(rerandom :num)
```

Command. Makes the results of `random` reproducible. Usually the sequence of random numbers is different each time Lhogho is started, unless `rerandom` is used.

If called with no arguments, sets same sequence each time. If you need the more than one sequence of pseudo-random numbers repeatedly, you can give `rerandom` an integer input which selects a unique sequence of pseudo-random numbers.

```
rerandom
(print random 4 random 4 random 4)
2 0 3
rerandom
(print random 4 random 4 random 4)
2 0 3
```

(f) Sequences

Functions for generating sequences return a list of numbers within a given range. These functions could be entirely written in Lhogho, but are also defined as primitives for higher performance.

`iseq :from :to`

Function. Outputs a list of the integers between `from` and `to`, inclusive.

```
print iseq 5 10
5 6 7 8 9 10
```

`rseq :from :to :count` DE: `lseq`

Function. Outputs a list of `count` equally spaced rational numbers between `from` and `to`, inclusive.

```
print rseq 5 3 5
5 4.5 4 3.5 3
```

(g) Operations with bits

`lshift :number :bits` DE: `lwechsel`

Function. Outputs `number` logical-shifted to the left by `bits` bits. If `bits` is negative, the shift is to the right with zero fill. Both inputs must be integers.

```
print lshift 1 2
4
print lshift 16 -2
5
```

`ashift :number :bits` DE: `awechsel`

Function. Outputs `number` arithmetic-shifted to the left by `bits` bits. If `bits` is negative, the shift is to the right with sign extension. Both inputs must be integers.

```
print ashift 1 2
4
print ashift -16 -2
-4
```

```
 bitand :value :value                                DE: bitund
(bitand :value :value :value … )
```

Function. Outputs the bitwise *and* of its inputs, which must be integers.

```
print bitand 7 12
4
```

```
 bitor :value :value                                 DE: bitoder
(bitor :value :value :value … )
```

Function. Outputs the bitwise *or* of its inputs, which must be integers.

```
print bitor 7 12
15
```

```
 bitxor :value :value                                DE: bitxoder
(bitxor :value :value :value … )
```

Function. Outputs the bitwise *exclusive or* of its inputs, which must be integers.

```
print  bitxor 7 12
11
```

```
bitnot :value                                        DE: bitnicht
```

Function. Outputs the bitwise *not* of its input, which must be integer.

```
print bitnot 3
-4
```

2. Predicates and Boolean operations

(a) Compare predicates

Predicates are functions and operators which are used to test if their input parameters has specific properties or are in specific relations. Predicates return Boolean values as result.

```
:value = :value

 equalp :value :value                                   DE: gleichp
 equal? :value :value                                   DE: gleich?
```

Operator and function. Outputs `true` if the inputs are equal, `false` otherwise. Two numbers are equal if they have the same numeric value. Two non-numeric words are equal if they contain the same characters in the same order. If there is a variable named `caseignoredp` whose value is `true`, then an upper case letter is considered the same as the corresponding lower case letter which is the case by default. Two lists are equal if their members are equal.

```
print 5 = "5
true
print equal? "One "two
false
```

```
:value <> :value

 notequalp :value :value                              DE: ungleichp
 notequal? :value :value                              DE: ungleich?
```

Operator and function. Outputs `false` if the inputs are equal, `true` otherwise.

```
print 5 <> "5
false
print notequal? "One "two
true
```

```
:value < :value

 lessp :value :value                                   DE: kleinerp
 less? :value :value                                   DE: kleiner?
```

Operator and function. Outputs `true` if its first input is strictly less than its second. Inputs must be numbers.

```
print 5 < 10
true
print less? 5 5
false
```

```
:value > :value

 greaterp :value :value                             DE: größerp
 greater? :value :value                             DE: größer?
```

Operator and function. Outputs `true` if its first input is strictly greater than its second. Inputs must be numbers.

```
print 15 > 10
true
print greater? 5 5
false
```

```
:value <= :value

 lessequalp :value :value                      DE: kleinergleichp
 lessequal? :value :value                      DE: kleinergleich?
```

Operator and function. Outputs `true` if its first input is less than or equal to its second. Inputs must be numbers.

```
print 5 <= 10
true
print lessequal? 5 5
true
```

```
:value >= :value

 greaterequalp :value :value                   DE: größergleichp
 greaterequal? :value :value                   DE: größergleich?
```

Operator and function. Outputs `true` if its first input is greater than or equal to its second. Inputs must be numbers.

```
print 5 >= 10
false
print greaterequal? 5 5
true
```

```
beforep :value :value                               DE: vorherp
before? :value :value                               DE: vorher?
```

Function. Outputs `true` if first argument comes before second in ASCII collating sequence. Case-sensitivity is determined by the value of `caseignoredp`.

Note: if the inputs are numbers, the result may not be the same as with `less?`.

```
print beforep 3 12
```

```
false
print before? "one "two
true
```

(b) Type predicates

```
wordp :value                                   DE: wortp
word? :value                                   DE: wort?
```

Function. Outputs `true` if the input is a word, `false` otherwise.

```
print wordp "123
true
print word? [123]
false
```

```
listp :value                                   DE: listep
list? :value                                   DE: liste?
```

Function. Outputs `true` if the input is a list, `false` otherwise.

```
print listp "123
false
print list? 123
true
```

```
numberp :value                                 DE: zahlp
number? :value                                 DE: zahl?
```

Function. Outputs `true` if the input is a number, `false` otherwise.

```
print numberp 123
true
print number? [123]
false
```

```
emptyp :value                                  DE: leerp
empty? :value                                  DE: leer?
```

Function. Outputs `true` if the input is the empty list or the empty word, `false` otherwise.

```
print emptyp "123
false
print empty? [123]
true
```

```
backslashedp :char                                  DE: backslashp
backslashed? :char                                  DE: backslash?
```

Function. Outputs `true` only if the input is a character which has been backslashed or barred. Characters which do not need to be backslashed or barred are always reported as non-backslashed even if they were actually backslashed. The backslashable characters are: `+`, `-`, `*`, `/`, `=`, `<`, `>`, `(`, `)`, `|` and`?` .

```
print backslashed? item 4 "123-456\-789
false
print backslashed? item 8 "123-456\-789
true
```

(c) Inclusion predicates

```
memberp :elem :value                             DE: elementp, elp
member? :elem :value                             DE: element?, el?
```

Function. If `value` is a list, outputs `true` if `elem` is `equal?` to any member of `value`, `false` otherwise. If `value` is a word, outputs `true` if `elem` is a one-character word `equal?` to a character of `value`, `false` otherwise.

```
print member? 345 [123 345 567]
true
print memberp "a 123
false
```

```
substringp :text1 :text2
substring? :text1 :text2
```

Function. Outputs `true` if `text1` is a substring of `text2`. If inputs are not words outputs `false`.

```
print substringp 123 456123456
true
```

(d) Logical functions

```
 and :value :value                                         DE: und
(and :value :value :value … )

 all? :value :value                                       DE: alle?
(all? :value :value :value … )
```

Function. Outputs `true` if all the inputs are `true`, `false` otherwise.

```
print and 1 < 2 3 = 3
```

```
true
print (and 1 < 2 3 <> 4 5 = 5)
true
```

```
 or :value :value                                        DE: oder
(or :value :value :value … )

 any? :value :value                                      DE: eines?
(any? :value :value :value … )
```

Function. Outputs `false` if all the inputs are `false`, `true` otherwise.

```
print or 1 > 2 3 <> 3
false
print (or 1 < 2 3 = 4 5 = 5)
true
```

```
not :value                                              DE: nicht
```

Function. Outputs `false` if argument is `true`, true if argument is `false`.

```
print not (1 > 2)
true
```

3. Word and list operations

(a) Selectors

Selectors are functions that extract part of its input. The input must be word or list.

```
first :value                                        DE: erstes, er
```

Function. If the input is a word, outputs the first character of the word. If the input is a list, outputs the first member of the list.

```
print first 123
1
print first [abc xyz]
abc
```

```
firsts :list                                    DE: alleerstes, aer
```

Function. Outputs a list containing the `first` of each member of the input list. It is an error if any member of the input list is empty. The input itself may be empty, in which case the output is also empty.

```
print firsts [123 456 789]
1 4 7
```

```
butfirst :value                                DE: ohneerstes
bf :value                                              DE: oe
```

Function. If the input is a word, outputs a word containing all but the first character of the input. If the input is a list, outputs a list containing all but the first member of the input.

```
print butfirst 123
23
print bf [abc xyz klmn]
xyz klmn
```

```
butfirsts :list                            DE: ohneerstesalle
bfs :list                                             DE: oea
```

Function. Outputs a list containing the `butfirst` of each member of the input list. It is an error if any member of the input list is empty. The input itself may be empty, in which case the output is also empty.

```
print butfirsts [123 456 789]
23 56 89
```

```
last :value                                   DE: letztes, lz
```

Function. If the input is a word, outputs the last character of the word. If the input is a list, outputs the last member of the list.

```
print last 123
3
print last [abc xyz]
xyz
```

```
butlast :value                                DE: ohneletztes
bl :value                                              DE: ol
```

Function. If the input is a word, outputs a word containing all but the last character of the input. If the input is a list, outputs a list containing all but the last member of the input.

```
print butlast 123
12
print bl [abc xyz klmn]
abc xyz
```

`item :index :value` DE: `element, el`

Function. If `value` is a word, outputs the `index`-th character of the word. If `value` is a list, outputs the `index`-th member of the list. `index` starts at 1.

```
print item 1 "abc
a
print item 2 [abc xyz klmn]
xyz
```

`member :elem :value` DE: `elementab`

Function. If `value` is a word, outputs a subword starting from the first occurrence of `elem` to the end or empty word if `elem` is not member of `value`. If `value` is a list, outputs a new list containing elements of `value` starting from the first occurrence of `elem` to the end or empty list it `elem` is not member of `value`.

```
print member "e "Test
est
print member 2 [1 2 3]
2 3
```

`substring :text1 :text2`

Function. Outputs the position of `text1` in `text2` or outputs 0 if `text1` is not a substring of `text2`. Both inputs must be words.

```
print substring [ope] "onomatopeia
7
print substring "a "onomatopeia
5
print substring "b "onomatopeia
0
```

`pick :list` DE: `picke`

Function. Outputs randomly selected element of its input, which must be a list.

```
print pick [1 2 3]
2
print pick [1 2 3]
1
```

```
remdup :value                                                    DE: entfdup
```

Function. Outputs a copy of `value` with duplicate members removed. If two or more members of the input are equal, the rightmost of those members is the one that remains in the output.

```
print remdup [1 2 1 3 2 1 4 2 5 1 6 1]
3 4 2 5 6 1
print remdup "121321425161
342561
```

```
remove :elem :value                                              DE: entferne
```

Function. Outputs a copy of `value` with every member equal to `elem` removed.

```
print remove 1 [1 2 1 3 2 1 4 2 5 1 6 1]
2 3 2 4 2 5 6
print remove 1 121321425161
2324256
```

(b) Constructors

```
 word :value :value                                              DE: wort
(word :value :value :value … )
```

Function. Outputs a word formed by concatenating its inputs.

```
print word "Hello "-World
Hello-World
```

```
 list :value :value                                              DE: liste
(list :value :value :value … )
```

Function. Outputs a list whose members are its inputs, which can be any word or list.

```
print list "test 123
test 123
print (list [123] [123 123] "123)
[123] [123 123] 123
```

```
 sentence :value :value                                          DE: satzbilden
(sentence :value :value :value … )
 se :value :value                                                DE: satz
(se :value :value :value … )
```

Function. Outputs a list whose members are its word-inputs that are words and the members of its list-inputs.

```
print (se [12] [34 56] "78)
12 34 56 78
```

```
lastput :value1 :value2                    DE: mitletztem
lput :value1 :value2                               DE: ml
```

Function. If the second input is a list outputs a list equal to its second input with one extra member, the first input, at the end. If the second input is a word, then the first input must be a one-letter word, outputs word equal to second argument, but with first input appended to the end.

```
print lput 1 123
1231
print lput [1 2 3] [1 2 3]
1 2 3 [1 2 3]
```

```
firstput :value1 :value2                    DE: miterstem
fput :value1 :value2                                DE: me
```

Function. If the second input is a list outputs a list equal to its second input with one extra member, the first input, at the beginning. If the second input is a word, then the first input must be a one-letter word, outputs word equal to second argument, but with first argument inserted at the beginning.

```
print fput 1 123
1123
print fput [1 2 3] [1 2 3]
[1 2 3] 1 2 3
```

```
reverse :value                              DE: umkehrung
```

Function. If `value` is a list, outputs a list whose members are the members of the input list, in reverse order. Otherwise outputs a word with the reversed order of characters of `value`.

```
print reverse [1 2 3 4 5 6 7]
7 6 5 4 3 2 1
print reverse "abcde
edcba
```

```
combine :value1 :value2                   DE: kombinieren
```

Function. If `value2` is a word, works like `word :value1 :value2`. If `value2` is a list, works like `fput :value1 :value2`.

```
print combine 1 [1 2 3]
```

```
1 1 2 3
print combine "Hello "-World
Hello-World
```

`gensym` `DE: gisym`

Function. Outputs a unique word each time it's invoked. The words are of the form `G1`, `G2`, etc.

```
print gensym
G1
print gensym
G2
```

`quoted :value` `DE: zitiert`

Function. If `value` is a list outputs `value` otherwise outputs `value` with quotation mark prepended.

```
print quoted 123
"123
```

(c) Transformers

`count :value` `DE: länge, anzahl`

Function. Outputs the number of characters in `value`, if it is a word; or the number of members, if it is a list;

```
print count "Test
4
```

`char :value` `DE: zeichen`

Function. Outputs the character represented in the ASCII code by `value`, which must be an integer between 0 and 255.

```
print char 67
C
```

`ascii :value` `DE: asc`

Function. Outputs an integer (between 0 and 255) that represents the input character `value` in the ASCII code. Interprets some control characters as representing punctuation in bars `|…|`, and returns the character code for the corresponding punctuation character itself without vertical bars.

```
print ascii "a
```

```
97
```

`rawascii :value` DE: `ascii`

Function. Outputs an integer (between 0 and 255) that represents the input character `value` in the ASCII code.

```
print rawascii "|(|
14
```

`uppercase :value` DE: `großschrift, groß`

Function. Outputs a copy of the input word, but with all lowercase letters changed to the corresponding uppercase letters.

```
print uppercase "Test
TEST
```

`lowercase :value` DE: `kleinschrift, klein`

Function. Outputs a copy of the input word, but with all uppercase letters changed to the corresponding loweracase letters.

```
print lowercase "Test
test
```

(d) Formatting

`form :num :width :precision`

Function. Outputs a word containing a printable representation of `num`, possibly preceded by spaces, with at least `width` characters, including exactly `precision` digits after the decimal point. If `precision` is -1 interprets `width` like format string.

```
(print "! form 3.1415926 20 5)
!               3.14159
(print "! form 3.1415926 "|%.15lf| -1)
! 3.141592600000000
```

`format :data :format`

Function. Outputs a word containing a printable representation of `data`, according to `format` string. For a list of supported date and time format string see *Format strings* at page 119.

```
(print "! format 1234567 "%.8X )
! 0012D687
```

`formattime :data :format` DE: `zeitformat`

Function. Outputs a word containing a printable representation of `data`, according to `format` string. The `data` is an integer number containing time measured in seconds elapsed since 00:00:00 on January 1, 1970. For a list of supported date and time format string see *Format strings* at page 119.

```
make "time first filetimes "lhogho.exe
print formattime :time "|%d-%b-%Y %H:%M:%S|
23-Jan-2012 13:12:16
```

`timezone` DE: `zeitzone`

Function. Outputs the number of seconds between the local time and the corresponding GMT time. The number is positive for time zones ahead of GMT, and negative otherwise. For example, if the system's time zone is GMT+2, then the function returns 7200 (=2*60*60)

```
print timezone
7200
```

4. Control structures

Control structures determine how user program is executed – this includes loops, conditional execution, passing result from callee to caller, and so on.

(a) Conditional execution

`if :condition :command-list` DE: `wenn`
`if :condition :command-list :command-list`

Command. If the `condition` is `true` then `if` executes the first command-list. If the condition is `false` and there is a second command-list, then `if` executes it.

```
to neg? :x
   if :x=0
     [ (print :x [is zero]) ]
   if :x<0
     [ (print :x [is negative]) ]
     [ (print :x [is not negative]) ]
end
neg? 5
5 is not negative
```

`if :condition :expression-list :expression-list` DE: `wenn`

Function. When used as a function `if` has three inputs. The last two are lists containing an expression each. The value of one of these expressions is the output of the `if` function.

```
repeat 4 [print repcount*if repcount>2 [1] [-1]]
-1
-2
3
4
```

`ifelse :condition :command-list :command-list` DE: `wennsonst`

Command. The command `ifelse` is equivalent to the command `if` and the only difference is that `ifelse` expects exactly two command-lists, while `if` accepts either one or two.

`ifelse :condition :expression-list :expression-list` DE: `wennsonst`

Function. The function `ifelse` is equivalent to the function `if`.

`test :condition`

Command. The command `test` remembers the condition which must be either `true` or `false`. The condition is later used by commands `iftrue` and `iffalse`.

```
make "a sin 45
test :a>0.5
iftrue [print [sin(45) > 0.5]]
sin(45) > 0.5
```

`iftrue :commands` DE: `wennwahr, ww`

Command. Executes the `commands` if the input of the latest `test` command within the current procedure was `true`.

`iffalse :commands` DE: `wennfalsch, wf`

Command. Executes the `commands` if the input of the latest `test` command within the current procedure was `false`.

(b) Loops

`repeat :count :command-list` DE: `wiederhole, wh`

Command. Executes the `command-list` `count` number of times. The number of repetitions must be an integer number from 0 to 2147483647 inclusive. In case of 0 the `command-list` is not executed.

```
make "a 1
repeat 3
[
  (print (word :a "* :a) "= :a*:a)
  make "a :a+1
]
1*1 = 1
2*2 = 4
3*3 = 9
```

`repcount` DE: `whzahl`

Function. It is used only inside `repeat` loop and returns the iteration number of this loop. The first iteration is number 1.

```
repeat 3 [print repcount]
1
2
3
```

`forever :command-list` DE: `andauernd`

Command. Executes the `command-list` forever. This infinite cycle can be exited with `stop` or `output` commands only.

```
make "a 1
forever
[
   print int :a
   if :a>100 [output]
   make "a pi*:a
]
1
3
9
```

```
31
97
306
```

`for :var :limits :command-list` DE: `für.bis`

Command. Executes the `command-list` predefined number of times. The first input must be a word literal, which will be used as the name of a control variable. The second input must be a list of two or three expressions – the starting value of the control variable, the limit value of the variable; and optionally the step size. If the third member is missing, the step size will be 1 or -1 depending on whether the limit value is greater than or less than the starting value, respectively. The third input is a list of commands. The effect of `for` is to run `command-list` repeatedly, assigning a new value to the control variable each time.

```
for "i [1 4]
[
  (type :i "-)
  for "j [1 :i] [type :j]
  print
]
1-1
2-12
3-123
4-1234
```

`while :condition :command-list` DE: `.solange`

Command. Executes the `command-list` while the `condition` is true. If the first evaluation of the `condition` is `false`, then the `command-list` is not executed at all.

```
make "a 1
make "n 1
while :a<10
[
  (print word "2^ :n "= :a)
  make "a 2*:a
  make "n :n+1
]
2^1 = 1
```

```
2^2 = 2
2^3 = 4
2^4 = 8
```

`do.while :command-list :condition` DE: `führeaus.solange`

Command. The command `do.while` is the same as `while`, only the order of inputs is reversed and the `command-list` is executed once before the first check of the `condition`.

`until :condition :command-list` DE: `.bis`

Command. Executes the `command-list` until the `condition` becomes `true`. If the first evaluation of the `condition` is true, then the `command-list` is not executed at all.

```
make "a 1
make "n 1
until :a>=10
[
  (print word "2^ :n "= :a)
  make "a 2*:a
  make "n :n+1
]
2^1 = 1
2^2 = 2
2^3 = 4
2^4 = 8
```

`do.until :command-list :condition` DE: `führeaus.bis`

Command. The command `do.until` is the same as `until`, only the order of inputs is reversed and the `command-list` is executed once before the first check of the `condition`.

(c) Execution

`run :instructions` DE: `tue`

Command and function. Runs the `instructions` which must be a list. If the instructions are commands, then return nothing. Otherwise, if the `instructions` is a list containing an expression, evaluate it and return its value.

```
run [print 2*3]
6
print run [2*3]
6
```

`runresult :instructions` DE: `tuewert`

Function. Runs the `instructions` which must be in a list. If the instructions are commands, then return an empty list. Otherwise, if the `instructions` is a list containing an expression, evaluate it and return a list containing its value.

```
make "x 10
print runresult [print :x*:x]
100

print runresult [:x*:x]
100
```

`runmacro :instructions` DE: `tuemakro`

Command. Runs the `instructions` which must be a list. Variables, functions and commands created as local entities inside `runmacro` persist after the end of the execution of `instructions`.

```
runmacro [
  local "a
  make "a 5
  to double :x
    output 2*:x
  end ]
run [
  print :a
  print double 4
]
5
8
```

`load :filename` DE: `lade`

Command. Runs the instructions in a text file with a given `filename` as if they are included in the place of the `load` command. The `filename` may include relative or absolute path. If there is no path then the file is searched in the current folder. If there is relative path, it is relative to the current folder. If the file is not found, then it is searched in the `lib` subfolder of the folder where the compiler is located.

Note: Lhogho processes the `load` command during parsing, i.e. before actually running the program. This requires that the filename is a literal constant.

If `LIB.LGO` contains these commands :

```
make "libver "1.2beta
to max :a :b
   if :a>:b [output :a] [output :b]
end
```

then it can be loaded by `load` command:

```
load "lib.lgo
(print "Version :libver)
Version 1.2beta
print max 6 10
10
```

(d) Exits and tags

```
output :value                                          DE: rückgabe
op :value                                                    DE: rg
```

Command. Ends the execution of the current function and returns to the caller the value of its input. Note that `output` is not a function and it does not return a value – it forces the current function to end and return a value.

```
to mysqr :x
   output :x*:x
end
print mysqr 2
4
```

```
maybeoutput :source                              DE: magseinrückgabe
```

Function or Command. Ends the execution of the current function and returns to the caller the value of its input (if any). If `source` is an expression, then `maybeoutput` acts like `output`. If `source` is not an expression, then `maybeoutput` acts like `stop`.

```
to func :pattern :x
   maybeoutput run :pattern
```

```
end
print func [:x*sin :x] 30
15
func [print :x*sin :x] 30
15
```

`stop` DE: `rückkehr, rk`

Command. Ends the execution of the current function without passing any return value.

```
to mysqr :x
   (print "x "= :x*:x)
   stop
   print [Beyond stop]
end
mysqr 2
x = 4
```

`bye` DE: `ade`

Command. This command is used to terminate program execution.

```
print [Before bye]
Before bye
bye
print [Never reach this code]
```

`tag :word` DE: `schildchen`

Command. The `tag` command defines a place within the current procedure. This place can be used by the `goto` command to change the execution path. The name of the tag must be a quoted word. Tags are always local to the procedure where they are defined.

```
make "a 1
tag "again
make "a 3*:a
if :a<30 [ goto "again ]
3
9
27
```

`goto :word` DE: `gehe`

Command. The `goto` command forces the execution to continue from the place named by a tag. The tag could be a quoted word or an expression which value is the name of an existing tag. Tags are always local to the procedure where they are defined, so it is not possible to jump from one procedure to another.

(e) Miscallaneous

`ignore :value` DE: `ignoriere`

Command. This command is used to evaluate an expression and ignore its value. This operation makes sense only if the evaluation of the expression has side effects, which are not ignored.

```
to mysqr :x
   (print "x "= :x*:x)
   output :x*:x
end
ignore mysqr 2
x = 2
```

`wait :value` DE: `warte`

Command. This command is used to suspend program execution for `value` 60ths of a second.

```
print [Sleeping for 5 seconds]
Sleeping for 5 seconds
wait 300
print [After sleep]
After sleep
```

5. Files and folders

Commands and function for files and folders are used to work with the directory structure. All names of folders are words that may contain just a folder’s name, a relative folder path or an abstract folder path. There are two folders with special names: the folder called `.` represents the current folder, and `..` represents the parent folder.

(a) Folders

`currentfolder` DE: `aktuellerordner`

Function. Returns a word containing the name of the current folder.

```
print currentfolder
D:\lhogho\
```

`changefolder :name` DE: `ändereordner`

Command. Changes the current folder.

```
print currentfolder
c:\lhogho\
changefolder "..
print currentfolder
c:\
```

`makefolder :name` DE: `setzeordner`

Command. Creates a new folder with a given `name`. The new folder is placed in the current folder unless `name` contains relative or absolute path.

The following example creates two folders: `main` in the current folder, and folder `other` in `main`.

```
makefolder "main
makefolder "main/other
```

`erasefolder :name` DE: `löscheordner`

Command. Erases a folder with a given `name`. The folder must be empty, otherwise the folder will not be erased.

The following example deletes two folders. Note, that `other` is erased before `main`.

```
erasefolder "main/other
erasefolder "main
```

`renamefolder :name :newname` DE: `umbenennenordner`

Command. Renames a folder with a given `name` to a new name given in `newname`.

```
renamefolder "main "mainobj
```

`folder? :name` DE: `ordner?`
`folderp :name` DE: `ordnerp`

Function. If `name` is a valid name (or path) of a folder and this folder exists, outputs `true`, otherwise `false`.

```
print folder? "lhogho.exe
false
makefolder "main
print folder? "main
true
```

`folders :name` DE: `ordner`

Function. Returns a list of the names of all folders in a folder with a given `name`. The order of the names in the returned list is undefined.

```
makefolder "main
makefolder "main/this
makefolder "main/that
print folders "main
. .. that this
```

To get a list of folders in the current folder use one of the following two ways:

```
print folders currentfolder
print folders ".
```

(b) Files

`files :name`

Function. Returns a list of the names of all files in a folder with a given `name`. The order of the names in the returned list is undefined.

```
print files "main
```

To get a list of files in the current folder use one of the following two ways:

```
print files currentfolder
print files ".
```

`file? :name` DE: `Datei?`
`filep :name` DE: `Datei`

Function. If `name` is a valid name (with or without path) of an existing file then outputs `true`, otherwise `false`.

```
print file? "readme.txt
```

```
erasefile :name                                    DE: löschedatei
erf :name                                             DE: vgdatei
```

Command. Erases a file with a given `name`.

```
erasefile "readme.txt
```

```
renamefile :name :newname                      DE: umbenennendatei
```

Command. Renames a file with a given `name` to a new name given in `newname`.

```
renamefile "readme.txt "readmenow.txt
```

```
filesize :name                                    DE: dateigröße
```

Function. Returns the size of a file with given `name`. The size is measured in bytes. If a file with such name does not exist or is inaccessible, then outputs -1.

```
print filesize "readme.txt
```

```
filetimes :name                                  DE: dateizeiten
```

Function. Returns a list of three integer numbers representing the times of a file with given `name`. The times are: time of file creation, time of last modification and time of last access.. If a file with such name does not exist or is inaccessible, then outputs an empty list.

The times represent the number of seconds elapsed since 00:00:00 on January 1, 1970. They can be converted into a calendar time with `formattime`.

```
make "time filetimes "lhogho.exe
print :time
1327317136 1327328139 1327330287
print formattime first :time "|%d-%b-%Y|
23-Jan-2012
```

(c) Opening and closing files

Whenever you want to work with the content of a file it must be first open with the command `openfile` or one of its variations: `openread`, `openwrite`, `openappend` and `openupdate`. The successful opening of a file generates a unique number called *handle* which is used to manage the content of the file. There is a system defined number of maximal 20 simultaneously opened files. A list of the names of all opened files can be retrieved with `allopen`.

Some file operations may use either file handles or file names to identify a file. Lhogho does not know which one is actually used, so it first searches for opened handles, and if not found it searches for opened file names.

When the managing of the file content is done, the file must be closed with the commands `closefile` or `closeall`. Closing a file releases a slot so that a new file can be opened. When Lhogho exits it will automatically close all opened files.

`openfile :filename :mode` DE: `öffnedatei`

Function or command. Opens a file with given `filename`. The `mode` determines how the file is opened and what operations will be performed on it:

`r` opens an existing file for reading

`w` opens (or creates) a file for writing

`a` opens (or creates) a file for appending

`r+`	opens an existing file for reading and writing
`w+`	opens (or creates) a file for reading and writing
`a+`	opens (or creates) a file for reading and appending

An additional character may appear after these:

`x`	does not allow `openfile` to overwrite existing file
`b`	the file is a binary file

If `openfile` is used as a function, then the returned value of is an OS-dependent file handle (i.e. a number), which identifies the file. This handle maybe used by the other file functions.

If `openfile` is used as a command, then the file handle is not returned. Other commands that refer to the opened file should use its name.

`openread :filename` DE: `öffnenlesen`

Function or command. Opens a file with a given `filename` for reading. The read position is initially at the beginning of the file. The function is equivalent to `openfile` with mode `r`.

`openwrite :filename` DE: `öffnenschreiben`

Function or command. Opens a file with a given `filename` for writing. If the file already existed, the old version is deleted and a new, empty file created. The function is equivalent to `openfile` with mode `w`.

`openappend :filename` DE: `öffnenanhängen`

Function or command. Opens a file with a given `filename` for appending. If the file already exists, the write position is initially set to the end of the old file, so that newly written data will be appended to it. The function is equivalent to `openfile` with mode `a`.

`openupdate :filename` DE: `öffnenaktualisieren`

Function or command. Opens a file with a given `filename` for updating. The read and write positions are initially set to the end of the old file, if any. The function is equivalent to `openfile` with parameter mode `r+`.

`allopen` DE: `allesoffen`

Function. Outputs a list of the names of all files opened with any of the functions `openfile`, `openread`, `openwrite`, `openappend` or `openupdate`. The names are not ordered.

`closefile :file` DE: `schließedatei`

Command. Closes a file opened with `openfile`. The input must be either a name of an opened file or a valid file handle.

`closeall` DE: `schließealles`

Command. Closes all files opened with any of the functions `openfile`, `openread`, `openwrite`, `openappend` or `openupdate`.

(d) Accessing file contents

The functions for accessing the file content work only with already opened files. Some file operations use file handles as input. If they cannot find opened file with this handle, they assume that the input is a name of a file and search the list of opened files by name.

For historical reasons Lhogho assign roles of two of the files:

Reading file – this is a file from which reading is done. At every moment there is at most one reading file. By default, reading is done from the default input device (usually the terminal or the keyboard).

Writing file – this is a file to which writing is done. At every moment there is at most one writing file. By default, writing is done to the default output device (usually the terminal or the console window).

Setting and querying the roles of the files is done by `reader`, `writer`, `setread`, and `setwrite`. If a reading/writing file is a true file (i.e. it is not the terminal), then it is possible to set or to query the reading/writing position with `readpos`, `writepos`, `setreadpos`, and `setwritepos`.

The function `eof?` can be used to check whether there is more text data to read from the current input. If the input is the terminal, the typing of Ctlr-Z (for Windows) or Ctrl-D (for Linux) is considered as the end of the input.

Follows an example of writing the numbers 1 to 10 and their squares into the text file `square.txt`:

```
make "file openwrite "square.txt
setwrite :file
for "i [1 10] [(print :i "* :i "= :i*:i)]
closefile :file
```

The contents of square.txt file will be:

```
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
8 * 8 = 64
9 * 9 = 81
10 * 10 = 100
```

The next example shows how to read line-by-line the already created `square.txt` text file and print its content to the terminal. The example assumes the number of lines in the text file is unknown.

```
make "file openread "square.txt
setread :file
while not eof? [print readword]
closefile :file
```

`setread :file` DE: `setzelesen`

Command. Makes an opened file the default file for reading. If `file` is the empty list, then reading is done from the default input device (e.g. the terminal), otherwise `file` must be either a name or a handle of an opened file. Changing the reading does not close the file that was previously used for reading, so it is possible to alternate between files.

`setwrite :file` DE: `setzeschreiben`

Command. Makes an opened file the default file for writing. If `file` is the empty list, then writing is done to the default output device (e.g. the terminal or the console), otherwise `file` must be either a name or a handle of an opened file. Changing the writing does not close the file that was previously used for writing, so it is possible to alternate between files.

`reader` `DE: leser`

Function. Outputs the name of the file currently used for reading, or the empty list if reading is done from the default input device (e.g. the terminal or the console).

`writer` `DE: schreiber`

Function. Outputs the name of the file currently used for writing, or the empty list if writing is done to the default output device (e.g. the terminal or the console).

`setreadpos :position` `DE: setzeleseort`

Command. Sets the reading position of the current reading file to a given `position`. The first byte of a file has position 0.

`setwritepos :position` `DE: setzeschreibeort`

Command. Sets the writing position of the current writing file to a given `position`. The first byte of a file has position 0.

`readpos` `DE: leseort`

Function. Outputs the current reading position of the reading file.

`writepos` `DE: schreibeort`

Function. Outputs the current writing position of the writing file.

(e) Text input/output

Lhogho supports several commands to write text to the console or to a file; as well as several functions to read text from the console or from a file. Initially text reading and text writing is associated with the terminal or the console.

If `setwrite` defines a writing file, then the text output commands `print`, `type` and `show` will start writing text to that file. If `setread` defines a reading file, then the text input functions `readchar`, `readchars`, `readrawline`, `readword` and `readlist` will start reading text from that file.

```
print :value                                      DE: druckezeile
(print :value :value ...)
 pr :value                                                 DE: dz
(pr :value :value ...)

? :value
(? :value :value ...)
```

Command. Prints its input(s) to the current output device. The command can accept more inputs, and in this case they are printed separated by a space. After each `print` a newline character is automatically printed. If the input is a list the outer square brackets are omitted. A `print` without any inputs creates an empty line.

```
print "a
a
(print "a "b "c)
a b c
```

The actual printed text depends on the values of variables `printdepthlimit`, `printwidthlimit` and `fullprintp`.

```
 type :value                                       DE: drucke, dr
(type :value :value … )
```

Command. Prints its input(s) like `print`, except that no newline character is printed at the end and multiple inputs are not separated by spaces.

```
(type 12 [1 2 3] "test)
type [1 2 3]
121 2 3test1 2 3
```

```
 show :value                                               DE: zg
(show :value :value … )
```

Command. Prints its input(s) like `print`, except that outermost square brackets of lists are printed.

```
show [1 2 3]
[1 2 3]
```

```
readchar                                          DE: lieszeichen
rc                                                   DE: lzeichen
```

Function. Reads a character from the standard input (usually the keyboard) and returns it as a word. If there are no more characters returns an empty list or waits for the user to type a character. Note that depending on the input policy of the operating system characters typed at the command prompt could be made available to `readchar` only when a complete line has been entered by pressing the `[Enter]` key.

`Readchar` function does not process any special characters.

```
readchars :number                                DE: lieszeichenkette
rcs :number                                                 DE: lzk
```

Function. Reads `number` characters from the standard input (usually the keyboard) and returns them as a word. If there are no more characters returns an empty list or waits for the user to type characters. Note that depending on the input policy of the operating system characters typed at the command prompt could be made available to `readchars` only when a complete line has been entered by pressing the `[Enter]` key.

`Readchars` function does not process any special characters.

```
readrawline                                           DE: liestasten
```

Function. Reads a line from the standard input (usually the keyboard) and returns it as a word. If there are no more characters returns an empty list or waits for the user to type characters.

`Readrawline` function does not process any special characters.

```
readword                                                DE: lieswort
rw                                                            DE: lw
```

Function. Reads a line from the standard input (usually the keyboard) and returns it as a word. If the word contains backslashes `\` or vertical bars `|…|` then they are processed according to the data tokenization rules of Lhogho. If there are no more characters `readword` returns an empty list or waits for the user to type characters.

```
readlist                                               DE: liesliste
rl                                                            DE: ll
```

Function. Reads a line from the standard input (usually the keyboard) and returns it as a list. All special characters except for the semicolon `;` are processed according to the data tokenization rules of Lhogho. If there are no more characters `readword` returns an empty word (not an empty list) or waits for the user to type characters.

```
eof?                                                  DE: dateiende?
eofp                                                  DE: dateiendep
```

Function. Outputs `true` if there are no more characters to be read from the standard input, `false` otherwise.

```
dribble :filename
```

Command. Creates a new text file with a given `filename` and begins recording in that file everything that is read from the keyboard or written to the terminal. This writing is in addition to the writing to the `writer` file. Using dribble while there is an active dribble file will first close it and then will create the new dribble file.

```
nodribble
```

Command. Stops copying information into the dribble file, and closes the file.

(f) Binary input/output

Operations for reading and writing from binary files require that the files are opened with `openfile` with mode referring explicitly binary files (e.g. `rb` or `wb`). Using the other file opening commands (`openread`, `openwrite`, etc.) may case wrong data to be transferred, because it is treated as text – some bytes have special meaning in text files, like `LF` (Line feed), `CR` (Carriage return) and others.

Binary input and output in Lhogho uses blocks to transfer binary data. For more information about blocks refer to page 73.

```
readblock :size                                  DE: liespackung
readblock :blockdef
```

Function. Reads `size` bytes from the current reading file and returned them in a newly allocated memory block. If there are not enough data in the file or if the reading file is the default input device (usually the terminal or the keyboard), then an empty list is returned. If the input is a list defining a block structure, then the size of the block is calculated based on the block structure.

The function `blocktolist` can be used to convert the read data into Logo data.

Function `eof?` could be used to check for the end of file. However, if the last unread portion of the file is smaller than `size`, then `eof?` before the reading will be `false`, `readpack` will still return an empty list, and just after that, `eof?` will start returning `true`.

```
readinblock :block
```

Function or command. Reads from the current reading file as much data as to fill in an existing `block`. If there are not enough data in the file or if the reading file is the default input device (usually the terminal or the keyboard), then an empty list is returned, otherwise the same block is returned.

The main feature of `readinblock` is that it reuses a block as a buffer for reading. The other function, `readblock`, creates a new block for each execution.

The next example shows a typical usage of `readinblock` (assuming it uses the block `buf` as a buffer):

```
make "buf readblock :buf
```

If the result of the reading (success or failure) is not needed, then `readinblock` can be used as a command:

```
readblock :buf
```

`writeblock :block` DE: `schreibepackung`

Command. Writes the data from a memory block to the current writing file, which must be opened for writing, updating or appending. If the writing file is the default output device (usually the terminal or the console) then nothing is written.

The functions `listtoblock` and `listintoblock` can be used to convert Logo data into a memory block.

The following example creates a binary file of 6 bytes and then reads them back:

```
make "def  [u1 u1 u1 u1 u1 u1]
make "file openfile "packopen.dat "wb
setwrite "packopen.dat
writepack listtoblock [1 2 3 4 5 -1] :def
closefile "packopen.dat

make "file openfile "packopen.dat "rb
setread "packopen.dat
make "data blocktolist readblock :def :def
closefile "packopen.dat
print :data
1 2 3 4 5 255
```

6. Variables

`make :varname :value` DE: `setze`

Command. Sets the `value` of variable called `varname`. If the variable does not exist, then it is created as a global one.

```
make "a 10
make "c :a+count [some text]
print :c
12
```

```
name :value :varname
```

Command. Sets the `value` of variable called `varname`. If the variable does not exist, then it is created as a global one. The `name` command is the same as `make` except that its inputs are in reversed order.

```
name 10 "a
name :a+count [some text]
print :c
12
```

```
local :varname                                          DE: lokal
(local :varname :varname :varname … )
```

Command. Create local variables with given names. The variables are local to the currently running procedure. The initial value of the variables is set to the empty list `[]`.

```
make "a 10
to test
   local "a
   make "a 20
   print :a
end
print :a
10
test
20
print :a
10
```

```
thing :name                                             DE: wert
```

Function. Outputs the value of the variable whose name is the input `name`. If there is more than one such variable, the innermost local variable of that name is chosen.

```
make "a [One mouse]
print thing "a
```

```
One mouse
```

```
defined? :name                                    DE: def?
definedp :name                                    DE: defp
```

Function. If the value of `name` is a name of a user-defined function or command (but not a variable), then outputs `true`. Otherwise outputs `false`.

```
make "a 10
to test
end
print defined? "a
false
print defined? "test
true
```

```
define :name :text                       DE: definiere, def
```

Command. Defines or redefines a procedure with given `name` and `text`. The `text` input must be a list whose elements are lists. The first element is a list of inputs `[[left-inputs] right-inputs]`. The remaining elements make up the body of the procedure.

The command `define` can define prefix, infix and suffix procedures. This is controlled by how formal inputs are described.

A prefix procedure can be defined in these ways:

```
to mux :a :b
end
define "mux [[a b] ...]
define "mux [[[] a b] ...]
```

An infix procedure can be defined in these ways:

```
to :a mux :b
end
define "mux [[[a] b] ...]
```

A suffix procedure can be defined in these ways

```
to :a :b mux
end
define "mux [[[a b]] ...]
```

There are two ways to execute a procedure which is defined at run-time. Note that Lhogho is a compiler, so it should process the call *after* the called procedure is defined. To implement this, execution can be delayed with the `run` command. In the next example the `print` statement is processed at run-time after `mux` is already defined and compiled.

```
define "mux [[x y] [output :x+:y]]
run [print mux 1 2]
3
```

The other alternative is to provide an empty prototype of the procedure.

```
to mux :x :y
end
define "mux [[x y] [output :x+:y]]
print mux 1 2
3
```

`procedure? :name` DE: `prozedur?`
`procedurep :name` DE: `prozedurp`

Function. If the value of `name` is a name of a function or command (but not a variable), then outputs `true`. Otherwise outputs `false`.

```
make "a 1
to test
end
print procedure? "a
print procedure? "test
print procedure? "make
false
true
true
```

`primitive? :name` DE: `grundwort?`
`primitivep :name` DE: `grundwortp`

Function. If the value of `name` is a name of a primitive function or command (but not a variable), then outputs `true`. Otherwise outputs `false`.

```
to test
end
print primitive? "test
print primitive? "make
```

```
false
true
```

```
name? :name
namep :name
```

Function. If the value of `name` is a name of a variable, then outputs `true`. Otherwise outputs `false`.

```
make "a 1
print name? "a
true
print name? "make
false
```

7. Advanced primitives

(a) Lhogho System variables

```
:logoplatform                                    DE: logoplattform
```

Variable. This variable contains a word describing the platform. Currently it is either `Windows` or `Linux`.

```
print :logoplatform
Windows
```

```
:logoversion
```

Variable. This variable contains a number describing the Logo major and minor version – i.e the first two numbers from the full version number.

```
print :logoversion
0.0
```

```
:logodialect                                      DE: logodialekt
```

Variable. This variable contains a word describing the Logo dialect. Currently it is `Lhogho`.

```
print :logodialect
Lhogho
```

`:printdepthlimit` DE: `druckelistentiefe`

Variable. A variable called `printdepthlimit` indicates how many levels of list nesting to print. If the variable is a non-negative integer value, the list will be printed only to the allowed depth.

```
make "a [a [b [c d] e [f g] h] i [j]]
make "printdepthlimit 2
print :a
a [b ... e ... h] i [j]
make "printdepthlimit 0
print :a
...
```

It is possible to create a local `printdepthlimit` and it will be used instead of the global system-defined one.

`:printwidthlimit` DE: `druckelistenmächtigkeit`

Variable `printwidthlimit` affects how many elements form the beginning of list or a word to print. If the variable is a non-negative integer value, only the first `printwidthlimit` elements will be printed. All the rest will be replaced by a single elipses. For words values between 0 and 9 (inclusive) are treated as if `printwidthlimit` is 10.

```
make "printwidthlimit 2
print [a [b [c d] e [f g] h] i [j]]
a [b [c d] ...] ...
print "abcdefghijklmnopqrstuvwxyz
abcdefghij...
```

It is possible to create a local `printwidthlimit` and it will be used instead of the global system-defined one.

`:fullprintp` DE: `vollelistentiefep`

Variable. If a variable called `fullprintp` is true then words are printed with vertical bars and backslashed in a way to allow Lhogho to reread and reparse them. Words printed under the effect of `fullprintp` does not necessarily have the same characters as in the original source.

```
make "a [spaced\ word |barred word| |barred \| bar|]
print :a
spaced word barred word barred | bar
make "fullprintp "true
```

```
print :a
|spaced word| |barred word| |barred \| bar|
```

If `funnprintp` is `true` then the empty word is printed as `||`.

It is possible to create a local `fullprintp` and it will be used instead of the global system-defined one.

`:caseignorep` DE: `großkleinschrift.ignoriertp`

Variable. If a variable called `caseignoredp` is `true` or is not defined, then words are compared case insensitively (`ABC` equals `Abc`). If the variable is `false`, then words are compared case sensitively (`ABC` differs from `Abc`).

The `caseignoredp` variable affects `equalp`, `equal?`, `=`, `notequalp`, `notequal?`, `<>`, `memberp`, `member?`, and `member`.

```
print "ABC = "abc
true
make "caseignoredp "false
print "ABC = "abc
false
```

It is possible to create a local `caseignoredp` and it will be used instead of the global system-defined one.

(b) Run-time functions and commands

`text :name`

Function. Outputs a list containing the definition of a user-defined function or command with given `name`. The first element of the list is a list of the inputs' names. The other elements represent individual lines from the body of the function.

```
to proc :a :b :c
   print 1
   print 2 print 3
   print 4
end
print text "proc
[a b c] [print 1] [print 2 print 3] [print 4]
```

If the user-defined function is infix or postfix, then the names of the left inputs are grouped in a sublist within the first element of the result – i.e. just before the first right input.

```
to :a proc1 :b :c
end
to :a :b proc2 :c
end
to :a :b :c proc3
end
print text "proc1
[[a] b c] [print 1]
print text "proc2
[[a b] c] [print 2]
print text "proc3
[[a b c]] [print 3]
```

The result of `text` is accepted as an input of `define`.

`fulltext :name` `DE: volltext`

Function. Outputs a word containing the definition of a user-defined function or command with given name. The result includes the definition from `to` to `end` inclusive. Spacing, formatting, continuation characters etc. are preserved. Dynamically created function for which there is no source, produce the same result with `fulltext` as with `text`.

```
to myfunc :a ;test function
   print :a
   (print :a ~
       :a*:a   )
end
print fulltext "myfunc
to myfunc :a ;test function
   print :a
   (print :a ~
       :a*:a   )
end
```

The result of `fulltext` can not be accepted as an input of `define`.

(c) Parsers

```
parse :value                                   DE: parsatz
```

Function. Parses `value` as if it contains Logo data. The value can be a word or a list. If it is a list, then each element is parsed individually.

```
print parse [print 1+count "boza]
print 1+count "boza
print parse [1+?37 1-555]
1+?37 1-555
```

```
runparse :value                                DE: tueparsatz
```

Function. Parses `value` as if it contains Logo commands. The value can be a word or a list. If it is a list, then each element is parsed individually. Nested sublist are not parsed as commands, but as data.

```
print runparse [print 1+count "boza]
print 1 + count "boza
print runparse [1+?37 1-555]
1 + ( ? 37 ) 1 - 555
```

(d) Error handling

```
 throw :tag                                    DE: wirf
(throw :tag :value)
```

Command. This command is used to generate a special event (called *exception*). Exceptions cause the program to terminate unless they are captured and processed by a `catch` command with the same `tag`. The `throw` command has 6 variants depending on the number of inputs and the contents of the tags.

```
 throw "toplevel                               DE: wirf "ausstieg
(throw "toplevel :value)
```

The tag `toplevel` forces the program to terminate and to return to the top level – if a GUI is running the top level is the GUI itself; if a console version of Lhogho is used, then top level is the command prompt.

```
 throw "system                                 DE: wirf "system
(throw "system :value)
```

To exit the running program and the GUI the tag `system` can be used.

```
print "before
throw "system
```

```
print "after
before
```

```
throw "error                                                DE: wirf "fehler
(throw "error :value)
```

The tag `error` causes `throw` to generate a user-defined error exception. Using tag `error` without a second input cases the error to be reported at the tag. If a second input is present then it is used as an error message. In this case the error is reported at the command in which `throw` is used.

```
to diff :x :y
   if not number? :x [(throw "error [Not a number])]
   if not number? :y [(throw "error [Not a number])]
   output :x-:y
end
print diff 5 pi
print diff 10 "pi
1.85840734641021
{ERR#30@217} - Not a number
print diff 10 "pi
      ^
```

If the tag is another word, then it is expected that the exception will be captured by `catch` with the same `tag`. If a second input is not provided then `catch` does not output any value too. If a second input is present, then it is the value output by the `catch` with the same tag.

```
to reciprocal :x
   if not number? :x [(throw "oops 0)]
   output 1/:x
end
print catch "oops [reciprocal 5]
0.2
print catch "oops [reciprocal "five]
0
```

```
catch :tag :commands                                        DE: fange
```

Command and function. This command is used to catch exceptions generates in the commands during their execution. Catching is successful only if the `tag` of `catch` and `throw` are the same, or if the `tag` of `catch` is `error` and the exception in commands is caused by an error.

`catch` can be a command and a function. The result of `catch` is the value provided as a second input to the corresponding `throw`. However, the second input to error thows are used as error messages, not as outputs of `catch`.

Only run-time errors related to the execution of user-program can be captured. Errors related to source parsing, for example, could not be captures because they are triggered before corresponding `catch` is activated.

```
catch "error
[
  print 1/5
  print 1/"five
]
0.2
```

`error` DE: `fehler`

Function. The function `error` is used to get information about the last captured error with `catch`. If there was no any captured error, then the result is an empty list. Otherwise it is a list with four elements:

- `code` – an integer number identifying the type of the error;
- `message` – a word containing the error message as text;
- `procedure` – a word containing the name of the user-defined procedure where the error has occurred. If the error happened at top level, this element is an empty list;
- `source` – a list containing the statement where the error has occurred. The statement is represented in a prefinx notation, fully parenthesized.

The last captured error is forgotten after using `error`.

```
to test :n
    print 1/:n
end
catch "error [test "two]
make "err error
(print [Error code:] item 1 :err)
Error code: 13
(print [Error text:] item 2 :err)
Error text: Not a number
```

```
(print [Error place:] item 3 :err)
Error place: test
(print [Error statement:] item 4 :err)
Error statement: (test "two)
```

(e) OS-related functions

This section describes functions which support the communication between Logo programs and the operating system.

`commandline` DE: `kommandozeile`

Function. Returns a list of all command-line parameters which are not processed by Lhogho itself. These parameters are the one after the name of the Logo program being executed. For example, the command line for:

```
D:\lhogho\>lhogho -Zm primes.lgo 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
{MEM#0}
```

is the list `[50]`, while `-Zm` and `primes.lgo` are processed by Lhogho and are not available to the user program. The same command line is for this case of running compiled `primes`:

```
D:\lhogho\>lhogho -x primes.lgo
D:\lhogho\>primes 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

`getenv :name` DE: `gibumgebungsvariable`

Function. Returns the value of an environment variable with given `name`. An empty word is returned in the name does not exist.

```
type "USER=
print getenv "USER
USER=Lhogho
```

`getenvs` DE: `gibumgebungsvariablen`

Function. Returns a list with the names and the values of all environment variables. Each name and value are grouped: `[[name1 value1] [name2 value2] … ]`. To get the value of a single variable it is faster to use `getenv`. The next example prints the names and the values of 5 environment variables.

```
make "a getenvs
repeat 5
```

```
[
  (type first first :a "= char 9)
  print last first :a
  make "a bf :a
]
WINDIR= C:\WINDOWS
USER=   Lhogho
TERM=        cygwin
PROMPT= $P$G
MAKE_MODE=   unix
```

8. Low-level access

This section describes function and commands used to access external functions and to deal with data stored in computer's memory.

IMPORTANT NOTE! THESE FUNCTIONS ARE SENSITIVE. THEY DEAL WITH CODE AND DATA, WHICH ARE BEYOND LHOGHO’S CONTROL. IF MISUSED THEY MAY CAUSE SYSTEM INSTABILITY AND CRASH.

(a) Native data types

Native data types express data in an efficient processor-friendly manner. Examples for native types are *bytes* and *pointers*. Logo data (numbers, words, lists) are not native types.

Lhogho uses native data types for various purposes:

- To communicate with modules written in another programming language, so that Lhogho can use their functions, and they can use Lhogho functions.
- To read and write data to binary files. These data could be structured (e.g. a file of 128-bytes records of patients) or unstructured (e.g. a sequence of bytes).
- To convert Logo data to binary data and vice versa.

Native types supported by Lhogho are named by a character indicating the group and a number indicating the size of the type.

The following table lists the native types.

Type	Size (in bytes)	Explanation	Externals	Blocks
`I1`	1	Signed integer number	Yes	Yes
`I2`	2			
`I4`	4			
`I8`	8			
`U1`	1	Unsigned integer number	Yes	Yes
`U2`	2			
`U4`	4			
`U8`	8			
`F4`	4	Floating point number	Yes	Yes
`F8`	8			
`S1`	4	Pointer to string of 1-byte characters	Yes	No
`S2`	4	Pointer to string of 2-byte characters		
`P4`	4	Pointer	Yes	Yes
`A4`	4	Atom (the internal Lhogho datum)	No	Yes
`V0`	4	Void	Yes	No

Note that not all native types are available for all functions that use them. For example, prototypes of external functions cannot use `A4` format, while block functions cannot use `S1`, `S2` and `V0` formats.

In many cases communication with native data types uses larger structures that are composed of several and usually different native types.

Lhogho represents the definition of such structures (called *block definitions*) as list of native types. The next example shows a block definition of a 3D point with integer coordinates:

```
[i4 i4 i4]
```

It is possible to put block definitions inside other block definitions. Thus, a segment defined by two points could be expressed as:

```
[[i4 i4 i4] [i4 i4 i4]]
```

When an element in a block definition is repeated many times it is possible to use *multipliers*. A multiplier is a number that determines how many times to repeat the next element in the definition. Instead of writing:

```
[i2 i2 i2 i2 i2 i2 i2 i2 i2 i2]
```

it is possible to write:

```
[10 i2]
```

Similarly, segment definition above could be written as:

```
[[3 i4] [3 i4]]
```

or even:

```
[2 [3 i4]]
```

For convenience it is possible to give custom names of native types and block definitions and then use these names in other block definitions. The following example defines `byte` and `int` as alternative names for `u1` and `i4`.

```
make "byte "u1
make "int "i4
```

Thus `point` and `segment` could be also defined as:

```
make "point [int int int]
make "segment [point point]
```

(b) Blocks

A block is a continuous area in the memory filled with binary data. Neither Lhogho or any other program or even the processor can tell what actually the meaning of these data is. When a block is paired to a block definition then Lhogho has a means to interpret the contents of the block as a collection of native data types. In this respect blocks provide a similar functionality as `struct` in C and `record` in Pascal.

The functions described in this section are used to manage memory blocks containing binary data (i.e. a sequence of bytes). Reading and writing blocks from binary files is described in page 58.

```
blocksize :block                                    DE: packgröße
blocksize :blockdef
```

Function. Outputs the size of an actual `block` or a block definition `blockdef`.

```
print blocksize [i1 i2 f4]
7
print blocksize [i1 i2 [i1 i8] [u1 [u4 f4]]]
21
```

```
listtoblock :list :blockdef                           DE: packen
```

Function. Converts the elements of a `list` according to the given block definition `blockdef` and returns a memory block containing the elements converted to native format. This corresponds (roughly!) to converting Logo data into a C structure. Data in a memory block can be converted back to Logo list with `blocktolist` function.

```
make "Byte "u1
make "Float "f4
make "struct listtoblock [1 [2 2.5]] [Byte [Float
Float]]
print blocktolist :struct [Byte [Float Float]]
1 [2 2.5]
```

If input data are less than the required by `blockdef` all empty slots in the memory block are set to 0.

```
make "a listtoblock [10 [5]] [u2 [u2 u2 u2] u2 u2]
print blocktolist :a [u2 [u2 u2 u2] u2 u2]
10 [5 0 0] 0 0
```

```
listintoblock :data :block :blockdef                DE: packenzu
listintoblock :data :address :blockdef
```

Command. Converts `data` according to the given block definition `blockdef` into destination `block` or `address`. The second input should be either a block or an integer number for a memory address. Converting to destination requires that there is enough space for all data. Otherwise extra data may overwrite essential data and make the whole system unstable.

```
make "pair [[i1 i1] [i1 i1]]
make "a listtoblock [[1 2] [3 4]] :pair
print blocktolist :a :pair
[1 2] [3 4]
listintoblock [[11 12] [13 14]] :a :pair
print blocktolist :a :pair
[11 12] [13 14]
```

```
blocktolist :block :blockdef                        DE: entpacken
blocktolist :address :blockdef
```

Function. Converts into a list data stored in `block` or at given memory `address` according to `blockdef`.

```
make "Byte "u1
make "Float "f4
make "struct listtoblock [1 [2 2.5]] [Byte [Float
Float]]
print blocktolist :struct [Byte [Float Float]]
1 [2 2.5]
```

Together with `dataaddr` this function can be used to peek the raw structure of Logo data. The following example prints the reference count of a Logo datum. This count is unsigned 32-bit integer stored at the beginning of each Logo datum.

```
make "a [one two]
make "b blocktolist dataaddr :a [u4]
(print [Reference count] first :b)
Reference count 3
```

The tandem `listtoblock` and `blocktolist` can be used to split and join integer numbers. The next example demonstrates the values of the four bytes in a 32-bit integer.

```
make "a listtoblock [1000] [i4]
print blocktolist :a [u1 u1 u1 u1]
232 3 0 0
```

(c) Shared libraries

This section describes functions that can be used to manage functional communication between Lhogho and external non-Lhogho compiled functions in shared or dynamic libraries.

`libload :libname` DE: `bibliothekladen`

Command. Loads dynamic/shared library with file name given by `libname`. If the name is without extension then the default extension for the current operating system will be used (`*.DLL` for Windows and `*.SO` for Linux). If the name contains a path, then the library is searched in this path. Otherwise the library is searched in the default for the operating system places. If the file is not found, then it is searched in the `lib` subfolder of the folder where the compiler is located. The returned value is an OS-dependent handle (i.e. a number) which identifies the loaded library.

```
make "handle libload "testlib
if :handle = 0
  [print [TestLib not loaded]]
  [print [TestLib loaded]]
TestLib loaded
```

`libfree :handle` DE: `bibliothekfrei`

Command. Unloads dynamic/shared library. The input should be the handle returned by `libload` when the library has been loaded for the first time.

```
make "handle libload "testlib
libfree :handle
```

`external :name :prototype :handle` DE: `extern`

Command. Defines that function called `name` corresponds to an external function with given `prototype` and its binary code is in a library corresponding to `handle`. The prototype is a list in this format: `[result extname param1 param2 …]` where `result` is the type name of the result (it could be native or user-defined), `extname` is the name of the external function the way it is exported by the library. The rest elements `param1`, `param2`, etc are the type names of the parameters.

In the example, the Lhogho definition of `byteadd` creates an empty function, which is bound to `addup` function from `testlib.so` or `testlib.dll`. The external function (that might have been compiled in C) uses two parameters, which are unsigned bytes, and produces a result, which is also an unsigned byte.

```
make "handle libload "testlib
to byteadd :a :b
end
external "byteadd [u1 addub u1 u1] :handle
print byteadd 100 100
200
libfree :handle
```

The `external` command helps Lhogho to use external functions. During the execution, Lhogho does the following steps:

- It converts inputs, which are in Logo data format, into native format
- It calls the external function providing the native data
- If receives the result of the function (a native datum)
- Converts the result into a Logo datum.

`internal :name :prototype` DE: `intern`

Command. Defines that function called `name` is a Logo function which will be called by an external function written in another language. The `prototype` defines the native data type of the result and the inputs of the Logo function: `[result param1 param2 …]` where `result` is the type name of the result (it could be native or user-defined), `param1`, `param2`, etc are the type names of the parameters.

The `internal` command helps Lhogho to define a function that is called by non-Lhogho functions (e.g. callbacks). During the execution, Lhogho does the following steps:

- It converts inputs, which are in native data format, into Logo format
- It calls the Lhogho function providing the Logo data
- If receives the result of the function (a Logo datum)
- Converts the result into a native datum and returns it to the calling function.

`funcaddr :name` DE: `funktionadr`

Function. Returns the address of the compiled body of function with a given `name`. The result can be used in hook functions – i.e. functions from a shared library that calls a Lhogho function by its address.

In the following example the external function `apply` calls directly the compiled code of `add` or `sub`, which are defined completely as user functions.

```
if equal? :logoplatform "Windows
  [ make "lib libload "testlib]
  [ make "lib libload "./libtestlib.so]
to apply.func :func :arg1 :arg2
end
to add :x :y
   output :x+:y
end
to sub :x :y
   output :x-:y
end
external "apply.func [i4 apply p4 i4 i4] :lib
internal "add [i4 i4 i4]
internal "sub [i4 i4 i4]
print apply.func funcaddr "add 10 5
print apply.func funcaddr "sub 10 5
libfree :lib
```

`dataaddr :thing` DE: `dataadr`

Function. This function returns the address of memory block where the Logo data of `thing` is stored. The address of Logo data can be used to manage the internal representation of Logo data. However, this is dangerous as long as going to invalid address or writing inappropriate data may lead to a system crash.

The following example demonstrates that two variables with the same value do not necessarily share the same memory.

```
make "a 45.8+1
make "b 44.8+2
make "c :a
(print equal? dataaddr :a dataaddr :b)
false
(print equal? dataaddr :a dataaddr :c)
true
```

(d) System stack

`_int3`

Command. Generates a software interrupt. This command is useful only when Lhogho is being debugged with an external debugger, which understands software interrupts. Using `_int3` without debugger will cause the program to terminate with an exception.

`_stackframe :frame :offset` DE: `_stackrahmen`

Function. Returns an integer number, which is found at an `offset` of a stack `frame`. Frame 0 is the stack frame of the currently executing procedure (the one, which uses `_stackframe`). Each procedure has two parent procedures (which could be different or the same) – *a static parent* and *a dynamic parent*. The static parent is the parent procedure that has the current procedure defined as a local procedure. The static parent of a procedure can never change once a Logo program is compiled. The dynamic parent is the procedure that called the current procedure. During the lifetime of a procedure it may have different dynamic parents (depending on which other procedures use it).

If `frame` is greater than 0 it denotes a static parent, 1 means the static parent, 2 means the static parent of the static parent and so on.

If `frame` is negative it denotes a dynamic parent, -1 means the dynamic parent, -2 means the dynamic parent of the dynamic parent and so on.

For portability `offset` is always measured in terms of the processor-specific data size, which is large enough to hold a memory address. Thus, `offset` is not measured in bytes.

The function `_stackframe` can be used to get the number of actual inputs of a procedure.

```
to inputs
   output _stackframe -1 2
end

to test :a :b ...
   print inputs
end

test 1 2
2
(test 1 2 3 4 5 6)
6
```

`_stackframeatom :frame :offset` DE: `_stackrahmenatom`

Function. This function is the same as `_stackframe`, except that it assumes the extracted value from the `offset` in the given stack `frame` is a Lhogho datum – number, word or list. Use `_stackframeatom` to get only data which is guaranteed to be Lhogho data; otherwise the user program may stop working.

The next example retrieves the values of all actual inputs of a user-defined command. Note that the physical order of the inputs places inputs in reversed order and named inputs are positioned before the others.

```
to inputs
   output _stackframe -1 2
end

to input :n
   output _stackframeatom -1 2+:n
end

to test :a :b ...
   repeat inputs [type input repcount]
   (print)
end

test 1 2
21
```

```
(test 1 2 3 4 5 6)
216543
```

Chapter IV. Libraries

1. TGA

The TGA library can be used to create TGA (Targa) image files. Only the most basic and simpliest file format is supported, i.e. the 24-bit uncompressed RGB format. For more information about TGA format check:

http://www.fileformat.info/format/tga/egff.htm

`tgaopen :filename :width :height` DE: `tgaöffnen`

Command. This command creates a new TGA image file called `filename` and defines its header. The size of the image is set to `width` and `height` pixels.

`tgawrite :filename :red :green :blue` DE: `tgaschreiben`

Command. This command writes a pixel color information to a TGA file created with `tgaopen`. The color is defined by three numbers (from 0 to 255) corresponding to the red, green and blue components of a color in RGB colorspace. The writer file is automatically set to the `filename`.

`tgaclose :filename` DE: `tgaschließen`

Command. This command closes a TGA file. Closing should be done only when the exact number of pixels is being written to the file with `tgawrite`. The number of pixels is the product of the `width` and the `height` of the image, as defined when `tgaopen` has been called. For example, if the image is 200x70 pixels, the number of pixels is 14,000.

```
load "tga.lgo
tgaopen "image.tga 128 128
for "x [0 127]
[ for "y [0 127]
  [
    make "r round 128+127*sin 10*:x
    make "g round 128+127*cos 10*:y
    make "b round 128+127*sin :x+:y
    tgawrite "image.tga :r :g :b
  ]
]
tgaclose "image.tga
```

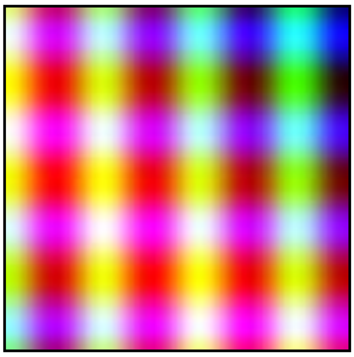

Figure 1 Using TGA library

2. GL

The `GL` library provides an interface to OpenGL. The binary file (`OPENGL32.DLL` for Windows or `libGL.so` for Linux) is not a part of Lhogho and should be available before using `GL`.

This section provides a list of supported functions. For a complete documentation about each of them consult OpenGL's documentation.

For an example of how to use `GL` check the `Cube3d` application.

(a) Matrix transformations

Most of the geometrical operations done by OpenGL are performed by matrices. The functions in this section are used to change these matrices.

```
glLoadIdentity
```

Command. Replaces the current matrix with the identity matrix.

```
glTranslatef :x :y :z
```

Command. Multiplies the current matrix by a translation matrix.

```
glScalef :sx :sy :sz
```

Command. Multiplies the current matrix by a general scaling matrix.

```
glRotatef :angle :x :y :z
```

Command. Multiplies the current matrix by a rotation matrix.

```
glOrtho :left :rigth :bottom :top :near :far
```

Command. Multiplies the current matrix by an orthographic matrix.

`glMatrixMode :mode`

Command. Specifies which matrix is the current matrix. The value of `mode` is one of these constants:

`GL_MODELVIEW`	Applies subsequent matrix operations to the modelview matrix stack.
`GL_PROJECTION`	Applies subsequent matrix operations to the projection matrix stack.

(b) Display lists

Display lists are used to collect a list of OpenGL commands which can be executed multiple times like a subroutine.

`glNewList :list :mode`

Command. Starts the definition of a display list. The value of `mode` is one of these constants:

`GL_COMPILE`	Commands are merely compiled.
`GL_COMPILE_AND_EXECUTE`	Commands are executed as they are compiled into the display list.

`glEndList`

Command. Finishes the definition of a display list.

`glCallList :list`

Command. Executes a display list.

(c) Graphical primitives

`glBegin :mode`

Command. Starts a definition of a group of graphical primitives. The value of `mode` is one of these constants:

`GL_POINTS`	Each vertex as a single point.
`GL_QUADS`	Each group of four vertices as an independent quadrilateral.

`glEnd`

Command. Ends a definition of a group of graphical primitives.

`glVertex3f :x :y :z`

Command. Specifies a vertex.

```
glRectf :x1 :y1 :x2 :y2
```

Command. Draw a rectangle.

(d) Properties of graphical primitives

```
glNormalf :sx :sy :sz
```

Command. Sets the current normal vector.

```
glColor3f :r :g :b
```

Command. Sets the current color.

```
glPointSize :size
```

Command. Specifies the diameter of rasterized points.

(e) Buffers

OpenGL uses buffers to represent the rasterized image. The following function manage buffers.

```
glClearColor :r :g :b :a
```

Command. Specifies clear values for the color buffers.

```
glDrawBuffer :buffer
```

Command. Specifies which color buffers are to be drawn into. The value of `buffer` is one of these constants:

`GL_BACK` Only the back-left and back-right color buffers are written. If there is no back-right color buffer, only the back-left color buffer is written.

```
glClear :buffer
```

Command. Clears buffers to preset values. The value of `buffer` is one or a combination of these constants:

`GL_COLOR_BUFFER_BIT`	Clears the buffers currently enabled for color writing.
`GL_DEPTH_BUFFER_BIT`	Clears the depth buffer.

(f) Miscellaneous

```
glGetError
```

Function. Returns OpenGL error information.

`glEnable :mode`

Command. Enables OpenGL capabilities. The value of `mode` is one of these constants:

`GL_DEPTH_TEST` If enabled, do depth comparisons and update the depth buffer.

`glViewport :x :y :w :h`

Command. Sets the viewport.

`glFlush`

Command. Forces execution of OpenGL functions in finite time.

3. GLU

The `GLU` library provides an interface to the OpenGL Utility Library GLU. The binary file (`GLU32.DLL` for Windows or `libGLU.so` for Linux) is not a part of Lhogho and should be available before using `GL`.

This section provides a list of supported functions. For a complete documentation about each of them consult their documentation.

For an example of how to use `GLU`, check the `Cube3d` application.

`gluLookAt :eyeX :eyeY :eyeZ :cX :cY :cZ :upX :upY :upZ`

Command. Creates a viewing matrix derived from an `eye` point, a reference point indicating the center `c` of the scene, and an `up` vector.

`gluPerspective :fovy :aspect :zNear :zFar`

Command. Sets up a perspective projection matrix.

4. GLUT

The `GLUT` library provides an interface to the OpenGL Utility Toolkit GLUT. The original GLUT is not updated for quite some time, so Lhogho uses an open-source alternative called FreeGLUT. The binary file (`FREEGLUT.DLL` for Windows or `libglut.so` for Linux) is not a part of Lhogho and should be available before using `GL`. For convenience, `FREEGLUT.DLL` is distributed with the Windows package of Lhogho.

This section provides a list of supported functions. For a complete documentation about each of them consult their documentation.

For an example of how to use `GLU`, check the `Cube3d` application. For an alternative of `GLUT` see `GLFW`.

(a) Window initialization

GLUT provides interfaces to several functions for window creation, initialization and modification.

```
glutInit :argc :argv
```

Command. Initializes the GLUT library.

```
glutFullScreen
```

Command. Requests that the current window be made full screen.

```
glutCreateWindow :caption
```

Function. Creates a top-level window with a given `caption`. Returns a handle to the new window.

```
glutCreateSubWindow :window :x :y :width :height
```

Function. Creates a subwindow. Returns a handle to the new subwindow.

```
glutInitDisplayMode :mode
```

Command. Sets the initial display `mode`, which is a combination of these constants:

`GLUT_DOUBLE`	Bit mask to select a double buffered window.
`GLUT_RGB`	Bit mask to select an RGBA mode window.
`GLUT_DEPTH`	Bit mask to select a window with a depth buffer.

```
glutReshapeWindow :x :y
```

Command. Requests a change to the size of the current window.

```
glutSetWindow :window
```

Command. Sets the current `window` identified by its window handle, which was returned by `glutCreateWindow` or `glutCreateSubWindow`.

```
glutSetCursor :cursor
```

Command. Changes the cursor image of the current window.

(b) Main loop

GLUT has a specific main loop that captures events and allows the user program to react on them. The following functions are used to manage the main loop.

`glutMainLoop`

Command. Enters the FreeGLUT event processing loop.

`glutLeaveMainLoop`

Command. Causes FreeGLUT to stop the event loop.

`glutPostRedisplay`

Command. Marks the current window as needing to be redisplayed.

`glutSwapBuffers`

Command. Swaps the front and the back buffers of the current window.

(c) Events

FreeGLUT supports callbacks to respond to events, like reshaping a window and keyboard input.

`glutHook :type :func`

Command. This command is not a part of GLUT or FreeGLUT. It defines that function with name the value of `func` is responsible to a callback link. I.e. the FreeGLUT library will call the hooked Lhogho user-defined command to handle specific events.

`glHook` is practically equivalent to defining that `func` is internal and then registering it with the appropriate callback function from FreeGLUT. The hook function is determined by `type`, which can be any of the words `idle`, `special`, `keyboard`, `display`, and `reshape`.

The following code:

```
to kbd :key :x :y
  if equal? :key 27 [ glutLeaveMainLoop ]
end
internal "kbd [v0 i1 i4 i4]
glutKeyboardFunc funcaddr "kbd
```

is equivalent to:

```
to kbd :key :x :y
  if equal? :key 27 [ glutLeaveMainLoop ]
```

```
end
glHook "keyboard "kbd
```

Note that `glHook` must be called only if `func` is a user-defined command that is not internalized with the `internal` command. If you need to hook an internal function, then use directly the corresponding functions listed below:

`glutDisplayFunc :func`

Command. Sets the display callback function for the current window. This function should be declared like this:

```
to displayFunc
end
```

`glutIdleFunc :func`

Command. Sets the global idle callback function. This function should be declared like this:

```
to idleFunc
end
```

`glutKeyboardFunc :func`

Command. Sets the keyboard callback function for the current window. This function should be declared like this:

```
to keyboardFunc :key :x :y
end
```

`glutSpecialFunc :func`

Command. Sets the special keyboard callback function for the current window. This function should be declared like this:

```
to specialFunc :key :x :y
end
```

`glutMotionFunc :func`

Command. Sets the motion callback function respectively for the current window. This function should be declared like this:

```
to motionFunc :x :y
end
```

`glutMouseFunc :func`

Command. Sets the mouse callback function for the current window. This function should be declared like this:

```
to mouseFunc :button :state :x :y
end
```

`glutReshapeFunc :func`

Command. Sets the reshape callback function for the current window. This function should be declared like this:

```
to reshapeFunc :w :h
end
```

5. GLFW

The `GLFW` library provides an interface to the OpenGL Framework GLFW (`http://www.glfw.org/`). The binary file (`glfw.dll` for Windows) is not a part of Lhogho and should be available before using `GL`. For convenience, `glfw.dll` is distributed with the Windows package of Lhogho. If the binary file cannot be located or loaded, the library will generate the custom error message: `Error loading GLFW`.

This section provides a list of supported functions. For a complete documentation about each of them consult their documentation.

For an example of how to use `GLFW`, check the `Cube3d` application. For an alternative of `GLFW` see `GLUT`.

(a) Initialization

GLFW provides interfaces to functions initialization, termination and version querying

`glfwInit`

Command. Initializes `GLFW`. No other `GLFW` functions may be called before this function has succeeded. If the initialization fails, the function will generate the custom error message: `Error initializing GLFW`.

`glfwTerminate`

Command. Terminates `GLFW`. If needed it closes the window and kills any running threads.

`glfwGetVersion`

Function. Returns a list of three numbers – the major version number, the minor version number and the revision number.

```
print glfwGetVersion
```

```
2 7 2
```

`glfwGetGLVersion`

Function. Returns a list of three numbers – the major version number, the minor version number and the revision number of OpenGL.

```
print glfwGetGLVersion
2 7 2
```

`glfwGetNumberOfProcessors`

Function. Returns the number of processors in the system.

`glfwEnable :feature`

Command. Enables a certain `feature` of GLFW. The feature is one of these values:

`GLFW_AUTO_POLL_EVENTS`
`glfwPollEvents` is automatically called immediately after `glfwSwapBuffers` is called.

`GLFW_KEY_REPEAT`
The key and character callback functions are called repeatedly when a key is held down long enough.

`GLFW_MOUSE_CURSOR`
The mouse cursor is visible and the mouse coordinates are limited to the interior area of the window.

`GLFW_STICKY_KEYS`
Keys which are pressed will not be released until they are physically released and checked with `glfwGetKey`.

`GLFW_STICKY_MOUSE_BUTTONS`
Mouse buttons that are pressed will not be released until they are physically released and checked with `glfwGetMouseButton`.

`GLFW_SYSTEM_KEYS`
Pressing standard system key combinations, such as Alt+Tab under Windows, will give the normal behavior.

`glfwDisable :feature`

Command. Disables a certain `feature` of GLFW. The feature is one of these values:

`GLFW_AUTO_POLL_EVENTS`
`glfwPollEvents` is not called automatically after `glfwSwapBuffers` is called.

`GLFW_KEY_REPEAT`
The callback functions are only called once when a key is pressed and once when it is released.

`GLFW_MOUSE_CURSOR`
The mouse cursor is invisible, and mouse coordinates are not limited.

`GLFW_STICKY_KEYS`
The status of a key is always the current physical state of the key.

`GLFW_STICKY_MOUSE_BUTTONS`
The status of a mouse button is always the current physical state of the mouse button.

`GLFW_SYSTEM_KEYS`
Пressing standard system key combinations like Alt+Tab under Windows will have no effect, since those key combinations are blocked by GLFW.

(b) Window handling

Functions for window handling are used to open and manage windows accepting OpenGL commands.

```
glfwOpenWindow :width :height :color :alpha
                                 :depth :stencil :mode
```

Command. Opens a window that best matches the parameters given to the function: `width` and `height` in pixels, number of bits for each `color`, number of bits for the `alpha`, `depth` and `stencil` buffers and `mode` which is one of these constants:

`GLFW_WINDOW`	generates a normal desktop window
`GLFW_FULLSCREEN`	generates a window which covers the entire screen.

A typical example for opening a normal window of 600x300 pixels supporting 24-bit colours, and 16-bit depth buffer would be:

```
glfwOpenWindow 600 300 8 0 16 0 :GLFW_WINDOW
```

Note: The original GLFW's `glfwOpenWindow` is a function (not a command) and it has different number of inputs. For example, it has separate input for each colour component (red, green and blue) If you need to use it; it is available under the name `_glfwOpenWindow`.

`glfwOpenWindowHint :param :value`

Command. Suggests a `value` of a `param` parameter of a GLFW window. A value of `0` or `GL_FALSE` sets the default system-defined hint. The `glfwOpenWindowHint` command must be called before `glfwOpenWindow`. The parameters (for complete details see the original GLFW Reference) are:

`GLFW_REFRESH_RATE`

`GLFW_ACCUM_RED_BITS`

`GLFW_ACCUM_GREEN_BITS`

`GLFW_ACCUM_BLUE_BITS`

`GLFW_ACCUM_ALPHA_BITS`

`GLFW_AUX_BUFFERS`

`GLFW_STEREO`

`GLFW_WINDOW_NO_RESIZE`

`GLFW_FSAA_SAMPLES`

`GLFW_OPENGL_VERSION_MAJOR`

`GLFW_OPENGL_VERSION_MINOR`

`GLFW_OPENGL_FORWARD_COMPAT`

`GLFW_OPENGL_DEBUG_CONTEXT`

`GLFW_OPENGL_PROFILE`

`glfwCloseWindow`

Command. Closes the opened window and destroys the associated OpenGL context.

`glfwSetWindowTitle :title`

Command. Sets the title (caption text) of a window.

`glfwSetWindowSize :width :height`

Command. Changes the size of an opened window. The `width` and `height` define the size of the interior area of the window (i.e. excluding any window borders).

`glfwGetWindowSize :width :height`

Function. Returns a list of the width and height of the interior area of an opened window.

`glfwSetWindowPos :x :y`

Command. Sets the horizontal position `x` and the vertical position `y` of the window, relative to the upper left corner of the desktop.

`glfwIconifyWindow`

Command. Minimizes an opened window to an icon.

`glfwRestoreWindow`

Command. Restores a minimized window.

`glfwGetWindowParam :param`

Function. Returns the value of a window parameter `param` which is one of the following (for complete details see the original GLFW Reference):

`GLFW_OPENED`

`GLFW_ACTIVE`

`GLFW_ICONIFIED`

`GLFW_ACCELERATED`

`GLFW_RED_BITS`

`GLFW_GREEN_BITS`

`GLFW_BLUE_BITS`

`GLFW_ALPHA_BITS`

`GLFW_DEPTH_BITS`

`GLFW_STENCIL_BITS`

`GLFW_REFRESH_RATE`

`GLFW_ACCUM_RED_BITS`

`GLFW_ACCUM_GREEN_BITS`

`GLFW_ACCUM_BLUE_BITS`

`GLFW_ACCUM_ALPHA_BITS`

`GLFW_AUX_BUFFERS`

`GLFW_STEREO`

`GLFW_WINDOW_NO_RESIZE`

`GLFW_FSAA_SAMPLES`

`GLFW_OPENGL_VERSION_MAJOR`

`GLFW_OPENGL_VERSION_MINOR`

`GLFW_OPENGL_FORWARD_COMPAT`

`GLFW_OPENGL_DEBUG_CONTEXT`

`GLFW_OPENGL_PROFILE`

`glfwSwapBuffers`

Command. Swaps the back and front color buffers of the window. `glfwPollEvents` is called after swapping if `GLFW_AUTO_POLL_EVENTS` is enabled (which is the default).

`glfwSwapInterval :interval`

Command. Selects the minimum number of monitor vertical retraces that should occur between two buffer swaps. If the selected swap interval is one, the rate of buffer swaps will never be higher than the vertical refresh rate of the monitor. If the selected swap interval is zero, the rate of buffer swaps is only limited by the speed of the software and the hardware.

(c) Video Modes

A video mode is a list of five number determining the raster and the colour resolution `[width height redbits greenbits bluebits]`. The raster resolution is the number of horizontal `width` and vertical `height` pixels. The colour resolution is the number of bits for each colour component.

`glfwGetDesktopMode`

Function. Returns the desktop video mode used by the desktop at the time the GLFW window was opened, not the current video mode (which may differ from the desktop video mode if the GLFW window is a fullscreen window).

The following example reports that the desktop screen is 1280x800 pixels and supports 24-bit colours.

```
print glfwGetDesktopMode
1280 800 8 8 8
```

`glfwGetVideoModes`

Function. Returns a list of supported video modes in ascending order (sorted by colour resolution and then by raster resolution). The last element of the list is the mode with largest resolutions.

```
print glfwGetVideoModes
```

```
[320 200 5 5 6]  [320 240 5 5 6]  [400 300 5 5 6]
[512 384 5 5 6]  [640 400 5 5 6]  [640 480 5 5 6]
[800 600 5 5 6]  [1024 768 5 5 6] [1280 768 5 5 6]
[1280 800 5 5 6] [320 200 8 8 8]  [320 240 8 8 8]
[400 300 8 8 8]  [512 384 8 8 8]  [640 400 8 8 8]
[640 480 8 8 8]  [800 600 8 8 8]  [1024 768 8 8 8]
[1280 768 8 8 8] [1280 800 8 8 8]
```

(d) Timing

GLFW supports internal high-precision timer – a floating-point number measuring seconds and fractions of seconds.

`glfwGetTime`

Function. Returns the current GLFW time. Initially, `glfwInit` sets the timer to 0.

`glfwSetTime :time`

Command. Sets the current GLFW time.

`glfwSleep :time`

Command. Puts the calling thread to sleep for the requested period of time. There is usually a system dependent minimum time for which it is possible to sleep (generally in the range 1 ms to 20 ms).

(e) Input handling

`glfwPollEvents`

Command. Polls for events, such as user input and window resize events. Upon calling this command, all window states, keyboard states and mouse states are updated. If any related callback functions or commands are registered, they are called. Events are implicitly polled during `glfwSwapBuffers` if `GLFW_AUTO_POLL_EVENTS` is enabled (as it is by default).

`glfwWaitEvents`

Command. Waits for events and then polls them as `glfwPollEvents`.

`glfwGetKey :key`

Function. Queries the current state of a specific keyboard `key`. The function returns `:GLFW_PRESS` if the key is held down, or `:GLFW_RELEASE` if the key is not held down. The value of `key` can be either the ASCII code of an uppercase printable ISO 8859-1 (Latin 1) character (e.g. ’`A`’, ’`3`’ or ’`.`’), or a special key identifier:

`GLFW_KEY_SPACE`	Space
`GLFW_KEY_ESC`	Escape
`GLFW_KEY_F`*n*	Function key *n* (*n* can be in the range 1..25)
`GLFW_KEY_UP`	Cursor up
`GLFW_KEY_DOWN`	Cursor down
`GLFW_KEY_LEFT`	Cursor left
`GLFW_KEY_RIGHT`	Cursor right
`GLFW_KEY_LSHIFT`	Left shift key
`GLFW_KEY_RSHIFT`	Right shift key
`GLFW_KEY_LCTRL`	Left control key
`GLFW_KEY_RCTRL`	Right control key
`GLFW_KEY_LALT`	Left alternate function key
`GLFW_KEY_RALT`	Right alternate function key
`GLFW_KEY_LSUPER`	Left super key, WinKey, or command key
`GLFW_KEY_RSUPER`	Right super key, WinKey, or command key
`GLFW_KEY_TAB`	Tabulator
`GLFW_KEY_ENTER`	Enter
`GLFW_KEY_BACKSPACE`	Backspace
`GLFW_KEY_INSERT`	Insert
`GLFW_KEY_DEL`	Delete
`GLFW_KEY_PAGEUP`	Page up
`GLFW_KEY_PAGEDOWN`	Page down
`GLFW_KEY_HOME`	Home
`GLFW_KEY_END`	End
`GLFW_KEY_KP_`*n*	Keypad numeric key *n* (*n* can be in the range 0..9)
`GLFW_KEY_KP_DIVIDE`	Keypad divide (_)
`GLFW_KEY_KP_MULTIPLY`	Keypad multiply (_)
`GLFW_KEY_KP_SUBTRACT`	Keypad subtract (□)

`GLFW_KEY_KP_ADD`	Keypad add (+)
`GLFW_KEY_KP_DECIMAL`	Keypad decimal (. or ,)
`GLFW_KEY_KP_EQUAL`	Keypad equal (=)
`GLFW_KEY_KP_ENTER`	Keypad enter
`GLFW_KEY_KP_NUM_LOCK`	Keypad num lock
`GLFW_KEY_CAPS_LOCK`	Caps lock
`GLFW_KEY_SCROLL_LOCK`	Scroll lock
`GLFW_KEY_PAUSE`	Pause key
`GLFW_KEY_MENU`	Menu key

`glfwGetMouseButton :button`

Function. Queries the current state of a specific mouse `button`. The function returns `:GLFW_PRESS` if the button is held down, or `:GLFW_RELEASE` if the button is not held down. The value of `button` can be:

`GLFW_MOUSE_BUTTON_LEFT`	Left mouse button (button 1)
`GLFW_MOUSE_BUTTON_RIGHT`	Right mouse button (button 2)
`GLFW_MOUSE_BUTTON_MIDDLE`	Middle mouse button (button 3)
`GLFW_MOUSE_BUTTON_`*n*	Mouse button *n* (*n* can be in the range 1..8)

`glfwGetMousePos`

Function. Returns the current mouse position as a list of two numbers. If the cursor is not hidden, the mouse position is the cursor position, relative to the upper left corner of the window.

`glfwSetMousePos :xpos :ypos`

Command. Changes the position of the mouse. If the cursor is visible (not disabled), the cursor will be moved to the specified position, relative to the upper left corner of the window. If the cursor is hidden (disabled), only the mouse position that is reported by GLFW is changed.

`glfwGetMouseWheel`

Function. Returns the current mouse wheel position.

`glfwSetMouseWheel :pos`

Command. Changes the position of the mouse wheel.

(f) Callback functions

Callback functions are used to provide user-functionality executed when a special event occurs.

```
glfwSetCallback :glfwFunc :userFunc
```

Command. This command is a Lhogho extension to GLFW. It hooks a user function or command given by its name `userFunc` to a dedicated GLFW function with name in `glfwFunc`. The values of `glfwFunc` are predetermined and they could be:

`glfwSetWindowClose`
The user function with a name in `userFunc` will be called whenever a user requests that the window should be closed, typically by clicking the window close icon. The user function should have no inputs. The return value of the user function indicates whether or not the window close action should continue. If the returned value is `GL_TRUE`, the window will be closed. If the returned value is `GL_FALSE`, the window will not be closed.

`glfwSetWindowSize`
The user command with a name in `userFunc` will be called whenever the window size changes. The user command should have two inputs (`width` and `height`).

`glfwSetWindowRefresh`
The user command with a name in `userFunc` will be called whenever there are window refresh events, which occur when any part of the window client area has been damaged, and needs to be repainted (for instance, if a part of the window that was previously occluded by another window has become visible). The user command should have no inputs.

`glfwSetKey`
The user command with a name in `userFunc` will be called whenever a key is pressed or released. The user command should have two inputs. The first one is the key (for its values see `glfwGetKey`) and the second is the action, which values are `:GLFW_PRESS` or `:GLFW_RELEASE`.

`glfwSetChar`
The user command with a name in `userFunc` will be called whenever a printable character is generated by the keyboard. The user command should have two inputs. The first one is the character, which values are Unicode characters, and the second is the action, which values are `:GLFW_PRESS` or `:GLFW_RELEASE`.

`glfwSetMouseButton`

The user command with a name in `userFunc` will be called whenever a mouse button is pressed or released. The user command should have two inputs. The first one is the button (see `glfwGetMouseButton` for details) and the second is the action, which values are `:GLFW_PRESS` or `:GLFW_RELEASE`.

`glfwSetMousePos`

The user command with a name in `userFunc` will be called whenever the mouse is moved. The user command should have two inputs – the X and Y co-ordinates of the mouse.

`glfwSetMouseWheel`

The user command with a name in `userFunc` will be called whenever the mouse wheel is rolled. The user command should have one input – the mouse wheel position.

Follows an example how to capture window resizes:

```
to resize :w :h
  (print [New size] :w "x :h)
end
glfwSetCallback "glfwSetWindowSize "resize
```

The command `resize` is called whenever the window is resized.

`glfwClearCallback :glfwFunc`

Command. This command is a Lhogho extension to GLFW. It unhooks a dedicated GLFW function with name in `glfwFunc` from whatever user function it is hooked to by `glfwSetCallback`. The values of `glfwFunc` are predetermined and are the same as in `glfwSetCallback`.

Follows an example how to stop capturing window resizes:

```
glfwClearCallback "glfwSetWindowSize
```

`glfwSetWindowSizeCallback :address`

Command. Sets (hooks) or clears (unhooks) the callback function for window size change events. It is preferred to use the more user-friendly command `glfwSetCallback` and `glfwClearCallback` instead of manually setting the call back function.

`glfwSetWindowCloseCallback :address`

Command. Sets (hooks) or clears (unhooks) the callback function for window close events. It is preferred to use the more user-friendly command `glfwSet-`

`Callback` and `glfwClearCallback` instead of manually setting the call back function.

`glfwSetWindowRefreshCallback :address`

Command. Sets (hooks) or clears (unhooks) the callback function for window refresh events. It is preferred to use the more user-friendly command `glfwSetCallback` and `glfwClearCallback` instead of manually setting the call back function.

6. Euler

The `Euler` library provides numerical functions for integers of arbitrary length as well as other functions like sorting and permutation.

`sort :value`

Function. Outputs the input rearranged into alphabetical or numerical order. If the input is a number or word, the characters are ordered from 0 to z. If the input is a list of words they are ordered alphabetically, if a list of numbers, they are ordered by size. If the input is a list of numbers *and* words, the numbers are ordered first by size followed by the words ordered alphabetically. If the input is a list of lists, the sublists are ordered internally, but the list order is unchanged. If the input is a list of lists *and* numbers or words or both, the lists, ordered internally, are moved to the left followed by the remainder ordered as in previous cases.

```
print sort 132546
123456
print sort "b2a5c4
245abc
show sort [2 13 a c 3 b 1]
[1 2 3 13 a b c]
show sort [2 3 a [2 3 2 1] c 3 b 1]
[[1 2 2 3] 1 2 3 3 a b c]
```

The maximum length of the input is about 10 000 members for a list of numbers but this drops to about 250 otherwise. The procedure uses a partition sort in the former case and a selection sort in the latter.

`factors :value` `DE: faktoren`

Function. Outputs a list of the factors of its input which must be a positive integer.

```
show factors 36
[1 2 3 4 6 9 12 18]
show factors 1
[]
```

`perms :value` DE: `permut`

Function. Outputs a list of the permutations of its input word or number. The number of permutations is n! where n is the length of the input.

```
show perms 123
[123 132 231 213 312 321]
```

Note, that for long inputs the number of permutations may be too big to fit in the available memory.

`allperms :value` DE: `allepermut`

Function. Outputs a list of all the permutations of its input word or number and its subsets. The last member is the empty word.

```
show allperms 123
[123 12 132 13 1 231 23 213 21 2 312 31 321 32 3 ]
show count allperms "a
2
```

`prperms :value` DE: `dzpermut`

Command. Prints each of the permutations of its input word or number. The number of permutations is $n!$ where n is the length of the input.

```
prperms "gas
gas
gsa
asg
ags
sga
sag
prperms 1
1
```

`prallperms :value` DE: `dzallepermut`

Command. Prints each of the permutations of its input and the subsets of its input. The last one printed is the empty word.

```
prallperms "gas
gas
ga
gsa
gs
g
asg
as
ags
ag
a
sga
sg
sag
sa
s
```

`ppt :value` `DE: gpt`

Function. Outputs a list of primitive Pythagorean triples generated from [3 4 5] using a UAD Tree[1]. Each triple in a level generates three triples in the next level. The input, which must be a non-negative integer, specifies the number of levels to generate. The UAD tree contains only *primitive* Pythagorean triples (i.e. 3, 4, 5 but not 6, 8, 10) and generates all primitive Pythagorean triples, i.e. any primitive Pythagorean triple will eventually be generated by using enough levels of the UAD tree.

```
show ppt 2
[[3 4 5] [5 12 13] [21 20 29] [15 8 17] [7 24 25] [55
48 73] [45 28 53] [39 80 89] [119 120 169] [77 36 85]
[33 56 65] [65 72 97] [35 12 37]]
print count ppt 7
3280
```

---

[1] Knott R., *Pythagorean Triangles and Triples: The UAD Tree of Primitive Pythagorean Triangles*; http://www.mcs.surrey.ac.uk/Personal/R.Knott/Pythag/pythag.html#uadgen

A commandlist stored in a variable named `"uadrun` will be executed once for each generated triple. Each generated triple is created and stored as part of a group of three using the variables `:utriple,` `:atriple` and `:dtriple`. An example of the use of `:uadrun,` below, calculates the percentage of triples up to generated level 5 whose element sum is divisible by 10 (an example is 5, 12, 13 with element sum 30)

```
local "divby10
make "divby10 0
make "uadrun [
   if 0=last(sumlist :utriple)[make "divby10
1+:divby10]
   if 0=last(sumlist :atriple)[make "divby10
1+:divby10]
   if 0=last(sumlist :dtriple)[make "divby10
1+:divby10]
]
local "total
make "total count ppt 5
(pr :divby10 "/ :total "= word :divby10/:total*100 "%)
111 / 364 = 30.4945054945055%
```

Note that the "seed" for the UAD tree, `[3 4 5]` must be considered separately. It is counted in `:total` but is not examined as part of `:uadrun`.

`prppt :value` DE: `dzgpt`

Command. Prints a listing of primitive Pythagorean triples generated from `[3 4 5]` using a UAD tree. Each triple in a level generates three triples in the next level. The input, which must be a non-negative integer, specifies the number of levels to generate. The UAD tree contains only *primitive* Pythagorean triples (ie 3,4,5 but not 6,8,10) and any primitive Pythagorean triple will eventually be generated by using enough levels of the UAD tree.

```
prppt 0
[3 4 5]
prppt 2
[3 4 5]
[5 12 13] [21 20 29] [15 8 17]
[7 24 25] [55 48 73] [45 28 53] [39 80 89] [119 120
169] [77 36 85] [33 56 65] [65 72 97] [35 12 37]
```

A commandlist stored in a variable named `"uadrun` will be executed for each generated triple (see further discussion under procedure ppt).

`fibonacci :digitlimit`

Function. Outputs a list of Fibonacci numbers[2] up to the specified maximum digit length. Provides for "long integers" to an arbitrary number of digits in the answer.

```
show fibonacci 2
[1 1 2 3 5 8 13 21 34 55 89]
show count fibonacci 500
2394
```

A commandlist stored in a variable named `"fibrun` will be executed once for each generated Fibonacci number. The generated number can be accessed through the variable `:current`. An example of the use of `:fibrun`, below, determines how many Fibonacci numbers up to 100 digits in length do not contain the digit 6.

```
local "n
make "n 0
make "fibrun [if not member? 6 :current [make "n
1+:n]]
make "tot count fibonacci 100
(pr :n "/ :tot [fibs up to 100 digits long don’t
contain 6])
57 / 480 fibs up to 100 digits long don't contain 6
```

`sumlist :list` DE: `sumliste`

Function. Outputs the sum of the elements of its input list.

```
print sumlist [1 2 3 4]
10
print sumlist [1.1 2.2 3.3]
6.6
```

---

[2] Chandra, Pravin and Weisstein, Eric W. *Fibonacci Number.* From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/FibonacciNumber.html

`sumlistl :list` DE: `sumlistel`

Function. Outputs the sum of the elements of its input list. It provides for "long integers" which can be of arbitrary length. Any member of the input list that is not in the form of an integer will cause the sum to terminate at that point.

```
print sumlistl [1 99999999999999999999999999999999 1]
100000000000000000000000000000001
print sumlistl [1 2 3.1 4]
3
```

`integerp :value` DE: `ganzp`
`integer? :value` DE: `ganz?`

Function. Outputs `true` if the input is an integer, `false` otherwise. Maximum integer value is `9223372036854775296` (approx `9.22e18`)

```
print integerp 123
true
print integer? "123
true
print integerp "123s
false
print integer? 9.22e18
true
print integer? 9.23e18
false
```

`lessthanp :value :value` DE: `kleineralsp`
`lessthan? :value :value` DE: `kleinerals?`

Function. If both inputs are numbers, outputs `true` if the first input is numerically smaller than the second, `false` otherwise. If either or both inputs are not numbers, outputs `true` if the first comes alphabetically before the second.

```
print lessthanp 4 123
true
print lessthan? "d "abc
false
```

```
gcd :value :value                                    DE: ggt
```

Function. Outputs the greatest common divisor[3] (highest common factor) of its inputs. Both must be integers and the result is an integer with the same sign as the smaller valued one.

```
print gcd 35 14
7
print gcd -3 8
-1
```

```
palindromep :value                           DE: palindromp
palindrome? :value                           DE: palindrom?
```

Function. Outputs `true` if the input word or list is a palindrome, i.e. the same forwards as it is backwards, `false` otherwise.

```
print palindromep "level
true
print palindrome? "a
true
print palindrome? "abc
false
print palindrome? [a bbc a]
true
```

```
decrement :value                              DE: dekrement
```

Function. Outputs a word that is numerically one less than its input which must be in the form of an integer. It provides for "long integers" which can contain an arbitrary number of characters.

```
print decrement 6
5
print decrement "99999999999999999999999
99999999999999999999998
print decrement "-99999999999999999999999
-100000000000000000000000
```

---

[3] Weisstein, Eric W. *Greatest Common Divisor.* From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/GreatestCommonDivisor.html

Failure to quote the input can lead to incorrect results if the value lies outside Lhogho’s integer limit (~ 9.22e18)

```
print decrement 99999999999999999999999 ; this is
wrong
99999999999999991999999
```

increment :value DE: inkrement

Function. Outputs a word that is numerically one more than its input which must be in the form of an integer. It provides for "long integers" which can contain an arbitrary number of characters.

```
print increment 5
6
print increment "99999999999999999999999
100000000000000000000000
print increment "-99999999999999999999999
-99999999999999999999998
```

Failure to quote the input can lead to incorrect results if the value lies outside Lhogho’s integer limit (~ 9.22e18)

```
show increment 99999999999999999999999 ; this is wrong
99999999999999992000001
```

primep :value DE: primp
prime? :value DE: prim?

Function. Outputs `true` if the positive integer input is prime, `false` otherwise.

```
print primep 17
true
print prime? 1
false
```

A global variable named `"primes` exists with initial value `[2 3 5]` that is used in this procedure and in the procedure `prime.factors` to store the list of primes. Additional primes are automatically added to this variable (up to the square root of the input) as needed using the procedure `generate.primes`.

```
show :primes
[2 3 5]
print primep 1001
false
```

```
show :primes
[2 3 5 7 11 13 17 19 23 29 31 37]
```

`prime.factors :value` DE: `primfaktoren`

Function. Outputs a list of the prime factors of its input.

```
show prime.factors 21
[3 7]
show prime.factors 48
[2 2 2 2 3]
show prime.factors 1
[]
```

`generate.primes :value` DE: `erzeuge.primz`

Function. Outputs a list of all the primes less than its input plus the next one.

```
show :primes
[2 3 5]
make "primes generate.primes 70
show :primes
[2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71]
print count generate.primes 10000
1230
```

This procedure begins with the existing `:primes` rather than re-generating from the start so a pre-calculated list of primes can be used and `generate.primes` will simply add to it as required rather than re-calculating from scratch.

`add :value :value`

Function. Outputs the sum of its inputs. It provides for "long integers" which can be of arbitrary length. The inputs should always be quoted if entered explicitly, otherwise erroneous results can occur. If either input is not in the form of an integer, the value output is 0.

```
print add "1 "2
3
print add "99999999999999999999999
"99999999999999999999999
199999999999999999999998
print add "1.1 "2
```

```
0
```

```
prod :value :value
```

Function. Outputs the product of its inputs. It provides for "long integers" which can be of arbitrary length. The inputs should always be quoted if entered explicitly, otherwise erroneous results can occur. If either input is not in the form of an integer, the value output is 0.

```
print prod "3 "2
6
print prod "99999999999999999999999
"99999999999999999999999
9999999999999999999999800000000000000000000001
print prod "1.1 "2
0
```

```
factorial :value                                    DE: faktoriell

:value !                                                    EN, DE
```

Function. Outputs the factorial[4] of its input. Provides for "long integers" to an arbitrary number of digits in the answer.

```
print factorial 4 ; 4x3x2x1
24
print 28 !
304888344611713860501504000000
```

```
ncr :n :r                                                  DE: nkr
binomial :value value                                 DE: binominal
```

Function. Outputs "n choose r" or the binomial coefficient[5] specified by its inputs, i.e. $\frac{n!}{r!(n-r)!}$, as represented in Pascal's triangle.

```
print ncr 6 2
15
print binomial 50 25
```

---

[4] Weisstein, Eric W. *Factorial*. From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/Factorial.html

[5] Weisstein, Eric W. *Binomial Coefficient*. From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialCoefficient.html

```
126410606437752
```

`rotatel :value` `DE: rotierel`

Function. Outputs its input rotated to the left by one, ie with the first element moved to the end.

```
show rotatel 1234
2341
show rotatel [a b c d]
[b c d a]
```

`rotater :value` `DE: rotierer`

Function. Outputs its input rotated to the right by one, ie with the last element moved to the beginning.

```
show rotater 1234
4123
show rotater [a b c d]
[d a b c]
```

`dec2bin :value` `DE: dekzubin`

Function. Outputs the decimal input number converted to binary representation.

```
print dec2bin 9
1001
print dec2bin 1048576
100000000000000000000
```

`bin2dec :value` `DE: binzudek`

Function. Outputs the binary input number converted to decimal representation. It is advisable to quote large input numbers (>20 digits).

```
print bin2dec 1001
9
print bin2dec "100000000000000000000
1048576
```

`dec2hex :value` `DE: dekzuhex`

Function. Outputs the decimal input number converted to hexadecimal representation.

```
print dec2hex 27
```

```
1B
print dec2hex 1048575
FFFFF
```

`hex2dec :value` DE: `hexzudek`

Function. Outputs the input word in hexadecimal number format converted to decimal representation.

```
print hex2dec "1b
27
print hex2dec "FFFFF
1048575
```

`list2word :value` DE: `listezuwort`

Function. Outputs a word formed by concatenating the words in the input list.

```
print list2word [no w he re]
nowhere
print word list2word [1 2 3] 10
12310
```

`word2list :value` DE: `wortzuliste`

Function. Outputs a list of the characters or digits in the input word or number.

```
show word2list "now
[n o w]
show word2list 99*99
[9 8 0 1]
```

Chapter V. Applications

Applications described in this section are Lhogho programs that can be compiled and used as standalone applications. Each application can be run by Lhogho, but can also be compiled in a standalone executable and be used without Lhogho.

To run an application with Lhogho (and without compiling it into a standalone executable file), use the command:

```
lhogho app params
```

Where `app` is the name of the source code of the application (together with the file extension `.lgo`) and `params` are the parameters given to the application.

To create a standalone executable application use the command:

```
lhogho -x app
```

which will create file `app.exe` (in Windows) or `app` (in Linux). To use this file execute it as any other application:

```
app params
```

1. Hello World

This application is the famous Hello World program. It just prints the text `Hello world`.

```
hello
Hello world
```

2. Simple CLI

CLI stand for command-line interpreter. `CLI.lgo` implements a simple CLI, which accepts one-line commands. The command prompt is `Lhogho>`. To exit the interpreter type `RETURN` key without any command. Note that this CLI accepts only commands on a single line.

```
cli
Lhogho> make "a 100
Lhogho> print :a
100
Lhogho> to mid :x output :x/2 end
Lhogho> make "b mid mid :a
Lhogho> print :b
25
Lhogho>
```

3. Prime Numbers

The application `Primes` is used to print the primes numbers up to a given upper boundary. It is based on a simple search for primary numbers by building a list of already found primes.

```
primes
Lhogho Primes 1.0 - Prints the prime numbers up to a
limit
Usage: primes limit
```

The application requires a single parameter – the upper limit. It will print all prime numbers from 2 to the upper limit (inclusive). To get the prime numbers not greater than 40, execute this command:

```
primes 10
2 3 5 7

primes 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97

primes 500
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97 101 103 107 109 113 127 131 137 139
149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359 367
373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499
```

4. Calculator

The application `Calc` is used to calculate a mathematical expression. The expression is provided as one or more parameters to the application.

```
calc
```

```
Lhogho Calculator 1.0 - Calculates a mathematical
expression
Usage: calc "expression"
```

It is better to frame the expression in double quotes; otherwise the current shell may try to parse them. Simple expression without parentheses does not require double quotes:

```
calc 1+2+3
6

calc "10*(sin 30) - 1/2"
4.5

calc exp 1
2.71828182845905
```

5. Square Root

The application `Sqrt` is used to calculate the square root of a number with the Newton's method.

```
lhogho sqrt.lgo
Lhogho SquareRoot 1.0 - Calculates square root with
Newton's method
Usage: sqrt number
```

The application requires a single parameter – a positive number. It will print all iterations starting from 1 until the difference between two successive iterations becomes too small. The last printed iteration is the final calculation:

```
sqrt 2
 -> 1
 -> 1.5
 -> 1.41666666666667
 -> 1.41421568627451
 -> 1.41421356237469
 -> 1.41421356237309

sqrt 121
 -> 1
 -> 61
 -> 31.49180327868853
 -> 17.66703646495592
 -> 12.25797485371191
 -> 11.06454984414054
 -> 11.00018828973782
 -> 11.00000000161147
 -> 11
```

6. Cube3D

`Cube3D` is a graphical application animating a cube rotation using OpenGL commands. The animation can be accelerated or slowed down by pressing left or right arrows. `ESC` key closes the application.

There are two versions of the program – one uses `GLUT` (`cube3d-glut.lgo`) and one uses `GLFW` (`cube3d-glfw.lgo`).

```
lhogho cube3d-glfw.lgo
Cube3D - A rotating OpenGL cube
Press ESC to exit
Use left and right arrows to change speed
```

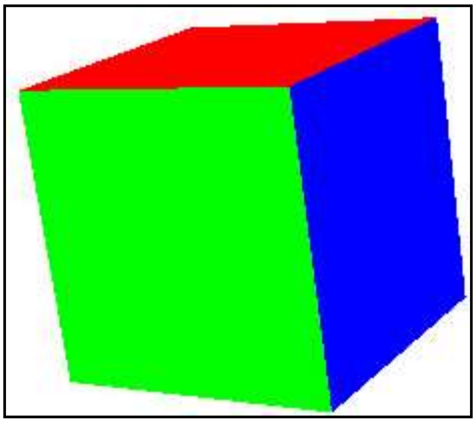

Figure 2 Snapshot of the Cube3D application

7. Mandelbrot

`Mandelbrot` is a graphical application drawing the Mandelbrot set fractal. Parameters of the program control which area of the set is explored, as well as at what magnification and color scheme. One the application is started users may zoom in (with a click of the left mouse button), zoom out (click with the right mouse button) or exit the application (by pressing the `ESC` key).

The parameters of `Mandelbrot` are not compulsory. They are:

- `width` – width of the graphical window in pixels (default value `600`)
- `height` – height of the graphical window in pixels (default value `400`)
- `center x` – abscissa of the central point (default value `0`)
- `center y` – ordinate of the central point (default value `0`)
- `scale` – zoom factor (default value `100`)
- `loops` – number of repetitions of color band (default value `1`)
- `bandname` – the name of a color band as defined in mandelbrot.colors.lgo (default value is rainbow)

The size of graphical window is set by the command line parameters, but the operating system determines the actual size of the window following the GUI policies. Thus, `width` and `height` define only the desired windows size.

The command line parameters for `center x` and `y` shifts the viewing area in a way that this center matches the center of the graphical window.

The initial zoom factor is defined by `scale`. Scale equal to 1 sets one mathematical unit length to be one pixel. Because the Mandelbrot set fits in a circle with radius 2, scale 1 will make the fractal just few pixels big. A starting scale of 100 or 150 is suggested for viewing the whole Mandelbrot set.

The precision of floating point operations in Lhogho permits scaling up to 10^{15}. Larger scales produce images with artifacts. While zooming in or out the current scale is displayed in the caption of the graphical window.

Clicking with the left mouse button restarts fractal drawing at a ten times higher scale and a new center point (defined by the click location). Zooming out with the right mouse button switched to a 10 times smaller scale.

While the mouse I moved over the graphical window, a small rectangle shows the area which will be visible if zoomed in. To hide this rectangle move the mouse pointer near the top of bottom area of the graphical window.

Drawing Mandelbrot set computes a number for each pixel. This number determines the color of the pixel. Colors for all possible values are defined as bands in an external file `mandelbrot.colors.lgo`. Each band is defined as a list which first element is the background color (it is used for every pixel with undetermined color). The next parameters are pairs of color index and colors. Color indices must be in ascending order. Colors which fall in-between two indices are interpolated. For example, the default rainbow band is defined as:

```
(make "band.rainbow [
  [0 0 0]         ; background
```

```
        000 [0 0 0]      ; black
        100 [1 0 0]      ; red
        200 [1 1 0]      ; yellow
        300 [0 1 0]      ; green
        400 [0 1 1]      ; cyan
        500 [0 0 1]      ; blue
        600 [0 0 0]      ; black
    ])
```

which corresponds to this color band:

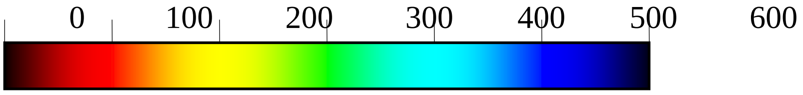

Figure 3 Default rainbow color band

There are three predefined bands – rainbow (default), bw and gold.

The `loop` parameter defines how many tiled bands will be used. If it is one, then only 1 band is considered. In the case of rainbow band, all pixels for which calculated value is above 600 are treated as pixels with background color. If `loops` is 5, then the rainbow band is tiled five times and pixels with value up to 3000 (i.e. loops×bandsize) will pick color from the band. Note that rainbow band tiled 5 times will have 5 areas with reds, yellows, greens, etc. Longer bands and higher loop counts make calculations much slower especially if there are many areas with background colors.

```
mandelbrot 600 600 -1 0 200 1 bw
```

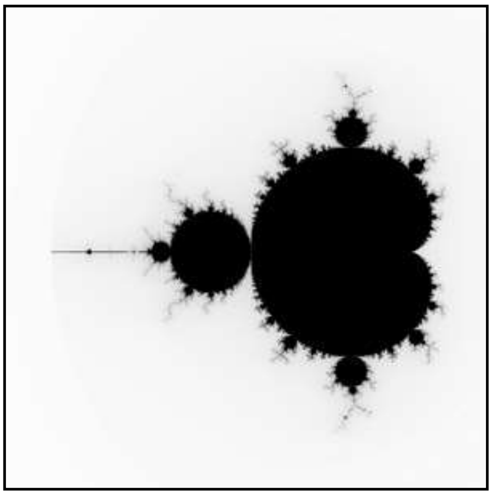

Figure 4 The whole Mandelbrot set based on the bw color band

The next example shows full-length command line parameters (both lines are actually a single long line):

```
mandelbrot 600 600 -0.74662112123098 -0.11151627135542
3e14 10 gold
```

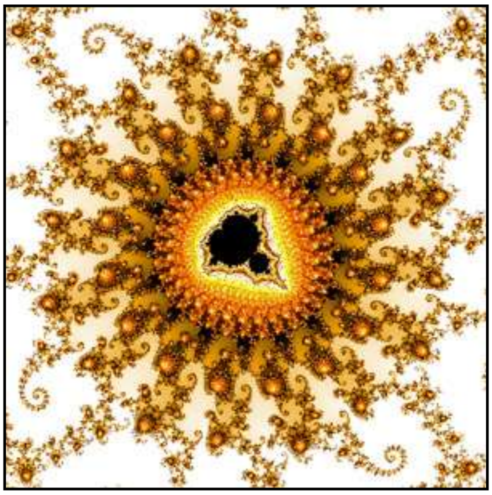

Figure 5 The gold color band and extreme zoom of 3^{14}

```
mandelbrot 600 600 -0.1185105 -0.8830802 1e7 2 rainbow
```

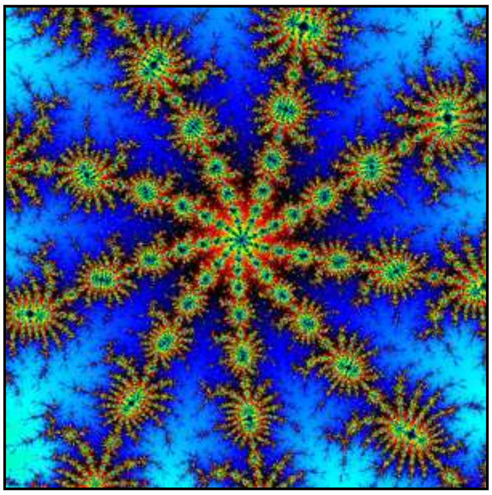

Figure 6 The rainbow band

Chapter VI. Appendices

1. Format strings

This appendix lists format strings used by `format` (for numbers) and `formattime` (for times and dates). For more details about the strings and discussion about subtle nuances check an online documentation about GCC functions `printf()` and `strftime()`.

(a) Format strings for numbers

Format		Explanation
Integer	`%d`	An integer as a signed decimal number.
	`%i`	An integer as a signed decimal number.
	`%u`	An integer as an unsigned decimal number.
	`%x`	An integer as an unsigned hexadecimal number with lower-case letters.
	`%X`	An integer as an unsigned hexadecimal number with upper-case letters.
	`%o`	An integer as an unsigned octal number.
Floating point	`%f`	A floating-point number in normal (fixed-point) notation.
	`%e`	A floating-point number in exponential notation with lower-case letters.
	`%E`	A floating-point number in exponential notation with upper-case letters.
	`%g`	A floating-point number in either normal or exponential notation, whichever is more appropriate for its magnitude with lower-case letters.
	`%G`	A floating-point number in either normal or exponential notation, whichever is more appropriate for its magnitude with upper-case letters.
	`%a`	A floating-point number in a hexadecimal fractional notation which the exponent to base 2 represented in decimal digits with lower-case letters.
	`%A`	A floating-point number in a hexadecimal fractional notation which the exponent to base 2 represented in decimal digits with upper-case letters.
Char	`%c`	A single character.
	`%C`	A single UTF-16 character.
String	`%s`	A string.
	`%S`	A UTF-16 string.
Pointer	`%p`	A pointer (i.e. an address in the memory).

(b) Format strings for date

Format		Explanation
Date	`%D`	The date using the format `%m/%d/%y`.
	`%F`	The date using the format `%Y-%m-%d`.
	`%x`	The preferred date representation for the current locale.
Century	`%C`	The century of the year.
Year	`%g`	The year corresponding to the week number (00…99).
	`%G`	The year corresponding to the week number.
	`%y`	The year without a century as a decimal number (00…99).
	`%Y`	The year as a decimal number.
Month	`%b`	The abbreviated month name according to the current locale.
	`%B`	The full month name according to the current locale.
	`%h`	The abbreviated month name according to the current locale.
	`%m`	The month as a decimal number (01…12).
Week	`%a`	The abbreviated weekday name according to the current locale.
	`%A`	The full weekday name according to the current locale.
	`%U`	The week number of the current year (00… 53), starting with the first Sunday as the first day of the first week. Days preceding the first Sunday in the year are considered to be in week 00.
	`%V`	The week number (01…53). Starts with Monday and end with Sunday. Week 01 of a year is the first week which has the majority of its days in that year.
	`%W`	The week number (00…53), starting with the first Monday as the first day of the first week. All days preceding the first Monday in the year are considered to be in week 00.
Day	`%d`	The day of the month (01…31).
	`%e`	The day of the month padded with blank (1… 31).
	`%j`	The day of the year (001…366).
	`%u`	The day of the week (1…7), Monday being 1.
	`%w`	The day of the week (0…6), Sunday being 0.

(c) Format strings for time

Format		Explanation
Zone	`%Z`	The time zone abbreviation (empty if the time zone can't be determined).
	`%z`	RFC 822/ISO 8601:1988 style numeric time zone (e.g., -0600 or +0100), or nothing if no time zone is determinable.
Time	`%r`	The complete calendar time using the AM/PM format of the current locale.
	`%T`	The time of day using decimal numbers using the format %H:%M:%S.
	`%R`	The hour and minute in decimal numbers using the format %H:%M.
	`%c`	The preferred calendar time representation for the current locale.
	`%X`	The preferred time of day representation for the current locale.
AM/PM	`%p`	Either ‘AM’ or ‘PM’ if the current locale supports ‘AM’/‘PM’ format, empty string otherwise.

	`%P`	Either ‘am’ or ‘pm’ if the current locale supports ‘AM’/‘PM’ format, empty string otherwise.
Hour	`%H`	The hour using a 24-hour clock (00…23).
	`%I`	The hour using a 12-hour clock (01…12).
	`%k`	The hour using a 24-hour clock padded with blank (0…23).
	`%l`	The hour using a 12-hour clock padded with blank (1…12).
Minute	`%M`	The minute (00…59).
Second	`%s`	The number of seconds since 1970-01-01 00:00:00 UTC.
	`%S`	The seconds (00…60).

2. Index of primitives