

newLISPTM

For BSDs, Linux, Mac OS X, Solaris and Win32

Users Manual and Reference v.9.0 rev 2

Copyright © 2006 Lutz Mueller. www.nuevatec.com. All rights reserved.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled [GNU Free Documentation License](#).

The accompanying software is protected by the [GNU General Public License](#) V.2, June 1991.

newLISP is a trademark of Lutz Mueller.

Contents

Contents.....	3
newLISP Users Manual.....	15
1. Introduction.....	15
newLISP-tk	16
Licensing	16
2. Deprecated functions and future changes	16
3. Command-line options and directories	17
Stack size	17
Maximum memory usage	17
Suppressing the prompt	17
newLISP as a TCP/IP server	18
TCP/IP daemon mode	18
inetd daemon mode.....	19
Daemon mode with handler function	21
Direct execution mode	22
Logging I/O	22
Command line help summary	22
The initialization file init.lsp	23
Directories on Linux, BSD, and Mac OS X	23
Directories on Win32/newLISP-tk	23
4. Shared library module for Linux/BSD versions	23
5. DLL module for Win32 versions	25
6. Evaluating newLISP expressions	25
Integer data, floating point data, and operators	26
Evaluation rules and data types	27
7. Lambda expressions in newLISP	30
8. nil, true, cons, and ()	31
9. Arrays	32
10. Dictionaries (hash tables)	33
11. Indexing elements of strings, lists, and arrays	34
Implicit indexing for nth	34
Implicit indexing and the default functor	34
Implicit indexing for rest and slice	35
Implicit indexing for nth-set and set-nth.....	35
12. Destructive versus nondestructive functions	36
13. Dynamic and lexical scoping	37

newLISP Users Manual and Reference

14. Early return from functions, loops, and blocks	38
Using catch and throw	38
Using and and or	39
15. Contexts	39
Scoping rules for contexts	40
Changing scoping	42
Symbol protection	43
Overwriting global symbols and built-ins	43
Variables containing contexts	43
Sequence of creating or loading contexts	44
Symbol creation in contexts	45
16. Programming with context objects	45
Late binding of context symbols	46
The context default function	47
Passing objects by reference	48
Contexts as prototypes	48
Lexical and static scoping in newLISP	49
Serializing context objects	50
17. XML, S-XML, and XML-RPC	51
18. Customization, localization, and UTF-8	53
Switching the locale	53
Decimal point and decimal comma	54
Unicode and UTF-8 encoding	54
19. Commas in parameter lists	57
20. Linking newLISP source and executable	57
newLISP Function Reference.....	59
1. Syntax of symbol variables and numbers.....	59
Symbols for variable names.....	59
Numbers.....	60
2. Data types and names in the reference.....	61
bool.....	61
int.....	61
num.....	61
matrix.....	62
str.....	62
sym.....	63
context.....	63
sym-context.....	63
func.....	63
list.....	63
array.....	64
exp.....	64
body.....	64
3. Functions in groups.....	64

newLISP Users Manual and Reference

List processing, flow control, and integer arithmetic.....	64
Bit operators.....	66
Floating point math and special functions.....	66
Matrix functions.....	68
Array functions.....	68
Financial math functions.....	68
Simulation and modeling math functions.....	68
Time and date functions.....	69
String and conversion functions.....	69
Input/output and file operations.....	70
Processes, pipes and threads.....	71
File and directory management.....	71
System functions and predicates.....	72
HTTP networking API.....	73
Socket TCP/IP and UDP network API.....	73
Importing libraries.....	74
newLISP internals API.....	74
Functions in alphabetical order.....	75
!	75
\$	75
+, -, *, / , %	76
<, >, =, <=, >=, !=	77
<<, >>	78
&	78
	79
^	79
~	79
abs	79
acos	80
add	80
address	80
amb	81
and	81
append	81
append-file	82
apply	83
args	84
array	85
array-list	86
array?	87

newLISP Users Manual and Reference

asin.....	87
assoc.....	87
atan.....	88
atan2.....	88
atom?.....	88
base64-dec.....	89
base64-enc.....	89
bayes-query.....	90
R.A. Fisher Chi ² method.....	91
Chain Bayesian method.....	92
Specifying probabilities instead of counts.....	92
bayes-train.....	93
begin.....	94
beta.....	95
betai.....	95
binomial.....	95
case.....	96
catch.....	96
ceil.....	97
change-dir.....	98
char.....	98
chop.....	99
clean.....	99
close.....	100
command-line.....	100
cond.....	100
cons.....	101
constant.....	102
context.....	103
context?.....	105
copy-file.....	105
cos.....	106
count.....	106
cpymem.....	106
crc32.....	107
crit-chi2.....	108
crit-z.....	108
current-line.....	108

newLISP Users Manual and Reference

date.....	109
date-value.....	111
debug.....	111
dec.....	112
define.....	112
define-macro.....	114
def-new.....	116
delete.....	116
delete-file.....	117
det.....	117
device.....	118
difference.....	118
directory.....	119
directory?.....	119
div.....	120
dolist.....	120
dotimes.....	121
dotree.....	121
do-until.....	122
do-while.....	122
dump.....	123
dup.....	123
empty?.....	124
encrypt.....	124
ends-with.....	125
env.....	126
erf.....	126
error-event.....	127
error-number.....	127
error-text.....	127
eval.....	128
eval-string.....	128
exec.....	129
exit.....	130
exp.....	130
expand.....	130
explode.....	132

newLISP Users Manual and Reference

factor.....	132
fft.....	133
file-info.....	133
file?.....	134
filter.....	134
find.....	135
find-all.....	136
first.....	137
flat.....	137
fn.....	138
float.....	138
float?.....	139
floor.....	139
flt.....	140
for.....	140
fork.....	141
format.....	142
fv.....	144
gammai.....	145
gammaln.....	145
gcd.....	145
get-char.....	146
get-float.....	146
get-int.....	147
get-string.....	148
get-url.....	148
global.....	150
if.....	150
ifft.....	151
import.....	151
inc.....	154
index.....	154
int.....	155
integer?.....	156
intersect.....	156
invert.....	156
irr.....	157

newLISP Users Manual and Reference

join.....	158
lambda.....	158
lambda-macro.....	158
lambda?.....	158
last.....	159
legal?.....	159
length.....	160
let.....	160
letex.....	161
letn.....	162
list.....	162
list?.....	163
load.....	163
local.....	164
log.....	164
lookup.....	164
lower-case.....	165
macro?.....	165
main-args.....	166
make-dir.....	167
map.....	167
match.....	168
max.....	169
member.....	169
min.....	170
mod.....	170
mul.....	171
multiply.....	171
name.....	171
NaN?.....	172
net-accept.....	172
net-close.....	173
net-connect.....	173
UDP communications.....	174
UDP multicast communications.....	174
UDP broadcast communications.....	175
net-error.....	175
net-eval.....	176

newLISP Users Manual and Reference

Raw mode.....	178
net-listen.....	179
UDP communications.....	180
UDP multicast communications.....	181
net-local.....	181
net-lookup.....	182
net-peek.....	182
net-peer.....	182
net-ping.....	183
net-receive.....	184
net-receive-from.....	185
net-receive-udp.....	185
net-select.....	186
net-send.....	188
net-send-to.....	188
net-send-udp.....	189
net-service.....	189
net-sessions.....	190
new.....	190
nil?.....	191
not.....	191
normal.....	192
now.....	192
null?.....	193
nper.....	194
npv.....	194
nth.....	194
nth-set.....	196
number?.....	198
open.....	198
or.....	199
pack.....	199
parse.....	201
peek.....	202
pipe.....	202
pmt.....	203
pop.....	203
post-url.....	204

newLISP Users Manual and Reference

Additional parameters.....	205
pow.....	205
pretty-print.....	206
primitive?.....	206
print.....	207
println.....	207
prob-chi2.....	207
prob-z.....	208
process.....	208
push.....	210
put-url.....	211
Additional parameters.....	213
pv.....	213
quote.....	213
quote?.....	214
rand.....	214
random.....	214
randomize.....	215
read-buffer.....	215
read-char.....	216
read-file.....	216
read-key.....	217
read-line.....	217
real-path.....	218
ref.....	218
ref-all.....	219
regex.....	219
remove-dir.....	221
rename-file.....	222
replace.....	222
List replacement.....	222
String replacement without regular expression.....	222
Regular expression replacement.....	222
List removal.....	223
replace-assoc.....	224
reset.....	225
rest.....	225
reverse.....	226
rotate.....	226

newLISP Users Manual and Reference

save.....	227
search.....	228
seed.....	228
seek.....	229
select.....	229
semaphore.....	230
sequence.....	232
series.....	232
set.....	233
setq.....	234
set-locale.....	234
set-nth.....	235
sgn.....	235
share.....	236
signal.....	238
silent.....	240
sin.....	240
sleep.....	240
slice.....	241
sort.....	241
source.....	242
sqrt.....	243
starts-with.....	243
string.....	244
string?.....	244
sub.....	245
swap.....	245
sym.....	246
symbol?.....	247
symbols.....	247
sys-error.....	248
sys-info.....	248
tan.....	249
throw.....	249
throw-error.....	250
time.....	250
time-of-day.....	251

newLISP Users Manual and Reference

timer.....	251
title-case.....	252
trace.....	253
trace-highlight.....	253
transpose.....	254
trim.....	255
true?.....	255
unicode.....	255
unify.....	256
unique.....	259
unless.....	259
unpack.....	259
until.....	261
upper-case.....	261
utf8.....	261
uuid.....	262
wait-pid.....	262
while.....	263
write-buffer.....	263
write-char.....	264
write-file.....	265
write-line.....	265
xml-error.....	266
xml-parse.....	266
xml-type-tags.....	270
zero?.....	270
newLISP APPENDIX.....	271
Error codes.....	271
TCP/IP and UDP Error Codes.....	272
GNU Free Documentation License.....	273
GNU GENERAL PUBLIC LICENSE.....	279



newLISP Users Manual and Reference

newLISP Users Manual

1. Introduction

newLISP focuses on the core components of LISP: *lists*, *symbols*, and *lambda expressions*. To these, newLISP adds *arrays*, *implicit indexing* on lists and arrays, and *dynamic* and *lexical scoping*. Lexical scoping is implemented using separate namespaces called *contexts*.

The result is an easier-to-learn LISP that is even smaller than most Scheme implementations, but which still has about 300 built-in functions. Approximately 200k in size, newLISP is built for high portability using only the most common UNIX system C-libraries. It loads quickly and has a small memory footprint. newLISP is as fast or faster than other popular scripting languages and uses very few resources.

newLISP is dynamically scoped inside lexically separated contexts (namespaces). Contexts can be used to create isolated protected expansion packages and to write *prototype*-based *object-oriented* programs.

Both built-in and user-defined functions, along with variables, share the same namespace and are manipulated by the same functions. Lambda expressions and user-defined functions can be handled like any other list expression. Like Scheme, newLISP evaluates the operator element of a list expression. Symbols can be protected from accidental change.

Contexts in newLISP facilitate the development of larger applications comprising independently developed modules with their own separate namespaces. They can be copied, dynamically assigned to variables, and passed by reference to functions as arguments. In this way, contexts can serve as dynamically created objects packaging symbols and methods. Lexical separation of namespaces also enables the definition of statically scoped functions.

newLISP's efficient *red-black tree* implementation can handle millions of symbols without degrading performance. Contexts can hold symbol-value pairs, allowing them to be used as hash-tables. Functions are also available to iteratively access symbols inside contexts.

newLISP allocates and reclaims memory automatically, without using traditional asynchronous garbage collection (except under error conditions). All objects — except for contexts, built-in primitives, and symbols — are passed by value and are referenced only once. When objects are no longer referenced, their memory is automatically deallocated. This results in predictable processing times, without the pauses found in traditional garbage collection. newLISP's unique automatic memory management makes it the fastest interactive LISP available.

Many of newLISP's built-in functions are polymorphic and accept a variety of data types. This greatly reduces the number of functions and syntactic forms it is necessary to learn and implement. High-level functions are available for distributed computing, financial math, statistics, and AI.

newLISP has functions to modify, insert, or delete elements inside complex *nested* lists or *multidimensional* array structures.

Since strings can contain null characters in newLISP, they can be used to process binary data.

newLISP can also be extended with a shared library interface to import functions that access data in foreign binary data structures. The distribution contains a module for importing MySQL and ODBC database APIs.

newLISP's HTTP, TCP/IP, and UDP socket interfaces make it easy to write distributed networked applications. Its built-in XML interface, along with its text-processing features — Perl Compatible Regular Expressions (PCRE) and text-parsing functions — make newLISP a useful tool for CGI processing. The source distribution includes an example of HTML forms processing, along with a simple web server written in newLISP.

The source distribution can be compiled for Linux, BSDs, Mac OS X/Darwin, Solaris, and Win32.

newLISP-tk

newLISP-tk is a graphical user interface (GUI) front-end for newLISP written in Tcl/Tk. With newLISP-tk, applications can be built based on the host operating system's native GUI. Third-party interfaces to the GTK libraries and OpenGL graphics library are also available.

Because newLISP and Tcl/Tk are available for most operating systems, newLISP-tk is a platform-independent solution for writing GUI applications in LISP.

For more information on newLISP-tk, see newlisp-tk.html.

Licensing

newLISP and newLISP-tk are licensed under version 2 of the [GPL \(General Public License\)](http://www.gnu.org/licenses/gpl-2.0.html). Both the newLISP and the newLISP-tk manuals are licensed under the [GNU Free Documentation License](http://www.gnu.org/licenses/faq.html). If this license is unsuitable for your application, please contact <http://nuevatec.com> for special arrangements.

(§)

2. Deprecated functions and future changes

<i>old function</i>	<i>new function</i>
get-integer	Use the shorter get-int . The longer get-integer has been eliminated.
integer	Use the shorter int . The longer integer will be eliminated.
symbol	Use the shorter sym . The longer symbol has been eliminated.
newlisp -x	Use other server modes like -p and -d. The -x server mode will be eliminated. HTTP mode for serving static pages is built into newLISP since version 8.9.1 (currently only available on UNIX).

3. Command-line options and directories

When starting newLISP from the command line, it looks for the initialization file `init.lisp` and loads it if present. In this way, one or more options and newLISP source files can be specified at startup. The options and source files are executed in the order listed in `init.lisp`. For options such as `-p` and `-d`, it makes sense to load source files *first*; other options, like `-m` and `-s`, should be specified *before* the source files. The `-e` switch is used to evaluate the program text and then exit; otherwise, evaluation continues interactively (unless an exit occurs while the files are loading).

Stack size

```
newlisp -s 4000
newlisp -s 100000 aprog bprog
newlisp -s 6000 myprog
```

The above examples show starting newLISP with different stack sizes using the `-s` option, as well as loading one or more newLISP source files. When no stack size is specified, the stack defaults to 2048.

Maximum memory usage

```
newlisp -m 128
```

This example limits LISP cell memory to 128 megabytes. In newLISP, each LISP cell consumes 16 bytes, so the argument 128 would represent a maximum of 8,388,608 LISP cells. This information is returned by [sys-info](#) as the list's second element. Although LISP cell memory is not the only memory consumed by newLISP, it is a good estimate of overall memory usage.

Suppressing the prompt

The command-line prompt and initial copyright banner can be suppressed:

```
newlisp -c
```

Listen and connection messages in `-p` and `-d` modes are suppressed if logging is not enabled. The `-c` option is useful when controlling newLISP from other programs; it is mandatory when setting it up as a [net-eval](#) server.

The `-c` option also enables newLISP server nodes running on UNIX like operating systems to answer simple HTTP GET and PUT requests.

A capital C forces prompts when running newLISP in pipe I/O mode inside the Emacs editor:

```
newlisp -C
```

To suppress return values from evaluations, use [silent](#).

newLISP as a TCP/IP server

```
newlisp some.lsp -p 9090
```

This example shows how newLISP can listen for commands on a TCP/IP socket connection. In this case, standard I/O is redirected to the port specified with the `-p` option. `some.lsp` is an optional file loaded during startup, before listening for a connection begins.

The `-p` option is also used to control newLISP from another application, such as a newLISP GUI front-end or a program written in another language.

A telnet application can be used to test running newLISP as a server. First enter:

```
newlisp -p 4711 &
```

The `&` indicates to a UNIX shell to run the process in the background. Now connect with a telnet application:

```
telnet localhost 4711
```

If connected, the newLISP sign-on banner and prompt appear. Instead of 4711, any other port number could be used.

When the client application closes the connection, newLISP will exit, too.

TCP/IP daemon mode

When the connection to the client is closed in `-p` mode, newLISP exits. To avoid this, use the `-d` option instead of the `-p` option:

```
newlisp -d 4711 &
```

This works like the `-p` option, but newLISP does not exit after a connection closes. Instead, it stays in memory, listening for a new connection and preserving its state. An [exit](#) issued from

a client application closes the network connection, and the newLISP daemon remains resident, waiting for a new connection. Any port number could be used in place of 4711.

When running in `-p` or `-d` mode, the opening and closing tags `[cmd]` and `[/cmd]` must be used to enclose multiline statements. They must each appear on separate lines. This makes it possible to transfer larger portions of code from controlling applications. This technique is used in newLISP-tk, a Tcl/Tk front-end to newLISP. It can also be used in the newLISP shell console.

Note that the daemon mode only works correctly on Linux and BSD systems. On Win32-based systems, newLISP may not be able to reconnect at all times.

The tags `[cmd]` and `[/cmd]` can be used to enclose multiline statements, with the opening and closing tags each appearing on separate lines.

The following variant of the `-d` mode is frequently used in a distributed computing environment, together with [net-eval](#) on the client side:

```
newlisp -c -d 4711 &
```

The `-c` spec suppresses prompts, making this mode suitable for receiving requests from the [net-eval](#) function.

newLISP server nodes running on UNIX like operating systems, will also answer simple HTTP GET and PUT requests. This can be used to retrieve and store files with [get-url](#), [put-url](#), [read-file](#), [write-file](#) and [append-file](#), or to load and save programs using [load](#) and [save](#) from and to remote server nodes. On Win32 newLISP server nodes do not answer HTTP requests.

inetd daemon mode

The `inetd` server running on virtually all Linux/UNIX OSes can function as a proxy for newLISP. The server accepts TCP/IP or UDP connections and passes on requests via standard I/O to newLISP. `inetd` starts a newLISP process for each client connection. When a client disconnects, the connection is closed and the newLISP process exits.

`inetd` and newLISP together can handle multiple connections efficiently because of newLISP's small memory footprint, fast executable, and short program load times. When working with [net-eval](#), this mode is preferred for efficiently handling multiple requests in a distributed computing environment.

Two files must be configured: `services` and `inetd.conf`. Both are ASCII-editable and can usually be found at `/etc/services` and `/etc/inetd.conf`.

Put one of the following lines into `inetd.conf`:

```
net-eval stream tcp nowait root /usr/bin/newlisp -c
# as an alternative, a program can also be preloaded
net-eval stream tcp nowait root /usr/bin/newlisp -c
myprog.lsp
```

newLISP Users Manual and Reference

Instead of `root`, another user and optional group can be specified. For details, see the UNIX man page for `inetd`.

The following line is put into the `services` file:

```
net-eval          4711/tcp      # newLISP net-eval requests
```

On Mac OS X and some UNIX systems, `xinetd` can be used instead of `inetd`. Save the following to a file named `net-eval` in the `/etc/xinetd.d/` directory:

```
service net-eval
{
    socket_type = stream
    wait = no
    user = root
    server = /usr/bin/newlisp
    port = 4711
    server_args = -c
    only_from = localhost
}
```

For security reasons, `root` should be changed to a different user. The `only_from` spec can be left out to permit remote access.

See the man pages for `xinetd` and `xinetd.conf` for other configuration options.

After configuring the daemon, `inetd` or `xinetd` must be restarted to allow the new or changed configuration files to be read:

```
kill -HUP <pid>
```

Replace `<pid>` with the process ID of the running `xinetd` process.

A number or network protocol other than 4711 or TCP can be specified.

newLISP handles everything as if the input were being entered on a newLISP command line without a prompt. To test the `inetd` setup, the `telnet` program can be used:

```
telnet localhost 4711
```

newLISP expressions can now be entered, and `inetd` will automatically handle the startup and communications of a newLISP process. Multiline expressions can be entered by bracketing them with `[cmd]` and `[/cmd]` tags, each on separate lines.

newLISP server nodes running on UNIX like operating systems, will also answer simple HTTP GET and PUT requests. This can be used to retrieve and store files with [get-url](#), [put-url](#), [read-file](#), [write-file](#) and [append-file](#), or to load and save programs using [load](#) and [save](#) from and to remote server nodes. On Win32 newLISP server nodes do not answer HTTP requests.

Daemon mode with handler function

This is similar to `-d` mode, but instead of executing the input as a newLISP command line, daemon mode passes it to a handler function named `x-event` with the request line as argument. The function `x-event` must be defined by the user and loaded on startup:

```
newlisp -x 8080 my-httpd
```

In this example, `my-httpd` — the file containing the definition of `x-event` — is loaded before newLISP begins listening for requests on port 8080. This file could contain the following definition:

```
;; my-httpd - simple web server for static text/html pages
(define (x-event request)
  (find "GET /(.*) HTTP" request 0)
  (while (!= "" (read-line))) ; skip header
  (set 'page (or (read-file $1) "Error 404: Page not found"))
  (print
    "HTTP/1.0 200 OK\r\n"
    "content-length: " (length page) "\r\n"
    "Content-type: text/html\r\n\r\n"
    page))
```

In a web browser, type `http://localhost:8080/mypage.html`, where `mypage.html` is a file in the current directory.

With the `-x` server option, it is possible to write handlers for any protocol. Using only `print`, `println`, `read-line` or `write-line`, handlers can be defined in the `x-event` function to communicate through the TCP/IP port with the connected client. After returning from `x-event`, newLISP awaits further incoming requests. The client is responsible for closing the connection after a client/server exchange. In `-x` server mode, newLISP automatically listens for a new connection on the specified port.

Whenever HTTP requests contain `Content-length:` parameters or line-oriented reading of input does not apply, [net-receive](#) can be used with the current connection socket obtained from [net-sessions](#). The following code was snipped from `xmlrpc-server`, one of the example files shipped with the source distribution:

```
;; get header info
(while (!= "" (read-line))
  (if (find "Content-length:(.*)" (current-line) 0)
    (set 'contentLength (int $1))))

;; read XML
(net-receive (first (net-sessions)) 'XML contentLength)
```

The `(net-sessions)` statement retrieves an internal list whose first element is always the last opened connection or listen-socket.

Direct execution mode

Small pieces of newLISP code can be executed directly from the command line:

```
newlisp -e "(+ 3 4)" → 7
```

The expression enclosed in quotation marks is evaluated, and the result is printed to standard out (STDOUT). In most UNIX system shells, single quotes can also be used as command-line delimiters. Note that there is a space between `-e` and the quoted command string.

Logging I/O

When in `-p`, `-d` or console mode, input from the command line and output from newLISP can be written to a log file. The two logging modes work differently, depending on whether newLISP is in console or server mode:

<i>mode</i>	<i>result</i>
<code>newlisp -l</code>	log only user input
<code>newlisp -L</code>	log user input and newLISP output (w/o prompt)
<code>newlisp -l -p 4711</code> <code>newlisp -l -d 4711</code>	} log client IP and connection time
<code>newlisp -L -p 4711</code> <code>newlisp -L -d 4711</code>	} log client IP, connection time and user input

All logging output is written to the file `newlisp-log.txt` within newLISP's startup directory. Instead of 4711, any other port number can be supplied.

Command line help summary

The `-h` command-line switch prints a copyright notice and summary of options:

```
newlisp -h
```

On Linux and other UNIX systems, a `newlisp` *man page* can be found:

```
man newlisp
```

This will display a man page in the Linux/UNIX shell.

The initialization file `init.lsp`

On Linux, BSDs, Mac OS X, and Cygwin, the initialization file is installed and expected in `/usr/share/newlisp/init.lsp`. newLISP compiled with MinGW or Borland BCC looks for `init.lsp` in the same directory where `newlisp.exe` is installed. Along with any files specified on the command line, `init.lsp` is loaded before the banner and prompt are shown. When newLISP is executed or launched by a program or process other than a shell, the banner and prompt are not shown, and newLISP communicates by standard I/O. `init.lsp`, however, is still loaded and evaluated if present.

While newLISP does not require `init.lsp` to run, it is convenient for defining functions and systemwide variables. `init.lsp` is not included in the newLISP-tk distribution, but it can be found in the source distribution.

The last part of `init.lsp` contains OS-specific code, which loads a second `.init.lsp` (starting with a dot). On Linux/UNIX, this file is expected in the directory specified by the `HOME` environment variable. On Win32, this file is expected in the directory specified by the `USERPROFILE` or `DOCUMENT_ROOT` environment variable.

Directories on Linux, BSD, and Mac OS X

The directory `/usr/share/newlisp/` contains modules with useful functions for a variety of tasks, such as database management with MySQL, procedures for statistics, POP3 mail, etc. The directory `/usr/share/newlisp/doc` contains documentation in HTML format.

Directories on Win32/newLISP-tk

On Win32 systems, all files are installed in the default directory `$PROGRAMFILES\newlisp`. `$PROGRAMFILES` is a Win32 environment variable that resolves to `C:\Program files\newlisp\` in English language installations. If an `init.lsp` file is required, it should be in the same directory where `newlisp.exe` resides.

(§)

4. Shared library module for Linux/BSD versions

newLISP can be compiled as a UNIX shared library called `newlisp.dylib` on Mac OS X and as `newlisp.so` on Linux and BSDs. A newLISP shared library can be used like any other UNIX shared library.

To use `newlisp.so` or `newlisp.dylib`, import the function `newlispEvalStr`. Like [eval-string](#), this function takes a string containing a newLISP expression and stores the result in a string address. The result can be converted using [get-string](#). The returned string is formatted

4. Shared library module for Linux/BSD versions

like output from a command-line session. It contains terminating line-feed characters, but without the prompt strings.

The first example shows how `newlisp.so` is imported from newLISP itself.

```
(import "/usr/lib/newlisp.so" "newlispEvalStr")
(get-string (newlispEvalStr "(+ 3 4)") → "7\n"
```

The second example shows how to import `newlisp.so` into a program written in C:

```
/* libdemo.c - demo for importing newlisp.so
 *
 * compile using:
 * gcc -ldl libdemo.c -o libdemo
 *
 * use:
 *
 * ./libdemo '(+ 3 4)'
 * ./libdemo '(symbols)'
 */
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char * argv[])
{
    void * hLibrary;
    char * result;
    char * (*func)(char *);
    char * error;

    if((hLibrary = dlopen("/usr/lib/newlisp.so",
                        RTLD_GLOBAL | RTLD_LAZY)) == 0)
    {
        printf("cannot import library\n");
        exit(-1);
    }

    func = dlsym(hLibrary, "newlispEvalStr");
    if((error = dlerror()) != NULL)
    {
        printf("error: %s\n", error);
        exit(-1);
    }

    printf("%s\n", (*func)(argv[1]));

    return(0);
}

/* eof */
```

This program will accept quoted newLISP expressions and print the evaluated results.

When calling `newlisp.so`'s function `newlispEvalStr`, output normally directed to the console (e.g., return values or [print](#) statements) is returned in the form of an integer string pointer. The output can be accessed by passing this pointer to the `get-string` function. To silence the output from return values, use the [silent](#) function.

(§)

4. Shared library module for Linux/BSD versions

5. DLL module for Win32 versions

On the Win32 platforms, newLISP can be compiled as a DLL (Dynamic Link Library). In this way, newLISP functions can be made available to other programs (e.g., MS Excel, Visual Basic, Borland Delphi, or even newLISP itself).

When the DLL is loaded, it looks for the file `init.lsp` in the current directory of the calling process.

To access the functionality of the DLL, use `newlispEvalStr`, which takes a string containing a valid newLISP expression and returns a string of the result:

```
(import "newlisp.dll" "newlispEvalStr")
(get-string (newlispEvalStr "(+ 3 4)")) → "7"
```

The above example shows the loading of a DLL using newLISP. The [get-string](#) function is necessary to access the string being returned. Other applications running on Win32 allow the returned data type to be declared when importing the function.

When using `newlisp.so`, output normally directed to the console — like [print](#) statements or return values — will be returned in a string pointed to by the call to `newlispEvalStr`. To silence the output from return values, use the [silent](#) directive.

(§)

6. Evaluating newLISP expressions

The following is a short introduction to LISP statement evaluation and the role of integer and floating point arithmetic in newLISP.

Top-level expressions are evaluated when using the [load](#) function or when entering expressions in console mode on the command line. As shown in the following snippet from an interactive session, multiline expressions can be entered by enclosing them between `[cmd]` and `/cmd]` tags:

```
> [cmd]
(define (foo x y)
  (+ x y))
[/cmd]
(lambda (x y) (+ x y))
> (foo 3 4)
7
> _
```

Each `[cmd]` and `/cmd]` tag is entered on a separate line. This mode is useful for pasting multiline code into the interactive console.

Integer data, floating point data, and operators

newLISP functions and operators accept integer and floating point numbers, converting them into the needed format. For example, a bit-manipulating operator converts a floating point number into an integer by omitting the fractional part. In the same fashion, a trigonometric function will internally convert an integer into a floating point number before performing its calculation.

The symbol operators (+ - * / % \$ ~ | ^ << >>) return values of type integer. Functions and operators named with a word instead of a symbol (e.g., add rather than +) return floating point numbers. Integer operators truncate floating point numbers to integers, discarding the fractional parts.

newLISP has two types of basic arithmetic operators: integer (+ - * /) and floating point (add sub mul div). The arithmetic functions convert their arguments into types compatible with the function's own type: integer function arguments into integers, floating point function arguments into floating points. To make newLISP behave more like other scripting languages, the integer operators +, -, *, and / can be redefined to perform the floating point operators add, sub, mul, and div:

```
(constant '+ add)
(constant '- sub)
(constant '* mul)
(constant '/ div)

;; or all 4 operators at once
(constant '+ add '- sub '* mul '/ div)
```

Now the common arithmetic operators +, -, *, and / accept both integer and floating point numbers and return floating point results.

Note that the looping variables in [dotimes](#) and [for](#), as well as the result of [sequence](#), use floating point numbers for their values.

Care must be taken when importing from libraries that use functions expecting integers. After redefining +, -, *, and /, a double floating point number may be unintentionally passed to an imported function instead of an integer. In this case, floating point numbers can be converted into integers by using the function [int](#). Likewise, integers can be transformed into floating point numbers using the [float](#) function:

```
(import "mylib.dll" "foo") ; importing int foo(int x) from C
(foo (int x))             ; passed argument as integer
(import "mylib.dll" "bar") ; importing C int bar(double y)
(bar (float y))           ; force double float
```

Some of the modules shipping with newLISP are written assuming the default implementations of +, -, *, and /. This gives imported library functions maximum speed when performing address calculations.

The newLISP preference is to leave +, -, *, and / defined as integer operators and use add, sub, mul, and div when explicitly required. Since version 8.9.7 integer operations in newLISP are 64 bit operations, while 64 bit double floating point numbers only offer 52 bits of resolution in the integer part of the number.

Evaluation rules and data types

Evaluate expressions by entering and editing them on the command line. More complicated programs can be entered using editors like Emacs and VI, which have modes to show matching parentheses while typing. Load a saved file back into a console session by using the [load](#) function.

A line comment begins with a `;` (semicolon) or a `#` (number sign) and extends to the end of the line. newLISP ignores this line during evaluation. The `#` is useful when using newLISP as a scripting language in Linux/UNIX environments, where the `#` is commonly used as a line comment in scripts and shells.

When evaluation occurs from the command line, the result is printed to the console window.

The following examples can be entered on the command line by typing the code to the left of the `→` symbol. The result that appears on the next line should match the code to the right of the `→` symbol.

nil and **true** are boolean data types that evaluate to themselves:

```
nil      → nil
true     → true
```

Integers and **floating point** numbers evaluate to themselves:

```
123      → 123
0xE8     → 232      ; hexadecimal prefixed by 0x
055      → 45       ; octal prefixed by 0 (zero)
1.23     → 1.23
123e-3   → 0.123    ; scientific notation
```

Integers are 64-bit numbers (including the sign bit, 32-bit before version 8.9.7). Valid integers are numbers between -9,223,372,036,854,775,808 and +9,223,372,036,854,775,807. Larger numbers converted from floating point numbers are truncated to one of the two limits. Integers internal to newLISP, which are limited to 32-bit numbers overflow to either +2,147,483,647 or -2,147,483,648. Floating point numbers are IEEE 754 64-bit doubles. Unsigned numbers up to 18,446,744,073,709,551,615 can be displayed using special formatting characters for [format](#).

Strings may contain null characters and can have different delimiters. They evaluate to themselves.

```
"hello"      → "hello"
"\032\032\065\032" → "  A  "
"\x20\x20\x41\x20" → "  A  "
"\t\r\n"     → "\t\r\n"
"\x09\x0d\x0a" → "\t\r\n"

;; null characters are legal in strings:
"\000\001\002" → "\000\001\002"
{this "is" a string} → "this \"is\" a string"

;; use [text] tags for text longer than 2048 bytes:
[text]this is a string, too[/text]
→ "this is a string, too"
```

Strings delimited by " (double quotes) will also process the following characters escaped with a \ (backslash):

<i>escaped character</i>	<i>description</i>
\"	for a double quote inside a quoted string
\n	for a line-feed character (ASCII 10)
\r	for a return character (ASCII 13)
\t	for a TAB character (ASCII 9)
\nnn	for a three-digit ASCII number (nnn format between 000 and 255)
\xnn	for a two-hex-digit ASCII number (xnn format between x00 and xff)

Quoted strings cannot exceed 2,048 characters. Longer strings should use the [text] and [/text] tag delimiters. newLISP automatically uses these tags for string output longer than 2,048 characters.

The { (left curly bracket), } (right curly bracket), and [text], [/text] delimiters do not perform escape character processing.

Lambda and lambda-macro expressions evaluate to themselves:

```
(lambda (x) (* x x))           → (lambda (x) (* x x))
(lambda-macro (a b) (set (eval a) b)) → (lambda (x) (* x x))
(fn (x) (* x x))               → (lambda (x) (* x x)) ;
an alternative syntax
```

Symbols evaluate to their contents:

```
(set 'something 123) → 123
something           → 123
```

Contexts evaluate to themselves:

```
(context 'CTX) → CTX
CTX           → CTX
```

Built-in functions also evaluate to themselves:

```
add           → add <B845770D>
(eval (eval add)) → add <B845770D>
(constant '+ add) → add <B845770D>
+             → add <B845770D>
```

In the above example, the number between the < > (angle brackets) is the hexadecimal memory address (machine-dependent) of the add function. It is displayed when printing a built-in primitive.

Quoted expressions lose one ' (single quote) when evaluated:

```
'something → something
'''any     → '''any
'(a b c d) → (a b c d)
```

A single quote is often used to *protect* an expression from evaluation (e.g., when referring to the symbol itself instead of its contents or to a list representing data instead of a function).

In newLISP, a **list**'s first element is evaluated before the rest of the expression (as in Scheme). The result of the evaluation is applied to the remaining elements in the list and must be one of the following: a *lambda* expression, *lambda-macro* expression, or *primitive* (built-in) function.

```
(+ 1 2 3 4)           → 10
(define (double x) (+ x x)) → (lambda (x) (+ x x))
```

or

```
(set 'double (lambda (x) (+ x x)))
(double 20)           → 40
((lambda (x) (* x x)) 5) → 25
```

For a user-defined lambda expression, newLISP evaluates the arguments from left to right and binds the results to the parameters (also from left to right), before using the results in the body of the expression.

Like Scheme, newLISP evaluates the *functor* (function object) part of an expression before applying the result to its arguments. For example:

```
((if (> X 10) * +) X Y)
```

Depending on the value of X, this expression applies the * (product) or + (sum) function to X and Y.

Because their arguments are not evaluated, **lambda-macro** expressions are useful for extending the syntax of the language. Most built-in functions evaluate their arguments from left to right (as needed) when executed. Some exceptions to this rule are indicated in the reference section of this manual. LISP functions that do not evaluate all or some of their arguments are called *special forms*.

Arrays evaluate to themselves:

```
(set 'A (array 2 2 '(1 2 3 4))) → ((1 2) (3 4))
(eval A)                       → ((1 2) (3 4))
```

Shell commands: If an ! (exclamation mark) is entered as the first character on the command line followed by a shell command, the command will be executed. For example, !ls on Unix or !dir on Win32 will display a listing of the present working directory. No spaces are permitted between the ! and the shell command. Symbols beginning with an ! are still allowed inside expressions or on the command line when preceded by a space. Note: This mode only works when running in the shell and does not work when controlling newLISP from another application.

To exit the newLISP shell on Linux/UNIX, press Ctrl-D; on Win32, type (exit) or Ctrl-C, then the x key.

Use the [exec](#) function to access shell commands from other applications or to pass results back to newLISP.

(§)

7. Lambda expressions in newLISP

Lambda expressions in newLISP evaluate to themselves and can be treated just like regular lists:

```
(set 'double (lambda (x) (+ x x)))
(set 'double (fn (x) (+ x x)))      ; alternative syntax

(last double) → (+ x x)             ; treat lambda as a list
```

Note: No ' is necessary before the lambda expression since lambda expressions evaluate to themselves in newLISP.

The second line uses the keyword `fn`, an alternative syntax first suggested by Paul Graham for his Arc language project.

A lambda expression is a *lambda list*, a subtype of *list*, and its arguments can associate from left to right or right to left. When using [append](#), for example, the arguments associate from left to right:

```
(append (lambda (x)) '((+ x x))) → (lambda (x) (+ x x))
```

[cons](#), on the other hand, associates the arguments from right to left:

```
(cons '(x) (lambda (+ x x))) → (lambda (x) (+ x x))
```

Note that the `lambda` keyword is not a symbol in a list, but a designator of a special *type* of list: the *lambda list*.

```
(length (lambda (x) (+ x x))) → 2
(first (lambda (x) (+ x x))) → (x)
```

Lambda expressions can be mapped or applied onto arguments to work as user-defined, anonymous functions:

```
((lambda (x) (+ x x)) 123) → 246
(apply (lambda (x) (+ x x)) '(123)) → 246
(map (lambda (x) (+ x x)) '(1 2 3)) → (2 4 6)
```

A lambda expression can be assigned to a symbol, which in turn can be used as a function:

```
(set 'double (lambda (x) (+ x x))) → 246
(double 123) → (lambda (x) (+ x x))
```

The [define](#) function is just a shorter way of assigning a lambda expression to a symbol:

```
(define (double x) (+ x x)) → (lambda (x) (+ x x))
(double 123) → 246
```

In the above example, the expressions inside the lambda list are still accessible within `double`:

```
(set 'double (lambda (x) (+ x x))) → (lambda (x) (+ x x))
(last double) → (+ x x)
```

A lambda list can be manipulated as a first-class object using any function that operates on lists:

```
(set-nth 1 double '(mul 2 x)) → (lambda (x) (mul 2 x))
double                        → (lambda (x) (mul 2 x))
(double 123)                  → 246
```

All arguments are optional when applying lambda expressions and default to `nil` when not supplied by the user. This makes it possible to write functions with multiple parameter signatures.

(§)

8. `nil`, `true`, `cons`, and `()`

In newLISP, `nil` and `true` represent both the symbols and the boolean values *true* and *false*. Depending on their context, `nil` and `true` are treated differently. The following examples use `nil`, but they can be applied to `true` by simply reversing the logic.

Evaluation of `nil` yields a boolean false and is treated as such inside control flow expressions, such as `if`, `unless`, `while`, `until`, and `not`. Likewise, evaluating `true` yields `true`.

```
(set 'lst '(nil nil nil)) → (nil nil nil)
(map symbol? lst)         → (true true true)
```

In the above example, `nil` represents a symbol. In the following example, `nil` and `true` are evaluated and represent boolean values:

```
(if nil "no" "yes") → "yes"
(if true "yes" "no") → "yes"
(map not lst)       → (true true true)
```

In newLISP, `nil` and the empty list `()` are not the same as in some other LISP. Only in conditional expressions are they treated as a boolean false, as in `and`, `or`, `if`, `while`, `unless`, `until`, and `cond`.

The expression `(list? '())` is true, but `(list? nil)` is not. This is because in newLISP, `nil` results in a boolean false when evaluated.

Evaluation of `(cons x '())` yields `(x)`, but `(cons x nil)` yields `(x nil)` because `nil` is treated as a boolean value when evaluated instead of as an empty list. The `cons` of two atoms in newLISP does not yield a dotted pair, but rather a two-element list. The predicate `atom?` is true for `nil`, but false for the empty list. The empty list in newLISP is only an empty list and not equal to `nil`.

A list in newLISP is a LISP cell of type list. It acts like a container for the linked list of elements making up the list cell's contents. There is no *dotted pair* in newLISP because the *cdr* (tail) part of a LISP cell always points to another LISP cell and never to a basic data type, such as a number or a symbol. Only the *car* (head) part may contain a basic data type. Early LISP implementations used `car` and `cdr` for the names *head* and *tail*.

(§)

9. Arrays

newLISP's arrays enable fast element access within large lists. New arrays can be constructed and initialized with the contents of an existing list using the function [array](#). Lists can be converted into arrays, and vice versa. Some of the same functions used for modifying and accessing lists can be applied to arrays, as well. Arrays can hold any type of data or combination thereof.

In particular, the following functions can be used for creating, accessing, and modifying arrays:

function description

append	appends arrays
array	creates and initializes an array with up to 16 dimensions
array-list	converts an array into a list
array?	checks if expression is an array
det	returns the determinant of a matrix
first	returns the first row of an array
invert	returns the inversion of a matrix
last	returns the last row of an array
multiply	multiplies two matrices
nth	returns an element of an array
nth-set	changes the element, returning the old; significantly faster than <code>set-nth</code>
rest	returns all but the first row of an array
set-nth	changes the element and returns the changed array
slice	returns a slice of an array
transpose	transposes a matrix

newLISP represents multidimensional arrays with an array of arrays (i.e., the elements of the array are themselves arrays).

When used interactively or with the newLISP-tk front-end, newLISP prints and displays arrays as lists, with no way of distinguishing between them.

Use the [source](#) or [save](#) functions to serialize arrays (or the variables containing them). The [array](#) statement is included as part of the definition when serializing arrays.

Like lists, negative indices can be used to enumerate the elements of an array, starting from the last element.

An out-of-bounds index will cause an error message on an array. In contrast, lists pick the last or first element when an out-of-bounds occurs.

Arrays can be non-rectangular, but they are made rectangular during serialization when using [source](#) or [save](#). The [array](#) function always constructs arrays in rectangular form.

The matrix functions [det](#), [transpose](#), [multiply](#), and [invert](#) can be used on matrices built with nested lists or arrays built with [array](#).

For more details, see [array](#), [array?](#), and [array-list](#) in the reference section of this manual.

(§)

10. Dictionaries (hash tables)

newLISP has no built-in *hash table* data type. Instead, it uses symbols for associative memory access. Symbols in newLISP are implemented using an efficient *red-black tree* algorithm. This algorithm balances the binary symbol tree for faster symbol access. In newLISP, symbol trees are represented as namespaces called *contexts*, which are themselves part of the MAIN namespace.

For a more detailed introduction to *namespaces*, see the chapter on [Contexts](#).

The [context](#) function can be used to make associations. It can also be used to create and switch contexts.

```
;; create a symbol and store data into it
(context 'MyHash "John Doe" 123)      → 123
(context 'MyHash "@#$%^" "hello world") → "hello world"

;; retrieve contents from the symbol
(context 'MyHash "John Doe")          → 123
(context 'MyHash "@#$%^")              → "hello world"
```

The first two statements create the symbols "John Doe" and "@#\$%^", storing the values 123 and "hello world" into them. The hash context named MyHash is created in the first statement, while the second merely adds the new association to the existing one.

Note that hash symbols can contain spaces or other special characters not typically allowed in variable names.

Internally, [context](#) is just a shorter and faster form of:

```
;; create a symbol and store the data in it
(set (sym "John Doe" 'MyHash) 123) → 123

;; retrieve contents from the symbol
(eval (sym "John Doe" MyHash)) → 123
```

The context [default function](#) can be used for a very short definition of a hash function:

```
(define (myhash:myhash key value)
  (if value
    (context myhash key value)
    (context myhash key)))

;; create a dictionary key value pair
(myhash "hello" 123) → 123

;; retrieve the key value
(myhash "hello") → 123
```

(§)

11. Indexing elements of strings, lists, and arrays

Some functions take array, list, or string elements (characters) specified by one or more *int-index* (integer index). The positive indices run 0, 1, ..., N-2, N-1, where N is the number of elements in the list. If int-index is negative, the sequence is -N, -N+1, ..., -2, -1. Adding N to the negative index of an element yields the positive index. Unless a function does otherwise, an index greater than N-1 returns the last element in a list; it returns the first element for indices less than -N. An error message is produced for any indexing occurring outside an array's boundaries.

Implicit indexing for `nth`

In versions 8.5 and later, implicit indexing can be used instead of [nth](#) to retrieve the characters of a string or the elements of a list or array:

```
(set 'lst '(a b c (d e) (f g)))

(lst 0)    → a      ; same as (nth 0 lst)
(lst 3)    → (d e)
(lst 3 1)  → e      ; same as (nth 3 1 lst)
(lst -1)   → (f g)

(set 'myarray (array 3 2 (sequence 1 6)))

(myarray 1)    → (3 4)
(myarray 1 0)  → 3
(myarray 0 -1) → 2

("newLISP" 3) → "L"
```

Indices may also be supplied from a list. In this way, implicit indexing works together with functions that take or produce index vectors, such as [push](#), [pop](#), [ref](#) and [ref-all](#).

```
(lst '(3 1))      → e
(set 'vec (ref 'e lst)) → (3 1)
(lst vec)         → e
```

Implicit indexing is both slightly faster than [nth](#) and capable of taking an unlimited number of indices.

Note that in the UTF-8-enabled version of newLISP, implicit indexing of strings using the [nth](#) function works on character rather than byte boundaries.

Implicit indexing and the default functor

The *default functor* is a functor inside a context with the same name as the context itself. See [The context default function](#) chapter. A default functor can be used together with implicit indexing to serve as a mechanism for referencing lists:

```
(set 'MyList:MyList '(a b c d e f g))

(MyList 0)    → a
(MyList 3)    → d
(MyList -1)   → g

(3 2 MyList) → (d e)
(-3 MyList)  → (e f g)

(set 'aList MyList)

(aList 3)    → d
```

In this example, `aList` references `MyList:MyList`, not a copy of it. For more information about contexts, see [Programming with context objects](#).

The default functor can also be used with [nht-set](#) as shown in the following example:

```
(nth-set (MyList 3) 999) → d
(MyList 3)              → 999
```

Implicit indexing for `rest` and `slice`

Implicit forms of [rest](#) and [slice](#) can be created by prepending a list with one or two numbers for offset and length:

```
(set 'lst '(a b c d e f g))
; or as array
(set 'lst (array 7 '(a b c d e f g)))

(1 lst)      → (b c d e f g)
(2 lst)      → (c d e f g)
(2 3 lst)    → (c d e)
(-3 2 lst)   → (e f)

(set 'str "abcdefg")

(1 str)      → "bcdefg"
(2 str)      → "cdefg"
(2 3 str)    → "cde"
(-3 2 str)   → "ef"
```

Implicit indexing for [rest](#) works on character rather than byte boundaries when using the UTF-8-enabled version of newLISP, whereas implicit indexing for [slice](#) will always work on byte boundaries and can be used for binary content.

Implicit indexing for `nth-set` and `set-nth`

```
(set 'aList '(a b c (d e (f g) h i) j k))
```

```

(nth-set (aList 0) 1) → a
(nth-set (aList 3 2) '(1 2 3 4)) → (f g)
(set 'i 3 'j 2 'k 2)
(nth-set (aList i j k) 99) → 3
aList
→ (1 b c (d e (1 2 99 4) h i) j k)
(set-nth (aList -3 -3 2) 999)
→ (1 b c (d e (1 2 999 4) h i) j k)

```

(§)

12. Destructive versus nondestructive functions

Most of the primitives in newLISP are nondestructive (no *side effects*) and leave existing objects untouched, although they may create new ones. There are a few destructive functions, however, that *do* change the contents of a list, string, or variable:

<i>function</i>	<i>description</i>
constant	sets the contents of a variable and protects it
dec	decrements the value in a variable
inc	increments the value in a variable
net-receive	reads into a buffer variable
push	pushes a new element onto a list or string
pop	pops an element from a list or string
read-buffer	reads into a buffer variable
replace	replaces elements in a list or string
replace-assoc	replaces elements inside a list or string
reverse	reverses a list or string
rotate	rotates the elements of a list or characters of a string
set, setq	changes an element inside a list or string
set-nth, nth-set	changes an element in a list or string
sort	sorts the elements of a list
swap	swaps two elements inside a list or string
write-buffer	writes to a string buffer
write-line	writes to a string buffer

Note that the last two functions, [write-buffer](#) and [write-line](#), are only destructive in one of their syntactic forms: when taking a string buffer instead of a file handle.

(§)

13. Dynamic and lexical scoping

newLISP uses dynamic scoping *inside* contexts and lexical scoping *outside* of them. In this way, newLISP programs can take advantage of both scoping mechanisms at once.

When the parameter symbols of a lambda expression are bound to its arguments, the old bindings are pushed on a stack. newLISP automatically restores the original variable bindings when leaving the lambda function.

The following example illustrates the *dynamic scoping* mechanism. The text in bold is the output from newLISP:

```
> (define (add-three-nums x y z) (print-vars) (+ x y z))
(lambda (x y z) (print-vars) (+ x y z))
> (define (print-vars) (print "X=" x " Y=" y " Z= " z "\n"))
(lambda () (print "X=" x " Y=" y " Z= " z "\n"))
> (set 'x 4)
4
> (set 'y 5)
5
> (set 'z 6)
6
> (print-vars)
X=4 Y=5 Z=6
6
> (add-three-nums 70 80 90)
X=70 Y=80 Z=90
240
> (print-vars)
X=4 Y=5 Z=6
6
> _
```

The example shows `add-three-nums`, which returns the sum of its arguments using `print-vars` to display the contents of the symbols `x`, `y`, and `z`. Before `add-three-nums` is called, the symbols `x`, `y`, and `z` are bound to the values 4, 5, and 6.

Note: Different values will be printed for `x`, `y`, and `z` depending on where `print-vars` is called from. While in the scope of `add-three-nums`, the symbols `x`, `y`, and `z` have local bindings. The old bindings are restored after returning from `add-three-nums`. This is different from the *lexical scoping* mechanisms found in languages like C, Java, and most current LISPs, where the binding of local parameters occurs inside the function only. In lexically scoped languages like C, `print-vars` would always print the global bindings of the symbols `x`, `y`, and `z` (4, 5, and 6).

Be aware that passing quoted symbols to a user-defined function causes a name clash if the same variable name is used as a function parameter:

```
(define (inc-symbol x y) (inc x y))
(set 'y 200)
(inc-symbol 'y 123) → 246
y                  → 200 ; y is still 200
```

Since `'y` shares the same name as the function's second parameter, `inc-symbol` returns 246 (123 + 123), leaving `'y` unaffected. Dynamic scoping's *variable capture* can be a disadvantage when passing symbol references to user-defined functions.

The problem is avoided entirely by grouping related user-defined functions into a [context](#). A symbol name clash cannot occur when accessing symbols and calling functions from *outside* of the defining context.

Contexts should be used to group related functions when creating interfaces or function libraries. This surrounds the functions with a lexical "fence," thus avoiding variable name clashes with the calling functions.

newLISP uses contexts for different forms of lexical scoping. See the chapters [Contexts](#) and [Programming with context objects](#), as well as the sections [Lexical, static scoping in newLISP](#) and [default functions](#) for more information.

(§)

14. Early return from functions, loops, and blocks

What follows are methods of interrupting the control flow inside both loops and the [begin](#) expression.

The looping functions [dolist](#) and [dotimes](#) can take optional conditional expressions to leave the loop early. [catch](#) and [throw](#) are a more general form to break out of a loop body and are also applicable to other forms or statement blocks.

Using catch and throw

Because newLISP is a functional language, it uses no `break` or `return` statements to exit functions or iterations. Instead, a block or function can be exited at any point using the functions [catch](#) and [throw](#):

```
(define (foo x)
  (...)
  (if condition (throw 123))
  (...))
456)

;; if condition is true

(catch (foo p)) → 123

;; if condition is not true

(catch (foo p)) → 456
```

Breaking out of loops works in a similar way:

```
(catch
  (dotimes (i N)
    (if (= (foo i) 100) (throw i))))
→ value of i when foo(i) equals 100
```

The example shows how an iteration can be exited before executing N times.

Multiple points of return can be coded using [throw](#):

```
(catch (begin
  (foo1)
  (foo2)
  (if condition-A (throw 'x))
  (foo3)
  (if condition-B (throw 'y))
  (foo4)
  (foo5)))
```

If `condition-A` is true, `x` will be returned from the `catch` expression; if `condition-B` is true, the value returned is `y`. Otherwise, the result from `foo5` will be used as the return value.

As an alternative to [catch](#), the [throw-error](#) function can be used to catch errors caused by faulty code or user-initiated exceptions.

Using `and` and `or`

Using the logical functions [and](#) and [or](#), blocks of statements can be built that are exited depending on the boolean result of the enclosed functions:

```
(and
  (func-a)
  (func-b)
  (func-c)
  (func-d))
```

The [and](#) expression will return as soon as one of the block's functions returns `nil` or an `()` (empty list). If none of the preceding functions causes an exit from the block, the result of the last function is returned.

[or](#) can be used in a similar fashion:

```
(or
  (func-a)
  (func-b)
  (func-c)
  (func-d))
```

The result of the [or](#) expression will be the first function that returns a value which is *not* `nil` or `()`.

(§)

15. Contexts

In newLISP, symbols can be separated into namespaces called *contexts*. Each context has a private symbol table lexically separate from all other contexts. Symbols known in one context are unknown in others, so the same name may be used in different contexts without conflict.

Contexts are used to build modules of isolated variable and function definitions. They can also be copied and dynamically assigned to variables or passed as arguments. Because contexts in newLISP have lexically separated namespaces, they allow programming with *lexical scoping* and software object styles of programming.

Contexts are identified by symbols that are part of the root or MAIN context. While context symbols are uppercased in this chapter, lowercase symbols may also be used.

In addition to context names, MAIN contains the symbols for built-in functions and special symbols such as `true` and `nil`. The MAIN context is created automatically each time newLISP is run. To see all the symbols in MAIN, enter the following expression after starting newLISP:

```
(symbols)
```

Scoping rules for contexts

Special symbols like `nil` and `true`, as well as context and built-in function symbols, are global (visible to all contexts). Any symbol can be made global by using the [global](#) function.

The following simulates a command-line session in newLISP:

```
> (context 'FOO)
FOO
FOO> _
```

If the FOO context already exists, newLISP switches to it. Otherwise, the context is created before the switch occurs. All symbols now read from the command line are created and known only within the context FOO. Note that the symbol used for the context name must be quoted ('FOO in this example) the first time a context is created. Subsequent uses of context do not require the quote. After the switch, the command-line prompt changes to FOO> :

```
FOO> (set 'x 123)
123
FOO> (set 'y 456)
456
FOO> (symbols)
(x y)
FOO> _
```

To switch back to the MAIN context, use:

```
FOO> (context MAIN)
MAIN
> _
```

A symbol can be referenced from outside its defining context by prepending a context name and a colon to it:

```
> FOO:x
123
> _
```


The same symbol may also be used in another context:

```
> (context 'FOO-B)
FOO-B
FOO-B> (set 'x 777)
777
FOO-B> FOO:x
123
> _
```

When quoting a fully qualified symbol (`context:symbol`), the quote precedes the context name:

```
> (set 'FOO-B:x 555)
555
> _
```

A context is implicitly created when referring to one that does not yet exist. Unlike the `context` function, the context is not switched. The following statements are all executed inside the `MAIN` context:

```
> (set 'ACTX:var "hello")
"hello"
> ACTX:var
"hello"
> _
```

The same symbol (`x` in this case) used in a context can also be used in `MAIN`. Now we have three versions of `x`, all in a different context:

```
> (set 'x "I belong to MAIN")
"I belong to MAIN"
> FOO:x
123
> FOO-B:x
555
> x
"I belong to MAIN"
> _
```

Symbols owned by a context (or `MAIN`) are *not* accessible unless prefixed by the context name:

```
FOO> MAIN:x
"I belong to MAIN"
FOO> FOO-B:x
555
FOO> x
123
> _
```

When loading source files on the command line with [load](#), or when executing the functions [eval-string](#) or [sym](#), the `context` function tells newLISP where to put all of the symbols and definitions:

```
;;; file MY_PROG.LSP
;;
;; everything from here on goes into GRAPH
(context 'GRAPH)
```

```

(define (draw-triangle x y z)
  (...))
(define (draw-circle)
  (...))

;; show the runtime context, which is GRAPH
(define (foo)
  (context))

;; switch back to MAIN
(context 'MAIN)

;; end of file

```

The `draw-triangle` and `draw-circle` functions — along with their `x`, `y`, and `z` parameters — are now part of the `GRAPH` context. These symbols are known only to `GRAPH`. To call these functions from another context, prefix them with `GRAPH`:

```

(GRAPH:draw-triangle 1 2 3)
(GRAPH:foo) → GRAPH

```

The last statement shows how the runtime context has changed to `GRAPH` (`foo`'s context). This feature was introduced in version 8.7.8. In older versions, the runtime context would still be `MAIN`.

A symbol's name and context are used when comparing symbols from different contexts. The [name](#) function can be used to extract the name part from a fully qualified symbol.

```

;; same symbol name, but different context name
(= 'A:val 'B:val) → nil
(= (name 'A:val) (name 'B:val)) → true

```

Note: The symbols are quoted with a `'` (single quote) because we are interested in the symbol itself, not in the contents of the symbol.

Changing scoping

By default, only built-in functions and symbols like `nil` and `true` are visible inside contexts other than `MAIN`. To make a symbol visible to every context, use the [global](#) function:

```

(set 'aVar 123) → 123
(global 'aVar) → aVar

(context 'FOO) → FOO

aVar → 123

```

Without the `global` statement, the second `aVar` would have returned `nil` instead of `123`. If `FOO` had a previously defined symbol (`aVar` in this example) *that* symbol's value — and not the global's — would be returned instead. Note that only symbols from the `MAIN` context can be made global.

Once it is made visible to contexts through the [global](#) function, a symbol cannot be hidden from them again.

Symbol protection

By using the [constant](#) function, symbols can be both set and protected from change at the same time:

```
(constant 'aVar 123) → 123
(set 'aVar 999)
error: symbol is protected in function set : aVar
```

A symbol needing to be both a constant and a global can be defined simultaneously:

```
(constant (global 'aVar) 123)
```

In the current context, symbols protected by `constant` can be overwritten by using the `constant` function again. This protects the symbols from being overwritten by code in other contexts.

Overwriting global symbols and built-ins

Global and built-in function symbols can be overwritten inside a context by prefixing them with their *own* context symbol:

```
(context 'Account)

(define (Account:new ...)
  (...))
(context 'MAIN)
```

In this example, the built-in function [new](#) is overwritten by `Account:new`, a different function that is private to the `Account` context.

Variables containing contexts

Variables can be used to refer to contexts:

```
(set 'FOO:x 123)

(set 'ctx FOO) → FOO

ctx:x → 123

(set 'ctx:x 999) → 999
```

FOO:x

→ 999

Context variables are used when creating contexts with the [new](#) function (*objects*), as well as when writing functions for uninstantiated contexts.

They also allow for *pass-by-reference* of large data objects when contained inside contexts and passed to functions as context variables.

Sequence of creating or loading contexts

The sequence in which contexts are created or loaded can lead to unexpected results. Enter the following code into a file called `demo`:

```
;; demo - file for loading contexts
(context 'FOO)
  (set 'ABC 123)
(context MAIN)

(context 'ABC)
  (set 'FOO 456)
(context 'MAIN)
```

Now load the file into the newlisp shell:

```
> (load "demo")
symbol is protected in function set : FOO
> _
```

Loading the file causes an error message for FOO, but not for ABC. When the first context FOO is loaded, the context ABC does not exist yet, so a local variable FOO:ABC gets created. When ABC loads, FOO already exists as a global protected symbol and will be correctly flagged as protected.

FOO could still be used as a local variable in the ABC context by explicitly prefixing it, as in ABC:FOO.

The following pattern can be applied to avoid unexpected behavior when loading contexts being used as modules to build larger applications:

```
;; begin of file - MyModule.lsp
(load "This.lsp")
(load "That.lsp")
(load "Other.lsp")

(context 'MyModule)

...

(define (func x y z) (...))

...

(context 'MAIN)

(MyModule:func 1 2 3)
```

```
(exit)

;; end of file
```

Always load the modules required by a context *before* the module's `context` statement. Always finish by switching back to the `MAIN` context, where the module's functions and values can be safely accessed.

Symbol creation in contexts

The following rules should simplify the process of understanding contexts by identifying which ones the created symbols are being assigned to.

1. newLISP first parses and translates the expression, then evaluates it if on the top level. The symbols are created during the parsing and translation phase.
2. A symbol is created when newLISP first *sees* it, when calling the [load](#), [sym](#), or [eval-string](#) functions. When newLISP reads a source file, symbols are created *before* evaluation occurs.
3. Once a symbol is created and assigned to a specific context, it will belong to that context permanently.
4. When an unknown symbol is encountered during code translation, a search for its definition begins inside the current context. Failing that, the search continues inside `MAIN` for a built-in function, context, or global symbol. If no definition is found, the symbol is created locally inside the current context.
5. Expressions and user-defined functions are evaluated in the context they are defined in.

(§)

16. Programming with context objects

Because contexts hold variables and functions and are lexically separated from each other, they can be used for *prototype-based* programming.

```
(context 'ACCOUNT)
  (set 'full-name "")
  (set 'balance 0.0)
  (set 'phone "")

  (define (deposit amount)
    (inc 'balance amount))

  (define (withdraw amount)
    (dec 'balance amount))
(context MAIN)
```

The ACCOUNT context serves as a *prototype* for account objects:

```
(new ACCOUNT 'John)    ; this creates a new context copy of
                        ; ACCOUNT called 'John'

(set 'John:full-name "John Doe")
(set 'John:phone "555-123-456")

(John:deposit 100.00)
(John:withdraw 60)

(new ACCOUNT 'Anne)
(set 'Anne:full-name "Anne Somebody")
(set 'Anne:phone "555-456-123")

(Anne:deposit 120.00)
(Anne:withdraw 50)
```

The previous example uses the function [new](#) to create a pair of contexts cloned from the ACCOUNT prototype. Object-oriented-programming (OOP) purists would use *getter* and *setter* functions to access the object's variables. This is unnecessary in newLISP because prefixing context variables with a context/object name makes them public. *Mixins* are possible using [new](#), which allows for various contexts to be combined into one. See [new](#)'s description for details.

Late binding of context symbols

Once a context is assigned to a variable, it can be referenced through the variable name. In the following example, the `report` function contains a parameter named `acct`, which refers to the passed context:

```
(define (report acct)
  (println
    (format "%-20s %8.2f" acct:full-name acct:balance)))

(report John)
John Doe           40.00

(report Anne)
Anne Somebody      70.00

;; eval symbols to contexts first. John and Anne are symbols
;; in a list, with the contexts inside.

(map report (map eval '(John Anne)))
John Doe           40.00
Anne Somebody      70.00
```

Here, [map](#) applies the function `report` to the context objects, John and Anne. The inner `map` evaluates the context symbols, producing the actual contexts, which are referenced inside `report` through the `acct` parameter.

The `report` function can be defined before any contexts passed to it. The `acct` context, along with the variables `acct:full-name` and `acct:balance`, are not resolved until the function is evaluated. This *late binding* of variable symbols facilitates using contexts as dynamic referent software objects, which are available at runtime.

The context default function

A *default function* is a user-defined function or macro with the same name as its context. When the context is used as the name of a function, newLISP executes the default function.

```
(define (foo:foo a b c) (+ a b c))
(foo 1 2 3) → 6
```

This allows a function defined inside a context to be called whenever the context is applied as a function. A default function can update the lexically isolated static variables contained inside its context:

```
(define (gen:gen x)
  (if gen:acc
      (inc 'gen:acc x)
      (set 'gen:acc x)))

(gen 1) → 1
(gen 1) → 2
(gen 2) → 4
(gen 3) → 7

gen:acc → 7
```

The first time the `gen` function is called, its accumulator is set to the value of the argument. Each successive call increments `gen`'s accumulator by the argument's value.

If a default function is called from a context other than `MAIN`, the context must already exist or be declared with a *forward declaration*, which creates the context and the function symbol:

```
; forward declaration of default function
(define fubar:fubar)

(context 'foo)
(define (foo:foo a b c)
  ...
  (fubar a b)          ; forward reference
  (...))               ; to default function

(context MAIN)

;; definition of previously declared default function

(context 'fubar)
(define (fubar:fubar x y)
  (...))

(context MAIN)
```

Default functions work like global functions, but they are lexically separate from the context in which they are called. The arguments in a default function macro are safe from variable capture.

Like a lambda or lambda-macro function, default functions can be used with [map](#) or [apply](#).

Passing objects by reference

In newLISP, all parameters are passed *by value*. This poses a potential problem when passing large lists or strings to user-defined functions or macros. Symbols and context objects can also be passed by reference. This allows memory-intensive objects to be passed without the overhead of copying the entire list or string:

```
;; pass an object by reference

(set 'mydb:data (sequence 1 100000))

(define (change-db obj idx value)
  (nth-set (obj:data idx) value))

(change-db mydb 1234 "abcdefg")

(nth 1234 mydb:data) → "abcdefg"
```

This example shows how objects can be passed *by reference* to a user-defined function using [context variables](#), without the overhead of passing them by value. String buffers or data objects enclosed in a context can also be passed using this technique.

As shown in the following variation, using [default functor](#) can further simplify the syntax:

```
;; pass a context containing a default functor

(set 'mydb:mydb (sequence 1 100000))

(define (change-db obj idx value)
  (nth-set (obj idx) value))

(change-db mydb 1234 "abcdefg")

(mydb 1234) → "abcdefg"
```

This shows that the function `change-db` does not need to know the name of the variable inside the context. This technique works for arrays and strings, as well.

Contexts as prototypes

To create object prototypes, use dynamic context variables defined inside a context. As with `make-new` in the example below, a method can be defined that initializes variables inside instantiated objects.


```

(context 'Account)

(define (make-new ctx nme bal ph)
  (new Account ctx)
  (set 'ctx (eval ctx))          ; get context out of symbol

  (set 'ctx:full-name nme)      ; initialize new object
  (set 'ctx:balance bal)
  (set 'ctx:phone ph))

(define (Account:deposit amount)
  (inc 'balance amount))

(define (Account:withdraw amount)
  (dec 'balance amount))

(context MAIN)

(Account:make-new 'JD-001 "John Doe" 123.45 "555-555-1212")

;; or when creating an account from inside a different context

(Account:make-new 'MAIN:JD-001 "John Doe" 123.45 "555-555-1212")

JD-001:balance → 123.45

```

Note: Before initialization can occur, the symbol passed as the context name and bound to the parameter `ctx` must be extracted using `eval`.

Lexical and static scoping in newLISP

A default function looks and behaves like statically scoped functions found in other programming languages. Several functions can *share one* lexical closure.

Using [def-new](#), a function or macro can be defined to define other statically scoped functions:

```

;; define static functions (use only in context MAIN)
;;
;; Example:
;;
;; (def-static (foo x) (+ x x))
;;
;; foo:foo → (lambda (foo:x) (+ foo:x foo:x))
;;
;; (foo 10) → 20
;;
(define-macro (def-static)
  (let (temp (append (lambda) (list (1 (args 0)) (args 1)))))
    (def-new 'temp (sym (args 0 0) (args 0 0)))))

```

When execution is complete, a lambda function's parameters are set to `nil`. The contents of a variable are kept inside its own defining context (lexical environment):

```

(def-static (acc x)
  (if sum
    (inc 'sum x)
    (set 'sum x)))

```

```

(acc 5) → 5
(acc 5) → 10
(acc 2) → 12

acc:sum → 12
acc:x   → nil

```

The example shows `acc:x` and `acc:sum` acting like an *automatic local* and a *local static* variable.

When forward referencing a statically defined function inside *another* statically defined function, the forwarded function must have been declared beforehand:

```

(define foo:foo) ; declare so it can be
                  ; referenced before definition

;; foo is forward referenced
(def-static (forward x) (foo x))
(def-static (foo x) (+ x x))

(forward 10) → 20

```

Without having pre-declared `foo:foo`, it would not be possible to reference it in another statically defined function.

Use the `def-static` function inside the `MAIN` context only.

Note that the keywords `fn` and `lambda` have the same effect and are interchangeable.

Serializing context objects

Serialization makes a software object *persistent* by converting it into a character stream, which is then saved to a file or string in memory. In newLISP, any object can be serialized to a file by using the [save](#) function. Like other symbols, contexts are saved just by using their names:

```

(save "mycontext.lsp" 'MyCtx) ; save MyCtx to
mycontext.lsp

(load "mycontext.lsp") ; loads MyCtx into
memory

(save "mycontexts.lsp" 'Ctx1 'Ctx2 'Ctx3) ; save multiple
contexts at once

```

For details, see the functions [save](#) (mentioned above) and [source](#) (for serializing to a newLISP string).

(§)

17. XML, S-XML, and XML-RPC

newLISP's built-in support for XML-encoded data or documents comprises three functions: [xml-parse](#), [xml-type-tags](#), and [xml-error](#).

Use the [xml-parse](#) function to parse XML-encoded strings. When `xml-parse` encounters an error, `nil` is returned. To diagnose syntax errors caused by incorrectly formatted XML, use the function [xml-error](#). The [xml-type-tags](#) function can be used to control or suppress the appearance of XML type tags. These tags classify XML into one of four categories: text, raw string data, comments, and element data.

XML source:

```
<?xml version="1.0"?>
<DATABASE name="example.xml">
<!--This is a database of fruits-->
  <FRUIT>
    <NAME>apple</NAME>
    <COLOR>red</COLOR>
    <PRICE>0.80</PRICE>
  </FRUIT>
</DATABASE>
```

Parsing without options:

```
(xml-parse (read-file "example.xml"))
→ ((("ELEMENT" "DATABASE" (("name" "example.xml")) ("TEXT"
"\r\n")
  ("COMMENT" "This is a database of fruits")
  ("TEXT" "\r\n")
  ("ELEMENT" "FRUIT" () (
    ("TEXT" "\r\n\t")
    ("ELEMENT" "NAME" () (("TEXT" "apple")))
    ("TEXT" "\r\n\t\t")
    ("ELEMENT" "COLOR" () (("TEXT" "red")))
    ("TEXT" "\r\n\t\t")
    ("ELEMENT" "PRICE" () (("TEXT" "0.80")))
    ("TEXT" "\r\n\t\t"))
  ("TEXT" "\r\n")))))
```

S-XML can be generated directly from XML using [xml-type-tags](#) and the special option parameters of the [xml-parse](#) function:

S-XML generation using all options:

```
(xml-type-tags nil nil nil nil)
(xml-parse (read-file "example.xml") (+ 1 2 4 8 16))
→ ((DATABASE (@ (name "example.xml"))
  (FRUIT (NAME "apple")
    (COLOR "red")
    (PRICE "0.80"))))
```

S-XML is XML reformatted as LISP *S-expressions*. The @ (at symbol) denotes an XML attribute specification.

See [xml-parse](#) in the reference section of the manual for details on parsing and option numbers, as well as for a longer example.

XML-RPC

The remote procedure calling protocol XML-RPC uses HTTP post requests as a transport and XML for the encoding of method names, parameters, and parameter types. XML-RPC client libraries and servers have been implemented for most popular compiled and scripting languages.

For more information about XML, visit www.xmlrpc.com.

XML-RPC clients and servers are easy to write using newLISP's built-in support for XML and HTTP request functions. Version 8.4.0 of newLISP introduced a working newLISP XML-RPC server. This stand-alone server is part of the source distribution and does not require any other web server to run. It is located in `examples/xmlrpc-server` and is started on all platforms using the following shell command:

```
newlisp -x 8080 xmlrpc-server
```

This command assumes the newLISP executable is located on the shell path and that the file `xmlrpc-server` is in the current directory. A port other than 8080 could have been chosen instead. The server maintains *state* between function calls. If a stateless XML-RPC server is required, the file `examples/xmlrpc.cgi` can be used together with any web server, including Apache. Both XML-RPC service scripts implement the following methods:

<i>method</i>	<i>description</i>
<code>system.listMethods</code>	Returns a list of all method names
<code>system.methodHelp</code>	Returns help for a specific method
<code>system.methodSignature</code>	Returns a list of return/calling signatures for a specific method
<code>newLISP.evalString</code>	Evaluates a Base64 newLISP expression string

The first three methods are *discovery* methods implemented by most XML-RPC servers. The last one is specific to the newLISP XML-RPC server and implements remote evaluation of a Base64-encoded string of newLISP source code. newLISP's [base64-enc](#) and [base64-dec](#) functions can be used to encode and decode Base64-encoded information.

In the `modules` directory of the source distribution, the file `xmlrpc-client.lsp` implements a specific client interface for all of the above methods. In a future version, a *generic* `XMLRPC:call` function could be used to call any function in a remote XML-RPC server. After starting the server, the following code would be used to access it remotely:

```
(load "xmlrpc-client.lsp") ; load XML-RPC client routines

(XMLRPC:newLISP.evalString
 "http://localhost:8080"
 "(+ 3 4)" → "7")
```

In a similar fashion, standard `system.xxx` calls can be issued.

All functions return either a result if successful, or `nil` if a request fails. In case of failure, `XMLRPC:error` can be evaluated to return an error message.

For more information, please consult the header of the file `modules/xmlrpc-client.lsp`.

(§)

18. Customization, localization, and UTF-8

All built-in primitives in newLISP can be easily renamed:

```
(constant 'plus +)
```

Now, `plus` is functionally equivalent to `+` and runs at the same speed. As with many scripting languages, this allows for double precision floating point arithmetic to be used throughout newLISP.

The [constant](#) function, rather than the `set` function, must be used to rename built-in primitive symbols. By default, all built-in function symbols are protected against accidental overwriting.

```
(constant '+ add)
(constant '- sub)
(constant '* mul)
(constant '/ div)
```

All operations using `+`, `-`, `*`, and `/` are now performed as floating point operations.

Using the same mechanism, the names of built-in functions can be translated into languages other than English:

```
(constant 'wurzel sqrt) ; German for 'square-root'
(constant 'imprime print) ; Spanish for 'print'
...
```

Switching the locale

newLISP can switch locales based on the platform and operating system. On startup, newLISP attempts to set the ISO C standard default POSIX locale, available for most platforms and locales. Use the [set-locale](#) function to switch to the default locale:

```
(set-locale "")
```

This switches to the default locale used on your platform/operating system and ensures character handling (e.g., [upper-case](#)) work correctly.

Many Unix systems have a variety of locales available. To find out which ones are available on a particular Linux/UNIX/BSD system, execute the following command in a system shell:

```
locale -a
```

This command prints a list of all the locales available on your system. Any of these may be used as arguments to [set-locale](#):

```
(set-locale "en_US")
```

This would switch to a U.S. Spanish locale. Accents or other characters used in a U.S. Spanish environment would be correctly converted.

See the manual description for more details on the usage of [set-locale](#).

Decimal point and decimal comma

Many countries use a comma instead of a period as a decimal separator in numbers. newLISP correctly parses numbers depending on the locale set:

```
;; switch to German locale on a Linux system
(set-locale "de_DE")

;; newLISP source and output use a decimal comma
(div 1,2 3) → 0,4
```

The default POSIX C locale, which is set when newLISP starts up, uses a period as a decimal separator.

The following countries use a period as a decimal separator:

Australia, Botswana, Canada (English-speaking), China, Costa Rica, Dominican Republic, El Salvador, Guatemala, Honduras, Hong Kong, India, Ireland, Israel, Japan, Korea (both North and South), Malaysia, Mexico, Nicaragua, New Zealand, Panama, Philippines, Puerto Rico, Saudi Arabia, Singapore, Thailand, United Kingdom, and United States

The following countries use a comma as a decimal separator:

Albania, Andorra, Argentina, Austria, Belarus, Belgium, Bolivia, Brazil, Bulgaria, Canada (French-speaking), Croatia, Cuba, Chile, Colombia, Czech Republic, Denmark, Ecuador, Estonia, Faroes, Finland, France, Germany, Greece, Greenland, Hungary, Indonesia, Iceland, Italy, Latvia, Lithuania, Luxembourg, Macedonia, Moldova, Netherlands, Norway, Paraguay, Peru, Poland, Portugal, Romania, Russia, Serbia, Slovakia, Slovenia, Spain, South Africa, Sweden, Switzerland, Ukraine, Uruguay, Venezuela, and Zimbabwe

Unicode and UTF-8 encoding

Note that for many European languages, the [set-locale](#) mechanism is sufficient to display non-ASCII character sets, as long as each character is presented as *one* byte internally. UTF-8 encoding is only necessary for multibyte character sets as described in this chapter.

newLISP can be compiled as a UTF-8-enabled application. UTF-8 is a multibyte encoding of the international Unicode character set. A UTF-8-enabled newLISP running on an operating system with UTF-8 enabled can handle any character of the installed locale.

The following steps make UTF-8 work with newLISP on a specific operating system and platform:

(1) Use one of the makefiles ending in `utf8` to compile newLISP as a UTF-8 application. If no UTF-8 makefile is available for your platform, the normal makefile for your operating system contains instructions on how to change it for UTF-8.

The Mac OS X binary installer contains a UTF-8-enabled version by default.

(2) Enable the UTF-8 locale on your operating system. Check and set a UTF-8 locale on Unix and Unix-like OSes by using the `locale` command or the `set-locale` function within newLISP. On Linux, the locale can be changed by setting the appropriate environment variable. The following example uses `bash` to set the U.S. locale:

```
export LC_CTYPE=en_US.UTF-8
```

(3) The UTF-8-enabled newLISP automatically switches to the locale found on the operating system. Make sure the command shell is UTF-8-enabled. When using the Tcl/Tk front-end on Linux/UNIX, Tcl/Tk will automatically switch to UTF-8 display as long as the UNIX environment variable is set correctly. The U.S. version of WinXP's `notepad.exe` can display Unicode UTF-8-encoded characters, but the command shell and the Tcl/Tk front-end cannot. On Linux and other UNIXes, the Xterm shell can be used when started as follows:

```
LC_CTYPE=en_US.UTF-8 xterm
```

The following procedure can now be used to check for UTF-8 support. After starting newLISP, type:

```
(println (char 937))           ; displays Greek uppercase
omega                         ; displays lowercase omega
(println (lower-case (char 937))) ; displays lowercase omega
```

While the uppercase omega (Ω) looks like a big O on two tiny legs, the lowercase omega (ω) has a shape similar to a small w in the Latin alphabet.

Note: Only the output of `println` will be displayed as a character; `println`'s return value will appear on the console as a multibyte ASCII character.

When UTF-8-enabled newLISP is used on a non-UTF-8-enabled display, both the output and the return value will be two characters. These are the two bytes necessary to encode the omega character.

When UTF-8-enabled newLISP is used, the following string functions work on character rather than byte boundaries:

<i>function</i>	<i>description</i>
<code>char</code>	translates between characters and ASCII/Unicode
<code>chop</code>	chops characters from the end of a string
<code>date</code>	converts date number to string (when used with the third argument)

explode	transforms a string into a list of characters
first	gets first element in a list (car, head) or string
last	returns the last element of a list or string
lower-case	converts a string to lowercase characters
nth	gets the <i>nth</i> element of a list or string
nth-set	changes the <i>nth</i> element of a list or string
pop	deletes an element from a list or string
push	inserts a new element in a list or string
rest	gets all but the first element of a list (cdr, tail) or string
select	selects and permutes elements from a list or string
set-nth	changes an element in a list or string
title-case	converts the first character of a string to uppercase
trim	trims a string from both sides
upper-case	converts a string to uppercase characters

All other string functions work on bytes. When positions are returned, as in [find](#) or [regex](#), they are byte positions rather than character positions. The [slice](#) function takes not character offset, but byte offsets. The [reverse](#) function reverses a byte vector, not a character vector. The last two functions can still be used to manipulate binary non-textual data in the UTF-8-enabled version of newLISP.

To enable UTF-8 in Perl Compatible Regular Expressions (PCRE) — used by [directory](#), [find](#), [parse](#), [regex](#), and [replace](#) — set the option number accordingly (2048). See the [regex](#) documentation for details.

Use [explode](#) to obtain an array of UTF-8 characters and to manipulate characters rather than bytes when a UTF-8-enabled function is unavailable:

```
(join (reverse (explode str))) ; reverse UTF-8 characters
```

The above string functions (often used to manipulate non-textual binary data) now work on character, rather than byte, boundaries, so care must be exercised when using the UTF-8-enabled version. The size of the first 127 ASCII characters — along with the characters in popular code pages such as ISO 8859 — is one byte long. When working exclusively within these code pages, UTF-8-enabled newLISP is not required. The [set-locale](#) function alone is sufficient for localized behavior.

Two new functions are available for converting between four-byte Unicode (UCS-4) and multibyte UTF-8 code. The [UTF-8](#) function converts UCS-4 to UTF-8, and the [unicode](#) function converts UTF-8 or ASCII strings into USC-4 Unicode.

These functions are rarely used in practice, as most Unicode text files are already UTF-8-encoded (rather than UCS-4, which uses four-byte integer characters). Unicode can be displayed directly when using the "%ls" [format](#) specifier.

For further details on UTF-8 and Unicode, consult [UTF-8 and Unicode FAQ for Unix/Linux](#) by Markus Kuhn.

(§)

19. Commas in parameter lists

Some of the example programs contain functions that use a comma to separate the parameters into two groups. This is not a special syntax of newLISP, but rather a visual trick. The comma is a symbol just like any other symbol. The parameters after the comma are not required when calling the function; they simply declare local variables in a convenient way. This is possible in newLISP because parameter variables in lambda expressions are local and arguments are optional:

```
(define (my-func a b c , x y z)
  (set 'x ...)
  (...))
```

When calling this function, only `a`, `b`, and `c` are used as parameters. The others (`x`, `y`, and `z`) are initialized to `nil` and are local to the function. After execution, the function's contents are forgotten and the environment's symbols are restored to their previous values.

For other ways of declaring and initializing local variables, see [let](#), [letex](#), [letn](#) and [local](#).

(§)

20. Linking newLISP source and executable

Source code and the newLISP executable can be linked together to build a self-contained application by using `link.lsp`. This program is located in the `examples` directory of the source distribution. As an example, the following code is linked to the newLISP executable to form a simple, self-contained application:

```
;; uppercase.lsp - Link example
(println (upper-case (nth 1 (main-args))))
(exit)
```

This program, which resides in the file `uppercase.lsp`, takes the first word on the command line and converts it to uppercase.

To build this program as a self-contained executable, follow these four steps:

(1) Put the following files into the same directory: (a) a copy of the newLISP executable; (b) `newlisp` (or `newlisp.exe` on Win32); (c) `link.lsp`; and (d) the program to link with (`uppercase.lsp` in this example).

(2) In a shell, go to the directory referred to in step 1 and load `link.lsp`:

```
newlisp link.lsp
```

(3) In the newLISP shell, type one of the following:

```
(link "newlisp.exe" "uppercase.exe" "uppercase.lsp") ; Win32
(link "newlisp" "uppercase" "uppercase.lsp") ; Linux/BSD
```

(4) Exit the newLISP shell and type:

```
uppercase "convert me to uppercase"
```

newLISP Users Manual and Reference

The console should print:

```
CONVERT ME TO UPPERCASE
```

Note: On Linux/BSD, the new file must be marked executable for the operating system to recognize it:

```
chmod 755 uppercase
```

This gives the file executable permission (this step is unnecessary on Win32).

([o](#))

newLISP Function Reference

1. Syntax of symbol variables and numbers

Source code in newLISP is parsed according the rules outlined here. When in doubt, verify the behavior of newLISP's internal parser by calling [parse](#) without optional arguments.

Symbols for variable names

The following rules apply to the naming of symbols used as variables or functions:

1. Variable symbols may not start with any of the following characters:
; " ' () { } . , 0 1 2 3 4 5 6 7 8 9
2. Variable symbols starting with a + or – cannot have a number as the second character.
3. Any character is allowed inside a variable name, except for:
" ' () : , and the space character. These mark the end of a variable symbol.
4. A symbol name starting with [(left square bracket) and ending with] (right square bracket) may contain any character except the right square bracket.

All of the following symbols are legal variable names in newLISP:

example:

```
myvar
A-name
X34-zz
[* 7 5 ( )};]
*111*
```

Sometimes it is useful to create hash-like [lookup dictionaries](#) with keys containing characters that are illegal in newLISP variables. The functions [sym](#) and [context](#) can be used to create symbols containing these characters:

```
(set (sym "(#:L*)" 456) ⇒ 456
(eval (sym "(#:L*")) ⇒ 456
(set (sym 1) 123) ⇒ 123
(eval (sym 1)) ⇒ 123
```

```
1      ⇒ 1
(+ 1 2) ⇒ 3
```

The last example creates the symbol 1 containing the value 123. Also note that creating such a symbol does not alter newLISP's normal operations, since 1 is still parsed as the number one.

Numbers

newLISP recognizes the following number formats:

Integers are one or more digits long, optionally preceded by a + or – sign. Any other character marks the end of the integer or may be part of the sequence if parsed as a float (see float syntax below).

example:

```
123
+4567
-999
```

Hexadecimals start with a 0x (or 0X) followed by any combination of the hexadecimal digits: 0123456789abcdefABCDEF. Any other character ends the hexadecimal number.

example:

```
0xFF      ⇒ 255
0x10ab    ⇒ 4267
0X10CC    ⇒ 4300
```

Octals start with an optional + (plus) or – (minus) sign and a 0 (zero), followed by any combination of the octal digits: 01234567. Any other character ends the octal number.

example:

```
012      ⇒ 10
010      ⇒ 8
077      ⇒ 63
-077     ⇒ -63
```

Floating point numbers can start with an optional + (plus) or – (minus) sign, but they cannot be followed by a 0 (zero) if they are. This would make them octal numbers instead of floating points. A single . (decimal point) can appear anywhere within a floating point number, including at the beginning.

example:

```
1.23      ⇒ 1.23
-1.23     ⇒ -1.23
+2.3456   ⇒ 2.3456
.506      ⇒ 0.506
```

As described above, **scientific notation** starts with a floating point number called the *significand* (or *mantissa*), followed by the letter e or E and an integer *exponent*.

example:

```
1.23e3    ⇒ 1230
-1.23E3   ⇒ -1230
+2.34e-2  ⇒ 0.0234
.506E3    ⇒ 506
```

2. Data types and names in the reference

To describe the types and names of a function's parameters, the following naming convention is used throughout the reference section:

syntax: (format *str-format exp-data-1* [*exp-data-i* ...])

Arguments are represented by symbols formed by the argument's type and name, separated by a – (hyphen). Here, *str-format* (a string) and *exp-data-1* (an expression) are named "format" and "data-1", respectively.

bool

true, nil, or an expression evaluating to one of these two.

```
true, nil, (<= X 10)
```

int

An integer or an expression evaluating to an integer. Generally, if a floating point number is used when an int is expected, the value is truncated to an integer.

```
123, 5, (* X 5)
```

num

An integer, a floating point number, or an expression evaluating to one of these two. If an integer is passed, it is converted to a floating point number.

```
1.234, (div 10 3), (sin 1)
```

matrix

A list in which each row element is itself a list or an array in which each row element is itself an array. All element lists or arrays (rows) are of the same length. When using [det](#), [multiply](#), or [invert](#), all numbers must be floats or integers.

The dimensions of a matrix are defined by indicating the number of rows and the number of column elements per row. Functions working on matrices ignore superfluous columns in a row. For missing row elements, 0.0 is assumed by the functions [det](#), [multiply](#), and [invert](#), while [transpose](#) assumes nil. Special rules apply for [transpose](#) when a whole row is not a list or an array, but some other data type.

```
((1 2 3 4)
 (5 6 7 8)
 (9 10 11 12))      ; 3 rows 4 columns

((1 2) (3 4) (5 6)) ; 3 rows 2 columns
```

str

A string or an expression that evaluates to a string.

```
"Hello", (append first-name " Miller")
```

Special characters can be included in quoted strings by placing a \ (backslash) before the character or digits to escape them:

<i>escaped character</i>	<i>description</i>
\n	the line feed character (ASCII 10)
\r	the carriage return character (ASCII 13)
\t	the tab character (ASCII 9)
\nnn	a decimal ASCII code where nnn is between 000 and 255

```
"\065\066\067" ⇒ "ABC"
```

Instead of a " (double quote), a { (left curly bracket) and } (right curly bracket) can be used to delimit strings. This is useful when quotation marks need to occur inside strings. Quoting with the curly brackets suppresses the backslash escape effect for special characters. Balanced nested curly brackets may be used within a string. This aids in writing regular expressions or short sections of HTML.

```
(print "<A HREF=\"http://mysite.com\">" ) ; the cryptic way

(print {<A HREF="http://mysite.com">} )   ; the readable way

;; also possible because the inner brackets are balanced
(regex {abc{1,2}} line)

(print [text]
      this could be
```

```
a very long (> 2048 characters) text,  
i.e. HTML.  
[/text])
```

The tags `[text]` and `[/text]` can be used to delimit long strings and suppress escape character translation. This is useful for delimiting long HTML passages in CGI files written in newLISP or for situations where character translation should be completely suppressed. Always use the `[text]` tags for strings longer than 2048 characters.

sym

A symbol or expression evaluating to a symbol.

```
'xyz, (first '(+ - /)), '*', '- ', 'someSymbol,
```

context

An expression evaluating to a context (namespace) or a variable symbol holding a context.

```
MyContext, aCtx, TheCTX
```

Most of the context symbols in this manual start with an uppercase letter to distinguish them from other symbols.

sym-context

A symbol, an existing context, or an expression evaluating to a symbol from which a context will be created. If a context does not already exist, many functions implicitly create them (e.g., [bayes-train](#), [context](#), [eval-string](#), [load](#), [sym](#), and [xml-parse](#)). The context must be specified when these functions are used on an existing context. Even if a context already exists, some functions may continue to take symbols (e.g., [context](#)). For other functions, such as [context?](#), the distinction is critical.

func

A symbol or an expression evaluating to an operator symbol or lambda expression.

```
+, add, (first '(add sub)), (lambda (x) (+ x x))
```

list

A list of elements (any type) or an expression evaluating to a list.

```
(a b c "hello" (+ 3 4))
```

array

An array (constructed with the [array](#) function).

exp

Any of the above.

body

One or more expressions that can be evaluated. The expressions are evaluated sequentially if there is more than one.

```
1 7.8
nil
(+ 3 4)
"Hi" (+ a b) (print result)
(do-this) (do-that) 123
```

3. Functions in groups

Some functions appear in more than one group.

List processing, flow control, and integer arithmetic

+, -, *, /, %	integer arithmetic
<, >, =	compares any data type: less, greater, equal
<=, >=, !=	compares any data type: less-equal, greater-equal, not-equal
and	logical and
append	appends lists ,arrays or strings to form a new list, array or string
apply	applies a function or primitive to a list of arguments
args	retrieves the argument list of a macro expression
assoc	searches for keyword associations in a list
begin	begins a block of functions
case	branches depending on contents of control variable
catch	evaluates an expression, possibly catching errors
chop	chops elements from the end of a list
clean	cleans elements from a list

List processing, flow control, and integer arithmetic

newLISP Users Manual and Reference

<u>cond</u>	branches conditionally to expressions
<u>cons</u>	prepends an element to a list, making a new list
<u>constant</u>	defines a constant symbol
<u>count</u>	counts elements of one list that occur in another list
<u>define</u>	defines a new function or lambda expression
<u>define-macro</u>	defines a macro or lambda-macro expression
<u>def-new</u>	copies a symbol to a different context (namespace)
<u>difference</u>	returns the difference between two lists
<u>dolist</u>	evaluates once for each element in a list
<u>dotimes</u>	evaluates once for each number in a range
<u>dotree</u>	iterates through the symbols of a context
<u>do-until</u>	repeats evaluation of an expression until the condition is met
<u>do-while</u>	repeats evaluation of an expression while the condition is true
<u>dup</u>	duplicates a list or string a specified number of times
<u>ends-with</u>	checks the end of a string or list against a key of the same type
<u>eval</u>	evaluates an expression
<u>expand</u>	replaces a symbol in a nested list
<u>first</u>	gets the first element of a list or string
<u>filter</u>	filters a list
<u>find</u>	searches for an element in a list or string
<u>flat</u>	returns the flattened list
<u>fn</u>	defines a new function or lambda expression
<u>for</u>	evaluates once for each number in a range
<u>if</u>	evaluates an expression conditionally
<u>index</u>	filters elements from a list and returns their indices
<u>intersect</u>	returns the intersection of two lists
<u>lambda</u>	defines a new function or lambda expression
<u>last</u>	returns the last element of a list or string
<u>length</u>	calculates the length of a list or string
<u>let</u>	declares and initializes local variables
<u>letex</u>	expands local variables into an expression, then evaluates
<u>letn</u>	initializes local variables incrementally, like <i>nested lets</i>
<u>list</u>	makes a list
<u>local</u>	declares local variables
<u>lookup</u>	looks up members in an association list
<u>map</u>	maps a function over members of a list, collecting the results
<u>match</u>	matches patterns against lists; for matching against strings, see <u>find</u> and <u>regex</u>
<u>member</u>	finds a member of a list or string
<u>name</u>	returns the name of a symbol or its context as a string

List processing, flow control, and integer arithmetic

not	logical not
nth	gets the <i>nth</i> element of a list or string
nth-set	changes the <i>nth</i> element of a list or string
or	logical or
pop	deletes and returns an element from a list or string
push	inserts a new element into a list or string
quote	quotes an expression
ref	returns the position of an element inside a nested list
ref	returns a list of index vectors of element inside a nested list
rest	returns all but the first element of a list or string
replace	replaces elements inside a list or string
replace-assoc	replaces an association within a list
reverse	reverses a list or string
rotate	rotates a list or string
select	selects and permutes elements from a list or string
set	sets the binding or contents of a symbol
setq	sets the binding or contents of an unquoted symbol
set-nth	changes the <i>nth</i> element of a list or string
silent	works like begin but suppresses console output of the return value
slice	extracts a sublist or substring
sort	sorts the members of a list
starts-with	checks the beginning of a string or list against a key of the same type
swap	swaps two elements inside a list or string
unify	unifies two expressions
unique	returns a list without duplicates
unless	evaluates an expression conditionally
until	repeats evaluation of an expression until the condition is met
while	repeats evaluation of an expression while the condition is true

Bit operators

<<, >>	bit shift left, bit shift right
&	bitwise and
 	bitwise inclusive or
^	bitwise exclusive or
~	bitwise not

Floating point math and special functions

abs	calculates the absolute value of a number
---------------------	---

<u>acos</u>	calculates the arccosine of a number
<u>add</u>	adds floating point or integer numbers
<u>array</u>	creates an array
<u>array-list</u>	returns a list conversion from an array
<u>asin</u>	calculates the arcsine of a number
<u>atan</u>	calculates the arctangent of a number
<u>atan2</u>	computes the principal value of the arctangent of Y / X in radians
<u>beta</u>	calculates the beta function
<u>betai</u>	calculates the incomplete beta function
<u>binomial</u>	calculates the binomial function
<u>ceil</u>	rounds up to the next integer
<u>cos</u>	calculates the cosine of a number
<u>crc32</u>	calculates a 32-bit CRC for a data buffer
<u>crit-chi2</u>	calculates the Chi ² for a given probability
<u>crit-z</u>	calculates the normal distributed Z for a given probability
<u>dec</u>	decrements a number
<u>div</u>	divides floating point or integer numbers
<u>erf</u>	calculates the error function of a number
<u>exp</u>	calculates the exponential <i>e</i> of a number
<u>factor</u>	factors a number into primes
<u>fft</u>	performs a fast Fourier transform (FFT)
<u>floor</u>	rounds down to the next integer
<u>flt</u>	converts a number to a 32-bit integer representing a float
<u>gammai</u>	calculates the incomplete Gamma function
<u>gammaln</u>	calculates the log Gamma function
<u>gcd</u>	calculates the greatest common divisor of a group of integers
<u>ifft</u>	performs an inverse fast Fourier transform (IFFT)
<u>inc</u>	increments a number
<u>log</u>	calculates the natural or other logarithm of a number
<u>min</u>	finds the smallest value in a series of values
<u>max</u>	finds the largest value in a series of values
<u>mod</u>	calculates the modulo of two numbers
<u>mul</u>	multiplies floating point or integer numbers
<u>pow</u>	calculates <i>x</i> to the power of <i>y</i>
<u>sequence</u>	generates a list sequence of numbers
<u>series</u>	creates a geometric sequence of numbers
<u>sgn</u>	calculates the signum function of a number
<u>sin</u>	calculates the sine of a number
<u>sqrt</u>	calculates the square root of a number
<u>sub</u>	subtracts floating point or integer numbers

tan	calculates the tangent of a number
uuid	returns a UUID (Universal Unique Identifier)

Matrix functions

det	return the determinant of a matrix
invert	return the inversion of a matrix
multiply	multiplies two matrices
transpose	returns the transposition of a matrix

Array functions

append	appends arrays
array	creates and initializes an array with up to 16 dimensions
array-list	converts an array into a list
array?	checks if expression is an array
det	returns the determinant of a matrix
first	returns the first row of an array
invert	returns the inversion of a matrix
last	returns the last row of an array
multiply	multiplies two matrices
nth	returns an element of and array
nth-set	changes the element, returning the old; significantly faster than <code>set-nth</code>
rest	returns all but the first row of an array
set-nth	changes the element and returns the changed array
slice	returns a slice of an array
transpose	transposes a matrix

Financial math functions

fv	returns the future value of an investment
irr	calculates the internal rate of return
nper	calculates the number of periods for an investment
npv	calculates the net present value of an investment
pv	calculates the present value of an investment
pmt	calculates the payment for a loan

Simulation and modeling math functions

amb	randomly picks an argument and evaluates it
---------------------	---

bayes-query	calculates Bayesian probabilities for a data set
bayes-train	counts items in lists for Bayesian or frequency analysis
normal	makes a list of normal distributed floating point numbers
prob-chi2	calculates the cumulated probability of Chi ²
prob-z	calculates the cumulated probability of a Z-value
rand	generates random numbers in a range
random	generates a list of evenly distributed floats
randomize	shuffles all of the elements in a list
seed	seeds the internal random number generator

Time and date functions

date	converts a date-time value to a string
date-value	calculates the time in seconds since January 1, 1970 for a date and time
now	returns a list of current date-time information
time	calculates the time it takes to evaluate an expression in milliseconds
time-of-day	calculates the number of milliseconds elapsed since the day started

String and conversion functions

address	gets the memory address of a number or string
append	appends lists, arrays or strings to form a new list, array or string
char	translates between characters and ASCII codes
chop	chops off characters from the end of a string
dup	duplicates a list or string a specified number of times
ends-with	checks the end of a string or list against a key of the same type
encrypt	does a one-time-pad encryption and decryption of a string
eval-string	compiles, then evaluates a string
explode	transforms a string into a list of characters
find	searches for an element in a list or string
find-all	returns a list of all pattern matches found in string
first	gets the first element in a list or string
float	translates a string or integer into a floating point number
format	formats numbers and strings as in the C language
get-char	gets a character from a memory address
get-float	gets a double float from a memory address
get-int	gets an integer from a memory address
get-string	gets a string from a memory address
int	translates a string or float into an integer
join	joins a list of strings

<u>last</u>	returns the last element of a list or string
<u>lower-case</u>	converts a string to lowercase characters
<u>member</u>	finds a list or string member
<u>name</u>	returns the name of a symbol or its context as a string
<u>nth</u>	gets the <i>nth</i> element in a list or string
<u>nth-set</u>	changes the <i>nth</i> element of a list or string
<u>pack</u>	packs LISP expressions into a binary structure
<u>parse</u>	breaks a string into tokens
<u>pop</u>	pops from a string
<u>push</u>	pushes onto a string
<u>regex</u>	performs a Perl-compatible regular expression search
<u>replace</u>	replaces elements in a list or string
<u>rest</u>	gets all but the first element of a list or string
<u>reverse</u>	reverses a list or string
<u>rotate</u>	rotates a list or string
<u>select</u>	selects and permutes elements from a list or string
<u>set-nth</u>	changes the element in a list or string
<u>slice</u>	extracts a substring or sublist
<u>source</u>	returns the source required to bind a symbol to a string
<u>starts-with</u>	checks the start of the string or list against a key string or list
<u>string</u>	transforms anything into a string
<u>sym</u>	translates a string into a symbol
<u>title-case</u>	converts the first character of a string to uppercase
<u>trim</u>	trims a string of one or both sides
<u>unicode</u>	converts ASCII or UTF-8 to UCS-4 Unicode
<u>utf8</u>	converts UCS-4 Unicode to UTF-8
<u>unpack</u>	unpacks a binary structure into LISP expressions
<u>upper-case</u>	converts a string to uppercase characters

Input/output and file operations

<u>append-file</u>	appends data to a file
<u>close</u>	closes a file
<u>command-line</u>	enables or disables interactive command line
<u>current-line</u>	retrieves contents of last read-line buffer
<u>device</u>	sets or inquires about current print device
<u>exec</u>	launches another program, then reads from or writes to it
<u>load</u>	loads and evaluates a file of LISP code
<u>open</u>	opens a file for reading or writing

peek	checks file descriptor for number of bytes ready for reading
print	prints to the console or a device
println	prints to the console or a device with a line feed
read-buffer	reads binary data from a file
read-char	reads an 8-bit character from a file
read-file	reads a whole file in one operation
read-key	reads a keyboard key
read-line	reads a line from the console or file
save	saves a workspace, context, or symbol to a file
search	searches a file for a string
seek	sets or reads a file position
write-buffer	writes binary data to a file or string
write-char	writes a character to a file
write-file	writes a file in one operation
write-line	writes a line to the console or a file

Processes, pipes and threads

!	shells out to the operating system
exec	runs a process, then reads from or writes to it
fork	launches a newLISP child process thread
pipe	creates a pipe for interprocess communication
process	launches a child process, remapping standard I/O and standard error
semaphore	creates and controls semaphores
share	shares memory with other processes and threads
wait-pid	waits for a child process to end

File and directory management

change-dir	changes to a different drive and directory
copy-file	copies a file
delete-file	deletes a file
directory	returns a list of directory entries
file-info	gets file size, date, time, and attributes
make-dir	makes a new directory
real-path	returns the full path of the relative file path
remove-dir	removes an empty directory
rename-file	renames a file or directory

System functions and predicates

<u>\$</u>	accesses system variables \$0 -> \$15
<u>atom?</u>	checks if an expression is an atom
<u>array?</u>	checks if an expression is an array
<u>catch</u>	evaluates an expression, catching errors and early returns
<u>context</u>	creates or switches to a different namespace
<u>context?</u>	checks if an expression is a context
<u>debug</u>	debugs a user-defined function
<u>delete</u>	deletes symbols from the symbol table
<u>directory?</u>	checks if a disk node is a directory
<u>empty?</u>	checks if a list or string is empty
<u>env</u>	gets or sets the operating system's environment
<u>error-event</u>	defines an error handler
<u>error-number</u>	gets the last error number
<u>error-text</u>	gets the error text for an error number
<u>exit</u>	exits newLISP, setting the exit value
<u>file?</u>	checks for the existence of a file
<u>float?</u>	checks if an expression is a float
<u>global</u>	makes a symbol accessible outside MAIN
<u>import</u>	imports a function from a shared library
<u>integer?</u>	checks if an expression is an integer
<u>lambda?</u>	checks if an expression is a lambda expression
<u>legal?</u>	checks if a string contains a legal symbol
<u>list?</u>	checks if an expression is a list
<u>macro?</u>	checks if an expression is a lambda-macro expression
<u>main-args</u>	gets command-line arguments
<u>NaN?</u>	checks if a float is NaN (not a number)
<u>new</u>	creates a copy of a context
<u>nil?</u>	checks if an expression is nil
<u>null?</u>	checks if an expression is nil, " ", () or 0.
<u>number?</u>	checks if an expression is a float or an integer
<u>pretty-print</u>	changes the pretty-printing characteristics
<u>primitive?</u>	checks if an expression is a primitive
<u>quote?</u>	checks if an expression is quoted
<u>reset</u>	goes to the top level
<u>set-locale</u>	switches to a different locale
<u>signal</u>	sets a signal handler
<u>sleep</u>	suspends processing for specified milliseconds

string?	checks if an expression is a string
symbol?	checks if an expression is a symbol
symbols	returns a list of all symbols in the system
sys-error	reports OS system error numbers
sys-info	gives information about system resources
throw	causes a previous catch to return
throw-error	throws a user-defined error
timer	starts a one-shot timer, firing an event
trace	sets or inquires about trace mode
trace-highlight	sets highlighting strings in trace mode
true?	checks if an expression is not <code>nil</code>
zero?	checks if an expression is 0 or 0.0

HTTP networking API

base64-enc	encodes a string into BASE64 format
base64-dec	decodes a string from BASE64 format
get-url	reads a file or page from the web
post-url	posts info to a URL address
put-url	uploads a page to a URL address
xml-error	returns last XML parse error
xml-parse	parses an XML document
xml-type-tags	shows or modifies XML type tags

Socket TCP/IP and UDP network API

net-accept	accepts a new incoming connection
net-close	closes a socket connection
net-connect	connects to a remote host
net-error	returns the last error
net-eval	evaluates expressions on multiple remote newLISP servers
net-listen	listens for connections to a local socket
net-local	returns the local IP and port number for a connection
net-lookup	returns the name for an IP number
net-peer	returns the remote IP and port for a net connect
net-peek	returns the number of characters ready to be read
net-ping	sends a ping packet (ICMP echo request) to one or more addresses
net-receive	reads data on a socket connection

net-receive-from	reads a UDP on an open connection
net-receive-udp	reads a UDP and closes the connection
net-select	checks a socket or list of sockets for status
net-send	sends data on a socket connection
net-send-to	sends a UDP on an open connection
net-send-udp	sends a UDP and closes the connection
net-service	translates a service name into a port number
net-sessions	returns a list of currently open connections

Importing libraries

address	gets the memory address of a number or string
flt	converts a number to a 32-bit integer representing a float
float	translates a string or integer into a floating point number
get-char	gets a character from a memory address
get-float	gets a double float from a memory address
get-int	gets an integer from a memory address
get-string	gets a string from a memory address
import	imports a function from a shared library
int	translates a string or float into an integer
pack	packs LISP expressions into a binary structure
unpack	unpacks a binary structure into LISP expressions

newLISP internals API

cpymem	copies memory between addresses
dump	shows memory address and contents of newLISP cells

(⌘)

Functions in alphabetical order



syntax: (! *str-command*)

Executes the command in *str-command* by shelling out to the operating system and executing. This function returns a different value depending on the host operating system.

example:

```
(! "vi")
(! "ls -ltr")
```

Use the [exec](#) function to execute a shell command and capture the standard output or to feed standard input. The [process](#) function may be used to launch a non-blocking child process and redirect std I/O and std error to pipes.

Note that ! (exclamation mark) can be also be used as a command-line shell operator by omitting the parenthesis and space after the !:

example:

```
> !ls -ltr      ; executed in the newLISP shell window
```

Used in this way, the ! operator is not a newLISP function at all, but rather a special feature of the newLISP command shell. The ! must be entered as the first character on the command line.



syntax: (\$ *int-idx*)

The functions that use regular expressions ([directory](#), [find](#), [parse](#), [regex](#), [search](#), and [replace](#)) all bind their results to the predefined system variables \$0, \$1, \$2–\$15 after or during the function's execution. Both [nth-set](#) and [set-nth](#) store the replaced expression in \$0. System variables can be treated the same as any other symbol. As an alternative, the contents of these variables may also be accessed by using (\$ 0), (\$ 1), (\$ 2), etc. This method allows indexed access (i.e., (\$ i), where i is an integer).

example:

```
(set 'str "http://newlisp.org:80")
(find "http://(.*):(.*)" str 0) ⇒ 0

$0 ⇒ "http://newlisp.org:80"
$1 ⇒ "newlisp.org"
$2 ⇒ "80"
```

```

($ 0) ⇒ "http://newlisp.org:80"
($ 1) ⇒ "newlisp.org"
($ 2) ⇒ "80"

(set-nth 2 '(a b c d e f g) 'x) ⇒ (a b x d e f g)

$0      ⇒ c
($ 0)   ⇒ c

```

For using captures within substitutions, the \$ system variables can be accessed from within the functions [nth-set](#), [set-nth](#), and [replace](#):

```

(set 'lst '(1 2 3 4))
(nth-set (lst 3) (* $0 3)) ⇒ 4
lst      ⇒ (1 2 3 12)

```

+, -, *, / ,%

syntax: (+ *int-1* [*int-2* ...])

Returns the sum of all numbers in *int-1* —.

syntax: (- *int-1* [*int-2* ...])

Subtracts *int-2* from *int-1*, then the next *int-i* from the previous result. If only one argument is given, its sign is reversed.

syntax: (* *int-1* [*int-2* ...])

The product is calculated for *int-1* to *int-i*.

syntax: (/ *int-1* [*int-2* ...])

Each result is divided successively until the end of the list is reached. Division by zero causes an error.

syntax: (% *int-1* [*int-2* ...])

Each result is divided successively by the next *int*, then the rest (modulo operation) is returned. Division by zero causes an error. For floating point numbers, use the [mod](#) function.

example:

```

(+ 1 2 3 4 5)      ⇒ 15
(+ 1 2 (- 5 2) 8)   ⇒ 14
(- 10 3 2 1)       ⇒ 4
(- (* 3 4) 6 1 2)   ⇒ 3
(- 123)            ⇒ -123
(map - '(10 20 30)) ⇒ (-10 -20 -30)
(* 1 2 3)          ⇒ 6
(* 10 (- 8 2))      ⇒ 60
(/ 12 3)           ⇒ 4
(/ 120 3 20 2)      ⇒ 1
(% 10 3)           ⇒ 1
(% -10 3)          ⇒ -1
(+ 1.2 3.9)        ⇒ 4

```

Floating point values in arguments to +, -, *, /, and % are truncated to their floor value.

Floating point values larger or smaller than the maximum

(9,223,372,036,854,775,807) or minimum (-9,223,372,036,854,775,808)

integer values are truncated to those values.

Calculations resulting in values larger than 9,223,372,036,854,775,807 or smaller than -9,223,372,036,854,775,808 wrap around from positive to negative or negative to positive.

For floating point values that evaluate to NaN (Not a Number), both +INF and -INF are treated as 0 (zero).

<, >, =, <=, >=, !=

syntax: (< *exp-1 exp-2 [exp-3 ...]*)

syntax: (> *exp-1 exp-2 [exp-3 ...]*)

syntax: (= *exp-1 exp-2 [exp-3 ...]*)

syntax: (<= *exp-1 exp-2 [exp-3 ...]*)

syntax: (>= *exp-1 exp-2 [exp-3 ...]*)

syntax: (!= *exp-1 exp-2 [exp-3 ...]*)

Expressions are evaluated and the results are compared successively. As long as the comparisons conform to the comparison operators, evaluation and comparison will continue until all arguments are tested and the result is `true`. As soon as one comparison fails, `nil` is returned.

All types of expressions can be compared: atoms, numbers, symbols, and strings. List expressions can also be compared (list elements are compared recursively).

When comparing lists, elements at the beginning of the list are considered more significant than the elements following (similar to characters in a string). When comparing lists of different lengths but equal elements, the longer list is considered greater (see examples).

In mixed-type expressions, the types are compared from lowest to highest. Floats and integers are compared by first converting them to the needed type, then comparing them as numbers.

Atoms: `nil`, `true`, integer or float, string, symbol, primitive

Lists: quoted list/expression, list/expression, lambda, lambda-macro

example:

```
(< 3 5 8 9)           ⇒ true
(> 4 2 3 6)           ⇒ nil
(< "a" "c" "d")       ⇒ true
(>= duba aba)         ⇒ true
(< '(3 4) '(1 5))     ⇒ nil
(> '(1 2 3) '(1 2))   ⇒ true
(= '(5 7 8) '(5 7 8)) ⇒ true
(!= 1 4 3 7 3)        ⇒ true
(< 1.2 6 "Hello" 'any '(1 2 3)) ⇒ true
```

<, >, =, <=, >=, !=

```

(< nil true)           ⇒ true
(< '(((a b))) '(((b c)))) ⇒ true
(< '(((a (b c)) '(a (b d)) '(a (b (d)))))) ⇒ true

```

<<, >>

syntax: (<< *int-1* *int-2* [*int-3* ...])

syntax: (>> *int-1* *int-2* [*int-3* ...])

syntax: (<< *int-1*)

syntax: (>> *int-1*)

The number *int-1* is arithmetically shifted to the left or right by the number of bits given as *int-2*, then shifted by *int-3* and so on. For example, 64-bit integers may be shifted up to 63 positions. When shifting right, the most significant bit is duplicated (*arithmetic shift*):

```

(>> 0x8000000000000000 1) ⇒ 0xC000000000000000 ; not
0x0400000000000000!

```

example:

```

(<< 1 3)      ⇒ 8
(<< 1 2 1)    ⇒ 8
(>> 1024 10)  ⇒ 1
(>> 160 2 2)  ⇒ 10

(<< 3)        ⇒ 6
(>> 8)        ⇒ 4

```

When *int-1* is the only argument << and >> shift by one bit.

&

syntax: (& *int-1* *int-2* [*int-3* ...])

A bitwise and operation is performed on the number in *int-1* with the number in *int-2*, then successively with *int-3*, etc.

example:

```

(& 0xAABB 0x000F) ⇒ 11 ; which is 0xB

```

|

syntax: (`|` *int-1* *int-2* [*int-3* ...])

A bitwise or operation is performed on the number in *int-1* with the number in *int-2*, then successively with *int-3*, etc.

example:

```
(| 0x10 0x80 2 1) ⇒ 147
```

^

syntax: (`^` *int-1* *int-2* [*int-3* ...])

A bitwise xor operation is performed on the number in *int-1* with the number in *int-2*, then successively with *int-3*, etc.

example:

```
(^ 0xAA 0x55) ⇒ 255
```

~

syntax: (`~` *int*)

A bitwise not operation is performed on the number in *int*, reversing all of the bits.

example:

```
(format "%X" (~ 0xFFFFF5AA)) ⇒ "55"
(~ 0xFFFFFFFF)                ⇒ 0
```

abs

syntax: (`abs` *num*)

Returns the absolute value of the number in *num*.

example:

```
(abs -3.5) ⇒ 3.5
```

acos

syntax: (acos *num*)

The arccosine function is calculated from the number in *num*.

example:

```
(acos 1) ⇒ 0
```

add

syntax: (add *num-1* [*num-2* ...])

All of the numbers in *num-1*, *num-2*, and on are summed. add accepts float or integer operands, but it always returns a floating point number. Any floating point calculation with NaN also returns NaN.

example:

```
(add 2 3.25 9) ⇒ 14.25
(add 1 2 3 4 5) ⇒ 15
```

address

syntax: (address *int*)

syntax: (address *float*)

syntax: (address *str*)

Returns the memory address of the integer in *int*, the double floating point number in *float*, or the string in *str*. This function is used for passing parameters to library functions that have been imported using the [import](#) function.

example:

```
(set 's "\001\002\003\004")
(get-char (+ (address s) 3)) ⇒ 4
(get-int (address 1234)) ⇒ 1234
(get-float (address 1.234)) ⇒ 1.234
```

When a string is passed, the address of the string is automatically used. As the example shows, address can be used to do pointer arithmetic on the string's address.

See also the [get-char](#), [get-int](#), and [get-float](#) functions.

amb

syntax: (amb *exp-1 exp-2 [exp-3...]*)

One of the expressions *exp-1 ... n* is selected at random, and the evaluation result is returned.

example:

```
(amb 'a 'b 'c 'd 'e) ⇒ one of: a, b, c, d, or e at random
(dotimes (x 10) (print (amb 3 5 7))) ⇒ 35777535755
```

Internally, newLISP uses the same function as [rand](#) to pick a random number. To generate random floating point numbers, use [random](#), [randomize](#), or [normal](#). To initialize the pseudo random number generating process at a specific starting point, use the [seed](#) function.

and

syntax: (and *exp-1 exp-2 [exp-3...]*)

The expressions *exp-1*, *exp-2*, *etc.* are evaluated in order, returning the result of the last expression. If any of the expressions yield `nil`, evaluation is terminated and `nil` is returned.

example:

```
(set 'x 10) ⇒ 10
(and (< x 100) (> x 2)) ⇒ true
(and (< x 100) (> x 2) "passed") ⇒ "passed"
(and '()) ⇒ nil
(and true) ⇒ true
(and) ⇒ nil
```

append

syntax: (append *list-1 [list-2 ...]*)

syntax: (append *array-1 [array-2 ...]*)

syntax: (append *str-1 [str-2 ...]*)

In the first form, `append` works with lists, appending *list-1* through *list-n* to form a new list. The original lists are left unchanged.

example:

```
(append '(1 2 3) '(4 5 6) '(a b)) ⇒ (1 2 3 4 5 6 a b)
(set 'aList '("hello" "world")) ⇒ ("hello" "world")
```

```
(append aList '("here" "I am")) ⇒ ("hello" "world" "here" "I
am")
```

In the second form `append` works on arrays:

example:

```
(set 'A (array 3 2 (sequence 1 6)))
⇒ ((1 2) (3 4) (5 6))
(set 'B (array 2 2 (sequence 7 10)))
⇒ ((7 8) (9 10))

(append A B)
⇒ ((1 2) (3 4) (5 6) (7 8) (9 10))

(append B B B)
⇒ ((7 8) (9 10) (7 8) (9 10) (7 8) (9 10))
```

In the third form, `append` works on strings. The strings in *str-n* are concatenated into a new string and returned.

example:

```
(set 'more " how are you") ⇒ " how are you"

(append "Hello " "world," more) ⇒ "Hello world, how are you"
```

`append` is also suitable for processing binary strings containing zeroes.

Linkage characters or strings can be specified using the [join](#) function. Use the [string](#) function to convert arguments to strings and `append` in one step.

Use the functions [push](#) or [write-buffer](#) (with its special syntax) to append to an existing string *in place*.

append-file

syntax: (append-file *str-filename str-buffer*)

Works similarly to [write-file](#), but the content in *str-buffer* is appended if the file in *str-filename* exists. If the file does not exist, it is created (in this case, `append-file` works identically to [write-file](#)). This function returns the number of bytes written.

example:

```
(write-file "myfile.txt" "ABC")
(append-file "myfile.txt" "DEF")

(read-file "myfile.txt") ⇒ "ABCDEF"
```

`append-file` can take a `http://` or `file://` URL in *str-file-name*. In this case `append-file` works exactly like [put-url](#) with `"Pragma: append\r\n"` in the header option and can

take the same additional parameters. The "Pragma: append\r\n" option is supplied automatically.

example:

```
(append-file "http://asite.com/message.txt" "More message text.")
```

The file `message.txt` is appended at a remote location `http://asite.com` with the contents of *str-buffer*. If the file does not yet exist, it will be created. In this mode, `append-file` can also be used to transfer files to remote newLISP server nodes.

See also [read-file](#) and [write-file](#).

apply

syntax: (apply *func list* [*int-reduce*])

Applies the contents of *func* (primitive, user-defined function, or lambda expression) to the arguments in *list*.

example:

```
(apply + '(1 2 3 4))           ⇒ 10
(set 'aList '(3 4 5))          ⇒ (3 4 5)
(apply * aList)                 ⇒ 60
(apply sqrt '(25))              ⇒ 5
(apply (lambda (x y) (* x y)) '(3 4)) ⇒ 12
```

The *int-reduce* parameter can optionally contain the number of arguments taken by the function in *func*. In this case, *func* will be repeatedly applied using the previous result as the first argument and taking the other arguments required successively from *list* (in left-associative order). For example, if `op` takes two arguments, then:

```
(apply op '(1 2 3 4 5) 2)

;; is equivalent to

(op (op (op (op 1 2) 3) 4) 5)

;; find the greatest common divisor
;; of two or more integers

(define (gcd_ a b)
  (let (r (% b a))
    (if (= r 0) a (gcd_ r a))))

(define-macro (gcd)
  (apply gcd_ (args) 2))

(gcd 12 18 6)    ⇒ 6
(gcd 12 18 6 4)  ⇒ 2
```

The last example shows how `apply`'s *reduce* functionality can be used to convert a two-argument function into one that takes multiple arguments.

`apply` should only be used on functions and operators that evaluate all of their arguments, not on *special forms* like [setq](#) or [case](#), which evaluate only some of their arguments. Doing so will cause the function to fail.

args

syntax: (args)

syntax: (args *int-idx-1* [*int-idx-2* ...])

Accesses a list of all unbound arguments passed to the currently evaluating [define](#), [define-macro](#) lambda, or lambda-macro expression. Only the arguments of the current function or macro that remain after local variable binding has occurred are available. The `args` function is useful for defining functions or macros with a variable number of parameters.

`args` can be used to define hygienic macros that avoid the danger of variable capture. See [define-macro](#).

example:

```
(define-macro (print-line)
  (dolist (x (args))
    (print x "\n")))

(print-line "hello" "World")
```

This example prints a line feed after each argument. The macro mimics the effect of the built-in function [println](#).

In the second syntax, `args` can take one or more indices (*int-idx-n*).

example:

```
(define-macro (foo)
  (print (args 2) (args 1) (args 0)))

(foo x y z)
zyx

(define (bar)
  (args 0 2 -1))

(bar '(1 2 (3 4))) ⇒ 4
```

The function `foo` prints out the arguments in reverse order. The `bar` function shows `args` being used with multiple indices to access nested lists.

Remember that `(args)` only contains the arguments not already bound to local variables of the current function or macro:

example:

```
(define (foo a b) (args))

(foo 1 2) ⇒ ()
```

```
(foo 1 2 3 4 5) ⇒ (3 4 5)
```

In the first example, an empty list is returned because the arguments are bound to the two local symbols, `a` and `b`. The second example demonstrates that, after the first two arguments are bound (as in the first example), three arguments remain and are then returned by `args`.

`(args)` can be used as an argument to a built-in or user-defined function call, but it should not be used as an argument to another macro, in which case `(args)` would not be evaluated and would therefore have the wrong contents in the new macro environment.

array

syntax: `(array int-n [int-n2 ... int-n16] [list-init])`

Creates an array with *int-n* elements, optionally initializing it with the contents of *list-init*. Up to sixteen dimensions may be specified for multidimensional arrays.

Internally, newLISP builds multidimensional arrays by using arrays as the elements of an array. newLISP arrays should be used whenever random indexing into a large list becomes too slow. Only a subset of the list functions may be used on arrays. For a more detailed discussion, see the chapter on [arrays](#).

example:

```
(array 5) ⇒ (nil nil nil nil nil)
(array 5 (sequence 1 5)) ⇒ (1 2 3 4 5)
(array 10 '(1 2)) ⇒ (1 2 1 2 1 2 1 2 1 2)
```

Arrays can be initialized with objects of any type. If fewer initializers than elements are provided, the list is repeated until all elements of the array are initialized.

```
(set 'myarray (array 3 4 (sequence 1 12)))
⇒ ((1 2 3 4) (5 6 7 8) (9 10 11 12))
```

Arrays are modified and accessed using the same list functions:

```
(set-nth 2 3 myarray 99) ; old syntax
(set-nth (myarray 2 3) 99) ; new preferred syntax
⇒ ((1 2 3 4) (5 6 7 8) (9 10 11 99))

(nth-set (myarray 1 1) "hello") ⇒ 6

myarray
⇒ ((1 2 3 4) (5 "hello" 7 8) (9 10 11 99))

(set-nth (myarray 1) (array 4 '(a b c d)))
⇒ ((1 2 3 4) (a b c d) (9 10 11 99))

(nth 1 myarray) ⇒ (a b c d) ; access a whole row
(nth 0 -1 myarray) ⇒ 4
```

```
;; use implicit indexing and slicing on arrays
(myarray 1)      ⇒ (a b c d)
(myarray 0 -1)   ⇒ 4
(2 myarray)      ⇒ (c d)
(-3 2 myarray)  ⇒ (b c)
```

Care must be taken to use an array when replacing a whole row.

[array-list](#) can be used to convert arrays back into lists:

```
(array-list myarray) ⇒ ((1 2 3 4) (a b c d) (1 2 3 99))
```

To convert a list back into an array, apply [flat](#) to the list:

```
(set 'aList '((1 2) (3 4)))          ⇒ ((1 2) (3 4))
(set 'aArray (array 2 2 (flat aList))) ⇒ ((1 2) (3 4))
```

The [array?](#) function can be used to check if an expression is an array:

```
(array? myarray)          ⇒ true
(array? (array-list myarray)) ⇒ nil
```

When serializing arrays using the function [source](#) or [save](#), the code includes the array statement necessary to create them. This way, variables containing arrays are correctly serialized when saving with [save](#) or creating source strings using [source](#).

```
(set 'myarray (array 3 4 (sequence 1 12)))

(save "array.lsp" 'myarray)

;; contents of file array.lsp ;;

(set 'myarray (array 3 4 (flat '(
  (1 2 3 4)
  (5 6 7 8)
  (9 10 11 12)))))
```

array-list

syntax: (array-list *array*)

Returns a list conversion from *array*, leaving the original array unchanged:

example:

```
(set 'myarray (array 3 4 (sequence 1 12)))
⇒ ((1 2 3 4) (5 6 7 8) (9 10 11 12))
```

```
(set 'mylist (array-list myarray))
⇒ ((1 2 3 4) (5 6 7 8) (9 10 11 12))

(list (array? myarray) (list? mylist))
⇒ (true true)
```

array?

syntax: (array? *expr*)

Checks if *expr* is an array:

example:

```
(set 'M (array 3 4 (sequence 1 4)))
⇒ ((1 2 3 4) (1 2 3 4) (1 2 3 4))

(array? M) ⇒ true
(array? (array-list M)) ⇒ nil
```

asin

syntax: (asin *num-radians*)

The arcsine function is calculated from the number in *num-radians*, and the result is returned.

example:

```
(asin 1) ⇒ 1.570796327
```

assoc

syntax: (assoc *exp-key list-alist*)

The value of *exp-key* is used to search *list-alist* for a *member-list* whose first element matches the key value. If found, the *member-list* is returned; otherwise, the result will be nil.

example:

```
(assoc 1 '((3 4) (1 2))) ⇒ (1 2)

(set 'data '((apples 123) (bananas 123 45) (pears 7)))
```

```
(assoc 'bananas data) ⇒ (bananas 123 45)
(assoc 'oranges data) ⇒ nil
```

For making replacements in association lists, use the [replace-assoc](#) function. The [lookup](#) function is used to perform association lookup and element extraction in one step.

atan

syntax: (atan *num-radians*)

The arctangent of *num-radians* is calculated and returned.

example:

```
(atan 1) ⇒ 0.7853981634
```

atan2

syntax: (atan2 *num-Y-radians num-X-radians*)

The `atan2` function computes the principal value of the arctangent of Y / X in radians. It uses the signs of both arguments to determine the quadrant of the return value. `atan2` is useful for converting Cartesian coordinates into polar coordinates.

example:

```
(atan2 1 1) ⇒ 0.7853981634
(div (acos 0) (atan2 1 1)) ⇒ 2
(atan2 0 -1) ⇒ 3.141592654
(= (atan2 1 2) (atan (div 1 2))) ⇒ true
```

atom?

syntax: (atom? *exp*)

Returns `true` if the value of *exp* is an atom, otherwise `nil`. An expression is an atom, if it evaluates to `nil`, `true`, an integer, a float, a string, a symbol or a primitive. Lists, lambda or lambda-macro expressions, and quoted expressions are not atoms.

example:

```
(atom? '(1 2 3)) ⇒ nil
(and (atom? 123)
  (atom? "hello")
  (atom? 'foo)) ⇒ true
```



```
(atom? 'foo)      ⇒ nil
```

base64-dec

syntax: (base64-dec *str*)

The BASE64 string in *str* is decoded. Note that *str* is not verified to be a valid BASE64 string. The decoded string is returned.

example:

```
(base64-dec "SGVsbG8gV29ybGQ=") ⇒ "Hello World"
```

For encoding, use the [base64-enc](#) function.

newLISP's BASE64 handling is derived from routines found in the UNIX [curl](#) utility.

base64-enc

syntax: (base64-enc *str*)

The string in *str* is encoded into BASE64 format. This format encodes groups of $3 * 8 = 24$ input bits into $4 * 8 = 32$ output bits, where each 8-bit output group represents 6 bits from the input string. The 6 bits are encoded into 64 possibilities from the letters A–Z and a–z; the numbers 0–9; and the characters + (plus sign) and / (slash). The = (equals sign) is used as a filler in unused 3- to 4-byte translations. This function is helpful for converting binary content into printable characters.

The encoded string is returned.

BASE64 encoding is used with many Internet protocols to encode binary data for inclusion in text-based messages (e.g., XML-RPC).

example:

```
(base64-enc "Hello World") ⇒ "SGVsbG8gV29ybGQ="
```

Note that `base64-enc` does not insert carriage-return/line-feed pairs in longer BASE64 sequences but instead returns a pure BASE64-encoded string.

For decoding, use the [base64-dec](#) function.

newLISP's BASE64 handling is derived from routines found in the UNIX [curl](#) utility.

bayes-query

syntax: (bayes-query list-L context-D [bool-chain] [bool-probs])

Takes a list of tokens (*list-L*) and a trained dictionary (*context-D*) and returns a list of the combined probabilities of the tokens in one category (*A* or *Mc*) versus a category (*B*) against all other categories (*Mi*). All tokens in *list-L* should occur in *context-D*. When using the default *R.A. Fisher Chi²* mode, nonexistent tokens will skew results toward equal probability in all categories.

Non-existing tokens will not have any influence on the result when using the true *Chain Bayesian* mode with *bool-chain* set to `true`. The optional last flag, *bool-probs*, indicates whether frequencies or probability values are used in the data set. The [bayes-train](#) function is typically used to generate a data set's frequencies.

Tokens can be strings or symbols. If strings are used, they are prepended with an underscore before being looked up in *context-D*. If [bayes-train](#) was used to generate *context-D*'s frequencies, the underscore was automatically prepended during the learning process.

Depending on the flag specified in *bool-probs*, [bayes-query](#) employs either the R. A. Fisher Chi² method of compounding probabilities or the Chain Bayesian method. By default, when no flag or `nil` is specified in *bool-probs*, the Chi² method of compounding probabilities is used. When specifying `true` in *bool-probs*, the Chain Bayesian method is used.

If the R.A. Fisher Chi² method is used, the total number of tokens in the different training set's categories should be equal or similar. Uneven frequencies in categories will skew the results.

For two categories *A* and *B*, [bayes-query](#) uses the following formula:

$$p(A|tkn) = p(tkn|A) * p(A) / p(tkn|A) * p(A) + p(tkn|B) * p(B)$$

For *N* categories, this formula is used:

$$p(Mc|tkn) = p(tkn|Mc) * p(Mc) / \text{sum-i-N}(p(tkn|Mi) * p(Mi))$$

The probabilities (*p(Mi)* or *p(A)*, along with *p(B)*) represent the *Bayesian prior probabilities*. *p(Mx|tkn)* and *p(A|tkn)* are the *posterior Bayesian* probabilities of a category or model.

Priors are handled differently, depending on whether the R.A. Fisher Chi² or the Chain Bayesian method is used. While in Chain Bayesian mode, posteriors from one token calculation get the priors in the next calculation. In the default R.A. Fisher method, priors are not passed on via chaining, but probabilities are compounded using the Chi² method.

In Chain Bayes mode, tokens with zero frequency in one category will effectively put the probability of that category to 0 (zero). This also causes all posterior priors to be set to 0 and the category to be completely suppressed in the result. Queries resulting in zero probabilities for all categories yield *NaN* values.

The default R.A. Fisher Chi² method is less sensitive about zero frequencies and still maintains a low probability for that token. This may be an important feature in natural language processing when using *Bayesian statistics*. Imagine that five different language *corpus* categories have been trained, but some words occurring in one category are not present in another. When the pure Chain Bayesian method is used, a sentence could never be

classified into its correct category because the zero-count of just one word token could effectively exclude it from the category to which it belongs.

On the other hand, the Chain Bayesian method offers exact results for specific proportions in the data. When using Chain Bayesian mode for natural language data, all zero frequencies should be removed from the trained dictionary first.

The return value of `bayes-query` is a list of probability values, one for each category. Following are two examples: the first for the default R.A. Fisher mode, the second for a data set processed with the Chain Bayesian method.

R.A. Fisher Chi² method

In the following example, the two data sets are books from Project Gutenberg. We assume that different authors use certain words with different frequencies and want to determine if a sentence is more likely to occur in one or the other author's writing. A similar method is frequently used to differentiate between spam and legitimate email.

```
;; from Project Gutenberg: http://www.gutenberg.org/catalog/
;; The Adventures of Sherlock Holmes - Sir Arthur Conan Doyle

(bayes-train (parse (lower-case (read-file "Doyle.txt")))
             "[^a-z]+" 0) '() 'DoyleDowson)

;; A Comedy of Masks - Ernest Dowson and Arthur Moore

(bayes-train '() (parse (lower-case (read-file "Dowson.txt")))
             "[^a-z]+" 0) 'DoyleDowson)

(save "DoyleDowson.lsp" 'DoyleDowson)
```

The two training sets are loaded, split into tokens, and processed by the [bayes-train](#) function. In the end, the DoyleDowson dictionary is saved to a file, which will be used later with the `bayes-query` function.

The following code illustrates how `bayes-query` is used to classify a sentence as *Doyle* or *Dowson*:

```
(load "DoyleDowson.lsp")
(bayes-query (parse "he was putting the last touches to a
picture")
             'DoyleDowson)
⇒ (0.03801673331 0.9619832667)

(bayes-query (parse "immense faculties and extraordinary powers
of observation")
             'DoyleDowson)
⇒ (0.9851075608 0.01489243923)
```

The queries correctly identify the first sentence as a *Dowson* sentence, and the second one as a *Doyle* sentence.

Chain Bayesian method

The second example is frequently found in introductory literature on Bayesian statistics. It shows the Chain Bayesian method of using `bayes-query` on the data of a previously processed data set:

example:

```
(set 'Data:test-positive '(8 18))
(set 'Data:test-negative '(2 72))
(set 'Data:total '(10 90))
```

A disease occurs in 10 percent of the population. A blood test developed to detect this disease produces a false positive rate of 20 percent in the healthy population and a false negative rate of 20 percent in the sick. What is the probability of a person carrying the disease after testing positive?

example:

```
(bayes-query '(test-positive) Data true)
⇒ (0.3076923077 0.6923076923)

(bayes-query '(test-positive test-positive) Data true)
⇒ (0.64 0.36)

(bayes-query '(test-positive test-positive test-positive) Data
true)
⇒ (0.8767123288 0.1232876712)
```

Note that the Bayesian formulas used assume statistical independence of events for the `bayes-query` to work correctly.

The example shows that a person must test positive several times before they can be confidently classified as sick.

Calculating the same example using the R.A. Fisher χ^2 method will give less-distinguished results.

Specifying probabilities instead of counts

Often, data is already available as probability values and would require additional work to reverse them into frequencies. In the last example, the data were originally defined as percentages. The additional optional *bool-probs* flag allows probabilities to be entered directly and should be used together with the Chain Bayesian mode for maximum performance:

example:

```
(set 'Data:test-positive '(0.8 0.2))
(set 'Data:test-negative '(0.2 0.8))
(set 'Data:total '(0.1 0.9))

(bayes-query '(test-positive) Data true true)
⇒ (0.3076923077 0.6923076923)
```

```
(bayes-query '(test-positive test-positive) Data true true)
⇒ (0.64 0.36)

(bayes-query '(test-positive test-positive test-positive) Data
true true)
⇒ (0.8767123288 0.1232876712)
```

As expected, the results are the same for probabilities as they are for frequencies.

bayes-train

syntax: (bayes-train *list-M1 list-M2 [list-M3 ...] sym-context-D*)

Takes two or more lists of tokens (*M1*, *M2*—) from a joint set of tokens. In newLISP, tokens can be symbols or strings (other data types are ignored). Tokens are placed in a common dictionary in *sym-context-D*, and the frequency is counted for each token in each category *Mi*. If the context does not yet exist, it must be quoted.

The *M* categories represent data models for which sequences of tokens can be classified (see [bayes-query](#)). Each token in *D* is a content-addressable symbol containing a list of the frequencies for this token within each category. String tokens are prepended with an `_` (underscore) before being converted into symbols. A symbol named `total` is created containing the total of each category. The `total` symbol cannot be part of the symbols passed as an *Mi* category.

The function returns a list of token frequencies found in the different categories or models.

example:

```
(bayes-train '(A A B C C) '(A B B C C C) 'L) ⇒ (5 6)

L:A      ⇒ (2 1)
L:B      ⇒ (1 2)
L:C      ⇒ (2 3)
L:total  ⇒ (5 6)

(bayes-train '("one" "two" "two" "three")
              '("three" "one" "three")
              '("one" "two" "three") 'S)
⇒ (3 2 3)

S:_one   ⇒ (1 1 1)
S:_two   ⇒ (2 0 1)
S:_three ⇒ (1 2 1)
S:total  ⇒ (3 2 3)
```

The first example shows training with two lists of symbols. The second example illustrates how an `_` is prepended when training with strings.

Note that these examples are just for demonstration purposes. In reality, training sets may contain thousands or millions of words, especially when training natural language models. But small data sets may be used when then the frequency of symbols just describe already-known proportions. In this case, it may be better to describe the model data set explicitly, without the `bayes-train` function:

```
(set 'Data:tested-positive '(8 18))
(set 'Data:tested-negative '(2 72))
(set 'Data:total '(10 90))
```

The last data are from a popular example used to describe the [bayes-query](#) function in introductory papers and books about *bayesian networks*.

Training can be done in different stages by using `bayes-train` on an existing trained context with the same number of categories. The new symbols will be added, then counts and totals will be correctly updated.

Training in multiple batches may be necessary on big text corpora or documents that must be tokenized first. These corpora can be tokenized in small portions, then fed into `bayes-train` in multiple stages. Categories can also be singularly trained by specifying an empty list for the absent corpus:

```
(bayes-train shakespeare1 '() 'data)
(bayes-train shakespeare2 '() 'data)
(bayes-train '() hemingway1 'data)
(bayes-train '() hemingway2 'data)
(bayes-train shakespeare-rest hemingway-rest 'data)
```

`bayes-train` will correctly update word counts and totals.

Using `bayes-train` inside a context other than `MAIN` requires the training contexts to have been created previously within the `MAIN` context via the [context](#) function.

`bayes-train` is not only useful with the [bayes-query](#) function, but also as a function for counting in general. For instance, the resulting frequencies could be analyzed using [prob-chi2](#) against a *null hypothesis* of proportional distribution of items across categories.

begin

syntax: (begin *body*)

The `begin` function is used to group a block of expressions. The expressions in *body* are evaluated in sequence, and the value of the last expression in *body* is returned.

example:

```
(begin
  (print "This is a block of 2 expressions\n")
  (print "====="))
```

Some built-in functions like [cond](#), [define](#), [dolist](#), [dotimes](#), and [while](#) already allow multiple expressions in their bodies, but `begin` is often used in an [if](#) expression.

The [silent](#) function works like `begin`, but suppresses console output on return.

beta

syntax: (beta *cum-a*, *num-b*)

The *Beta* function, `beta`, is derived from the *log Gamma* `gamma1n` function as follows:

$$\text{beta} = \exp(\text{gamma1n}(a) + \text{gamma1n}(b) - \text{gamma1n}(a + b))$$

example:

```
(beta 1 2) ⇒ 0.5
```

betai

syntax: (betai *num-x*, *num-a*, *num-b*)

The *Incomplete Beta* function, `betai`, equals the cumulative probability of the *Beta* distribution, `betai`, at *x* in *num-x*. The cumulative binomial distribution is defined as the probability of an event, *pev*, with probability *p* to occur *k* or more times in *N* trials:

$$\text{pev} = \text{Betai}(p, k, N - k + 1)$$

example:

```
(betai 0.5 3 8) ⇒ 0.9453125

;; probability of F ratio for df1/df2
;;
(define (f-prob f df1 df2)
  (let (prob (mul 2 (betai (div df2 (add df2 (mul df1 f)))
                           (mul 0.5 df2)
                           (mul 0.5 df1)))))
    (div (if (> prob 1) (sub 2 prob) prob) 2)))
```

The first example calculates the probability for an event, with a probability of 0.5 to occur 3 or more times in 10 trials ($8 = 10 - 3 + 1$). The incomplete Beta distribution can be used to derive a variety of other functions in mathematics and statistics. The second example calculates the one-tailed probability of a variance, *F ratio*. In similar fashion, *students t* could be calculated using `betai`. See also the [binomial](#) function.

binomial

syntax: (binomial *int-n* *int-k* *float-p*)

The binomial distribution function is defined as the probability for an event to occur *int-k* times in *int-n* trials if that event has a probability of *float-p* and all trials are independent of one another:

$$\text{binomial} = n! / (k! * (n - k)!) * \text{pow}(p, k) * \text{pow}(1.0 - p, n - k)$$

where $x!$ is the factorial of x and $\text{pow}(x, y)$ is x raised to the power of y .

example:

```
(binomial 10 3 0.5) ⇒ 0.1171875
```

The example calculates the probability for an event with a probability of 0.5 to occur 3 times in 10 trials. For a cumulated distribution, see the [betai](#) function.

case

syntax: (case exp-switch (*exp-1 body-1*) [(*exp-2 body-2*) ...])

The result of evaluating *exp-switch* is compared to each of the *unevaluated* expressions *exp-1*, *exp-2*, —. If a match is found, the corresponding expressions in *body* are evaluated. The result of the last match is returned as the result for the entire *case* expression.

example:

```
(define (translate n)
  (case n
    (1 "one")
    (2 "two")
    (3 "three")
    (4 "four")
    (true "Can't translate this")))
(translate 3) ⇒ "three"
(translate 10) ⇒ "Can't translate this"
```

The example shows how, if no match is found, the last expression in the body of a *case* function can be evaluated.

catch

syntax: (catch *exp*)

syntax: (catch *exp symbol*)

In the first syntax, *catch* will return the result of the evaluation of *exp* or the evaluated argument of a [throw](#) executed during the evaluation of *exp*:

example:

```
(catch (dotimes (x 1000)
  (if (= x 500) (throw x)))) ⇒ 500
```


This form is useful for breaking out of iteration loops and for forcing an early return from a function or expression block:

```
(define (foo x)
  ...
  (if condition (throw 123))
  ...
  456)

;; if condition is true

(catch (foo p)) ⇒ 123

;; if condition is not true

(catch (foo p)) ⇒ 456
```

In the second syntax, `catch` evaluates the expression *exp*, stores the result in *symbol*, and returns `true`. If an error occurs during evaluation, `catch` returns `nil` and stores the error message in *symbol*. This form can be useful when errors are expected as a normal potential outcome of a function and are dealt with during program execution.

example:

```
(catch (func 3 4) 'result) ⇒ nil
result
⇒ "invalid function in function catch : (func 3 4)"

(constant 'func +) ⇒ add <4068A6>
(catch (func 3 4) 'result) ⇒ true
result ⇒ 7
```

When a [throw](#) is executed during the evaluation of *expr*, `catch` will return `true`, and the throw argument will be stored in *symbol*:

```
(catch (dotimes (x 100)
  (if (= x 50) (throw "fin"))) 'result) ⇒ true

result ⇒ "fin"
```

As well as being used for early returns from functions and for breaking out of iteration loops (as in the first syntax), the second syntax of `catch` can also be used to catch errors. The [throw-error](#) function may be used to throw user-defined errors.

ceil

syntax: (ceil *number*)

Returns the next highest integer above *number* as a floating point.

example:

```
(ceil -1.5) ⇒ -1
(ceil 3.4) ⇒ 4
```

See also the [floor](#) function.

change-dir

syntax: (change-dir *str-path*)

Changes the current directory to be the one given in *str-path*. If successful, `true` is returned; otherwise `nil` is returned.

example:

```
(change-dir "/etc")
```

Makes `/etc` the current directory.

char

syntax: (char *str* [*int-index*])

syntax: (char *int*)

Given a string argument, extracts the character at *int-index* from *str*, returning the ASCII value of that character. If *int-index* is omitted, 0 (zero) is assumed.

See [Indexing elements of strings and lists](#).

Given an integer argument, `char` returns a string containing the ASCII character with value *int*.

On UTF-8-enabled versions of newLISP, the value in *int* is taken as Unicode and a UTF-8 character is returned.

example:

```
(char "ABC")      ⇒ 65 ; ASCII code for "A"
(char "ABC" 1)    ⇒ 66 ; ASCII code for "B"
(char "ABC" -1)   ⇒ 67 ; ASCII code for "C"
(char "B")        ⇒ 66 ; ASCII code for "B"

(char 65) ⇒ "A"
(char 66) ⇒ "B"

(char (char 65)) ⇒ 65 ; two inverse applications
(map char (sequence 1 255)) ; returns current character set
```

chop

syntax: (chop *str* [*int-chars*])

syntax: (chop *list* [*int-elements*])

If the first argument evaluates to a string, chop returns a copy of *str* with the last *int-char* characters omitted. If the *int-char* argument is absent, one character is omitted. chop does not alter *str*.

If the first argument evaluates to a list, a copy of *list* is returned with *int-elements* omitted (same as for strings).

example:

```
(set 'str "newLISP") ⇒ "newLISP"

(chop str)          ⇒ "newLIS"
(chop str 2)        ⇒ "newLI"

str ⇒ "newLISP"

(set 'lst '(a b (c d) e))

(chop lst)          ⇒ (a b (c d))
(chop lst 2)        ⇒ (a b)

lst ⇒ (a b (c d) e)
```

clean

syntax: (clean *exp-predicate list*)

The predicate *exp-predicate* is applied to each element of *list*. In the returned list, all elements for which *exp-predicate* is true are eliminated.

clean works like [filter](#) with a negated predicate.

example:

```
(clean symbol? '(1 2 d 4 f g 5 h)) ⇒ (1 2 4 5)

(filter symbol? '(1 2 d 4 f g 5 h)) ⇒ (d f g h)

(define (big? x) (> x 5))           ⇒ (lambda (x) (> x 5))

(clean big? '(1 10 3 6 4 5 11)) ⇒ (1 3 4 5)

(clean (fn (x) (> x 5)) '(1 10 3 6 4 5 11)) ⇒ (1 3 4 5)
```

The predicate may be a built-in predicate or a user-defined function or lambda expression.

For cleaning numbers from one list using numbers from another, use [difference](#) or [intersect](#) (with the *list* option).

See also the related function [index](#), which returns the indices of the remaining elements, and [filter](#), which returns all elements for which a predicate returns true.

close

syntax: (close *int-file*)

Closes the file specified by the file handle in *int-file*. The handle would have been obtained from a previous [open](#) operation. If successful, `close` returns `true`; otherwise `nil` is returned.

example:

```
(close (device))  ⇒ true
(close 7)         ⇒ true
(close aHandle)   ⇒ true
```

Note that using `close` on [device](#) automatically resets it to 0 (zero, the screen device).

command-line

syntax: (command-line [*bool*])

Enables or disables the console's interactive command-line mode. The command line is switched off if *bool* evaluates to `nil`, and on for anything else. The command line is also switched on if reset or an error condition occurs.

example:

```
(command-line nil)
```

On Linux/UNIX, this will also disable the Ctrl-C handler.

cond

syntax: (cond (*exp-condition-1 body-1*) [(*exp-condition-2 body-2*) ...]

Like `if`, `cond` conditionally evaluates the expressions within its body. The *exp-conditions* are evaluated in turn, until some *exp-condition-i* is found that evaluates to anything other than `nil` or an empty list `()`. The result of evaluating *body-i* is then returned as the result of the entire *cond-expression*. If all conditions evaluate to `nil` or an empty list, `cond` returns the value of the last *cond-expression*.

example:

```

(define (classify x)
  (cond
    ((< x 0) "negative")
    ((< x 10) "small")
    ((< x 20) "medium")
    ((>= x 30) "big")))

(classify 15)    ⇒ "medium"
(classify 22)    ⇒ "nil"
(classify 100)   ⇒ "big"
(classify -10)   ⇒ "negative"

```

When a *body-n* is missing, the value of the last *cond-expression* evaluated is returned. If no condition evaluates to `true`, the value of the last conditional expression is returned (i.e., `nil` or an empty list).

```
(cond ((+ 3 4))) ⇒ 7
```

When used with multiple arguments, the function [if](#) behaves like `cond`, except it does not need extra parentheses to enclose the condition-body pair of expressions.

cons

syntax: (cons *exp-1* *exp-2*)

If *exp-2* evaluates to a list, then a list is returned with the result of evaluating *exp-1* inserted as the first element. If *exp-2* evaluates to anything other than a list, the results of evaluating *exp-1* and *exp-2* are returned in a list. Note that there is no *dotted pair* in newLISP: *consing* two atoms constructs a list, not a dotted pair.

example:

```

(cons 'a 'b)           ⇒ (a b)
(cons 'a '(b c))       ⇒ (a b c)
(cons (+ 3 4) (* 5 5)) ⇒ (7 25)
(cons '(1 2) '(3 4))   ⇒ ((1 2) 3 4)
(cons nil 1)           ⇒ (nil 1)
(cons 1 nil)           ⇒ (1 nil)
(cons 1)               ⇒ (1)
(cons)                ⇒ ()

```

Unlike other LISPs that return (`s`) as the result of the expression (`cons 's nil`), newLISP's `cons` returns (`s nil`). In newLISP, `nil` is a boolean value and is not equivalent to an empty list, and a LISP cell holds only one value.

`cons` behaves like the inverse operation of [first](#) and [rest](#) (or [first](#) and [last](#) if the list is a pair):

```

(cons (first '(a b c)) (rest '(a b c))) ⇒ (a b c)
(cons (first '(x y)) (last '(x y)))     ⇒ (x y)

```

constant

syntax: (constant *sym-1 exp-1* [*sym-2 exp-2 ...*])

Identical to [set](#) in functionality, `constant` further protects the symbols from subsequent modification. A symbol set with `constant` can only be modified using the `constant` function again. When an attempt is made to modify the contents of a symbol protected with `constant`, newLISP generates an error message. Only symbols from the current context can be used with `constant`. This prevents the overwriting of symbols that have been protected in their home context.

Symbols initialized with [set](#), [define](#), or [define-macro](#) can still be protected by using the `constant` function:

```
(constant 'aVar 123) ⇒ 123
(set 'aVar 999)
error: symbol is protected in function set: aVar

(define (double x) (+ x x))

(constant 'double)

;; equivalent to

(constant 'double (fn (x) (+ x x)))
```

The first example defines a constant, `aVar`, which can only be changed by using another `constant` statement. The second example protects `double` from being changed (except by `constant`). Because a function definition in newLISP is equivalent to an assignment of a lambda function, both steps can be collapsed into one, as shown in the last statement line. This could be an important technique for avoiding protection errors when a file is loaded multiple times.

The last value to be assigned can be omitted. `constant` returns the contents of the last symbol set and protected.

Built-in functions can be assigned to symbols or to the names of other built-in functions, effectively redefining them as different functions. There is no performance loss when renaming functions.

```
(constant 'sqrteroot sqrt) ⇒ sqrt <406C2E>
(constant '+ add)         ⇒ add <4068A6>
```

`sqrteroot` will behave like `sqrt`. The `+` (plus sign) is redefined to use the mixed type floating point mode of `add`. The hexadecimal number displayed in the result is the binary address of the built-in function and varies on different platforms and OSes.

context

syntax: (context [*sym-context*])

syntax: (context *sym-context str exp-value*)

syntax: (context *sym-context str*)

In the first syntax, context is used to switch to a different context namespace. Subsequent [loads](#) of newLISP source or functions like [eval-string](#) will put newly created symbols and function definitions in the new context.

If the context still needs to be created, the symbol for the new context should be specified. When no argument is passed to context, then the symbol for the current context is returned.

Because contexts evaluate to themselves, a quote is not necessary to switch to a different context if that context already exists.

example:

```
(context 'GRAPH)           ; create / switch context GRAPH

(define (foo-draw x y z)   ; function resides in GRAPH
  (...))

(set 'var 12345)
(symbols) ⇒ (foo-draw var) ; GRAPH has now two symbols

(context MAIN)             ; switch back to MAIN (quote not
required)

(print GRAPH:var) ⇒ 12345   ; contents of symbol in GRAPH

(GRAPH:foo-draw 10 20 30)   ; execute function in GRAPH
(set 'GRAPH:var 6789)       ; assign to a symbol in GRAPH
```

If a context symbol is referred to before the context exists, the context will be created implicitly.

```
(set 'person:age 0)        ; no need to create context first
(set 'person:address "")   ; useful for quickly defining data
structures
```

Contexts can be copied:

```
(new person 'JohnDoe) ⇒ JohnDoe

(set 'JohnDoe:age 99)
```

Contexts can be referred to by a variable:

```
(set 'human JohnDoe)

human:age ⇒ 99

(set 'human:address "1 Main Street")

JohnDoe:address ⇒ "1 Main Street"
```

An evaluated context (no quote) can be given as an argument:

```
> (context 'FOO)
FOO
FOO> (context MAIN)
MAIN
> (set 'old FOO)
FOO
> (context 'BAR)
BAR
BAR> (context MAIN:old)
FOO
FOO>
```

If an identifier with the same symbol already exists, it is redefined to be a context.

Symbols within the current context are referred to simply by their names, as are built-in functions and special symbols like `nil` and `true`. Symbols outside the current context are referenced by prefixing the symbol name with the context name and a `:` (colon). To quote a symbol in a different context, prefix the context name with a `'` (single quote).

Within a given context, symbols may be created with the same name as built-in functions or context symbols in MAIN. This overwrites the symbols in MAIN when they are prefixed with a context:

```
(context 'CTX)
(define (CTX:new var)
  (...))

(context 'MAIN)
```

`CTX:new` will overwrite `new` in MAIN.

In the second syntax, `context` is used to create *dictionaries* for *hash*-like associative memory access:

```
;; create a symbol and store data in it
(context 'MyHash "John Doe" 123)      ⇒ 123
(context 'MyHash "@#$%^" "hello world") ⇒ "hello world"

;; retrieve contents from symbol
(context 'MyHash "john Doe")           ⇒ 123
(context 'MyHash "@#$%^")              ⇒ "hello world"
```

The first two statements create a symbol and store a value of any data type inside. The first statement also creates the hash context named `MyHash`.

Hash symbols can contain spaces or any other special characters not typically allowed in newLISP symbols being used as variable names. This second syntax of `context` only creates the new symbol and returns the value contained in it. It does not switch to the new namespace.

The following function definition can be used as a comfortable shorter method to handle dictionaries:

```
(define (myhash:myhash key value)
  (if value
    (context 'myhash key value)
    (context 'myhash key)))
```



```
(myhash "hello" 123)
(myhash "hello") ⇒ 123
```

Note that `context` cannot be used to modify the [default functor](#). This protects a function like the previous against modifying itself.

context?

syntax: (context? *exp*)

syntax: (context? *exp str-sym*)

In the first syntax, *context?* is a predicate that returns `true` only if *exp* evaluates to a context; otherwise, it returns `nil`.

example:

```
(context? MAIN) ⇒ true
(set 'x 123)
(context? x) ⇒ nil

(set 'FOO:q "hola") ⇒ "hola"
(set 'ctx FOO)
(context? ctx) ⇒ true ; ctx contains context foo
```

The second syntax checks for the existence of a symbol in a context. The symbol is specified by its name string in *str-sym*.

```
(context? FOO "q") ⇒ true
(context? FOO "p") ⇒ nil
```

Use [context](#) to change and create namespaces and to create hash symbols in contexts.

copy-file

syntax: (copy-file *str-from-name str-to-name*)

Copies a file from a path-file-name given in *str-from-name* to a path-file-name given in *str-to-name*. Returns `true` or `nil`, depending on if the copy was successful or not.

example:

```
(copy-file "/home/me/newlisp/data.lsp" "/tmp/data.lsp")
```

cos

syntax: (cos *num-radians*)

The cosine function is calculated from *num*, and the result is returned.

example:

```

(cos 1)           ⇒ 0.5403023059
(set 'pi (mul 2 (acos 0))) ⇒ 3.141592654
(cos pi)          ⇒ -1

```

count

syntax: (count *list-1 list-2*)

Counts elements of *list-1* in *list-2* and returns a list of those counts.

example:

```

(count '(1 2 3) '(3 2 1 4 2 3 1 1 2 2)) ⇒ (3 4 2)
(count '(z a) '(z d z b a z y a))      ⇒ (3 2)

(set 'lst (explode (read-file "myFile.txt")))
(set 'letter-counts (count (unique lst) lst))

```

The second example counts all occurrences of different letters in `myFile.txt`.

The first list in `count`, which specifies the items to be counted in the second list, should be unique. For items that are not unique, only the first instance will carry a count; all other instances will display 0 (zero).

cpymem

syntax: (cpymem *int-from-address int-to-address int-bytes*)

Copies *int-bytes* of memory from *int-from-address* to *int-to-address*. This function can be used for direct memory writing/reading or for hacking newLISP internals (e.g., type bits in LISP cells, or building functions with binary executable code on the fly).

Note that this function should only be used when familiar with newLISP internals. `cpymem` can crash the system or make it unstable if used incorrectly.

example:

```

(cpymem (pack "c c" 0 32) (last (dump 'sym)) 2)

(set 's "0123456789")

(cpymem "xxx" (+ (address s) 5) 3)

```

```
s ⇒ "01234xxx89")
```

The first example would remove the protection bit in symbol `sym`. The second example copies a string directly into a string variable.

The following example creates a new function from scratch, runs a piece of binary code, and adds up two numbers. This assembly language snippet shows the x86 code to add up two numbers and return the result:

```
55      push ebp
8B EC   mov  ebp, esp
8B 45 08 mov  eax, [ebp+08]
03 45 0C add  eax, [ebp+0c]
5D      pop  ebp
C3      ret
```

The binary representation is attached to a new function created in newLISP:

```
;; set code
(set 'bindata (pack "cccccccccc"
  0x55 0x8B 0xEC 0x8B 0x45 0x08 0x03 0x45 0x0C 0x5D 0xC3))

;; get function template
(set 'foo print)

;; change type to library import and OS calling conventions
;(cpymem (pack "ld" 265) (first (dump foo)) 4) ; Win32 stdcall
;(cpymem (pack "ld" 264) (first (dump foo)) 4) ; Linux cdecl

;; set code pointer
(cpymem (pack "ld" (address bindata)) (+ (first (dump foo)) 12)
4)

;; execute
(foo 3 4) ⇒ 7
```

Use the [dump](#) function to retrieve binary addresses and the contents from newLISP cells.

crc32

syntax: (crc32 *str-data*)

Calculates a running 32-bit CRC (Circular Redundancy Check) sum from the buffer in *str-data*, starting with a CRC of 0xffffffff for the first byte. `crc32` uses an algorithm published by www.w3.org.

example:

```
(crc32 "abcdefghijklmnopqrstuvwxy") ⇒ 1277644989
```

`crc32` is often used to verify data integrity in unsafe data transmissions.

crit-chi2

syntax: (**crit-chi2** *num-probability* *num-df*)

Calculates the critical minimum Chi² for a given confidence probability *num-probability* and the degrees of freedom *num-df* for testing the significance of a statistical null hypothesis.

example:

```
(crit-chi2 0.99 4) ⇒ 13.27670443
```

See also the inverse function [prob-chi2](#).

crit-z

syntax: (**crit-z** *num-probability*)

Calculates the critical normal distributed Z value of a given cumulated probability (*num-probability*) for testing of statistical significance and confidence intervals.

example:

```
(crit-z 0.999) ⇒ 3.090232372
```

See also the inverse function [prob-z](#).

current-line

syntax: (**current-line**)

Retrieves the contents of the last [read-line](#) operation. `current-line`'s contents are also implicitly used when [write-line](#) is called without a string parameter.

The following source shows the typical code pattern for creating a UNIX command-line filter:

example:

```
#!/usr/bin/newlisp

(set 'inFile (open (main-args 2) "read"))
(while (read-line inFile)
  (if (starts-with (current-line) ";;")
      (write-line)))
(exit)
```

The program is invoked:

```
./filter myfile.lsp
```

This displays all comment lines starting with `;;` from a file given as a command-line argument when invoking the script `filter`.

date

syntax: `(date)`

syntax: `(date int-secs [int-offset])`

syntax: `(date int-secs int-offset str-format)`

The first syntax returns the local time zone's current date and time as a string representation.

In the second syntax, `date` translates the number of seconds in `int-secs` into its date/time string representation for the local time zone. The number in `int-secs` is usually retrieved from the system using [date-value](#). Optionally, a time-zone offset (in minutes) can be specified in `int-offset`, which is added or subtracted before conversion of `int-sec` to a string.

example:

```
(date)                ⇒ "Fri Oct 29 09:56:58 2004"

(date (date-value))   ⇒ "Sat May 20 11:37:15 2006"
(date (date-value) 300) ⇒ "Sat May 20 16:37:19 2006" ; 5 hours
offset
(date 0)              ⇒ "Wed Dec 31 16:00:00 1969"
```

Note that on some Win32-compiled versions, values resulting in dates earlier than January 1, 1970, 00:00:00 return `nil`. But the MinGW compiled version will also work with values that result in dates up to 24 hours prior to 1/1/1970, returning a date-time string for 12/31/1969.

The way the date and time are presented in a string depends on the underlying operating system.

The second example would show 1-1-1970 0:0 when in the Greenwich time zone, but it displays a time lag of 8 hours when in Pacific Standard Time (PST). `date` assumes the `int-secs` given are in Coordinated Universal Time (UCT; formerly Greenwich Mean Time (GMT)) and converts it according to the local time-zone.

The third syntax makes the date string fully customizable by using a format specified in `str-format`. This allows the day and month names to be translated into results appropriate for the current locale:

example:

```
(set-locale "german") ⇒ "de_DE"

(date (date-value) 0 "%A %-d. %B %Y")
⇒ "Montag 7. März 2005"
; on Linux - suppresses the leading 0

(set-locale "C") ; default POSIX

(date (date-value) 0 "%A %B %d %Y")
⇒ "Monday March 07 2005"
```

```

(date (date-value) 0 "%a %#d %b %Y")
⇒ "Mon 7 Mar 2005"
; suppressing leading 0 on Win32 using #

(set-locale "german")

(date (date-value) 0 "%x")
⇒ "07.03.2005"      ; day month year

(set-locale "C")

(date (date-value) 0 "%x")
⇒ "03/07/05"      ; month day year

```

The following table summarizes all format specifiers available on both Win32 and Linux/UNIX platforms. More format options are available on Linux/UNIX. For details, consult the documentation for the C function `strftime()` in the individual platform's C library.

***for
mat* description**

%a	abbreviated weekday name according to the current locale
%A	full weekday name according to the current locale
%b	abbreviated month name according to the current locale
%B	full month name according to the current locale
%c	preferred date and time representation for the current locale
%d	day of the month as a decimal number (range 01–31)
%H	hour as a decimal number using a 24-hour clock (range 00–23)
%I	hour as a decimal number using a 12-hour clock (range 01–12)
%j	day of the year as a decimal number (range 001–366)
%m	month as a decimal number (range 01–12)
%M	minute as a decimal number
%p	either 'am' or 'pm' according to the given time value or the corresponding strings for the current locale
%S	second as a decimal number 0–61 (60 and 61 to account for occasional leap seconds)
%U	week number of the current year as a decimal number, starting with the first Sunday as the first day of the first week
%w	day of the week as a decimal, Sunday being 0
%W	week number of the current year as a decimal number, starting with the first Monday as the first day of the first week
%x	preferred date representation for the current locale without the time
%X	preferred time representation for the current locale without the date
%y	year as a decimal number without a century (range 00–99)
%Y	year as a decimal number including the century
%z	time zone or name or abbreviation (same as %Z on Win32, different on

Linux/UNIX)
 %Z time zone or name or abbreviation (same as %z on Win32, different on Linux/UNIX)
 %% a literal '%' character

Leading zeroes in the display of decimal numbers can be suppressed using - (minus) on Linux/UNIX and # (number sign) on Win32.

See also [date-value](#), [time-of-day](#), [time](#), and [now](#).

date-value

syntax: (date-value *int-year int-month int-day* [*int-hour int-min int-sec*])

syntax: (date-value)

In the first syntax, `date-value` returns the time in seconds since 1970-1-1 00:00:00 for a given date and time. The parameters for the hour, minutes, and seconds are optional. The time is assumed to be Coordinated Universal Time (UCT), not adjusted for the current time zone.

In the second syntax, `date-value` returns the time value in seconds for the current time.

example:

```
(date-value 2002 2 28)      ⇒ 1014854400
(date-value 1970 1 1 0 0 0) ⇒ 0

(date (apply date-value (now))) ⇒ "Wed May 24 10:02:47 2006"
(date (date-value))           ⇒ "Wed May 24 10:02:47 2006"
(date)                        ⇒ "Wed May 24 10:02:47 2006"
```

The following function can be used to transform a `date-value` back into a list:

```
(define (value-date val)
  (append
    (slice (now (+ (/ (date-value) -60) (/ val 60))) 0 5)
    (list (% val 60)))

(value-date 1014854400) ⇒ (2002 2 28 0 0 0)
```

See also [date](#), [time-of-day](#), [time](#), and [now](#).

debug

syntax: (debug *func*)

Calls [trace](#) and begins evaluating the user-defined function in *func*. `debug` is a shortcut for executing `(trace true)`, then entering the function to be debugged.

`debug`

example:

```
;; instead of doing
(trace true)
(my-func a b c)
(trace nil)

;; use debug as a shortcut
(debug (my-func a b c))
```

See also the [trace](#) function.

dec

syntax: (dec *sym* [*num*])

The number in *sym* is decremented by one (or the optional number *num*) and returned. `dec` performs mixed integer and floating point arithmetic according to the rules outlined below.

If *sym* contains a float and *num* is absent, the input argument is truncated to an integer.

Integer calculations (without *num*) resulting in numbers greater than 9,223,372,036,854,775,807 wrap around to negative numbers. Results smaller than -9,223,372,036,854,775,808 wrap around to positive numbers.

If *num* is supplied, `dec` always returns the result as a floating point number (even for integer arguments).

example:

```
(set 'x 10)      ⇒ 10
(dec 'x)         ⇒ 9
x               ⇒ 9
(dec 'x 0.25)    ⇒ 8.75
x               ⇒ 8.75
```

Use the [inc](#) function to increment numbers.

define

syntax: (define (*sym-name* [*sym-param-1* ...]) [*body-1* ...])**syntax: (define (*sym-name* [(*sym-param-1* *exp-default*) ...]) [*body-1* ...])****syntax: (define *sym-name* *exp*)**

Defines the new function *sym-name*, with optional parameters *sym-param-1*—. `define` is equivalent to assigning a lambda expression to *sym-name*. When calling a defined function, all arguments are evaluated and assigned to the variables in *sym-param-1*—, then the *body-1* — expressions are evaluated. When a function is defined, the lambda expression bound to *sym-name* is returned.

All parameters defined are optional. When a user-defined function is called without arguments, those parameters assume the value `nil`. If those parameters have a default value specified in *exp-default*, they assume that value.

The return value of `define` is the assigned *lambda* expression. When calling a user-defined function, the return value is the last expression evaluated in the function body.

example:

```
(define (area x y) (* x y)) ⇒ (lambda (x y) (* x y))
(area 2 3)                  ⇒ 6
```

As an alternative, `area` could be defined as a function without using `define`.

```
(set 'area (lambda (x y) (* x y)))
```

lambda or *fn* expressions may be used by themselves as *anonymous* functions without being defined as a symbol:

```
((lambda (x y) (* x y)) 2 3) ⇒ 6
((fn (x y) (* x y)) 2 3)    ⇒ 6
```

`fn` is just a shorter form of writing `lambda`.

Parameters can have default values specified:

```
(define (foo (a 1) (b 2))
  (list a b))

(foo)      ⇒ (1 2)
(foo 3)    ⇒ (3 2)
(foo 3 4)  ⇒ (3 4)
```

Expressions in *exp-default* are evaluated in the function's current environment.

```
(define-macro (foo (a 10) (b (div a 2))))
  (list a b))

(foo)      ⇒ (10 5)
(foo 30)   ⇒ (30 15)
(foo 3 4)  ⇒ (3 4)
```

The second version of `define` works like the [set](#) function.

example:

```
(define x 123) ⇒ 123
;; is equivalent to
(set 'x 123)   ⇒ 123

(define area (lambda (x y) (* x y)))
;; is equivalent to
(set 'area (lambda (x y) (* x y)))
;; is equivalent to
(define (area x y) (* x y))
```

Trying to redefine a protected symbol will cause an error message.

define-macro

syntax: (define-macro (*sym-name* [*sym-param-1* ...]) *body*)

syntax: (define-macro (*sym-name* [(*sym-param-1* *expr-default*) ...]) *body*)

Defines the new macro *sym-name*, with optional arguments *sym-param-1*—. *define-macro* is equivalent to assigning a lambda-macro expression to a symbol. When a macro-defined function is called, arguments are assigned to the variables in *sym-param-1*—, without evaluating the arguments first. Then the *body* expressions are evaluated. When evaluating the *define-macro* function, the lambda-macro expression is returned.

example:

```
;; use underscores on symbols
(define-macro (my-setq _x _y) (set _x (eval _y)))
⇒ (lambda-macro (_x _y) (set _x (eval _y)))

(my-setq x 123) ⇒ 123
```

Macros in newLISP are similar to *define* functions, but they do *not* evaluate their arguments. New functions can be created to behave like built-in functions that delay the evaluation of certain arguments. Since macros can access the arguments inside a parameter list, they can be used to create flow-control functions like those already built into newLISP.

All parameters defined are optional. When a macro is called without arguments, those parameters assume the value *nil*. If those parameters have a default value specified in *exp-default*, they assume that default value.

```
(define-macro (foo (a 1) (b 2))
  (list a b))

(foo)      ⇒ (1 2)
(foo 3)    ⇒ (3 2)
(foo 3 4)  ⇒ (3 4)
```

Expressions in *exp-default* are evaluated in the function's current environment.

```
(define-macro (foo (a 10) (b (div a 2)))
  (list a b))

(foo)      ⇒ (10 5)
(foo 30)   ⇒ (30 15)
(foo 3 4)  ⇒ (3 4)
```

Note that in macros, the danger exists of passing a parameter with the same variable name as used in the macro definition. In this case, the macro internal variable would end up receiving *nil* instead of the intended value:

```
;; not a good definition!

(define-macro (my-setq x y) (set x (eval y)))

;; symbol name clash for x

(my-setq x 123) ⇒ 123
x                ⇒ nil
```

There are several methods that can be used to avoid this problem, known as *variable capture*, and to write *hygienic* macros:

- Prefix all macro variable names with an underscore character. Using this or a similar convention, the danger of symbol name clashes can be avoided.
- Put the macro into its own lexically closed namespace context. If the function has the same name as the context, it can be called by using the context name alone. A function with this characteristic is called a [default function](#).
- Use [args](#) to access arguments passed by the function.

example:

```
;; a macro as a lexically isolated function
;; avoiding variable capture in passed parameters

(context 'my-setq)

(define-macro (my-setq:my-setq x y) (set x (eval y)))

(context MAIN)

(my-setq x 123)  ⇒ 123 ; no symbol clash
```

The macro in the example is lexically isolated, and no variable capture can occur. Instead of the macro being called using `(my-setq:my-setq ...)`, it can be called with just `(my-setq ...)` because it is a [default function](#).

A third possibility is to refer to passed parameters using [args](#):

example:

```
;; avoid variable capture in macros using the args function

(define-macro (my-setq) (set (args 0) (eval (args 1))))
```

The last example shows how [letex](#) can be combined with `define-macro` to expand macro variables into an expression to be evaluated:

example:

```
(define-macro (dolist-while)
  (letex (var (args 0 0)
          lst (args 0 1)
          cnd (args 0 2)
          body (cons 'begin (1 (args))))
    (let (res)
      (catch (dolist (var lst)
                     (if (set 'res cnd) body (throw res))))))

> (dolist-while (x '(a b c d e f) (!= x 'd)) (println x))
a
b
c
nil
>
```

`dolist-while` loops through a list while the condition is true. Note that a similar feature is already built into [dolist](#) as a *break condition* optional parameter.

Also, the [expand](#) function performs variable expansion explicitly, without evaluating the expanded expression.

def-new

syntax: (def-new *sym-source* [*sym-target*])

This function works similarly to [new](#), but it only creates a copy of one symbol and its contents from the symbol in *sym-source*. When *sym-target* is not given, a symbol with the same name is created in the current context. All symbols referenced inside *sym-source* will be translated into symbol references into the current context. If an argument is present in *sym-target*, the copy will be made into a symbol and context as referenced by the symbol in *sym-target*. In addition to allowing renaming of the function while copying, this also enables the copy to be placed in a different context. All symbol references will be translated into symbol references of the target context. The target context cannot be MAIN.

def-new returns the symbol created:

example:

```
(set 'foo:var '(foo:x foo:y))

(def-new 'foo:var) ⇒ var

var ⇒ (x y)

(def-new 'foo:var 'myvar) ⇒ myvar

myvar ⇒ (x y)

(def-new 'foo:var 'ct:myvar) ⇒ ct:myvar

ct:myvar ⇒ (ct:x ct:y)
```

The function def-new can be used to configure contexts or context objects in a more granular fashion than is possible with [new](#), which copies a whole context.

delete

syntax: (delete *symbol* [*bool*])

syntax: (delete *sym-context* [*bool*])

Deletes a symbol, *symbol*, or a context in *sym-context* with all contained symbols from newLISP's symbol table. References to the symbol will be changed to nil. When the expression in *bool* evaluates to true or anything other than nil, symbols are only deleted when they are not referenced.

Protected symbols of built-in functions and special symbols like nil and true cannot be deleted.

`delete` returns `true` if the symbol was deleted, else it returns `nil`.

example:

```
(set 'lst '(a b aVar c d))

(delete 'aVar) ; aVar deleted, references marked nil

lst ⇒ (a b nil c d)

(set 'lst '(a b aVar c d))

(delete 'aVar true)
⇒ nil ; protect aVar if referenced

lst ⇒ (a b aVar c d)

(set 'foo:x 123)
(set 'foo:y "hello")

;; in contexts, the quote may be omitted
(delete foo) ⇒ foo:x, foo:y and foo will be deleted
```

Note that deleting a symbol that is part of a function which is currently executing can crash the system or have other unforeseen effects.

delete-file

syntax: (delete-file *str-file-name*)

Deletes a file given in *str-file-name*. Returns `true` or `nil` depending on the outcome of the delete operation.

example:

```
(delete-file "junk")
```

This deletes the file `junk` in the current directory.

det

syntax: (det *matrix*)

Returns the determinant of a square matrix. A matrix can either be a nested list or an [array](#).

example:

```
(set 'A '((-1 1 1) (1 4 -5) (1 -2 0)))

(det A) ⇒ -1
```

If the matrix is singular, `nil` is returned.

See also the other matrix functions [invert](#), [multiply](#), and [transpose](#).

device

syntax: (device [*int*])

int is an I/O device number, which is 0 (zero) for the default STD I/O console window. *int* may also be a file handle previously obtained using [open](#). When no argument is supplied, the current I/O device number is returned. The I/O channel specified by *device* is used internally by the functions [print](#) and [read-line](#). When the current I/O device is 0 (zero), [print](#) sends output to the console window and [read-line](#) accepts input from the keyboard. If the current I/O device has been set by opening a file, then [print](#) and [read-line](#) work on that file.

example:

```
(device (open "myfile" "write")) ⇒ 5
(print "This goes in myfile")    ⇒ "This goes in myfile"
(close (device))                 ⇒ true
```

Note that using [close](#) on *device* automatically resets *device* to 0 (zero).

difference

syntax: (difference *list-A list-B*)

syntax: (difference *list-A list-B bool*)

In the first syntax, *difference* returns the *set* difference between *list-A* and *list-B*. The resulting list only has elements occurring in *list-A*, but not in *list-B*. All elements in the resulting list are unique, but *list-A* and *list-B* need not be unique. Elements in the lists can be any type of LISP expression.

example:

```
(difference '(2 5 6 0 3 5 0 2) '(1 2 3 3 2 1)) ⇒ (5 6 0)
```

In the second syntax, *difference* works in *list* mode. *bool* specifies `true` or an expression not evaluating to `nil`. In the resulting list, all elements of *list-B* are eliminated in *list-A*, but duplicates of other elements in *list-A* are left.

example:

```
(difference '(2 5 6 0 3 5 0 2) '(1 2 3 3 2 1) true) ⇒ (5 6 0 5 0)
```

See also the set functions [intersect](#) and [unique](#).

directory

syntax: (directory [*str-path*])

syntax: (directory *str-path str-pattern* [*int-option*])

A list of directory entry names is returned for the directory path given in *str-path*. On failure, `nil` is returned. When *str-path* is omitted, the list of entries in the current directory is returned.

example:

```
(directory "/bin")

(directory "c:/")
```

The first example returns the directory of `/bin`, the second line returns a list of directory entries in the root directory of drive C:. Note that on Win32 systems, a forward slash (/) can be included in path names. When used, a backslash (\) must be preceded by a second backslash.

On Win32 systems, there should be no trailing slash character after the directory name, but the drive letter must be followed by a colon and a forward slash (: /). On Linux/UNIX systems, a trailing slash after the directory name will not cause problems.

In the second syntax, `directory` can take a regular expression pattern in *str-pattern*. Only filenames matching the pattern will be returned in the list of directory entries. In *int-options*, special regular expression options can be specified; see [regex](#) for details.

example:

```
(directory "." "\\c") ⇒ ("foo.c" "bar.c")
;; or using braces as string pattern delimiters
(directory "." {\c}) ⇒ ("foo.c" "bar.c")
```

The regular expression forces `directory` to return only file names containing the string `".c"`.

Other functions that use regular expressions are [find](#), [find-all](#), [parse](#), [regex](#), [replace](#), and [search](#).

directory?

syntax: (directory? *str-path*)

Checks if *str-path* is a directory. Returns `true` or `nil` depending on the outcome.

```
(directory? "/etc") ⇒ true
(directory? "/usr/bin/emacs") ⇒ nil
```

On Win32 systems, there should be no trailing slash character after the directory name, but the drive letter must be followed by a colon and a forward slash (: /). On Linux/UNIX systems, a trailing slash after the directory name will not cause problems.

div

syntax: (div *num-1* *num-2* [*num-3* ...])

syntax: (div *num-1*)

Successively divides *num-1* by the number in *num-2*—. *div* can perform mixed-type arithmetic, but it always returns floating point numbers. Any floating point calculation with NaN also returns NaN.

example:

```
(div 10 3)           ⇒ 3.333333333
(div 120 (sub 9.0 6) 100) ⇒ 0.4
(div 10)             ⇒ 0.1
```

When *num-1* is the only argument, *div* calculates the inverse of *num-1*.

dolist

syntax: (dolist (*sym list* [*exp-break*]) *body*)

The expressions in *body* are evaluated for each element in *list*. The variable in *sym* is set to each of the elements before evaluation of the body expressions. The variable used as loop index is local and behaves according to the rules of dynamic scoping.

Optionally, a condition for early loop exit may be defined in *exp-break*. If the break expression evaluates to any non-nil value, the *dolist* loop returns with the value of *exp-break*. The break condition is tested before evaluating *body*.

example:

```
(set 'x 123)
(dolist (x '(a b c d e f g)) ; prints: abcdefg
  (print x)) ⇒ g           ; return value

(dolist (x '(a b c d e f g) (= x 'e)) ; prints: abcd
  (print x))

;; x is local in do-list
;; x has still its old value outside the loop

x ⇒ 123 ; x has still its old value
```


This example prints `abcdefg` in the console window. After the execution of `dolist`, the value for `x` remains unchanged because the `x` in `dolist` has local scope. The return value of `dolist` is the result of the last evaluated expression.

The internal system variable `$idx` keeps track of the current offset into the list passed to `dolist`, and it can be accessed during its execution:

```
(dolist (x '(a b d e f g))
  (println $idx ":" x)) ⇒ g
```

```
0:a
1:b
2:d
3:e
4:f
5:g
```

The console output is shown in boldface. `$idx` is protected and cannot be changed by the user.

dotimes

syntax: (dotimes (*sym int [exp-break]*) *body*)

The expressions in *body* are evaluated *int* times. The variable in *sym* is set from 0 (zero) to (*int* - 1) each time before evaluating the body expression(s). The variable used as the loop index is local to the `dotimes` expression and behaves according the rules of dynamic scoping. The loop index is of integer type. `dotimes` returns the result of the last expression evaluated in *body*.

Optionally, a condition for early loop exit may be defined in *exp-break*. If the break expression evaluates to any non-`nil` value, the `dotimes` loop returns with the value of *exp-break*. The break condition is tested before evaluating *body*.

example:

```
(dotimes (x 10)
  (print x)) ⇒ 9 ; return value
```

This prints `0123456789` to the console window.

dotree

syntax: (dotree (*sym-context*) *body*)

The expressions in *body* are evaluated for all symbols in *sym-context*. The symbols are accessed in a sorted order. Before each evaluation of the body expression(s), the variable in *sym* is set to the next symbol from *sym-context*. The variable used as the loop index is local to the `dotree` expression and behaves according the rules of dynamic scoping.

example:

```
;; faster and less memory overhead
(dotree (s 'SomeCTX) (print s " "))

;; the quote can be omitted
(dotree (s SomeCTX) (print s " "))

;; slower and more memory usage
(dolist (s (symbols 'SomeCTX)) (print s " "))
```

This example prints the names of all symbols inside SomeCTX to the console window.

do-until

syntax: (do-until *exp-condition body*)

The expressions in *body* are evaluated before *exp-condition* is evaluated. If the evaluation of *exp-condition* is not nil, then the do-until expression is finished; otherwise, the expressions in *body* get evaluated again. Note that do-until evaluates the conditional expression *after* evaluating the body expressions, whereas [until](#) checks the condition *before* evaluating the body. The return value of the do-until expression is the last evaluation of the *body* expression.

example:

```
(set 'x 1)
(do-until (= x 1) (inc 'x))
x ⇒ 2

(set 'x 1)
(until (= x 1) (inc 'x))
x ⇒ 1
```

While do-until goes through the loop at least once, [until](#) never enters the loop.

See also the functions [while](#) and [do-while](#).

do-while

syntax: (do-while *exp-condition body*)

The expressions in *body* are evaluated before *exp-condition* is evaluated. If the evaluation of *exp-condition* is nil, then the do-while expression is finished; otherwise the expressions in *body* get evaluated again. Note that do-while evaluates the conditional expression *after* evaluating the body expressions, whereas [while](#) checks the condition *before* evaluating the body. The return value of the do-while expression is the last evaluation of the *body* expression.

example:

do-while

```
(set 'x 10)
(do-while (< x 10) (inc 'x))
x ⇒ 11

(set 'x 10)
(while (< x 10) (inc 'x))
x ⇒ 10
```

While `do-while` goes through the loop at least once, [while](#) never enters the loop.

See also the functions [until](#) and [do-until](#).

dump

syntax: (`dump` [*exp*])

Shows the binary contents of a newLISP cell. Without an argument, this function outputs a listing of all LISP cells to the console. When *exp* is given, it is evaluated and the contents of a LISP cell are returned in a list.

example:

```
(dump 'a) ⇒ (9586996 5 9578692 9578692 9759280)

(dump 999) ⇒ (9586996 130 9578692 9578692 999)
```

The list contains the following memory addresses and information:

```
(0) memory address of the LISP cell
(1) cell->type: major/minor type, see newlisp.h for details
(2) cell->next: linked list ptr
(3) cell->aux:
    string length+1 or
    low (little endian) or high (big endian) word of 64-bit
integer or
    low word of IEEE 754 double float
(4) cell->contents:
    string/symbol address or
    high (little endian) or low (big endian) word of 64-bit
integer or
    high word of IEEE 754 double float
```

This function is valuable for changing type bits in cells or hacking other parts of newLISP internals. See the function [cpymem](#) for a comprehensive example.

dup

syntax: (`dup` *exp* *int-n* [*bool*])

If the expression in *exp* evaluates to a string, it will be replicated *int-n* times within a string and returned. When specifying an expression evaluating to anything other than `nil` in *bool*, the string will not be concatenated but replicated in a list like any other data type.

If *exp* contains any data type other than string, the returned list will contain *int-n* evaluations of *exp*.

example:

```
(dup "A" 6)           ⇒ "AAAAAA"
(dup "A" 6 true)      ⇒ ("A" "A" "A" "A" "A" "A")
(dup "A" 0)           ⇒ ""
(dup "AB" 5)          ⇒ "ABABABABAB"
(dup 9 7)             ⇒ (9 9 9 9 9 9 9)
(dup 9 0)             ⇒ ()
(dup 'x 8)            ⇒ (x x x x x x x x)
(dup '(1 2) 3)        ⇒ ((1 2) (1 2) (1 2))
(dup "\000" 4)        ⇒ "\000\000\000\000"
```

The last example shows handling of binary information, creating a string filled with four binary zeroes.

See also the functions [sequence](#) and [series](#).

empty?

syntax: (empty? *exp*)

syntax: (empty? *str*)

exp is tested for an empty list (or *str* for an empty string). Depending on whether the argument contains elements, `true` or `nil` is returned.

example:

```
(set 'var '())
(empty? var)           ⇒ true
(empty? '(1 2 3 4))    ⇒ nil
(empty? "hello")       ⇒ nil
(empty? "")            ⇒ true
```

The first example checks a list, while the second two examples check a string.

encrypt

syntax: (encrypt *str-source* *str-pad*)

Performs a one-time-pad encryption of *str-source* using the encryption pad in *str-pad*. The longer *str-pad* is and the more random the bytes are, the safer the encryption. If the pad is as long as the source text, is fully random, and is used only once, then one-time-pad encryption

is virtually impossible to break, since the encryption seems to contain only random data. To retrieve the original, the same function and pad are applied again to the encrypted text:

example:

```
(set 'secret
  (encrypt "A secret message" "my secret key"))
⇒ ",YS\022\006\017\023\017TM\014\022\n\012\030E"

(encrypt secret "my secret key") ⇒ "A secret message"
```

The second example encrypts a whole file:

```
(write-file "myfile.enc"
  (encrypt (read-file "myfile") "29kH67*"))
```

ends-with

syntax: (ends-with *str-data str-key* [*option*])

syntax: (ends-with *list exp*)

In the first syntax, `ends-with` tests the string in *str-data* to see if it ends with the string specified in *str-key*. It returns `true` or `nil` depending on the outcome.

When `nil` or any expression evaluating to `nil` as a third parameter in *bool* is specified, the comparison is case-insensitive.

If a regular expression *option* number is specified *str-key* contains a regular expression pattern. See [regex](#) for valid numbers for *option*.

example:

```
(ends-with "newLISP" "LISP")           ⇒ true
(ends-with "newLISP" "lisp")           ⇒ nil
(ends-with "newLISP" "lisp" nil)       ⇒ true
;; use regular expressions
(ends-with "newLISP" "lisp|york" 1)    ⇒ true
```

In the second syntax, `ends-with` checks if a list ends with the list element in *exp*. `true` or `nil` is returned depending on outcome.

example:

```
(ends-with '(1 2 3 4 5) 5)             ⇒ true
(ends-with '(a b c d e) 'b)            ⇒ nil
(ends-with '(a b c (+ 3 4)) '(+ 3 4)) ⇒ true
```

The last example shows that *exp* could be a list by itself.

See also the [starts-with](#) function.

env

syntax: (env)

syntax: (env *var-str*)

syntax: (env *var-str value-str*)

In the first syntax (without arguments), the operating system's environment is retrieved as a list in which each entry is a string.

example:

```
(env) ⇒ ("PATH=/bin:/usr/bin:/sbin" "USER=LUTZ")
```

In the second syntax, the name of an environment variable is given in *var-str*. `env` returns the value of the variable or `nil` if the variable does not exist in the environment.

example:

```
(env "PATH") ⇒ "/bin:/usr/bin:/usr/local/bin"
```

The third syntax (variable name in *var-str* and value pair in *value-str*) sets or creates an environment variable.

example:

```
(env "NEWLISPDIR" "/usr/bin/") ⇒ true
(env "NEWLISPDIR")           ⇒ "/usr/bin/"
```

`env` replaces the deprecated `environ`, `getenv`, and `putenv` functions.

erf

syntax: (erf *num*)

`erf` calculates the error function of a number in *num*. The error function is defined as:

$$\text{erf}(x) = 2/\sqrt{\pi} * \text{integral from } 0 \text{ to } x \text{ of } \exp(-t^2) dt$$

example:

```
(map erf (sequence 0.0 6.0 0.5))
⇒
(0 0.5204998778 0.8427007929 0.9661051465 0.995322265 0.999593048
0.9999779095 0.9999992569 0.9999999846 0.9999999998 1 1 1)
```

error-event

syntax: (error-event *sym*)

syntax: (error-event *func*)

sym contains a user-defined function for handling errors. Whenever an error occurs, the system performs a [reset](#) and executes the user-defined error handler. The error handler can use the built-in function [error-number](#) to retrieve the number of the error.

example:

```
(define (my-handler)
  (print "error # " (error-number) " has occurred\n")
  (restart-program))

(error-event 'my-handler) ⇒ my-handler

;; specify a function directly

(error-event my-handler) ⇒ $error-event

(error-event
  (fn () (print "error # " (error-number) " has occurred\n")))

(error-event exit) ⇒ $error-event
```

For a different way of handling errors, see the [catch](#) function. Use [throw-error](#) to throw user-defined errors.

error-number

syntax: (error-number)

Returns the number of the last error.

example:

```
(define (my-handler)
  (print "error # " (error-number) " has occurred\n")
  (restart-program))

(error-event 'my-handler)
```

See also the functions [sys-error](#), [throw-error](#), [error-event](#), and [error codes](#) in the appendix.

error-text

syntax: (error-text [*int-error*])

Returns a descriptive text for the error number in *int-error*:

error-text

example:

```
(error-text 5) ⇒ "Not an expression"
```

If no *int-error* is given, the last error is assumed.

See also the list of [error codes](#) in the appendix and the functions [error-event](#), [catch](#), [sys-error](#), and [throw-error](#).

eval

syntax: (eval *exp*)

eval evaluates the result of evaluating *exp*.

example:

```
(set 'expr '(+ 3 4)) ⇒ (+ 3 4)
(eval expr)          ⇒ 7
(eval (list + 3 4))  ⇒ 7
(eval 'x)            ⇒ x
(set 'y 123)
(set 'x 'y)
x                    ⇒ y
(eval x)             ⇒ 123
```

newLISP passes all arguments by value. Using a quoted symbol, expressions can be passed by reference through the symbol. *eval* can be used to access the original contents of the symbol:

```
(define (change-list aList) (push 999 (eval aList)))

(set 'data '(1 2 3 4 5))

(change-list 'data) ⇒ (999 1 2 3 4 5)
```

In the example, the parameter *'data* is quoted, so *push* can work on the original list.

It is also possible to pass arguments by reference in newLISP by enclosing the data inside context objects. See the chapter [Programming with context objects](#) and the sub-chapter [Passing objects by reference](#).

eval-string

syntax: (eval-string *str* [*expr*] [*sym-context*])

Before being evaluated, the result of *str* is compiled into newLISP's internal format, and the result of the evaluation is returned. If the string contains more than one expression, the result of the last evaluation is returned.

A second optional argument, *expr*, can be passed that is evaluated and returned in case of an error. This permits programmatic control to be maintained if the evaluation of *str* produces errors. An optional third argument can be used to specify the context in which the string should be parsed and evaluated. When *sym-context* is specified, the failure expression in *expr* must be specified, as well.

example:

```
(eval-string "(+ 3 4)") ⇒ 7
(set 'X 123)           ⇒ 123
(eval-string "X")      ⇒ 123

(define (lisp)
  (while true
    (print "\n=>" (eval-string (read-line) "syntax error"))))

(set 'a 10)
(set 'b 20)
(set 'foo:a 11)
(set 'foo:b 22)

(eval-string "(+ a b)") ⇒ 30
(eval-string "(+ a b)" nil 'foo) ⇒ 33
```

The second example shows a simple LISP interpreter eval loop.

Use the [catch](#) function to catch errors resulting from the evaluation of expressions, as opposed to strings.

The last example shows how to specify a target context for evaluation. The symbols *a* and *b* now refer to the values in context *foo* instead of *MAIN*.

exec

syntax: (exec *str-process*)

syntax: (exec *str-process* [*str-stdin*])

In the first form, *exec* launches a process described in *str-process* and returns all standard output as an array of strings (one for each line in *stdout*). *exec* returns *nil* if the process could not be launched.

example:

```
(exec "ls *.c") ⇒ ("newlisp.c" "nl-math.c" "nl-string.c")
```

The example starts a process and performs the shell command *ls*, capturing the output in an array of strings.

In the second form, *exec* creates a process pipe, starts the process in *str-process*, and receives from *str-stdin* standard input for this process. The return value is *true* if the process was successfully launched; otherwise it is *nil*.

example:

```
(exec "cgiProc" query)
```

In this example, `cgiProc` could be a cgi processor (e.g., Perl or newLISP) that receives and processes standard input supplied by a string contained in the variable `query`.

exit

syntax: (`exit` [*int*])

Exits newLISP. An optional exit code, *int*, may be supplied. This code can be tested by the host operating system. When newLISP is run in [daemon server mode](#) using `-d` as a command-line option, only the network connection is closed, while newLISP stays resident, listening for a new connection.

example:

```
(exit 5)
```

exp

syntax: (`exp` *num*)

The expression in *num* is evaluated, and the exponential function is calculated based on the result. `exp` is the inverse function of `log`.

example:

```
(exp 1)           ⇒ 2.718281828
(exp (log 1))    ⇒ 1
```

expand

syntax: (`expand` *list sym* [*sym_2 ... sym_n*])

syntax: (`expand` *list list-assoc*)

syntax: (`expand` *list*)

In the first syntax, one symbol in *sym* (or more in *sym_2* through *sym_n*) is looked up in a simple or nested *list*. They are then expanded to the current binding of the symbol, and the expanded list is returned. The original list remains unchanged.

example:

```
(set 'x 2 'a '(d e))
(expand '(a x b) 'x)           ⇒ (a 2 b)
(expand '(a x (b c x)) 'x)    ⇒ (a 2 (b c 2))
```

```
(expand '(a x (b c x)) 'x 'a) ⇒ ((d e) 2 (b c 2))
```

`expand` is useful when composing lambda expressions or doing variable expansion inside macros.

```
(define (raise-to power)
  (expand (fn (base) (pow base power)) 'power))

(define square (raise-to 2))
(define cube (raise-to 3))

(square 5) ⇒ 25
(cube 5)   ⇒ 125
```

If more than one symbol is present, `expand` will work in an incremental fashion:

```
(set 'a '(b c))
(set 'b 1)

(expand '(a b c) 'a 'b) ⇒ ((1 c) 1 c)
```

Like the [apply](#) function, `expand` *reduces* its argument list.

syntax: (expand list list-assoc)

The second syntax of `expand` allows expansion bindings to be specified on the fly, without performing a [set](#) on the participating variables:

example:

```
(expand '(a b c) '((a 1) (b 2))) ⇒ (1 2 c)
(expand '(a b c) '((a 1) (b 2) (c (x y z))))) ⇒ (1 2 (x y z))
```

Note that the contents of the variables in the association list will not change. This is different from the [letex](#) function, where variables are set by evaluating and assigning their association parts.

This form of `expand` is frequently used in logic programming, together with the [unify](#) function.

syntax: (expand list)

A third syntax is used to expand only the contents of variables starting with an uppercase character. This PROLOG mode may also be used in the context of logic programming. As in the first syntax of `expand`, symbols must be preset. Only uppercase variables and those bound to anything other than `nil` will be expanded:

example:

```
(set 'A 1 'Bvar 2 'C nil 'd 5 'e 6)
(expand '(A (Bvar) C d e f)) ⇒ (1 (2) C d e f)
```

Only the symbols `A` and `Bvar` are expanded, since they have capitalized names and non-`nil` contents.

The *currying* function in the example demonstrating the first syntax of `expand` can now be written even more simply using an uppercase variable:

```
(define (raise-to Power)
  (expand (fn (base) (pow base Power))))

> (define cube (raise-to 3))
(lambda (base) (pow base 3))

> (cube 4)
64

> _
```

See the [letex](#) function, which also provides an expansion mechanism, and the function [unify](#), which is frequently used together with `expand`.

explode

syntax: (explode *str*)

Transforms the string in *str* into a list of single-character strings:

example:

```
(explode "newLISP") ⇒ ("n" "e" "w" "L" "I" "S" "P")
(join (explode "keep it together"))
⇒ "keep it together"
```

[join](#) and [append](#) are inverse operations of `explode`.

`explode` also works on binary content:

```
(explode "\000\001\002\003")
⇒ ("\000" "\001" "\002" "\003")
```

When called in UTF-8-enabled versions of newLISP, `explode` will work on character boundaries rather than byte boundaries. In UTF-8-encoded strings, characters may contain more than one byte.

factor

syntax: (factor *int*)

Factors the number in *int* into its prime components. Floating point numbers in *num* are truncated to their integer part.

example:

```
(factor 123456789123456789) ⇒ (3 3 7 11 13 19 3607 3803 52579)
```

```
;; check correctness of factoring
(= (apply * (factor 123456789123456789)) 123456789123456789)
⇒ true

;; factor the biggest integer
(factor 9223372036854775807) ⇒ (7 7 73 127 337 92737 649657)

;; primes.lsp - return all primes in a list, up to n

(define (primes n , p)
  (set 'p '())
  (dotimes (e n)
    (if (= (length (factor e)) 1)
        (push e p -1))) p)

(primes 20) ⇒ (2 3 5 7 11 13 17 19)
```

factor returns nil for numbers smaller than 2. For numbers larger than 9,223,372,036,854,775,807 (the largest 64-bit integer) converted from floating point numbers, the largest integer is factored.

fft

syntax: (fft *list-num*)

Calculates the discrete Fourier transform on the list of complex numbers in *list-num* using the FFT method (Fast Fourier Transform). Each complex number is specified by its real part, followed by its imaginary part. If only real numbers are used, the imaginary part is set to 0.0 (zero). When the number of elements in *list-num* is not a power of 2, *fft* increases the number of elements by padding the list with zeroes. When the imaginary part of a complex number is 0, simple numbers can be used instead.

example:

```
(ifft (fft '((1 0) (2 0) (3 0) (4 0))))
⇒ ((1 0) (2 0) (3 0) (4 0))

;; when imaginary part is 0, plain numbers work, too
;; complex numbers can be intermixed

(fft '(1 2 3 4)) ⇒ ((10 0) (-2 -2) (-2 0) (-2 2))
(fft '(1 2 (3 0) 4)) ⇒ ((10 0) (-2 -2) (-2 0) (-2 2))
```

The inverse operation of *fft* is the [ifft](#) function.

file-info

syntax: (file-info *str_name*)

Returns a list of information about the file or directory in *str_name*. newLISP uses the POSIX system call `stat()` to get the following information:

offset	contents
0	size
1	mode
2	device mode
3	user id
4	group id
5	access time
6	modification time
7	status change time

example:

```
(file-info ".bashrc")
⇒ (124 33188 0 500 0 920951022 920951022 920953074)

(date (last (file-info "/etc")))
⇒ "Mon Mar 8 18:23:17 1999"
```

In the second example, the last status change date for the directory */etc* is retrieved.

file?

syntax: (file? *str-name*)

Checks for the existence of a file in *str-name*. Returns `true` if the file exists; otherwise, it returns `nil`. This function will also return `true` for directories. The existence of a file does not imply anything about its read or write permissions. A file may exist while not having the permissions to read from or write to it by the current user.

example:

```
(if (file? "afile") (set 'fileNo (open "afile" "read")))
```

filter

syntax: (filter *exp-predicate exp-list*)

The predicate *exp-predicate* is applied to each element of the list *exp-list*. A list is returned containing the elements for which *exp-predicate* is true. `filter` works like [clean](#), but with a negated predicate.

example:

```
(filter symbol? '(1 2 d 4 f g 5 h)) ⇒ (d f g h)
```

```
(define (big? x) (> x 5)) ⇒ (lambda (x) (> x 5))
(filter big? '(1 10 3 6 4 5 11)) ⇒ (10 6 11)
```

The predicate may be a built-in predicate, a user-defined function, or a lambda expression.

For filtering a list of elements with the elements from another list, use the [difference](#) function or [intersect](#) (with the *list* option).

See also the related function [index](#), which returns the indices of the filtered elements and [clean](#), which returns all elements of a list for which a predicate is false.

find

syntax: (find *exp-key* *list*)

syntax: (find *str-key* *str-data*)

syntax: (find *str-pattern* *str-data* *int-option*)

syntax: (find *str-pattern* *list* *int-option*)

If the second argument evaluates to a *list*, then `find` returns the index position (offset) of the element derived from evaluating *exp-key*.

If the second argument, *str-data*, evaluates to a string, then the offset position of the string *str-key* (found in the first argument, *str-data*) is returned. In this case, `find` also works on binary *str-data*.

The presence of a third parameter specifies a search using the regular expression pattern specified in *str-pattern*, as well as an option number specified in *int-option* (i.e., 1 (one) for case-insensitive search or 0 (zero) for no special options).

In newLISP, regular expressions are standard Perl Compatible Regular Expression (PCRE) searches. Found expressions or subexpressions are returned in the system variables `$0`, `$1`, `$2`, etc., which can be used like any other symbol. As an alternative, the contents of these variables can also be accessed by using `($ 0)`, `($ 1)`, `($ 2)`, etc. This method allows indexed access (i.e., `($ i)`, where *i* is an integer).

See [regex](#) for the meaning of the option numbers and more information on regular expression searching.

example:

```
(find "world" '("hello" "world")) ⇒ 1
(find "hi" '("hello" "world")) ⇒ nil
(find '(1 2) '((3 4) 5 6 (1 2) (8 9))) ⇒ 3

(find "world" "Hello world") ⇒ 6
(find "WORLD" "Hello woRLd") ⇒ nil

; case-insensitive regex
(find "World" "Hello woRLd" 1) ⇒ 6
```

```

(find "hi" "hello world")      ⇒ nil
(find "Hello" "Hello world")  ⇒ 0

; regex with default options

(find "cat|dog" "I have a cat" 0) ⇒ 9
$0                               ⇒ "cat"
(find "cat|dog" "my dog" 0)    ⇒ 3
$0                               ⇒ "dog"
(find "cat|dog" "MY DOG" 1)    ⇒ 3
$0                               ⇒ "DOG"

; regex finds string at index 2 of a list

(find "a" '(1 2 "nnnammm" '(a b)) 0) ⇒ 2
$0                               ⇒ "a"

;; find with subexpressions in regular expression
;; and access with system variables

(set 'str "http://nuevatec.com:80")

(find "http://(.*):(.*)" str 0) ⇒ 0

$0 ⇒ "http://nuevatec.com:80"
$1 ⇒ "nuevatec.com"
$2 ⇒ "80"

;; system variables as an indexed expression (since 8.0.5)
($ 0) ⇒ "http://nuevatec.com:80"
($ 1) ⇒ "nuevatec.com"
($ 2) ⇒ "80"

```

For other functions using regular expressions, see [directory](#), [find-all](#), [parse](#), [regex](#), [replace](#), and [search](#).

To find expressions in nested or multidimensional lists, use the [ref](#) and [ref-all](#) functions.

find-all

syntax: (**findall** *str-pattern str-text* [*expr*] [*int-option*])

Finds all occurrences of *str-pattern* in the text *str-text*, returning a list containing all matching strings. *nil* is returned if no matches are found.

Optionally, an expression can be specified to process the found string or regular subexpressions before placing them into the returned list. An additional option, *int-option*, specifies special regular expression options (see [regex](#) for further details).

example:

```

(find-all {\d+} "lkjhlkljh34ghfdhgfd678gfdhfgd9")
⇒ ("34" "678" "9")

(find-all {(new)(lisp)} "newLISPisNEWLISP" (append $2 $1) 1)
⇒ ("LISPnew" "LISPNEW")

```



```
(unique (sort
  (find-all {[a-zA-Z]+}
    (replace "<[^>]+>" (get-url "http://newlisp.org") "" 0) )
))
⇒ ("A" "ACC" "AI" "API" "About" "All" "Amazing" "Apps"
  ...
  "where" "whole" "width" "wiki" "will" "with" "work" "written")
```

The first example discovers all numbers in a text. The second example shows how an optional expression in *expr* can work on subexpressions found by the regular expression pattern in *str-pattern*. The last example retrieves a web page, cleans out all HTML tags, and then collects all words into a unique and sorted list.

Note that `find-all` always performs a regular expression search, even if the option in *int-option* is omitted.

first

syntax: (first *list*)

syntax: (first *array*)

syntax: (first *str*)

Returns the first element of a list or the first character of a string. The operand is not changed. This function is equivalent to *car* or *head* in other LISP dialects.

example:

```
(first '(1 2 3 4 5))      ⇒ 1
(first '((a b) c d))     ⇒ (a b)
(set 'aList '(a b c d e)) ⇒ (a b c d e)
(first aList)             ⇒ a
aList                    ⇒ (a b c d e)

(set 'A (array 3 2 (sequence 1 6)))
⇒ ((1 2) (3 4) (5 6))
(first A)                 ⇒ (1 2)
```

In the second syntax, the first character is returned from the string in *str* as a string.

example:

```
(first "newLISP")        ⇒ "n"
(first (rest "newLISP")) ⇒ "e"
```

See also the functions [last](#) and [rest](#).

flat

syntax: (flat *list*)

Returns the flattened form of a list:

example:

```
(set 'lst '(a (b (c d))))
(flat lst) ⇒ (a b c d)

(map (fn (x) (ref x lst)) (flat lst))
⇒ ((0) (1 0) (1 1 0) (1 1 1))
```

`flat` can be used to iterate through nested lists.

fn

syntax: (fn (*list-parameters*) *exp-body*)

`fn` is used to define anonymous functions, which are frequently used in [map](#), [sort](#), and anywhere functions can be used as a argument.

Using an anonymous function eliminates the need to define a new function with [define](#). Instead, a function is defined on the fly:

example:

```
(map (fn (x) (+ x x)) '(1 2 3 4 5)) ⇒ (2 4 6 8 10)

(sort '("." "..." "." ".....") (fn (x y) (> (length x) (length y))))
⇒ ("....." "..." "..." ".")
```

The example defines the function *fn(x)*, which takes an integer (*x*) and doubles it. The function is *mapped* onto a list of arguments using [map](#). The second example shows strings being sorted by length.

The [lambda](#) function (the longer, traditional form) can be used in place of `fn`.

float

syntax: (float *exp* [*exp-default*])

If the expression in *exp* evaluates to a number or a string, the argument is converted to a float and returned. If *exp* cannot be converted to a float then `nil` or, if specified, the evaluation of *exp-default* will be returned. This function is mostly used to convert strings from user input or when reading and parsing text. The string must start with a digit or the + (plus sign), - (minus sign), or . (period). If *str* is invalid, `float` returns `nil` as a default value.

Floats with exponents larger than 1e308 or smaller than -1e308 are converted to +INF or -INF, respectively. The display of +INF and -INF differs on different platforms and compilers.

example:

```

(float "1.23")           ⇒ 1.23
(float " 1.23")          ⇒ 1.23
(float ".5")             ⇒ 0.50
(float "-1.23")          ⇒ -1.23
(float "-.5")            ⇒ nil
(float "#1.23")          ⇒ nil
(float "#1.23" 0.0)      ⇒ 0

(float? 123)             ⇒ nil
(float? (float 123))     ⇒ true

(float '(a b c))         ⇒ nil
(float '(a b c) 0)       ⇒ 0
(float nil 0)            ⇒ 0

(float "abc" "not a number") ⇒ "not a number"
(float "1e500")           ⇒ inf
(float "-1e500")          ⇒ -inf

(print "Enter a float num:")
(set 'f-num (float (read-line)))

```

Use the [int](#) function to parse integer numbers.

float?

syntax: (float? exp)

true is returned only if *exp* evaluates to a floating point number; otherwise, nil is returned.

example:

```

(set 'num 1.23)
(float? num) ⇒ true

```

floor

syntax: (floor number)

Returns the next lowest integer below *number* as a floating point.

example:

```

(floor -1.5) ⇒ -2
(floor 3.4)  ⇒ 3

```

See also the [ceil](#) function.

flt

syntax: (flt *number*)

Converts *number* to a 32-bit float represented by an integer. This function is used when passing 32-bit floats to library routines. newLISP floating point numbers are 64-bit and are passed as 64-bit floats when calling imported C library routines.

example:

```
(flt 1.23) ⇒ 1067282596

;; pass 32-bit float to C-function: foo(float value)
(import "mylib.so" "foo")
(foo (flt 1.23))

(get-int (pack "f" 1.23)) ⇒ 1067282596

(unpack "f" (pack "ld" (flt 1.2345))) ⇒ (1.234500051)
```

The last two statements illustrate the inner workings of `flt`.

Use the [import](#) function to import libraries.

for

syntax: (for (*sym num-from num-to* [*num-step*] [*exp-break*]) *body*)

Repeatedly evaluates the expressions in *body* for a range of values specified in *num-from* and *num-to*, inclusive. A step size may be specified with *num-step*. If no step size is specified, 1.0 is assumed.

Optionally, a condition for early loop exit may be defined in *exp-break*. If the break expression evaluates to any non-`nil` value, the `for` loop returns with the value of *exp-break*. The break condition is tested before evaluating *body*. If a break condition is defined *num-step* must be defined too.

The symbol *sym* is local in dynamic scope to the `for` expression. It takes on each value successively in the specified range as an integer value if no step size is specified, or as a floating point value when a step size is present.

example:

```
> (for (x 1 10 2) (println x))
1
3
5
7
9

> (for (x 8 6 0.5) (println x))
8
7.5
```

```

7
6.5
6

> (for (x 1 100 2 (> (* x x) 30)) (println x))
1
3
5
true
> _

```

The second example uses a range of numbers from highest to lowest. Note that the step size is always a positive number. In the third example, a break condition is tested.

Use the [sequence](#) function to make a sequence of numbers.

fork

syntax: (fork *exp*)

The expression in *exp* is launched as a newLISP child process thread of the platform's OS. The new process inherits the entire address space, but runs independently, so symbol or variable contents changed in the child thread will not affect the parent process or vice versa. The child process ends when the evaluation of *exp* finishes.

On success, `fork` returns with the child process ID; on failure, `nil` is returned. See also the [wait-pid](#) function, which waits for a child process to finish.

This function is only available on Linux/UNIX versions of newLISP and is based on the `fork()` implementation of the underlying OS.

example:

```

> (set 'x 0)
0
> (fork (while (< x 20) (println (inc 'x)) (sleep 1000)))
176

> 1
2
3
4
5
6

```

The example illustrates how the child process thread inherits the symbol space and how it is independent of the parent process. The `fork` statement returns immediately with the process ID 176. The child process increments the variable `x` by one each second and prints it to standard out (boldface). In the parent process, commands can still be entered. Type `x` to see that the symbol `x` still has the value 0 (zero) in the parent process. Although statements entered will mix with the display of the child process output, they will be correctly input to the parent process.

The second example illustrates how [pipe](#) can be used to communicate between threads.

example:

```
#!/usr/bin/newlisp

(define (count-down-thread x channel)
  (while (!= x 0)
    (begin
      (write-line (string x) channel)
      (dec 'x))))

(define (observer-thread channel)
  (do-until (= i "1")
    (println "thread " (setq i (read-line channel)))))

(map set '(in out) (pipe))
(set 'observer (fork (observer-thread in)))
(set 'counter (fork (count-down-thread 5 out)))

; avoid zombies
(wait-pid observer)
(wait-pid counter)

(exit)
```

The following output is generated by observer-thread

```
thread 5
thread 4
thread 3
thread 2
thread 1
```

The count-down-thread writes numbers to the communication pipe, where they are picked up by the observer-thread and displayed.

Use the [semaphore](#) function for synchronizing threads and [share](#) for sharing memory between threads.

format

syntax: (format *str-format* *exp-data-1* [*exp-data-i* ...])

syntax: (format *str-format* *list-data*)

Constructs a formatted string from *exp-data-1* using the format specified in the evaluation of *str-format*. The format specified is identical to the format used for the `printf()` function in the ANSI C language. Two or more *exp-data* arguments can be specified for more than one format specifier in *str-format*.

In an alternative syntax, the data to be formatted can be passed inside a list in *list-data*.

`format` checks for a valid format string, matching data type, and the correct number of arguments. Wrong formats or data types result in error messages. [int](#), [float](#), or [string](#) can be used to ensure correct data types and to avoid error messages.

The format string has the following general format:

"%w.pf"

The % (percent sign) starts a format specification. To display a % inside a format string, double it: %%

The w represents the width field. Data is right-aligned, except when preceded by a minus sign, in which case it is left-aligned. When preceded by a zero, the unused space is filled with leading zeroes. The width field is optional and serves all data types.

The p represents the precision number of decimals (floating point only) or strings and is separated from the width field by a period. Precision is optional. If preceded by a + (plus sign), positive numbers are displayed with a +. When using the precision field on strings, the number of characters displayed is limited to the number in p.

The f represents a type flag and is essential; it cannot be omitted.

Below are the types in f:

s	text string
c	character (value 1 - 255)
d	decimal (32-bit)
u	unsigned decimal (32-bit)
x	hexadecimal lowercase
X	hexadecimal uppercase
f	floating point
e	scientific floating point
E	scientific floating point
g	general floating point

Formatting 64-bit numbers using 32-bit format specifiers will truncate and format the lower 32 bits of the number.

For 64-bit numbers (since version 8.9.7) use the following format strings on UNIX like operating systems:

lld	decimal (64-bit)
llu	unsigned decimal (64-bit)
llx	hexadecimal (64-bit)
llX	hexadecimal uppercase(64-bit)

For 64-bit numbers (since version 8.9.7) use the following format strings on Tru64 UNIX:

ld	decimal (64-bit)
lu	unsigned decimal (64-bit)
lx	hexadecimal (64-bit)
lX	hexadecimal uppercase(64-bit)

On Win32 platforms the following characters apply for 64 bit numbers:

I64d	decimal (64-bit)
I64u	unsigned decimal (64-bit)
I64x	hexadecimal (64-bit)
I64X	hexadecimal uppercase(64-bit)

Other text may occur between, before, or after the format specs.

Note that on Tru64 UNIX the format character i can be used instead of d.

example:

```

(format ">>>%6.2f<<<" 1.2345)      ⇒ ">>> 1.23<<<"
(format ">>>%-6.2f<<<" 1.2345)     ⇒ ">>>1.23 <<<"
(format ">>>%+6.2f<<<" 1.2345)     ⇒ ">>> +1.23<<<"
(format ">>>%+6.2f<<<" -1.2345)    ⇒ ">>> -1.23<<<"
(format ">>>%-6.2f<<<" -1.2345)    ⇒ ">>>-1.23 <<<"

(format "%e" 123456789)             ⇒ "1.234568e+08"
(format "%12.10E" 123456789)        ⇒ "1.2345678900E+08"

(format "%10g" 1.23)                ⇒ " 1.23"
(format "%10g" 1.234)               ⇒ " 1.234"

(format "Result = %05d" 2)           ⇒ "Result = 00002"

(format "%-15s" "hello")             ⇒ "hello "
(format "%15s %d" "hello" 123)       ⇒ " hello 123"
(format "%5.2s" "hello")             ⇒ " he"
(format "%-5.2s" "hello")            ⇒ "he "

(format "%x %X" -1 -1)              ⇒ "fffffff FFFFFFFF"

(format "%c" 65)                    ⇒ "A"

```

The data to be formatted can be passed inside a list:

```

(set 'L '("hello" 123))
(format "%15s %d" L)                  ⇒ " hello 123"

```

If the format string requires it, newLISP's `format` will automatically convert integers into floating points or floating points into integers:

```

(format "%f" 123)                    ⇒ 123.000000
(format "%d" 123.456)                ⇒ 123

```

fv**syntax: (fv *num-rate num-nper num-pmt num-pv* [*int-type*])**

Calculates the future value of a loan with constant payment *num-pmt* and constant interest rate *num-rate* after *num-nper* period of time and a beginning principal value of *num-pv*. If payment is at the end of the period, *int-type* is 0 (zero); for payment at the end of each period, *int-type* is 1. If *num-type* is omitted, payment at the end of each period is assumed.

example:

```

(fv (div 0.07 12) 240 775.30 -100000) ⇒ -0.5544645052

```

The example illustrates how a loan of \$100,000 is paid down to a residual of \$0.55 after 240 monthly payments at a yearly interest rate of 7 percent.

See also the functions [irr](#), [nper](#), [npv](#), [pmt](#), and [pv](#).

gammai

syntax: (gammai *num-a num-b*)

Calculates the incomplete Gamma function of values *a* and *b* in *num-a* and *num-b*, respectively.

example:

```
(gammai 4 5) ⇒ 0.7349740847
```

The incomplete Gamma function is used to derive the probability of Chi² to exceed a given value for a degree of freedom, df, as follows:

$$Q(\text{Chi}^2|\text{df}) = Q(\text{df}/2, \text{Chi}^2/2) = \text{gammai}(\text{df}/2, \text{Chi}^2/2)$$

See also the [prob-chi2](#) function.

gammaLn

syntax: (gammaLn *num-x*)

Calculates the log Gamma function of the value *x* in *num-x*.

example:

```
(exp (gammaLn 6)) ⇒ 120
```

The example uses the equality of $n! = \text{gamma}(n + 1)$ to calculate the factorial value of 5.

The log Gamma function is also related to the Beta function, which can be derived from it:

$$\text{Beta}(z,w) = \text{Exp}(\text{GammaLn}(z) + \text{GammaLn}(w) - \text{GammaLn}(z+w))$$

gcd

syntax: (gcd *int_1 [int-2 ...]*)

Calculates the greatest common divisor of a group of integers. The greatest common divisor of two integers that are not both zero is the largest integer that divides both numbers. gcd will calculate the greatest common divisor for the first two integers in *int-i* and then further

reduce the argument list by calculating the greatest common divisor of the result and the next argument in the parameter list.

example:

```
(gcd 0)           ⇒ 0
(gcd 0 0)        ⇒ 0
(gcd 10)         ⇒ 10
(gcd 12 36)      ⇒ 12
(gcd 15 36 6)    ⇒ 3
```

See [Wikipedia](#) for details and theory about gcd numbers in mathematics.

get-char

syntax: (get-char *int-address*)

Gets a character from an address specified in *int-address*. This function is useful when using imported shared library functions with [import](#).

example:

```
char * foo(void)
{
  char * result;
  result = "ABCDEFGH";
  return(result);
}
```

Consider the above C function from a shared library, which returns a character pointer (address to a string).

```
(import "mylib.so" "foo")
(print (get-char (foo) )) ⇒ 65
(print (get-char (+ (foo) 1))) ⇒ 66
```

Note that it is unsafe to use the `get-char` function with an incorrect address in *int-address*. Doing so could result in the system crashing or becoming unstable.

See also the [address](#), [get-int](#), [get-float](#), [get-string](#), [pack](#), and [unpack](#) functions.

get-float

syntax: (get-float *int-address*)

Gets a 64-bit double float from an address specified in *int-address*. This function is helpful when using imported shared library functions (with `import`) that return an address pointer to a double float or a pointer to a structure containing double floats.

example:

get-float

```
double float * foo(void)
{
    double float * result;
    ...
    *result = 123.456;
    return(result);
}
```

The previous C function is compiled into a shared library.

```
(import "mylib.so" "foo")
(get-float (foo)) ⇒ 123.456
```

`foo` is imported and returns a pointer to a double float when called. Note that `get-float` is unsafe when used with an incorrect address in *int-address* and may result in the system crashing or becoming unstable.

See also the [address](#), [get-int](#), [get-char](#), [get-string](#), [pack](#), and [unpack](#) functions.

get-int

syntax: (get-int *int-address*)

Gets a 32-bit integer from the address specified in *int-address*. This function is handy when using imported shared library functions with `import`, a function returning an address pointer to an integer, or a pointer to a structure containing integers.

example:

```
int * foo(void)
{
    int * result;
    ...
    *result = 123;
    return(result);
}

int foo-b(void)
{
    int result;
    ...
    result = 456;
    return(result);
}
```

Consider the C function `foo` (from a shared library), which returns an integer pointer (address of an integer).

```
(import "mylib.so" "foo")
(get-int (foo)) ⇒ 123
(foo-b)      ⇒ 456
```

Note that using `get-int` with an incorrect address in *int-address* is unsafe and could result in the system crashing or becoming unstable.

See also the [address](#), [get-char](#), [get-float](#), [get-string](#), [pack](#), and [unpack](#) functions.

get-string

syntax: (get-string *int-address*)

Gets a character string from the address specified in *int-address*. This function is helpful when using imported shared library functions with [import](#).

example:

```
char * foo(void)
{
  char * result;
  result = "ABCDEFGH";
  return(result);
}
```

Consider the above C function from a shared library, which returns a character pointer (address to a string).

```
(import "mylib.so" "foo")
(print (get-string (foo))) ⇒ "ABCDEFGH"
```

When a string is passed as an argument, `get-string` will take its address as the argument. Because `get-string` always breaks off at the first first `\000` (null character) it encounters, it can be used to retrieve a string from a buffer:

example:

```
(set 'buff "ABC\000\000\000") ⇒ "ABC\000\000\000"
(length buff) ⇒ 6
(get-string buff) ⇒ "ABC"
(length (get-string buff)) ⇒ 3
```

See also the [get-char](#), [get-int](#), [get-float](#), [pack](#), and [unpack](#) functions.

Note that `get-string` can crash the system or make it unstable if the wrong address is specified.

get-url

syntax: (get-url *str-url* [*str-option*] [*int-timeout*] [*str-header*]))

Reads a web page or file specified by the URL in *str-url* using the HTTP GET protocol. "header" can be optionally specified to retrieve only the header. A list option, "list",

causes header and page information to be returned as separate strings in a list. The old "debug" option, which printed header information to the console, has been eliminated.

The optional argument *int-timeout* can specify a value in milliseconds. If no data is available from the host after the specified timeout, `get-url` returns the string `ERR: timeout`. When other error conditions occur, `get-url` returns a string starting with `ERR:` and the description of the error.

`get-url` requests are also understood by newLISP server nodes.

example:

```
(get-url "http://www.nuevatec.com")
(get-url "http://www.nuevatec.com" 3000)
(get-url "http://www.nuevatec.com" "header")
(get-url "http://www.nuevatec.com" "header" 5000)
(get-url "http://www.nuevatec.com" "list")

(env "HTTP_PROXY" "http://ourproxy:8080")
(get-url "http://www.nuevatec.com/newlisp/")
```

The index page from the site specified in *str-url* is returned as a string. In the third line, only the HTTP header is returned in a string. Lines 2 and 4 show a timeout value being used.

The second example illustrates the use of a proxy server. The proxy server's URL must be in the operating system's environment. As shown in the example, this can be added using the [env](#) function.

The *int-timeout* can be followed by an optional custom header in *str-header*:

Custom header

The custom header may contain options for browser cookies or other directives to the server. When no *str-header* is specified, newLISP sends certain header information by default. After the following request:

```
(get-url "http://somehost.com" 5000)
```

newLISP will configure and send the request and header below:

```
GET / HTTP/1.1
Host: somehost.com
User-Agent: newLISP v8800
Connection: close
```

As an alternative, the *str-header* option could be used:

```
(get-url "http://somehost.com" 5000
  "User-Agent: Mozilla/4.0\r\nCookie: name=fred\r\n")
```

newLISP will now send the following request and header:

```
GET / HTTP/1.1
Host: somehost.com
User-Agent: Mozilla/4.0
Cookie: name=fred
Connection: close
```

Note that when using a custom header, newLISP will only supply the GET request line, as well as the `Host:` and `Connection:` header entries. newLISP inserts all other entries supplied in the custom header between the `Host:` and `Connection:` entries. Each entry must end with a carriage return line-feed pair: `\r\n`.

See an HTTP transactions reference for valid header entries.

Custom headers can also be used in the [put-url](#) and [post-url](#) functions.

global

syntax: (global *sym-1* [*sym-2* ...])

One or more symbols in *sym-1* [*sym-2* ...] can be made globally accessible from contexts other than MAIN. The statement has to be executed in the MAIN context, and only symbols belonging to MAIN can be made global. `global` returns the last symbol made global.

example:

```
(global 'aVar 'x 'y 'z) ⇒ z

(define (foo x)
  (...))

(constant (global 'foo))
```

The second example shows how [constant](#) and `global` can be combined into one statement, protecting and making a previous function definition global.

if

syntax: (if *exp-condition* *exp-1* [*exp-2*])

syntax: (if *exp-cond-1* *exp-1* *exp-cond-2* *exp-2* [...])

If the value of *exp-condition* is not nil or an empty list, the result of evaluating *exp-1* is returned; otherwise, the value of *exp-2* is returned. If *exp-2* is absent, the value of *exp-condition* is returned.

example:

```
(set 'x 50)                ⇒ 50
(if (< x 100) "small" "big") ⇒ "small"
(set 'x 1000)              ⇒ 1000
(if (< x 100) "small" "big") ⇒ "big"
(if (> x 2000) "big")       ⇒ nil
```

The second form of `if` works similarly to [cond](#), except it does not take parentheses around the condition-body pair of expressions. In this form, `if` can have an unlimited number of arguments.

example:

```
(define (classify x)
  (if
    (< x 0) "negative"
    (< x 10) "small"
    (< x 20) "medium"
    (>= x 30) "big"
    "n/a"))

(classify 15)    ⇒ "medium"
(classify 100)   ⇒ "big"
(classify 22)    ⇒ "n/a"
(classify -10)   ⇒ "negative"
```

The last expression, "n/a", is optional. When this option is omitted, the evaluation of (\geq x 30) is returned, behaving exactly like a traditional [cond](#) but without requiring parentheses around the condition-expression pairs.

In any case, the whole `if` expression always returns the last expression or condition evaluated.

See also the [unless](#) function.

ifft

syntax: (ifft *list-num*)

Calculates the inverse discrete Fourier transform on a list of complex numbers in *list-num* using the FFT method (Fast Fourier Transform). Each complex number is specified by its real part, followed by its imaginary part. In case only real numbers are used, the imaginary part is set to 0.0 (zero). When the number of elements in *list-num* is not an integer power of 2, `ifft` increases the number of elements by padding the list with zeroes. When complex numbers are 0 in the imaginary part, simple numbers can be used.

example:

```
(ifft (fft '((1 0) (2 0) (3 0) (4 0))))
⇒ ((1 0) (2 0) (3 0) (4 0))

;; when imaginary part is 0, plain numbers work too

(ifft (fft '(1 2 3 4)))
⇒ ((1 0) (2 0) (3 0) (4 0))
```

The inverse operation of `ifft` is the [fft](#) function.

import

syntax: (import *str-lib-name str-function-name* ["cdecl"])

import

Imports the function specified in *str-function-name* from a shared library named in *str-lib-name*. The functions [address](#), [get-char](#), [get-int](#), [get-float](#), [get-string](#), [pack](#), and [unpack](#) can be used to retrieve return values or to unpack data from returned structure addresses. If the library is not located in the normal library search path, *str-lib-name* must contain the full path name.

To transform newLISP data types into the data types needed by the imported function, use the functions [float](#) for 64-bit double floats, [flt](#) for 32-bit floats, and [int](#) for 32-bit integers. By default, newLISP passes floating point numbers as 64-bit double floats, integers as 32-bit integers, and strings as 32-bit integers for string addresses.

example:

```
;; import in Linux
(import "libc.so.6" "printf") ⇒ printf <400862A0>

;; import in Mac OS X
(import "libc.dylib" "printf") ⇒ printf <90022080>

;; import in CYGWIN
(import "cygwin1.dll" "printf") ⇒ printf <6106B108>

(printf "%g %s %d %c\n" 1.23 "hello" 999 65)
1.23 hello 999 A
⇒ 17 ; return value

;; import Win32 DLLs in Win32 or CYGWIN version
(import "kernel32.dll" "GetTickCount") ⇒ GetTickCount
(import "user32.dll" "MessageBoxA") ⇒ MessageBoxA
(GetTickCount) ⇒ 3328896
```

In the first example, the string "1.23 hello 999 A" is printed as a side effect, and the value 17 (number of characters printed) is returned. Any C function can be imported from any shared library in this way.

The message box example pops up a Windows dialog box, which may be hidden behind the console window. The console prompt does not return until the 'OK' button is pressed in the message box.

```
;;this pops up a message box

(MessageBoxA 0 "This is the body" "Caption" 1)
```

The other examples show several imports of Win32 DLL functions and the details of passing values *by value* or *by reference*. Whenever strings or numbers are passed by reference, space must be reserved beforehand.

```
;; allocating space for a string return value

(import "kernel32.dll" "GetWindowsDirectoryA")
(set 'str (dup "\000" 64) ; reserve space and initialize

(GetWindowsDirectoryA str (length str))

str ⇒ "C:\\WINDOWS\\000"
```



```

(slice str 0 (find "\000" str)) ⇒ "C:\\WINDOWS"

;; or use trim
(trim str) ⇒ "C:\\WINDOWS"

;; passing an integer parameter by reference
(import "kernel32.dll" "GetComputerNameA")

(set 'str (dup "\000" 64) ; reserve space, initialize

;; get size in a buffer lpNum
(set 'lpNum (pack "lu" (length str)))

;; call the function
(GetComputerNameA str lpNum)

str ⇒ "LUTZ-PC\000"

(slice str 0 (find "\000" str)) ⇒ "LUTZ-PC"

;; or use trim
(trim str) ⇒ "LUTZ-PC"

```

`import` returns the address of the function, which can be used to assign a different name to the imported function.

```

(set 'imprime (import "libc.so.6" "printf"))
⇒ printf <400862A0>

(imprime "%s %d" "hola" 123)
⇒ "hola 123"

```

Note that the preceding examples are not displayed in the newLISP-tk GUI front-end, as the output of `'printf'` is directed to standard out (STDIO) and is not visible in the newLISP-tk console. This only affects function imports with output to standard out.

Note that the Win32 version of newLISP uses standard call *stdcall* conventions to call DLL library routines. This is necessary for calling DLLs that belong to the Win32 operating system (e.g., `odbc32.dll`). Most third-party DLLs are compiled for C declaration *cdecl* calling conventions and may need to specify the string `"cdecl"` as an additional last argument when importing functions. newLISP compiled for Linux and other UNIX systems uses the *cdecl* calling conventions by default and ignores any additional string.

```

;; force cdecl calling conventions on Win32
(import "sqlite.dll" "sqlite_open" "cdecl") ⇒ sqlite_open
<673D4888>

```

Imported functions may take up to fourteen arguments. Note that floating point arguments take up two spaces each (e.g., passing five floats takes up ten of the fourteen parameters).

inc

syntax: (inc *sym* [*num*])

Increments the number in *sym* by 1 or by the optional number *num* and returns the result. *inc* performs mixed int and float arithmetic according to the rules outlined below.

If *num* is absent, *inc* always returns an integer in *sym*. If the input arguments are floats and *num* is absent, the input arguments are truncated to integers.

Integer calculations (without *num*) resulting in numbers greater than 9,223,372,036,854,775,807 wrap around to negative numbers. Results smaller than -9,223,372,036,854,775,808 wrap around to positive numbers.

If *num* is supplied, *inc* always returns the result as floating point, even for integer input arguments.

example:

```

(set 'x 0)      ⇒ 0
(inc 'x)        ⇒ 1
x              ⇒ 1
(inc 'x 0.25)   ⇒ 1.25
x              ⇒ 1.25

(inc 'x) ⇒ 2 ; gets truncated

```

Use the [dec](#) function for decrementing.

index

syntax: (index *exp-predicate exp-list*)

Applies the predicate *exp-predicate* to each element of the list *exp-list* and returns a list containing the indices of the elements for which *exp-predicate* is true.

example:

```

(index symbol? '(1 2 d 4 f g 5 h)) ⇒ (2 4 5 7)

(define (big? x) (> x 5)) ⇒ (lambda (x) (> x 5))

(index big? '(1 10 3 6 4 5 11)) ⇒ (1 3 6)

```

The predicate may be a built-in predicate, a user-defined function, or a lambda expression.

Use the [filter](#) function to return the elements themselves.

int

syntax: (int *exp* [*exp-default*] [*int-base*])

If the expression in *exp* evaluates to a number or a string, the result is converted to an integer and returned. If *exp* cannot be converted to an integer, then `nil` or the evaluation of *exp-default* will be returned. This function is mostly used when translating strings from user input or from parsing text. If *exp* evaluates to a string, the string must start with a digit; one or more spaces; or the + or - sign. The string must begin with '0x' for hexadecimal strings or '0' (zero) for octal strings. If *str* is invalid, `integer` returns `nil` as a default value if not otherwise specified.

A second optional parameter can be used to force the number base of conversion to a specific value.

Integers larger than 9,223,372,036,854,775,807 are truncated to 9,223,372,036,854,775,807. Integers smaller than -9,223,372,036,854,775,808 are truncated to -9,223,372,036,854,775,808.

When converting from a float (as in the second form of `integer`), floating point values larger or smaller than the integer maximum or minimum are also truncated. A floating point expression evaluating to NaN is converted to 0 (zero).

example:

```

(int "123")           ⇒ 123
(int " 123")          ⇒ 123
(int "a123" 0)        ⇒ 0
(int (trim " 123"))   ⇒ 123
(int "0xFF")          ⇒ 255
(int "055")           ⇒ 45
(int "1.567")         ⇒ 1
(int 1.567)           ⇒ 1

(integer? 1.00)        ⇒ nil
(integer? (int 1.00))  ⇒ true

(int "1111" 0 2)      ⇒ 15    ; base 2 conversion
(int "0xFF" 0 16)     ⇒ 255   ; base 16 conversion

(int 'xyz)            ⇒ nil
(int 'xyz 0)          ⇒ 0
(int nil 123)         ⇒ 123

(int "abc" "not a number") ⇒ "not a number"

(print "Enter a num:")
(set 'num (int (read-line)))

```

Use the [float](#) function to convert arguments to floating point numbers.

integer?

syntax: (integer? *exp*)

Returns `true` only if the value of *exp* is an integer; otherwise, it returns `nil`.

example:

```
(set 'num 123)  ⇒ 123
(integer? num) ⇒ true
```

intersect

syntax: (intersect *list-A list-B*)

syntax: (intersect *list-A list-B bool*)

In the first syntax, `intersect` returns a list containing one copy of each element found both in *list-A* and *list-B*.

example:

```
(intersect '(3 0 1 3 2 3 4 2 1) '(1 4 2 5))
⇒ (2 4 1)
```

In the second syntax, `intersect` returns a list of all elements in *list-A* that are also in *list-B*, without eliminating duplicates in *list-A*. *bool* is an expression evaluating to `true` or any other value not `nil`.

example:

```
(intersect '(3 0 1 3 2 3 4 2 1) '(1 4 2 5) true)
⇒ (1 2 4 2 1)
```

See also the set functions [difference](#) and [unique](#).

invert

syntax: (invert *matrix*)

Returns the inversion of a two-dimensional matrix in *matrix*. The matrix must be square, with the same number of rows and columns, and *non-singular* (invertible). Matrix inversion can be used to solve systems of linear equations (e.g., multiple regression in statistics). newLISP uses LU-decomposition of the matrix to find the inverse.

The dimensions of a matrix are defined by the number of rows times the number of elements in the first row. For missing elements in non-rectangular matrices, 0.0 (zero) is assumed. A matrix can either be a nested list or an [array](#).

`invert` will return `nil` if the matrix is *singular* and cannot be inverted.

example:

```
(set 'A '((-1 1 1) (1 4 -5) (1 -2 0)))
(invert A) ⇒ ((10 2 9) (5 1 4) (6 1 5))
```

All operations shown here on lists can be performed on arrays, as well.

See also the matrix functions [det](#), [transpose](#), and [multiply](#).

irr

syntax: (`irr` *list-amounts* [*list-times* [*num-guess*]])

Calculate the internal rate of return of a cash flow per time period. The internal rate of return is the interest rate that makes the present value of a cash flow equal to 0.0 (zero). In-flowing (negative values) and out-flowing (positive values) amounts are specified in *list-amounts*. If no time periods are specified in *list-times*, amounts in *list-amounts* correspond to consecutive time periods increasing by 1 (1, 2, 3—). The algorithm used is iterative, with an initial guess of 0.5 (50 percent). Optionally, a different initial guess can be specified. The algorithm returns when a precision of 0.000001 (0.0001 percent) is reached. `nil` is returned if the algorithm cannot converge after 50 iterations.

`irr` is often used to decide between different types of investments.

example:

```
(irr '(-1000 500 400 300 200 100))
⇒ 0.2027

(npv 0.2027 '(500 400 300 200 100))
⇒ 1000.033848 ; ~ 1000

(irr '(-1000 500 400 300 200 100) '(0 3 4 5 6 7))
⇒ 0.0998

(irr '(-5000 -2000 5000 6000) '(0 3 12 18))
⇒ 0.0321
```

If an initial investment of 1,000 yields 500 after the first year, 400 after two years, and so on, finally reaching 0.0 (zero) after five years, then that corresponds to a yearly return of about 20.2 percent. The next line demonstrates the relation between `irr` and [npv](#). Only 9.9 percent returns are necessary when making the first withdrawal after three years.

In the last example, securities were initially purchased for 5,000, then for another 2,000 three months later. After a year, securities for 5,000 are sold. Selling the remaining securities after 18 months renders 6,000. The internal rate of return is 3.2 percent per month, or about 57 percent in 18 months.

See also the [fv](#), [nper](#), [npv](#), [pmt](#), and [pv](#) functions.

join

syntax: (join *list-of-strings* [*separator-string*])

Concatenates the given list of strings in *list-of-strings*. If *separator-string* is present, it is inserted between each string in the join.

example:

```
(set 'lst '("this" "is" "a" "sentence"))
(join lst " ") ⇒ "this is a sentence"
(join (map string (slice (now) 0 3)) "-") ⇒ "2003-11-26"
(join (explode "keep it together")) ⇒ "keep it together"
```

See also the [append](#), [string](#), and [explode](#) functions, which are the inverse of the join operation.

lambda

See the description of [fn](#) which is a shorter form of writing lambda.

lambda-macro

See the description of [define-macro](#).

lambda?

syntax: (lambda? *exp*)

Returns true only if the value of *exp* is a lambda expression and otherwise nil.

example:

```
(define (square x) (* x x))
(lambda? square) ⇒ true
```

See [define](#) and [define-macro](#) for more information about *lambda* expressions.

last

syntax: (last *list*)

syntax: (last *array*)

syntax: (last *str*)

Returns the last element of a list or a string.

example:

```
(last '(1 2 3 4 5)) ⇒ 5
(last '(a b (c d))) ⇒ (c d)

(set 'A (array 3 2 (sequence 1 6)))
⇒ ((1 2) (3 4) (5 6))
(last A) ⇒ (5 6)
```

In the second version the last character in the string *str* is returned as a string.

example:

```
(last "newLISP") ⇒ "P"
```

See also [first](#), [rest](#) and [nth](#).

legal?

syntax: (legal? *str*)

The token in *str* is verified as a legal newLISP symbol. Non legal symbols can be created using the [sym](#) function (e.g. symbols containing spaces, quotes, or other characters not normally allowed). Non legal symbols are created frequently when using them for associative data access:

example:

```
(symbol? (sym "one two")) ⇒ true
(legal? "one two") ⇒ nil ; contains a space
(set (sym "one two") 123) ⇒ 123
(eval (sym "one two")) ⇒ 123
```

The example shows that the string "one two" does not contain a legal symbol although a symbol can be created from this string and treated like a variable.

length

syntax: (length *expr*)

Returns the number of elements in a list, the number of rows in an array or the number of characters in a string.

length applied to a symbol returns the length of the symbol name. Applied to a number, length returns the number of bytes needed in memory to store that number: 4 for integers and 8 for floating point numbers.

example:

```
(length '(a b (c d) e))      ⇒ 4
(length '())                 ⇒ 0
(set 'someList '(q w e r t y)) ⇒ (q w e r t y)
(length someList)            ⇒ 6

(set 'ary (array 2 4 '(0)))  ⇒ ((1 2 3 4) (5 6 7 8))
(length ary)                 ⇒ 2

(length "Hello World")      ⇒ 11
(length "")                 ⇒ 0

(length 'someVar)           ⇒ 7
(length 123)                ⇒ 4
(length 1.23)               ⇒ 8
```

let

syntax: (let ((*sym1 exp-init1*) [(*sym2 exp-init2*) ...]) *body*)

syntax: (let (*sym1 exp-init1* [*sym2 exp-init2* ...]) *body*)

One or more variables *sym1*, *sym2*, ... are declared locally and initialized with expressions in *exp-init1*, *exp-init2*, etc. When the local variables are initialized the initializer expressions evaluate using symbol bindings as before the `let` statement. To incrementally use symbol bindings as evaluated during the initialization of locals in `let`, use [letn](#). One or more expressions in *exp-body* are evaluated using the local definitions of *sym1*, *sym2* etc. `let` is useful for breaking up complex expressions by defining local variables close to the place where they are used. The second form omits the parenthesis around the variable expression pairs but functions identical.

example:

```
(define (sum-sq a b)
  (let ((x (* a a)) (y (* b b)))
    (+ x y)))

(sum-sq 3 4) ⇒ 25

(define (sum-sq a b) ; alternative syntax
  (let (x (* a a) y (* b b))
    (+ x y)))
```


The variables `x` and `y` are initialized, then the expression `(+ x y)` is evaluated. The `let` form is just an optimized version and syntactic convenience for writing:

```
((lambda (sym1 [sym2 ...]) exp-body ) exp-init1 [ exp-init2 ])
```

See also [letn](#) for an incremental or nested form of `let`.

letex

syntax: `(letex ((sym1 exp-init1) [(sym2 exp-init2) ...]) body)`

syntax: `(letex (sym1 exp-init1 [sym2 exp-init2 ...]) body)`

This functions combines [let](#) and [expand](#) to expand local variables into an expression before evaluating it.

Both forms provide the same functionality, but in the second form the parentheses around the initializers can be omitted.

example:

```
(letex '(x 1 y 2 z 3) '(x y z))    ⇒ (1 2 3)
```

Before the expression `'(x y z)` gets evaluated, `x`, `y` and `z` are literally replaced with the initializers from the `letex` initializer list. The final expression which gets evaluated is `'(1 2 3)`.

The following is a more complex realistic example. `letex` and [define-macro](#) are used together to define a `dolist-while`, which loops through a list while certain condition is true:

example:

```
(define-macro (dolist-while)
  (letex (var (args 0 0)
          lst (args 0 1)
          cnd (args 0 2)
          body (cons 'begin (1 (args))))
    (let (res)
      (catch (dolist (var lst)
                     (if (set 'res cnd) body (throw res))))))

> (dolist-while (x '(a b c d e f) (!= x 'd)) (println x))
a
b
c
nil
>
```

The [args](#) function is used to access the unevaluated argument list from [define-macro](#).

letn

syntax: (letn ((*sym1 exp-init1*) [(*sym2 exp-init2*) ...]) *body*)

syntax: (letn (*sym1 exp-init1* [*sym2 exp-init2* ...]) *body*)

letn is like a *nested let* and works similar to [let](#), but will incrementally use the new symbol bindings when evaluating the initializer expressions as if several [let](#) were nested. The following comparison of [let](#) and letn show the difference:

example:

```
(set 'x 10)
(let ((x 1) (y (+ x 1)))
  (list x y))      ⇒ (1 11)

(letn ((x 1) (y (+ x 1)))
  (list x y))      ⇒ (1 2)
```

While in the first example using [let](#) the variable y is calculated using the binding of x before the [let](#) expression, in the second example using letn the variable y is calculated using the new local binding of x.

```
(letn (x 1 y x)
  (+ x y))          ⇒ 2

;; same as nested let's

(let (x 1)
  (let (y x)
    (+ x y)))        ⇒ 2
```

letn works like several *nested* [let](#). The parenthesis around the initializer expressions can be omitted.

list

syntax: (list *exp-1* [*exp-2* ...])

The *exp* are evaluated and the values used to construct a new list.

example:

```
(list 1 2 3 4 5)      ⇒ (1 2 3 4 5)
(list 'a '(b c) (+ 3 4) '() '*) ⇒ (a (b c) 7 () *)
```

See also [cons](#) and [push](#) for other forms of building lists.

list?

syntax: (list? *exp*)

Returns `true` only if the value of *exp* is a list; otherwise returns `nil`. Note that lambda and lambda-macro expressions are also recognized as special instances of a list expression.

example:

```
(set 'var '(1 2 3 4))    ⇒ (1 2 3 4)
(list? var)              ⇒ true

(define (double x) (+ x x))

(list? double)           ⇒ true
```

load

syntax: (load *str-file-name* [*str-file-name-2* ...] [*sym-context*])

Loads and translates newLISP from a source file specified in one or more *str-file-name* and evaluates the expressions contained in the file(s). When loading is successful `load` returns the result of the last expression in the last file evaluated. If a file cannot be loaded `load` throws an error.

An optional *sym-context* can be specified, which becomes the context of evaluation, unless such a context switch is already present in the file being loaded. By default, files which do not contain [context](#) switches will be loaded into the MAIN context.

The *str-file-name* specs can contain URLs. Both `http://` and `file://` URLs are supported.

example:

```
(load "myfile.lsp")

(load "a-file.lsp" "b-file.lsp")

(load "file.lsp" "http://mysite.org/mypro")

(load "http://192.168.0.21:6000//home/test/program.lsp")

(load "a-file.lsp" "b-file.lsp" 'MyCTX)

(load "file:///usr/share/newlisp/mysql5.lsp")
```

In case expressions evaluated during the `load` are changing the [context](#), this will not influence the programming module doing the `load`. The current [context](#) after the `load` statement will always be the same as before in the `load`.

Normal file specs and URLs can be mixed in the same `load` command.

`load` with HTTP URLs can also be used to load code remotely from newLISP server nodes running on UNIX like operating system. In this mode, `load` will issue an HTTP GET request to the target URL. Note that a double backslash is required when path names are specified relative to the root directory. `load` in HTTP mode will observe a 60-second timeout.

The second to last line causes the files to be loaded in to the context `MyCTX`. The quote forces the context to be created if it did not exist.

The `file://` URL is followed by a third `/` for the directory spec.

local

syntax: (local (*sym-1* [*sym-2* ...]) *body*)

Initializes one or more symbols in *sym-1*— to `nil`, evaluates the expressions in *body*, and returns the result of the last evaluation.

`local` works similarly to [let](#), but local variables are all initialized to `nil`.

`local` provides a simple way to localize variables without explicit initialization.

log

syntax: (log *num*)

syntax: (log *num num-base*)

In the first syntax the expression in *num* is evaluated and the natural logarithmic function is calculated from the result.

example:

```
(log 1)           ⇒ 0
(log (exp 1))    ⇒ 1
```

In the second syntax an arbitrary base can be specified in *num-base*.

example:

```
(log 1024 2)      ⇒ 10
(log (exp 1) (exp 1)) ⇒ 1
```

See also [exp](#), which is the inverse function to `log` with base *e*.

lookup

syntax: (lookup *exp assoc-list* [*int-index*])

`lookup`

Finds in *assoc-list* an *association* the *key* element of which has the same value as *exp* and returns the *int-index* element of *association* (or the last element if *int-index* is absent). See also [Indexing elements of strings and lists](#).

lookup is similar to [assoc](#) but goes one step further by extracting specific element found in the list.

example:

```
(set 'params '(
  (name "John Doe")
  (age 35)
  (gender "M")
  (balance 12.34)))

(lookup 'age params)    ⇒ 35

(set 'persons '(
  ("John Doe" 35 "M" 12.34)
  ("Mickey Mouse" 65 "N" 12345678)))

(lookup "Mickey Mouse" persons 2)    ⇒ "N"
(lookup "Mickey Mouse" persons -3)   ⇒ 65
(lookup "John Doe" persons 1)        ⇒ 35
(lookup "John Doe" persons -2)       ⇒ "M"
```

See also [assoc](#)

lower-case

syntax: (lower-case *str*)

Converts the characters of the string in *str* to lowercase. A new string is created, and the original is left untouched.

example:

```
(lower-case "HELLO WORLD") ⇒ "hello world"
(set 'Str "ABC")
(lower-case Str) ⇒ "abc"
Str              ⇒ "ABC"
```

See also the [upper-case](#) and [title-case](#) functions.

macro?

syntax: (macro? *exp*)

returns true if *exp* evaluates to a lambda-macro expression; otherwise, nil is returned.

example:

```
macro?
```

```
(define-macro (mysetq lv rv) (set lv (eval rv)))
(macro? mysetq) ⇒ true
```

main-args

syntax: (main-args)

syntax: (main-args *int-index*)

`main-args` returns a list with several string members, one for program invocation and one for each of the command-line arguments.

example:

```
newlisp 1 2 3

> (main-args)
("/usr/bin/newlisp" "1" "2" "3")
```

After `newlisp 1 2 3` is executed at the command prompt, `main-args` returns a list containing the name of the invoking program and three command-line arguments.

Optionally, `main-args` can take an *int-index* for indexing into the list:

```
newlisp a b c

> (main-args 0)
"/usr/bin/newlisp"
> (main-args -1)
"c"
> (main-args 2)
"b"
```

Note that when newLISP is executed from a script, `main-args` also returns the *name* of the script as the second argument:

```
#!/usr/bin/newlisp
#
# script to show the effect of 'main-args' in script file

(print (main-args) "\n")
(exit)

# end of script file

;; execute script in the OS shell:

script 1 2 3

("/usr/bin/newlisp" "./script" "1" "2" "3")
```

Try executing this script with different command-line parameters.

make-dir

syntax: (**make-dir** *str-dir-name* [*int-mode*])

Creates a directory as specified in *str-dir-name*, with the optional access mode *int-mode*. Returns `true` or `nil` depending on the outcome. If no access mode is specified, most UNIX systems default to `drwxr-xr-x`.

On UNIX systems, the access mode specified will also be masked by the OS's *user-mask* set by the system administrator. The *user-mask* can be retrieved on UNIX systems using the command `umask` and is usually `0022` (octal), which masks write (and creation) permission for non-owners of the file.

example:

```
;; 0 (zero) in front of 750 makes it an octal number
(make-dir "adir" 0750)
```

This example creates a directory named `adir` in the current directory with an access mode of `0750` (octal `750` = `drwxr-x---`).

map

syntax: (**map** *exp-functor* *list-args-1* [*list-args-2* ...])

Successively applies the primitive function, defined function, or lambda expression *exp-functor* to the arguments specified in *list-args-1*, *list-args-2*—, returning all results in a list.

example:

```
(map + '(1 2 3) '(50 60 70)) ⇒ (51 62 73)

(map if '(true nil true nil true) '(1 2 3 4 5) '(6 7 8 9 10))
⇒ '(1 7 3 9 5)

(map (fn (x y) (* x y)) '(3 4) '(20 10))
⇒ (60 40)
```

The second example shows how to dynamically create a function for `map`:

```
(define (foo op p)
  (append (lambda (x) (list (list op p 'x))))
```

We can also use the shorter `fn`:

```
(define (foo op p)
  (append (fn (x) (list (list op p 'x))))
```

`foo` now works like a function-maker:

```
(foo 'add 2) ⇒ (lambda (x) (add 2 x))
```

```
(map (foo add 2) '(1 2 3 4 5)) ⇒ (3 4 5 6 7 8)
(map (foo mul 3) '(1 2 3 4 5)) ⇒ (3 6 9 12 15)
```

Note that the quote before the operand can be omitted, since primitives evaluate to themselves in newLISP.

By incorporating map into the function definition, we can do the following:

```
(define (list-map op p lst)
  (map (lambda (x) (op p x)) lst))

(list-map + 2 '(1 2 3 4)) ⇒ (3 4 5 6)
(list-map mul 1.5 '(1 2 3 4)) ⇒ (1.5 3 4.5 6)
```

The number of arguments used is determined by the length of the first argument list. Arguments missing in other argument lists cause an error message. If an argument list contains too many elements, the extra ones will be ignored.

Note that only functions with *applicative order of evaluation* can be mapped. Functions with conditional or delayed evaluation of their arguments (e.g., [if](#) or [case](#)) cannot be mapped.

match

syntax: (match *list-pattern list-match* [*bool*])

The pattern in *list-pattern* is matched against the list in *list-match*, and the matching expressions are returned in a list. The three wildcard characters `?`, `+`, and `*` can be used in *list-pattern*.

Wildcard characters may be nested. `match` returns a list of matched expressions. For each `?` (question mark), a matching expression element is returned. For each `+` (plus sign) or `*` (asterisk), a list containing the matched elements is returned. If the pattern cannot be matched against the list in *list-match*, `match` returns `nil`.

Optionally, the boolean value `true` (or any other expression not evaluating to `nil`) can be supplied as a third argument. This causes `match` work as it did in versions prior to 8.2.3, showing all list elements in the returned result.

example:

```
(match '(a ? c) '(a b c)) ⇒ (b)
(match '(a ? ?) '(a b c)) ⇒ (b c)
(match '(a ? c) '(a (x y z) c)) ⇒ ((x y z))
(match '(a ? c) '(a x y z c)) ⇒ nil

(match '(a * c) '(a x y z c)) ⇒ ((x y z))
(match '(a (b c ?) x y z) '(a (b c d) x y z)) ⇒ (d)
```



```

(match '(a (*) x ? z) '(a (b c d) x y z)) ⇒ ((b c d) y)
(match '(+) '()) ⇒ nil
(match '(+) '(a)) ⇒ ((a))
(match '(+) '(a b)) ⇒ ((a b))
(match '(a (*) x ? z) '(a () x y z)) ⇒ (() y)
(match '(a (+) x ? z) '(a () x y z)) ⇒ nil

```

Note that the `*` operator tries to grab the fewest number of elements possible, but `match` backtracks and grabs more elements if a match cannot be found.

The `+` operator works similarly to the `*` operator, but it requires at least one list element.

The following example shows how the matched expressions can be bound to variables.

```

(map set '(x y) (match '(a (? c) d *) '(a (b c) d e f)))
x ⇒ b
y ⇒ (e f)

```

Note that `match` for strings has been eliminated. For more powerful string matching, use [regex](#), [find](#), [find-all](#) or [parse](#).

max

syntax: (`max` *num-1* [*num-2* ...])

Evaluates the expressions *num-1*— and returns the largest number.

example:

```
(max 4 6 2 3.54 7.1) ⇒ 7.1
```

See also the [min](#) function.

member

syntax: (`member` *exp list*)

syntax: (`member` *str str-key* [*num-option*])

In the first syntax, `member` searches for the element *exp* in the list *list*. If the element is a member of the list, a new list starting with the element found and the rest of the original list is constructed and returned. If nothing is found, `nil` is returned. When specifying *num-option* `member` performs a regular expression search.

example:

```
(set 'aList '(a b c d e f g h)) ⇒ (a b c d e f g h)
(member 'd aList)                ⇒ (d e f g h)
(member 55 aList)                ⇒ nil
```

In the second syntax, `member` searches for *str-key* in *str*. If *str-key* is found, all of *str* (starting with *str-key*) is returned. `nil` is returned if nothing is found.

example:

```
(member "LISP" "newLISP") ⇒ "LISP"
(member "LI" "newLISP")  ⇒ "LISP"
(member "" "newLISP")    ⇒ "newLISP"
(member "xyz" "newLISP") ⇒ nil
(member "li" "newLISP" 1) ⇒ "LISP"
```

See also the related functions [slice](#) and [find](#).

min

syntax: (min *num-1* [*num-2* ...])

Evaluates the expressions *num-1*— and returns the smallest number.

example:

```
(min 4 6 2 3.54 7.1) ⇒ 2
```

See also the [max](#) function.

mod

syntax: (mod *num-1* *num-2* [*num-3* ...])

Calculates the modular value of the numbers in *num-1* and *num-2*. `mod` computes the remainder from the division of the numerator *num-1* by the denominator *num-2* + 1. Specifically, the return value is *numerator* - *n* * *denominator*, where *n* is the quotient of the numerator divided by the denominator, rounded towards zero to an integer. The result has the same sign as the numerator and its magnitude is less than the magnitude of the denominator.

example:

```
(mod 10.5 3.3) ⇒ 0.6
(mod -10.5 3.3) ⇒ -0.6
```

Use the [%](#) (percent sign) function when working with integers only.

mul

syntax: (mul *num-1 num-2 [num-3 ...]*)

Evaluates all expressions *num-1*—, calculating and returning the product. `mul` can perform mixed-type arithmetic, but it always returns floating point numbers. Any floating point calculation with NaN also returns NaN.

example:

```
(mul 1 2 3 4 5 1.1) ⇒ 132
(mul 0.5 0.5)       ⇒ 0.25
```

multiply

syntax: (multiply *matrix-A matrix-B*)

Returns the matrix multiplication of matrices in *matrix-A* and *matrix-B*. If *matrix-A* has the dimensions *n* by *m* and *matrix-B* the dimensions *k* by *l* (*m* and *k* must be equal), the result is an *n* by *l* matrix. `multiply` can perform mixed-type arithmetic, but the results are always double precision floating points, even if all input values are integers.

The dimensions of a matrix are determined by the number of rows and the number of elements in the first row. For missing elements in non-rectangular matrices, 0.0 is assumed. A matrix can either be a nested list or [array](#).

example:

```
(set 'A '((1 2 3) (4 5 6)))
(set 'B '((1 2) (1 2) (1 2)))
(multiply A B) ⇒ ((6 12) (15 30))
```

All operations shown here on lists can be performed on arrays, as well.

See also the matrix operations [det](#), [transpose](#), and [invert](#).

name

syntax: (name *symbol [bool]*)

Returns as a string, the name of a *symbol* without the context prefix. If the expression in *bool* evaluates to anything other than `nil`, the name of the symbol's context is returned instead.

example:

```
(set 'CTX:var 123)
(set 'sm 'CTX:var)
(string sm)      ⇒ "CTX:var"
(name sm)        ⇒ "var"
```

```
(name sm true) ⇒ "ACTX"
```

NaN?

syntax: (NaN? *number*)

Tests if the result of a floating point math operation is a NaN. Certain floating point operations return a special IEEE 754 number format called a NaN for 'Not a Number'.

example:

```
(set 'x (sqrt -1)) ⇒ NaN
(add x 123)        ⇒ NaN
(mul x 123)        ⇒ NaN

(+ x 123) ⇒ 123
(* x 123) ⇒ 0

(> x 0) ⇒ nil
(<= x 0) ⇒ nil
(= x x) ⇒ true
(NaN? x) ⇒ true
```

Note that all floating point arithmetic operations with a NaN yield a NaN. All comparisons with NaN return nil, but true when comparing to itself. Comparison with itself, however, would result in *not* true when using ANSI C.

Integer operations treat NaN as 0 (zero) values.

net-accept

syntax: (net-accept *int-socket*)

Accepts a connection on a socket previously put into listening mode. Returns a newly created socket handle for receiving and sending data on this connection.

example:

```
(set 'socket (net-listen 1234))
(net-accept socket)
```

Note that for ports less than 1024, newLISP must be started in superuser mode on UNIX-like operating systems.

See also the [server](#) and [client](#) examples in the appendix.

net-close

syntax: (**net-close** *int-socket* [*true*])

Closes a network socket in *int-socket* that was previously created by a [net-connect](#) or [net-accept](#) function. Returns `true` on success and `nil` on failure.

example:

```
(net-close aSock)
```

The optional *true* flag suppresses immediate shutdown of sockets waiting for pending data transmissions to finish.

net-connect

syntax: (**net-connect** *str-remote-host* *int-port* [*str-mode* [*int-ttl*]])

Connects to a remote host computer specified in *str-remote-host* and a port specified in *int-port*. Returns a socket handle after having connected successfully; else it returns `nil`.

example:

```
(define (finger nameSite , socket buffer user site)
  (set 'user (nth 0 (parse nameSite "@")))
  (set 'site (nth 1 (parse nameSite "@")))
  (set 'socket (net-connect site 79))
  (if socket
    (net-send socket (append user "\r\n"))
    "no connection")
  (net-receive socket 'str 512)
  (print "\n" str "\n"))
```

The above program uses the *finger* service on a remote computer. This service returns information about an account holder on this computer. Some ISP and UNIX installations provide this service.

When executing:

```
(finger "johnDoe@someSite.com")
```

the program tries to connect to a server named "someSite.com" and sends the string "johnDoe". If "someSite.com" is running a finger service, it sends back information about the account "johnDoe" on this server. In case a connection cannot be made, the function returns the string "no connection."

`nameSite` is split up into the account name and host name parts. `net-connect` is used to connect to `someSite.com` and returns the socket handle, which processes incoming data.

UDP communications

As a third parameter, the string "udp" or "u" can be specified in the optional *str-mode* to create a socket suited for UDP (User Datagram Protocol) communications. In UDP mode, `net-connect` does *not* try to connect to the remote host, but only binds the socket to the remote address. A subsequent [net-send](#) will send a UDP packet containing that target address. Using [net-send-to](#) causes that address to be overwritten.

The functions [net-receive](#) and [net-receive-from](#) can also be used and will perform UDP communications. [net-select](#) and [net-peek](#) can be used to check for received data in a non-blocking fashion.

If data is never received when opening a client connection using `net-connect`, then calling [net-listen](#) with the "udp" option may be preferable for starting the client side of the connection. [net-listen](#) binds a specific local address and port to the socket. When `net-connect` is used, the local address and port will be picked by the socket-stack functions of the host OS.

UDP multicast communications

When specifying "multi" or "m" as a third parameter for *str-mode*, a socket for UDP multicast communications will be created. Optionally, the fourth parameter *int-ttl* can be specified as a TTL (time to live) value. If no *int-ttl* value is specified, a value of 3 is assumed.

Note that specifying UDP multicast mode in `net-connect` does not actually establish a connection to the target multicast address but only puts the socket into UDP multicasting mode. On the receiving side, use [net-listen](#) together with the UDP multicast option.

example:

```
;; example client
(net-connect "226.0.0.1" 4096 "multi") ⇒ 3
(net-send-to "226.0.0.1" 4096 "hello" 3)

;; example server
(net-listen 4096 "226.0.0.1" "multi") ⇒ 5
(net-receive-from 5 20)
⇒ ("hello" "192.168.1.94" 32769)
```

On the server side, [net-peek](#) or [net-select](#) can be used for non-blocking communications. In the above example, the server would block until a datagram is received.

The address 226.0.0.1 is just one multicast address in the Class D range of multicast addresses from 224.0.0.0 to 239.255.255.255.

The [net-send](#) and [net-receive](#) functions can also be used instead of [net-send-to](#) and [net-receive-from](#).

UDP broadcast communications

Specifying the string "broadcast" or "b" in the third parameter, *str-mode*, causes UDP broadcast communications to be set up. In this case, the broadcast address ending in 255 is used.

example:

```
;; example client
(net-connect "192.168.2.255" 3000 "broadcast") ⇒ 3
(net-send 3 "hello")

;; example server
(net-listen 3000 "" "udp") ⇒ 5
(net-receive 5 'buff 10)
buff ⇒ "hello"

;; or
(net-receive-from 5 10)
⇒ ("hello" "192.168.2.1" 46620)
```

Note that on the receiving side, [net-listen](#) should be used with the default address specified with an "" (empty string). Broadcasts will not be received when specifying an address. As with all UDP communications, [net-listen](#) does not actually put the receiving side in listen mode, but rather sets up the sockets for the specific UDP mode.

The [net-select](#) or [net-peek](#) functions can be used to check for incoming communications in a non-blocking fashion.

net-error

syntax: (net-error)

Retrieves the last error that occurred when calling a [net-*](#) function. When any of the following functions return nil, [net-error](#) can be called to get more information: [net-accept](#), [net-connect](#), [net-eval](#), [net-listen](#), [net-lookup](#), [net-receive](#), [net-receive-udp](#), [net-select](#), [net-send](#), [net-send-udp](#), and [net-service](#). Functions that communicate using sockets close the socket automatically and remove it from the [net-sessions](#) list. This makes for a very robust API in situations of unreliable net connections. Calling any of these functions successfully clears the last error.

The following messages are returned:

- 1: Cannot open socket
- 2: Host name not known
- 3: Not a valid service
- 4: Connection failed

```

5: Accept failed
6: Connection closed
7: Connection broken
8: Socket send() failed
9: Socket recv() failed
10: Cannot bind socket
11: Too many sockets in net-select
12: Listen failed
13: Badly formed IP
14: Select failed
15: Peek failed
16: Not a valid socket

```

example:

```

(net-connect "jhghjgkjhg" 80) ⇒ nil
(net-error) ⇒ (2 "ERR: Host name not known")

```

net-eval

syntax: (net-eval *str-host int-port str-expr* [*int-timeout* [*func-handler*]])

syntax: (net-eval '((*str-host int-port str-expr*) [(...) ...]) [*int-timeout*])

syntax: (net-eval '((*str-host int-port str-expr*) ...) *int-timeout func-handler*)

Can be used to evaluate source remotely on one or more newLISP servers. This function handles all communications necessary to connect to the remote servers, send source for evaluation, and wait and collect responses.

Beginning with version 8.9.8, *str-host*, *int-port*, and *str-expr* are evaluated. It is no longer necessary to specify them as constant. `net-eval` will evaluate these arguments.

The remote TCP/IP servers are started in the following way:

```

newlisp -c -d 4711 &

;; or with logging connections

newlisp -l -c -d 4711 &

```

Instead of 4711, any other port number can be used. Multiple nodes can be started on different hosts and with the same or different port numbers. The `-l` or `-L` logging options can be specified to log connections and remote commands.

The `-d` daemon mode allows newLISP to maintain state between connections. When keeping state between connections is not desired, the [inetd daemon mode](#) offers more advantages. The Internet `inetd` or `xinetd` services daemon will start a new newLISP process for each client connection. This makes for much faster servicing of multiple connections. In `-d` daemon mode, each new client request would have to wait for the previous request to be finished. See the chapter [inetd daemon mode](#) on how to configure this mode correctly.

In the first syntax, `net-eval` talks to only one remote newLISP server node, sending the host in *str-host* on port *int-port* a request to evaluate the expression *str-expr*. If *int-timeout*

is not given, `net-eval` will wait indefinitely for a response. Otherwise, if the timeout in milliseconds has expired, `nil` is returned; else, the evaluation result of *str-expr* is returned.

example:

```
(net-eval "192.168.1.94" 4711 "(+ 3 4)")      ⇒ 7
(net-eval "192.168.1.94" 4711 "(+ 3 4)" 1)    ⇒ nil ; timeout
to short
(net-eval "192.168.1.94" 4711 "(+ 3 4)" 1000) ⇒ 7
```

The second syntax of `net-eval` returns a list of the results after all of the responses are collected or timeout occurs. Responses that time out return `nil`. Connection errors or errors that occur when sending information to nodes are returned as a list of error numbers and descriptive error strings. See the function [net-error](#) for a list of potential error messages.

example:

```
(net-eval '(
  ("192.168.1.94" 4711 "(+ 3 4)")
  ("192.168.1.95" 4711 "(+ 5 6)")
) 5000)
⇒ (7 11)

(net-eval '(
  ("localhost" 8081 {(foo "abc")})
  ("localhost" 8082 "(myfunc 123)")
) 3000)

;; inetd or xinetd nodes on the same server and port

(net-eval '(
  ("localhost" 2000 {(foo "abc")})
  ("localhost" 2000 "(myfunc 123)")
) 3000)
```

The first example shows two expressions evaluated on two different remote nodes. In the second example, both nodes run on the local computer. This may be useful when debugging or taking advantage of multiple CPUs on the same computer.

When nodes are `inetd` or `xinetd`-controlled, several nodes may have the same node for the IP address and port number. In this case, the UNIX daemon `inetd` or `xinetd` will start multiple newLISP servers. This is useful when testing distributed programs on just one machine. The last example illustrates this case.

The source sent for evaluation can consist of entire multiline programs. This way, remote nodes can be loaded with programs first, then specific functions can be called. For large program files, the functions [put-url](#) or [save](#) (with a URL file name) can be used to transfer programs.

Optionally, a handler function can be specified. This function will be repeatedly called while waiting and once for every remote evaluation completion.

example:

```
(define (myhandler param)
  (if param
    (println param))
)
```

```
(set 'Nodes '(
  ("192.168.1.94" 4711)
  ("192.168.1.95" 4711)
))

(set 'Progs '(
  {(+ 3 4)}
  {(+ 5 6)}
))

(net-eval (map (fn (n p) (list (n 0) (n 1) p)) Nodes Progs)
  5000 myhandler)
⇒
("192.168.1.94" 4711 7)
("192.168.1.95" 4711 11)
```

The example shows how the list of node specs can be assembled from a list of nodes and sources to evaluate. This may be useful when connecting to a larger number of remote nodes. Since version 8.9.7, `net-eval` has also been able to evaluate the spec given in the node lists. This allows the following code:

```
(net-eval '(
  ((Nodes 0 0) (Nodes 0 1) (Progs 0))
  ((Nodes 1 0) (Nodes 1 1) (Progs 1))
) 3000 myhandler)
```

While waiting for input from remote hosts, `myhandler` will be called with `nil` as the argument to `param`. When a remote node result is completely received, `myhandler` will be called with `param` set to a list containing the remote host name or IP number, the port, and the resulting expression. `net-eval` will return `true` before a timeout or `nil` if the timeout was reached or exceeded. All remote hosts that exceeded the timeout limit will contain a `nil` in their results list.

Unless operating in raw mode, each piece of code sent to a remote node for evaluation should be one expression. This can be achieved by putting several statements into a [begin](#) block.

Raw mode

An additional parameter in each node specification can control whether the returned result is evaluated (the default behavior) or returned as a string as it comes over the communications channel. The following example illustrates the difference between the default *evaluated* and *raw* modes of `net-eval`:

```
(net-eval '(("localhost" 4711 {(+ 3 4)})) 1000) ⇒ (7)

(net-eval '(("localhost" 4711 {(+ 3 4)} true)) 1000) ⇒ ("7\n")

(net-eval '(("localhost" 4711 {(+ 3 4) (+ 5 6)})) 1000) ⇒ (11)

(net-eval '(("localhost" 4711 {(+ 3 4) (+ 5 6)} true)) 1000) ⇒
("7\n11\n")
```

While the evaluated mode always returns an evaluated expression, raw mode returns a string terminated by a line-feed. The last two statements reveal that in the default evaluated mode,

only the result of the last expression evaluation is returned, while in raw mode, both results are visible, each terminated by a line-feed.

Raw mode returns the same string as would be observed when entering expressions on the command line, while the evaluated mode returns LISP expressions ready for further newLISP processing. newLISP's `net-eval` protects the expression returned with a single quote before evaluating, thus ensuring that the expression string received is parsed in the receiving environment, but the resulting expression itself stays in the original form sent by the remote node. Only one quote gets prepended. For that reason, only one expression should be sent back when working in non-raw mode.

The following example shows this effect:

```
(set 'prog [text]
(list 1 2 3 4)
(list 'a 'b 'c)
[/text])

; raw mode
(net-eval '((host port prog true) ...))
⇒ ("(1 2 3 4)\n(a b c)\n")

; normal mode
(net-eval '((host port prog) ...))
invalid function in function net-eval : (a b c)

; brace statements with (begin ...)
(set 'prog [text]
(begin
  (list 1 2 3 4)
  (list 'a 'b 'c))
[/text])

; normal mode
(net-eval '((host port prog) ...))
⇒ ((a b c))
```

The `begin` in the definition of `prog` forces the return of only one expression, which then gets converted correctly by the receiving `net-eval`.

Note that *raw* mode has always been part of `net-eval`, but it was not documented prior to version 8.7.5.

net-listen

syntax: (`net-listen` *int-port* [*str-ip-addr*] [*str-mode*])

Listens on a port specified in *int-port*. A call to `net-listen` returns immediately with a socket number, which is then used by the blocking [net-accept](#) function to wait for a connection. As soon as a connection is accepted, [net-accept](#) returns a socket number that can be used to communicate with the connecting client.

example:

```
(set 'port 1234)
```

```
(set 'listen (net-listen port))
(unless listen (begin
  (print "listening failed\n")
  (exit)))
(print "Waiting for connection on: " port "\n")
(set 'connection (net-accept listen))
(if connection
  (while (net-receive connection 'buff 1024 "\n")
    (print buff)
    (if (= buff "\r\n") (exit)))
  (print "Could not connect\n"))
```

The example waits for a connection on port 1234, then reads incoming lines until an empty line is received. Note that listening on ports lower than 1024 may require superuser access on UNIX systems.

On computers with more than one interface card, specifying an optional interface IP address or name in *str-ip-addr* directs `net-listen` to listen on the specified address.

```
;; listen on a specific address
(net-listen port "192.168.1.54")
```

UDP communications

As a third parameter, the optional string "udp" or "u" can be specified in *str-mode* to create a socket suited for UDP (User Datagram Protocol) communications. A socket created in this way can be used directly with [net-receive-from](#) to await incoming UDP data *without* using `net-accept`, which is only used in TCP communications. The [net-receive-from](#) call will block until a UDP data packet is received. Alternatively, [net-select](#) or [net-peek](#) can be used to check for ready data in a non-blocking fashion. To send data back to the address and port received with [net-receive-from](#), use [net-send-to](#).

Note that [net-peer](#) will not work, as UDP communications do not maintain a connected socket with address information.

```
(net-listen 1002 "192.168.1.120" "udp")

(net-listen 1002 "" "udp")
```

The first example listens on a specific network adapter, while the second example listens on the default adapter. Both calls return a socket number that can be used in subsequent [net-receive](#), [net-receive-from](#), [net-send-to](#), [net-select](#), or [net-peek](#) function calls.

Both a UDP server *and* UDP client can be set up using `net-listen` with the "udp" option. In this mode, `net-listen` does not really *listen* as in TCP/IP communications; it just binds the socket to the local interface address and port.

For a working example, see [UDP client](#) and [UDP server](#) in the appendix of this manual.

Instead of `net-listen` and the "udp" option, the functions [net-receive-udp](#) and [net-send-udp](#) can be used for short transactions consisting only of one data packet.

`net-listen`, [net-select](#), and [net-peek](#) can be used to facilitate non-blocking reading. The listening/reading socket is not closed but is used again for subsequent reads. In contrast,

when the [net-receive-udp](#) and [net-send-udp](#) pair is used, both sides close the sockets after sending and receiving.

UDP multicast communications

If the optional string *str-mode* is specified as "multi" or "m", `net-listen` returns a socket suitable for multicasting. In this case, *str-ip-addr* contains one of the multicast addresses in the range 224.0.0.0 to 239.255.255.255. `net-listen` will register *str-ip-addr* as an address on which to receive multicast transmissions. This address should not be confused with the IP address of the server host.

example:

```
;; example client
(net-connect "226.0.0.1" 4096 "multi") ⇒ 3
(net-send-to "226.0.0.1" 4096 "hello" 3)

;; example server
(net-listen 4096 "226.0.0.1" "multi") ⇒ 5
(net-receive-from 5 20)
⇒ ("hello" "192.168.1.94" 32769)
```

On the server side, [net-peek](#) or [net-select](#) can be used for non-blocking communications. In the example above, the server would block until a datagram is received.

The [net-send](#) and [net-receive](#) functions can be used instead of [net-send-to](#) and [net-receive-from](#).

net-local

syntax: (net-local *int-socket*)

Returns the IP number and port of the local computer for a connection on a specific *int-socket*.

example:

```
(net-local 16) ⇒ ("204.179.131.73" 1689)
```

Use the [net-peer](#) function to access the remote computer's IP number and port.

net-lookup

syntax: `(net-lookup str-ip-number)`

syntax: `(net-lookup str-hostname [bool])`

Returns either a hostname string from *str-ip-number* in IP dot format or the IP number in dot format from *str-hostname*:

example:

```
(net-lookup "209.24.120.224")    ⇒ "www.nuevatec.com"
(net-lookup "www.nuevatec.com") ⇒ "209.24.120.224"

(net-lookup "216.16.84.66.sbl-xbl.spamhaus.org" true)
⇒ "127.0.0.2"
```

Optionally, a *bool* flag can be specified in the second syntax. If the expression in *bool* evaluates to anything other than `nil`, host-by-name lookup will be forced, even if the name string starts with an IP number.

net-peek

syntax: `(net-peek int-socket)`

Returns the number of bytes ready for reading on the network socket *int-socket*. If an error occurs or the connection is closed, `nil` is Returned.

example:

```
(set 'aSock (net-connect "aserver.com" 123))
(while ( = (net-peek aSock) 0) (do-something-else))
(net-receive aSock 'buff 1024)
```

After connecting, the program waits in a while loop until `aSock` can be read.

Use the [peek](#) function to check file descriptors and `stdin`.

net-peer

syntax: `(net-peer int-socket)`

Returns the IP number and port of the remote computer for a connection on *int-socket*.

example:

```
(net-peer 16) ⇒ ("192.100.81.100" 13)
```

Use the [net-local](#) function to access the local computer's IP number and port.

net-ping

syntax: (**net-ping** *str-address* [*int-timeout* [*int-response*]])

syntax: (**net-ping** *list-addresses* [*int-timeout* [*int-response*]])

In the first syntax, `net-ping` sends a ping ICMP 64-byte echo request to the address specified in *str-address*. If it is a broadcast address, the ICMP packet will be received by all addresses on the subnet. Note that for security reasons, many computers do not answer ICMP broadcast ping (ICMP_ECHO) requests. An optional timeout parameter can be specified in *int-timeout*. If no timeout is specified, a waiting time of 1000 milliseconds (one second) is assumed.

`net-ping` returns either a list of IP strings for which a response was received or an empty list if no response was received.

A return value of `nil` indicates a failure. Use the [net-error](#) function to retrieve the error message. If the message reads `Cannot open socket`, it is probably because newLISP is running without root permissions. newLISP can be started using:

```
sudo newlisp
```

Alternatively, newLISP can be installed with the set-user-ID bit set to run in superuser mode.

example:

```
(net-ping "newlisp.org")      ⇒ ("66.235.209.72")
(net-ping "127.0.0.1")       ⇒ ("127.0.0.1")
(net-ping "yahoo.com" 3000)  ⇒ nil
```

In the second syntax, `net-ping` is run in *batch mode*. Only one socket is opened in this mode, but multiple ICMP packets are sent out—one each to multiple addresses. In this case, multiple answers can be received.

To limit the number of responses to be waited for in broadcast or batch mode, an additional argument indicating the maximum number of responses to receive can be specified in *int-response*. Usage of this parameter can cause the function to return sooner than the specified timeout. When a given number of responses has been received, `net-ping` will return before the timeout has occurred.

example:

```
(net-ping '("newlisp.org" "yahoo.com" "192.168.1.255") 5000)

(net-ping "192.168.1.*" 500)
⇒ ("192.168.1.1" "192.168.1.2" "192.168.2.3" "192.168.2.254")

(net-ping "192.168.1.*" 500 2)
⇒ ("192.168.1.3" "192.168.1.1")
```

Broadcast or batch mode—as well as normal addresses and IP numbers or hostnames—can be mixed in one `net-ping` statement by putting all of the IP specs into a list.

The second and third line show how the batch mode of `net-ping` can be initiated by specifying the `*` (asterisk) as a wildcard character for the last subnet octet in the IP number. `net-ping` will iterate through all numbers from 1 to 254, sending an ICMP packet to each address. Note that this is different from the *broadcast* mode specified with an IP octet of 255. While in broadcast mode, `net-ping` sends out only one packet, which is received by multiple addresses. Batch mode explicitly generates multiple packets, one for each target address.

When sending larger lists of IPs in batch mode over one socket, a longer timeout may be necessary to allow enough time for all of the packets to be sent out over one socket. If the timeout is too short, the function `net-ping` may return `nil` with a message of `socket send failed`, which was returned by [net-error](#). In any case, `net-ping` will send out packages as quickly as possible.

This function is only available on UNIX-based systems and must be run in superuser mode.

net-receive

syntax: (`net-receive int-socket sym-buffer int-max-bytes [wait-string]`)

Receives data on the socket *int-socket* into a string contained in *sym-buffer*. A maximum of *int-max-bytes* is received. `net-receive` returns the number of bytes read. If there is a break in the connection, `nil` is returned. The space reserved in *sym-buffer* is exactly the size of bytes read.

Note that `net-receive` is a blocking call and does not return until the data arrives at *int-socket*. Use [net-peek](#) or [net-select](#) to find out if a socket is ready for reading.

Optionally, a *wait-string* can be specified as a fourth parameter. `net-receive` then returns after a character or string of characters matching *wait-string* is received. The *wait-string* will be part of the data contained in *sym-buffer*.

example:

```
(define (gettime)
  (net-connect "netcom.com" 13)
  (net-receive socket 'buf 256)
  (print buf "\n")
  (net-close socket))
```

When calling `gettime`, the program connects to port 13 of the server `netcom.com`. Port 13 is a date-time service on most server installations. Upon connection, the server sends a string containing the date and time of day.

```
(define (net-receive-line socket sBuff)
  (net-receive socket sBuff 256 "\n"))

(set 'bytesReceived (net-receive-line socket 'sm))
```

The second example defines a new function `net-receive-line`, which returns after receiving a newline character (a string containing one character in this example) or 256 characters. The `"\n"` string is part of the contents of `sBuff`.

Note that when the fourth parameter is specified, `net-receive` is slower than the normal version because information is read character by character. In most situations, the speed difference can be neglected.

net-receive-from

syntax: (net-receive-from *int-socket int-max-size*)

`net-receive-from` can be used to set up non-blocking UDP communications. The socket in *int-socket* must previously have been opened by either [net-listen](#) or [net-connect](#) (both using the "udp" option). *int-max-size* specifies the maximum number of bytes that will be received. On Linux/BSD, if more bytes are received, those will be discarded. On Win32, `net-receive-from` will return nil and close the socket.

example:

```
;; listen on port 1001 on the default address
(net-listen 1001 "" "udp") ⇒ 1980

;; optionally poll for arriving data with 100ms timeout
(while (not (net-select 1980 "r" 100000)) (do-something ...))

(net-receive-from 1980 20) ⇒ ("hello" "192.168.0.5" 3240)

;; send answer back to sender
(net-send-to "192.168.0.5" 3240 "hello to you" 1980)

(net-close 1980) ; close socket
```

The second line in this example is optional. Without it, the `net-receive-from` call would block until data arrives. A UDP server could be set up by listening and polling several ports serving them as they receive data.

Note that `net-receive` could not be used in this case because it does not return the sender's address and port information, which are required to talk back. In UDP communications, the data packet itself contains the address of the sender, *not* the socket over which communication takes place.

See also the [net-connect](#) function with the "udp" option and the [net-send-to](#) function for sending UDP data packets over open connections.

For blocking short UDP transactions, see the [net-send-udp](#) and [net-receive-udp](#) functions.

net-receive-udp

syntax: (net-receive-udp *int-port int-maxsize* [*int-microsec*][*str-addr-if*])

Receives a User Datagram Protocol (UDP) packet on port *int-port*, reading *int-maxsize* bytes. If more than *int-maxsize* bytes are received, bytes over *int-maxsize* are discarded on

Linux/BSD; on Win32, `net-receive-udp` returns `nil`. `net-receive-udp` blocks until a datagram arrives or the optional timeout value in *int-microsec* expires. When setting up communications between datagram sender and receiver, the `net-receive-udp` statement must be set up first.

No previous setup using `net-listen` or `net-connect` is necessary.

`net-receive-udp` returns a list containing a string of the UDP packet followed by a string containing the sender's IP number and the port used.

example:

```
;; wait for datagram with maximum 20 bytes
(net-receive-udp 1001 20)

;; or
(net-receive-udp 1001 20 5000000) ; wait for max 5 seconds

;; executed on remote computer
(net-send-udp "nuevatec.com" 1001 "Hello") ⇒ 4

;; returned from the net-receive-udp statement
⇒ ("Hello" "128.121.96.1" 3312)

;; sending binary information
(net-send-udp "ahost.com" 2222 (pack "c c c c" 0 1 2 3))
⇒ 4

;; extracting the received info
(set 'buff (first (net-receive-udp 2222 10)))

(print (unpack "c c c c" buff)) ⇒ (0 1 2 3)
```

See also the [net-send-udp](#) function for sending datagrams and the [pack](#) and [unpack](#) functions for packing and unpacking binary information.

To listen on a specified address on computers with more than one interface card, an interface IP address or name can be optionally specified in *str-addr-if*. When specifying *str-addr-if*, a timeout must also be specified in *int-wait*.

As an alternative, UDP communication can be set up using [net-listen](#), or [net-connect](#) together with the "udp" option to make non-blocking data exchange possible with [net-receive-from](#) and [net-send-to](#).

net-select

syntax: (net-select *int-socket str-mode int-micro-seconds*)

syntax: (net-select *list-sockets str-mode int-micro-seconds*)

In the first form, `net-select` finds out about the status of one socket specified in *int-socket*. Depending on *str-mode*, the socket can be checked if it is ready for reading or writing, or if the socket has an error condition. A timeout value is specified in *int-micro-seconds*.

In the second syntax, `net-select` can check for a list of sockets in *list-sockets*.

The following value can be given for *str-mode*:

"read" or "r" to check if ready for reading or accepting.
 "write" or "w" to check if ready for writing.
 "exception" or "e" to check for an error condition.

Read, send, or accept operations can be handled without blocking by using the `net-select` function. `net-select` waits for a socket to be ready for the value given in *int-micro-seconds*, then returns `true` or `nil` depending on the readiness of the socket. During the select loop, other portions of the program can run. On error, [net-error](#) is set. When `-1` is specified for *int-micro-seconds*, `net-select` will never time out.

example:

```
(set 'listen-socket (net-listen 1001))

;; wait for connection
(while (not (net-select listen-socket "read" 1000))
  (if (net-error) (print (net-error))))
(set 'connection (net-accept listen-socket))
(net-send connection "hello")

;; wait for incoming message
(while (not (net-select connection "read" 1000))
  (do-something))
(net-receive connection 'buff 1024)
```

When `net-select` is used, several listen and connection sockets can be watched, and multiple connections can be handled. When used with a list of sockets, `net-select` will return a list of ready sockets. The following example would listen on two sockets and continue accepting and servicing connections:

example:

```
(set 'listen-list '(1001 1002))

(while (not (net-error))
  (dolist (conn (net-select listen-list "r" 1000))
    (accept-connection conn)) ; build and accept-list

  (dolist (conn (net-select accept-list "r" 1000))
    (read-connection conn)) ; read on conn socket

  (dolist (conn (net-select accept-list "w" 1000))
    (write-connection conn)) ; write on conn socket
```

In the second syntax, a list is returned containing all the sockets that passed the test; if timeout occurred, an empty list is returned. An error causes [net-error](#) to be set.

Note that supplying a nonexistent socket to `net-select` will cause an error to be set in [net-error](#).

net-send

syntax: (net-send *int-socket sym-buffer* [*int-num-bytes*])

syntax: (net-send *int-socket str-buffer* [*int-num-bytes*])

Sends the contents of *sym-buffer* on the connection specified by *int-socket*. If *int-num-bytes* is specified, up to *int-num-bytes* are sent. If *int-num-bytes* is not specified, all contents in *sym-buffer* are sent. `net-send` returns the number of bytes sent or `nil` on failure.

rewrite;; Since `net-send` can use a string buffer directly, the symbol does not need to be quoted.

example:

```
(set 'buf "hello there")
(net-send sock 'buf)
(net-send sock 'buf 5)

;; a string buffer can be used unquoted
(net-send sock buf)
(net-send sock "bye bye")
```

The first `net-send` sends the string "hello there", while the second `net-send` sends only the string "hello".

net-send-to

syntax: (net-send-to *str-remotehost int-remoteport str-buffer int-socket*

Sends UDP data packets on open connections. The socket in *int-socket* must have previously been opened with a [net-connect](#) or [net-listen](#) function. Both functions must be opened with their "udp" option. The host in *str-remotehost* can be specified either as a hostname or as an IP-number string.

example:

```
(net-connect "asite.com" 1010 "udp")
⇒ 2058 ; get a UDP socket

(net-send-to "asite.com" 1010 "hello" 2058)

;; optionally poll for answer
(while (not (net-select 2058 "r" 100000))
  (do-something ...))

;; receive answering data from UDP server
(net-receive-from 2058 20)
⇒ ("hello to you" "10.20.30.40" 1010)

(net-close 2058)
```

The second line in the example is optional. Without it, the [net-receive-from](#) call would block until data arrives. Using polling, a client could maintain conversations with several UDP servers at the same time.

See also the [net-receive-from](#) function and the [net-listen](#) function with the "udp" option.

For blocking short UDP transactions, see [net-send-udp](#) and [net-receive-udp](#).

net-send-udp

syntax: (net-send-udp *str-remotehost* *int-remoteport* *str-buffer* [*bool*])

Sends a User Datagram Protocol (UDP) to the host specified in *str-remotehost* and to the port in *int-remoteport*. The data sent is in *str-buffer*.

No previous setup using `net-connect` or `net-listen` is necessary. `net-send-udp` returns immediately with the number of bytes sent and closes the socket used. If no `net-receive-udp` statement is waiting at the receiving side, the datagram sent is lost. When using datagram communications over insecure connections, setting up a simple protocol between sender and receiver is recommended for ensuring delivery. UDP communication by itself does not guarantee reliable delivery as TCP/IP does.

example:

```
(net-send-udp "somehost.com" 3333 "Hello") ⇒ 5
```

`net-send-udp` is also suitable for sending binary information (e.g., the zero character or other nonvisible bytes). For a more comprehensive example, see [net-receive-udp](#).

Optionally, the sending socket can be put in broadcast mode by specifying `true` or any expression not evaluating to `nil` in *bool*:

```
(net-send-udp "192.168.1.255" 2000 "Hello" true) ⇒ 5
```

The UDP will be sent to all nodes on the 192.168.1 network. Note that on some operating systems, sending the network mask 255 without the *bool* `true` option will enable broadcast mode.

As an alternative, the [net-connect](#) function using the "udp" option—together with the [net-send-to](#) function—can be used to talk to a UDP listener in a non-blocking fashion.

net-service

syntax: (net-service *str-service* *str-protocol*)

Makes a lookup in the *services* database and returns the standard port number for this service. Returns `nil` on failure.

example:

```
net-service
```

```
(net-service "ftp" "tcp")      ⇒ 21
(net-service "finger" "tcp")  ⇒ 79
(net-service "net-eval" "tcp") ⇒ 4711 ; if configured
```

net-sessions

syntax: (net-sessions)

Returns a list of active listening and connection sockets.

new

syntax: (new *context-source* *sym-context-target* [*bool*])

syntax: (new *context-source*)

In the first syntax, *context-source* is the name of an existing context, and *sym-context-target* is the name of a new context to be created just like the original, with the same variable names and user-defined functions. If the context in *sym-context-target* already exists, then new symbols and definitions are added. Existing symbols are overwritten when the expression in *bool* evaluates to anything besides *nil*; otherwise, the content of existing symbols will remain. This makes *mixins* of context objects possible. *new* returns the target context, which cannot be MAIN.

In the second syntax, the existing context in *context-source* gets copied into the current context as the target context.

All references to symbols in the originating context will be translated to references in the target context. This way, all functions and data structures referring to symbols in the original context will now refer to symbols in the target context.

example:

```
(new CTX 'CTX-2) ⇒ CTX-2

;; force overwrite of existing symbols
(new CTX MyCTX true) ⇒ MyCTX

(set 'CTX:x 123)
(new CTX) ⇒ MAIN ; copies x into MAIN
x        ⇒ 123

(map new '(Ct-a Ct-b Ct-c)) ; merge into current context
```

The first line in the example creates a new context called CTX-2 that has the exact same structure as the original one. Note that CTX is not quoted because contexts evaluate to themselves, but CTX-2 must be quoted because it does not exist yet.

The second line merges the context CTX into MyCTX. Any existing symbols of same name in MyCTX will be overwritten. Because MyCTX already exists, the quote before the context symbol can be omitted.

The last lines show how a foreign context gets merged into the current one and how [map](#) can be used to merge a list of contexts.

Context symbols need not be mentioned explicitly, but they can be contained in variables:

example:

```
(set 'foo:x 123)
(set 'bar:y 999)

(set 'ctxa foo)
(set 'ctxb bar)

(new ctxa ctxb) ; from foo to bar

bar:x ⇒ 123 ; x has been added to bar
bar:y ⇒ 999)
```

The example refers to contexts in variables and merges context foo into bar.

See also the function [def-new](#) for moving and merging single functions instead of entire contexts. See the [context](#) function for a more comprehensive example of new.

nil?

syntax: (nil? *expr*)

If the expression in *expr* evaluates to nil, then nil? returns true; otherwise, it returns nil.

example:

```
(map nil? '(x nil 1 nil "hi" ()))
⇒ (nil true nil true nil nil)

(nil? nil) ⇒ true
(nil? '()) ⇒ nil
```

The nil? predicate is useful for distinguishing between nil and the empty list ().

not

syntax: (not *exp*)

If *exp* evaluates to nil, then true is returned; otherwise, nil is returned.

example:

not

```

(not true)           ⇒ nil
(not nil)            ⇒ true
(not '())            ⇒ true
(not (< 1 10))       ⇒ nil
(not (not (< 1 10))) ⇒ true

```

normal

syntax: (normal *float-mean* *float-stdev* *int-n*)

syntax: (normal *float-mean* *float-stdev*)

In the first form, `normal` returns a list of length *int-n* of random, continuously distributed floating point numbers with a mean of *float-mean* and a standard deviation of *float-stdev*. The random generator used internally can be seeded using the [seed](#) function.

example:

```

(normal 10 3 10)
⇒ (7 6.563476562 11.93945312 6.153320312 9.98828125
   7.984375 10.17871094 6.58984375 9.42578125 12.11230469)

```

In the second form, `normal` returns a single normal distributed floating point number:

```

(normal 0 1) ⇒ 0.6630859375

```

See also the [random](#) and [rand](#) functions for evenly distributed numbers, [amb](#) for randomizing evaluation in a list of expressions, and [seed](#) for setting a different start point for pseudo random number generation.

now

syntax: (now [*int-offset*])

Returns information about the current date and time as a list of integers. An optional time-zone offset can be specified in minutes in *int-offset*. This causes the time to be shifted forward or backward in time, before being split into separate date values.

example:

```

(now) ⇒ (2002 2 27 18 21 30 140000 57 3 300 0)
(apply date-value (now)) ⇒ 1014834090

```

The numbers represent the following date-time fields:

<i>format</i>	<i>description</i>
year	Gregorian calendar
month	(1–12)

day	(1–31)
hour	(0–23) UCT
minute	(0–59)
second	(0–59)
microsecond	(0–999999) OS-specific, millisecond resolution
day of current year	Jan 1st is 1
day of current week	(1–7) starting Sunday
time zone offset in minutes	west of GMT
daylight savings time flag	(0–1) on Linux/UNIX bias and in minutes on Win32

The second example returns the UCT time value of seconds after January 1, 1970.

Ranging from 0 to 23, hours are given in Coordinated Universal Time (UCT) and are not adjusted for the local time zone. The resolution of the microsecond field depends on the operating system and platform. On some platforms, the last three digits of the microseconds field are always 0 (zero).

See also the [date](#), [date-value](#), [time](#), and [time-of-day](#) functions.

Note that on Solaris, the returned time offset value is not working correctly in some versions/platforms and may contain garbage values.

On many platforms, the daylight savings flag is not active and returns 0 (zero) even during daylight savings time.

null?

syntax: (null? expr)

Checks if an expression evaluates to `nil`, the empty list `()`, the empty string `" "`, `NaN` (not a number), or 0 (zero), in which case it returns `true`. In all other cases, `null?` returns `nil`. The predicate `null?` is useful in conjunction with the functions [filter](#) or [clean](#) to check the outcome of other newLISP operations.

example:

```
(map null? '(1 0 0.0 2 NaN "hello" "" (a b c) () nil true))
⇒ (nil true true true nil true nil true nil true true nil)

(filter null? '(1 0 2 0.0 NaN "hello" "" (a b c) () nil true))
⇒ (0 0 NaN "" () nil)

(clean null? '(1 0 2 0.0 NaN "hello" "" (a b c) () nil true))
⇒ (1 2 "hello" (a b c) true)
```

See also the predicates [empty?](#), [nil?](#) and [zero?](#).

nper

syntax: (**nper** *num-interest num-pmt num-pv* [*num-fv int-type*])

Calculates the number of payments required to pay a loan of *num-pv* with a constant interest rate of *num-interest* and payment *num-pmt*. If *num-fv* is omitted, the future value of the loan is assumed to be 0.0. If payment is at the end of the period, *int-type* is 0; else it is 1. If *int-type* is omitted, 0 is assumed.

example:

```
(nper (div 0.07 12) 775.30 -100000) ⇒ 239.9992828
```

The example calculates the number of monthly payments required to pay a loan of \$100,000 at a yearly interest rate of 7 percent with payments of \$775.30.

See also the [fv](#), [irr](#), [npv](#), [pmt](#), and [pv](#) functions.

npv

syntax: (**npv** *num-interest list-values*)

Calculates the net present value of an investment with a fixed interest rate *num-interest* and a series of future payments and income in *list-values*. Payments are represented by negative values in *list-values*, while income is represented by positive values in *list-values*.

example:

```
(npv 0.1 '(1000 1000 1000))
⇒ 2486.851991

(npv 0.1 '(-2486.851991 1000 1000 1000))
⇒ -1.434386832e-08 ; ~ 0.0 (zero)
```

In the example, an initial investment of \$2,481.85 would allow for an income of \$1,000 after the end of the first, second, and third years.

See also the [fv](#), [irr](#), [nper](#), [pmt](#), and [pv](#) functions.

nth

syntax: (**nth** *int-index-1* [*int-index-2 ...*] *list*)

syntax: (**nth** *int-index-1* [*int-index-2 ...*] *array*)

syntax: (**nth** *int-offset str*)

In the first version, *nth* evaluates *int-index* and uses it as an index into *list*, returning the element found at that index. See also [Indexing elements of strings and lists](#). Multiple indices

may be specified to recursively access elements in nested lists. If there are more indices than nesting levels, the extra indices are ignored. Up to 16 indices can be specified.

example:

```
(nth 0 '(a b c)) ⇒ a

(set 'names '(john martha robert alex))
⇒ (john martha robert alex)

(nth 2 names) ⇒ robert

(nth -1 names) ⇒ alex

(set 'persons '((john 30) (martha 120) ((john doe) 17)))

(nth 1 1 persons) ⇒ 120

(nth 2 0 1 persons) ⇒ doe

(nth -2 0 persons) ⇒ martha

(nth 10 persons) ⇒ ((john doe) 17) ; out of bounds
(nth -5 person) ⇒ (john 30) ; out of bounds
```

In the second version, `nth` works on [arrays](#) just like it does on lists, but out-of-bounds indices will cause an error message.

example:

```
(set 'aArray (array 2 3 '(a b c d e f)))
⇒ ((a b c) (d e f))

(nth 1 aArray) ⇒ (d e f)

(nth 1 0 aArray) ⇒ d

(nth -5 -3 aArray) ⇒ out of bounds error

(nth 10 10 aArray) ⇒ out of bounds error
```

In the third version, `nth` returns the character found at the position *int-index* in *str* and returns it as a string.

example:

```
(nth 0 "newLISP") ⇒ "n"

(nth 3 "newLISP") ⇒ "L"

(nth -1 "newLISP") ⇒ "P"
```

See also the [set-nth](#) function for accessing multidimensional nested lists and arrays. See the [push](#) and [pop](#) functions for accessing multidimensional lists.

Note that beginning with version 8.5, [implicit indexing](#) can be used as a shorter form of [nth](#). *Implicit indexing* is slightly faster than conventional indexing and can take an unlimited number of indices.

Note also that [nth](#) works on character boundaries rather than byte boundaries when using the UTF-8-enabled version of newLISP.

nth-set

syntax: (nth-set *int-nth-1* [*int-nth-2* ...] *list|array exp-replacement*)

syntax: (nth-set *int-nth-1* *str str-replacement*)

syntax: (nth-set (*list|array int-nth-1* [*int-nth-2* ...]) *exp-replacement*)

syntax: (nth-set (*str int-nth-1*) *str str-replacement*)

Sets the *int-nth* element of a list or array with the evaluation of *exp-replacement* and returns the old element. As shown in the last two syntax lines, [implicit indexing](#) syntax can be used for specifying indices. Because it is more readable, implicit indexing is the preferred form (since version 8.8.8) in *nth-set* and [set-nth](#), but both forms remain valid.

nth-set performs a destructive operation, changing the original list or array. More than one index can be specified to recursively traverse nested list structures or multidimensional arrays. An out-of-bounds index always returns the last or first element when indexing a list, but it causes an out-of-bound error when indexing an array. Up to 16 indices can be specified.

When replacing in lists, the old element is also contained in the system variable `$0` and can be used in the replacement expression itself.

In the second form, the *int-nth* character in *str* is replaced with the string in *str-replacement*. Out-of-bounds indices will pick the first or last character for replacement, and the system variable `$0` is set to the replaced character.

example:

```
;;;;;;;;;;;;; usage on lists ;;;;;;;;;;;;;;

(set 'aList '(a b c d e f g))
(nth-set 2 aList "I am the replacement") ⇒ c ; old syntax

;; or using implicit indexing

(nth-set (aList 2) "I am the replacement") ⇒ c ; new syntax

aList ⇒ (a b "I am the replacement" d e f g)

$0 ⇒ c

(set 'aList '(a b (c d (e f g) h) i)))
(nth-set (aList 2 2 0) 'x) ⇒ e

aList ⇒ (a b (c d (x f g) h) i)
$0 ⇒ e

(set-nth (aList -2 2 -1) 99) ⇒ g

aList ⇒ (a b (c d (x f 99) h) i)

;; usage on default functors
```

```
(set 'db:db '(a b c d e f g))

(nth-set (db 3) 99) ⇒ d

db:db ⇒ (a b c 99 e f g)
```

The following examples use `nth-set` to change the contents of [arrays](#).

example:

```
;;;;;;;;;;;; usage on arrays ;;;;;;;;;;;;;;

(set 'myarray (array 3 4 (sequence 1 12)))
⇒ ((1 2 3 4) (5 6 7 8) (9 10 11 12))

(nth-set 2 3 myarray 99) ⇒ 12
;; or with implicit indexing
(nth-set (myarray 2 3) 99) ⇒ 12
myarray
⇒ ((1 2 3 4) (5 6 7 8) (9 10 11 99))

(nth-set (myarray -2 1) "hello") ⇒ 6
myarray
⇒ ((1 2 3 4) (5 "hello" 7 8) (9 10 11 99))

(nth-set (myarray 1) (array 4 '(a b c d)))
⇒ (5 "hello" 7 8)
myarray
⇒ ((1 2 3 4) (a b c d) (9 10 11 99))

;; usage on default functors
(set 'myarray:myarray (array 7 '(a b c d e f g)))

(nth-set (myarray 3) 99) ⇒ d
myarray:myarray ⇒ (a b c 99 e f g)
```

When replacing whole rows as in the third example, care must be taken to replace it as an array. See also the array functions [array](#), [array?](#), and [array-list](#).

In second form, the *int-nth* character in *str* is replaced with the string in *str-replacement*.

example:

```
;;;;;;;;;;;; usage on strings ;;;;;;;;;;;;;;

(set 's "abcd")

(nth-set (s 0) "xyz") ⇒ "a"
s ⇒ "xyzbcd"
$0 ⇒ "a"
```

`nth-set` uses the system variable `$0` for the element found in lists and strings. This can be used in the replacement expression:

```
(set 'lst '(1 2 3 4))

(nth-set (lst 1) (+ $0 1)) ⇒ 2

lst ⇒ '(1 3 3 4)
```

See the [set-nth](#) function, which works like `nth-set` but returns the whole changed expression instead of the replaced element. `set-nth` is also slower when doing replacements in larger lists or string buffers.

Use the [nth](#), [push](#), and [pop](#) functions to access multidimensional nested lists. The [nth](#) function will also work with multidimensional nested arrays.

`nth-set` works exactly like [set-nth](#) but returns the replaced element instead of the whole changed list expression. `nth-set` is much faster when replacing elements in larger lists or arrays.

number?

syntax: (number? exp)

`true` is returned only if `exp` evaluates to a floating point number or an integer; otherwise, `nil` is returned.

example:

```
(set 'x 1.23)
(set 'y 456)
(number? x)      ⇒ true
(number? y)      ⇒ true
(number? "678")  ⇒ nil
```

See the functions [float?](#) and [integer?](#) to test for a specific number type.

open

syntax: (open str-path-file str-access-mode [str-option])

The *str-path-file* is a file name, and *str-access-mode* is a string specifying the file access mode. `open` returns an integer, which is a file handle to be used on subsequent read or write operations on the file. On failure, `open` returns `nil`. The access mode `"write"` creates the file if it doesn't exist, or it truncates an existing file to 0 (zero) bytes in length.

The following strings are legal access modes:

```
"read" or "r" for read only access
"write" or "w" for write only access
"update" or "u" for read/write access
"append" or "a" for append read/write access
```

example:

```
(device (open "newfile.data" "write")) ⇒ 5
(print "hello world\n") ⇒ "hello world"
(close (device)) ⇒ 5
```

```
(set 'aFile (open "newfile.data" "read"))
(seek aFile 6)
(set 'inChar (read-char aFile))
(print inChar "\n")
(close aFile)
```

The first example uses `open` to set the device for [print](#) and writes the word "hello world" into the file `newfile.data`. The second example reads a byte value at offset 6 in the same file (the ASCII value of 'w' is 119). Note that using `close` on [\(device\)](#) automatically resets [device](#) to 0 (zero).

As an additional *str-option*, "non-block" or "n" can be specified after the "read" or "write" option. Only available on UNIX systems, non-blocking mode can be useful when opening *named pipes* but is not required to perform I/O on named pipes.

To create a named pipe in newLISP, use the [exec](#) or [import](#) function:

example:

```
(exec "mkfifo myfifo")

;; or alternatively

(import "/lib/libc.so.6" "mkfifo")
(mkfifo "/tmp/myfifo" 0777)
```

The named pipe can now be used like a file with [open](#), [read-buffer](#), and [write-buffer](#).

or

syntax: (or *exp-1* [*exp-2 exp-3 ...*])

Evaluates expressions *exp-x* from left to right until finding a result that does not evaluate to `nil` or the empty list `()`. The result is the return value of the `or` expression.

example:

```
(set 'x 10)
(or (> x 100) (= x 10))      ⇒ true
(or "hello" (> x 100) (= x 10)) ⇒ "hello"
(or '())                    ⇒ nil
(or true)                   ⇒ true
(or)                        ⇒ nil
```

pack

syntax: (pack *str-format exp-1* [*exp-2 ... exp-n*])

Packs one or more expressions (*exp-1* to *exp-n*) into a binary format specified in the format string *str-format*, returning the binary structure in a string buffer. A symmetrical [unpack](#) function is used to unpack. `pack` and `unpack` are useful when reading and writing binary files (see [read-buffer](#) and [write-buffer](#)) or when unpacking binary structures from return values of imported C functions with `import`.

The following characters are used in *str-format*:

<i>format</i>	<i>description</i>
t	
c	a signed 8-bit number
b	an unsigned 8-bit number
d	a signed 16-bit short number
u	an unsigned 16-bit short number
ld	a signed 32-bit long number
lu	an unsigned 32-bit long number
f	a float in 32-bit representation
lf	a double float in 64-bit representation
sn	a string of <i>n</i> null padded ASCII characters
nn	<i>n</i> null characters
>	switch to big endian byte order
<	switch to little endian byte order

Note that newLISP only supports 32-bit signed integers and treats `lu` and `ld` the same way internally.

`pack` will convert all floats to integer types when passed to `b`, `c`, `d`, `ld`, or `lu` formats. `pack` will also convert from integer to float type when passing integers to `f` and `lf` formats.

example:

```
(pack "c c c" 65 66 67) ⇒ "ABC"
(unpack "c c c" "ABC") ⇒ (65 66 67)

(pack "c c c" 0 1 2) ⇒ "\000\001\002"
(unpack "c c c" "\000\001\002") ⇒ (0 1 2)

(set 's (pack "c d u" 10 12345 56789))
(unpack "c d u" s) ⇒ (10 12345 56789)

(set 's (pack "s10 f" "result" 1.23))
(unpack "s10 f" s)
⇒ ("result\000\000\000\000" 1.230000019)

(set 's (pack "s3 lf" "result" 1.23))
(unpack "s3 f" s) ⇒ ("res" 1.23)

(set 's (pack "c n7 c" 11 22))
(unpack "c n7 c" s) ⇒ (11 22)

(unpack "b" (pack "b" -1.0)) ⇒ (255)
(unpack "f" (pack "f" 123)) ⇒ (123)
```


The last two statements show how floating point numbers are converted into integers when required by the format spec.

The > and < specifiers can be used to switch between *little endian* and *big endian* byte order when packing or unpacking:

```
(pack "d" 1)    ⇒ "\001\000"    ;; on little endian CPU
(pack ">d" 1)   ⇒ "\000\001"    ;; force big endian

(pack "ld" 1)   ⇒ "\001\000\000\000" ;; on little endian CPU
(pack "<ld" 1)  ⇒ "\000\000\000\001" ;; force big endian
```

Switching the byte order will affect all number formats with 16-, 32-, or 64-bit sizes.

The pack and unpack format need not be the same:

```
(set 's (pack "s3" "ABC"))
(unpack "c c c" s) ⇒ (65 66 67)
```

The examples show spaces between the format specifiers. These are not required but can be used to improve readability.

To pack 64-bit integers, pack two 32-bit numbers:

```
(pack "lld" big (>> big 32)) ; for little endian byte order
(pack "lld" (>> big 32) big)  ; for big endian byte order
```

See also the [address](#), [get-int](#), [get-char](#), [get-string](#), and [unpack](#) functions.

parse

syntax: (parse *str-data* [*str-break* *int-option*])

Breaks the string that results from evaluating *str-data* into string tokens, which are then returned in a list. When no *str-break* is given, *parse* tokenizes according to newLISP's internal parsing rules. A string may be specified in *str-break* for tokenizing only at the occurrence of a string. If an *int-option* number is specified, a regular expression pattern may be used in *str-break*.

When *str-break* is not specified, the maximum token size is 2048 for quoted strings and 256 for identifiers. In this case, newLISP uses the same faster tokenizer it uses for parsing LISP source. If *str-data* is specified, there is no limitation on the length of tokens. A different algorithm is used that splits the source string *str-data* at the string in *str-break*.

example:

```
(parse "hello how are you") ⇒ ("hello" "how" "are" "you")

(parse "one:two:three" ":") ⇒ ("one" "two" "three")

(parse "one--two--three" "--")
⇒ ("one" "two" "three")

(parse "one-two--three---four" "-+" 0)
```

```
⇒ ("one" "two" "three" "four")

(parse "hello regular  expression 1, 2, 3" {,\s*|\s+} 0)
⇒ ("hello" "regular" "expression" "1" "2" "3")
```

The last two examples show a regular expression as the break string with the default option 0 (zero). Instead of { and } (left and right curly brackets), quotes can be used to limit the pattern. In this case, double backslashes must be used inside the pattern. The last pattern could be used for parsing CVS files. For the regular expression option numbers, see [regex](#).

parse will return empty fields around separators as empty strings:

```
(parse "1,2,3," ",") ⇒ ("1" "2" "3" "")
(parse "1,,,4" ",") ⇒ ("1" "" "" "4")
(parse ", " ",") ⇒ (" " "")

(parse "") ⇒ ()
(parse " " " ") ⇒ ()
```

This behavior is needed when parsing records with empty fields.

Parsing an empty string will always result in an empty list.

Use the [regex](#) function to break strings up and the [directory](#), [find](#), [find-all](#), [regex](#), [replace](#), and [search](#) functions for using regular expressions.

peek

syntax: (peek *int-handle*)

Returns the number of bytes ready to be read on a file descriptor; otherwise, it returns nil if the file descriptor is invalid. peek can also be used to check stdin. This function is only available on UNIX-like operating systems.

example:

```
(peek 0) ; check # of bytes ready on stdin
```

Use the [net-peek](#) function to check for network sockets, or for the number of available bytes on them. On UNIX systems, [net-peek](#) can be used to check file descriptors. The difference is that [net-peek](#) also sets [net-error](#).

pipe

syntax: (pipe)

Creates an inter-process communications pipe and returns the read and write handles to it within a list.

example:

```
(pipe) ⇒ (3 4) ; 3 for read, 4 for writing
```

The pipe handles can be passed to a child process or thread launched via [process](#) or to [fork](#) for inter-process communications.

Note that the pipe does not block when being written to, but it does block reading until bytes are available. A [read-line](#) blocks until a newline character is received. A [read-buffer](#) blocks when fewer characters than specified are available from a pipe that has not had the writing end closed by all processes.

More than one pipe can be opened if required.

newLISP can also use *named pipes*. See the [open](#) function for further information.

pmt

syntax: (parse *num-interest num-periods num-principal* [*num-future-value int-type*])

Calculates the payment for a loan based on a constant interest of *num-interest* and constant payments over *num-periods* of time. *num-future-value* is the value of the loan at the end (typically 0.0). When paying at the end of each period, *num-type* is 0 (zero); otherwise, it is 1. If omitted, *int-type* is assumed to be 0 (zero) for payment at the end of a period.

example:

```
(pmt (div 0.07 12) 240 100000) ⇒ -775.2989356
```

The above example calculates a payment of \$775.30 for a loan of \$100,000 at a yearly interest rate of 7 percent. It is calculated monthly and paid over 20 years (20 * 12 = 240 monthly periods). This illustrates the typical way payment is calculated for mortgages.

See also the [fv](#), [irr](#), [nper](#), [npv](#), and [pv](#) functions.

pop

syntax: (pop *list* [*int-index-1* [*int-index-2* ...]])

syntax: (pop *list* [*list-indexes*])

syntax: (pop *str* [*int-index*] [*int-length*])

Extracts an element from the list found by evaluating *list*. If a second parameter is present, the element at *int-index* is extracted and returned. See also [Indexing elements of strings and lists](#).

In the second version, indices are specified in the list *list-indices*. This way, `pop` works easily together with [ref](#) and [ref-all](#), which return lists of indices.

`pop` changes the contents of the target list. The popped element is returned.

example:

```
(set 'pList '((f g) a b c "hello" d e 10))

(pop pList)    ⇒ (f g)
(pop pList)    ⇒ a
pList         ⇒ (b c "hello" d e 10)

(pop pList 3)   ⇒ d
(pop pList 100) ⇒ 10
pList          ⇒ (b c "hello" e)

(pop pList -1)  ⇒ e
pList          ⇒ (b c "hello")

(pop pList -2)  ⇒ c
pList          ⇒ (b "hello")

(set 'pList '(a 2 (x y (p q) z)))

(pop pList -1 2 0) ⇒ p

;; use indices in a list
(set 'pList '(a b (c d () e)))

(push 'x pList '(2 2 0)) ⇒ x
pList
⇒ (a b (c d (x) e))

(ref 'x pList) ⇒ (2 2 0)

(pop pList '(2 2 0)) ⇒ x

;; use pop on strings

(set 'str "newLISP")

(pop str -4 4) ⇒ "LISP"

str ⇒ "new"

(pop str 1) ⇒ "e"

str ⇒ "nw"
```

See also the [push](#) function, the inverse operation to `pop`, and the [set-nth](#) and [nth](#) functions, which can take multidimensional indices into lists.

post-url

syntax: (`post-url str-url str-content [str-content-type] [str-option] [int-timeout [str-header]]`)

post-url

Sends an HTTP POST request to the URL in *str-url*. POST requests are used to post information collected from web entry forms to a web site. Most of the time, the function `post-url` mimics what a web browser would do when sending information collected in an HTML form to a server, but it can also be used to upload files (see an HTTP reference). The function returns the page returned from the server in a string.

When `post-url` encounters an error, it returns a string description of the error beginning with `ERR:`.

The last parameter, *int-timeout*, is for an optional timeout value, which is specified in milliseconds. When no response from the host is received before the timeout has expired, the string `ERR: timeout` is returned.

example:

```
(post-url "http://somesite.com/login.pl"
  "user=johnDoe&pass=12345"
  "application/x-www-form-urlencoded")

(post-url "http://somesite.com/login.pl"
  "user=johnDoe&pass=12345"
  "application/x-www-form-urlencoded" 8000)

;; assumes default content type
(post-url "http://somesite.com/login.pl"
  "user=johnDoe&pass=12345")

(post-url "http://somesite.com/login.pl"
  "user=johnDoe&pass=12345" 10000)
```

The above example uploads a user name and password using a special format called `application/x-www-form-urlencoded`. `post-url` can be used to post other content types such as files or binary data. See an HTTP reference for other content-type specifications and data encoding formats. When the content-type parameter is omitted, `post-url` assumes `application/x-www-form-urlencoded` as the default content type.

Additional parameters

When *str-content-type* is specified, the *str-option* "header" or "list" can be specified as the return page. If the *int-timeout* option is specified, the custom header option *str-header* can be specified, as well. See the function [get-url](#) for details on both of these options.

See also the [get-url](#) and [put-url](#) functions.

pow

syntax: (pow *num-1* *num-2* [*num-3* ...])

syntax: (pow *num-1*)

Calculates *num-1* to the power of *num-2* and so forth.

example:

```
(pow 100 2)      ⇒ 10000
(pow 100 0.5)    ⇒ 10
(pow 100 0.5 3)  ⇒ 1000

(pow 3)          ⇒ 9
```

When *num-1* is the only argument, `pow` assumes 2 for the exponent.

pretty-print

syntax: (pretty-print [*int-length* [*str-tab*]])

Reformats expressions for [print](#), [save](#), or [source](#). The first parameter, *int-length*, specifies the maximum line length, and *str-tab* specifies the string used to indent lines. All parameters are optional. `pretty-print` returns the current settings or the new settings when one or both parameters are specified.

example:

```
(pretty-print) ⇒ (64 " ") ; default setting

(pretty-print 90 "\t") ⇒ (90 "\t")

(pretty-print 100) ⇒ (100 "\t")
```

The first example reports the default settings of 64 for the maximum line length and a TAB character for indenting. The third example changes the line length only.

Note that `pretty-print` cannot be used to prevent line breaks from being printed. To completely suppress pretty printing, use the function [string](#) to convert the expression to a raw unformatted string as follows:

example:

```
;; print without formatting

(print (string my-expression))
```

primitive?

syntax: (primitive? *exp*)

Evaluates and tests if *exp* is a primitive symbol and returns `true` or `nil` depending on the result.

example:

```
(set 'var define)
```

`primitive?`

```
(primitive? var) ⇒ true
```

print

syntax: (print *exp-1* [*exp-2* ...])

Evaluates and prints *exp-1*— to the current I/O device, which defaults to the console window. See the built-in function [device](#) for details on how to specify a different I/O device.

List expressions are indented by the nesting levels of their opening parentheses.

Several special characters may be included in strings encoded with the escape character \:

<i>escaped character</i>	<i>description</i>
\n	the line-feed character (ASCII 10)
\r	the carriage-return character (ASCII 13)
\t	the tab character (ASCII 9)
\nnn	where <i>nnn</i> is a decimal ASCII code between 000 and 255

example:

```
(print (set 'res (+ 1 2 3)))
(print "the result is" res "\n")

"\065\066\067" ⇒ "ABC"
```

To finish printing with a line feed, use [println](#).

println

syntax: (println *exp-1* [*exp-2* ...])

Evaluates and prints *exp-1*— to the current I/O device, which defaults to the console window. A line feed is printed at the end. See the built-in function [device](#) for details on how to specify a different I/O device. `println` works exactly like [print](#) but emits a line-feed character at the end.

See also the [write-line](#) and [print](#) functions.

prob-chi2

syntax: (prob-chi2 *num-chi2 num-df*)

Returns the probability Q of an observed Chi² statistic in *num-chi2* with *num-df* degrees of freedom to be equal to or greater than. `prob-chi2` is derived from the incomplete Gamma function [gammai](#).

example:

```
(prob-chi2 10 6) ⇒ 0.1246520195
```

See also the inverse function [crit-chi2](#).

prob-z

syntax: (prob-z *num-z*)

Returns the probability of *num-z*, not to exceed the observed value where *num-z* is a normal distributed value with a mean of 0.0 and a standard deviation of 1.0.

example:

```
(prob-z 0.0) ⇒ 0.5
```

See also the inverse function [crit-z](#).

process

syntax: (process *str-command*)

syntax: (process *str-command int-pipe-in int-pipe-out [int-win32-option]*)

syntax: (process *str-command int-pipe-in int-pipe-out [int-pipe-error]*)

In the first syntax, `process` works similarly to `!` (exclamation mark) but in a non-blocking fashion, launching a child process specified in *str-command* and then returning immediately with the child process ID or `nil` if a child process could not be created.

example:

```
(process "notepad") ⇒ 1894
```

In the second syntax, standard input and output of the created process can be redirected to pipe handles. When remapping standard I/O of the launched application to a pipe, it is possible to communicate with the other application via [write-line](#) and [read-line](#) or [write-buffer](#) and [read-buffer](#) statements:

example:

```
;; Linux/UNIX
;; create pipes
(map set '(myin bcout) (pipe))
(map set '(bcin myout) (pipe))

;; launch UNIX 'bc' calculator application
```


newLISP Users Manual and Reference

```
(process "bc" bcin bcout)

(write-buffer myout "3 + 4\n") ; bc expects a linefeed

(read-line myin) ⇒ "7"

;; bc can use bignums with arbitrary precision

(write-buffer myout "123456789012345 * 123456789012345\n")

(read-line myin) ⇒ "15241578753238669120562399025"

;; Win32
(map set '(myin cmdout) (pipe))
(map set '(cmdin myout) (pipe))

(process "cmd" cmdin cmdout) ; Win32 command shell

(write-line "dir c:\\*.bat" myout)

(read-buffer myin 'buff 2000)

(println buff) ; directory listing
```

On Win32 versions of newLISP, a fourth optional parameter of *int-win32-option* can be specified to control the display status of the application. This option defaults to 1 for showing the application's window, 0 for hiding it, and 2 for showing it minimized on the Windows launch bar.

On both Win32 and Linux/UNIX systems, standard error will be redirected to standard out by default. On Linux/UNIX, an optional pipe handle for standard error output can be defined. In this case, [peek](#) can be used to check for information on the pipe handles:

```
;; create pipes
(map set '(myin bcout) (pipe))
(map set '(bcin myout) (pipe))
(map set '(errin errout) (pipe))

;; launch UNIX 'bc' calculator application
(process "bc" bcin bcout errout)

(write-buffer myout command)

;; wait for bc sending result or error info
(while (and (= (peek myin) 0)
            (= (peek errin) 0)) (sleep 10))

(if (> (peek errin) 0)
    (println (read-line errin)))

(if (> (peek myin) 0)
    (println (read-line myin)))
```

Not all interactive console applications can have their standard I/O channels remapped. Sometimes only one channel, *in* or *out*, can be remapped. In this case, specify 0 (zero) for the unused channel. The following statement uses only the launched application's output:

```
(process "app" 0 myout)
```

Normally, two pipes are used: one for communications to the child process and the other one for communications from the child process.

See also the [pipe](#) and [share](#) functions for inter-process communications and the [semaphore](#) function for synchronization of several processes. See the [fork](#) function for starting separate newLISP threads on Linux/UNIX.

push

syntax: (**push** *exp list* [*int-index-1* [*int-index-2* ...]])

syntax: (**push** *exp list* [*list-indexes*])

syntax: (**push** *str-1 str-2* [*int-index*])

Inserts the value of *exp* into the list *list*. If *int-index* is present, the element is inserted at that index. If the index is absent, the element is inserted at index 0 (zero), the first element. **push** is a destructive operation that changes the contents of the target list. The element inserted is returned. See also [Indexing elements of strings and lists](#).

If more than one *int-index* is present, the indices are used to access a nested list structure. Improper indices (those not matching list elements) are discarded.

The second version takes a list of *list-indexes* but is otherwise identical to the first. In this way, **push** works easily together with [ref](#) and [ref-all](#), which return lists of indices.

If *list* does not contain a list, *list* must contain a `nil` and will be initialized to the empty list.

Repeatedly using **push** to the end of a list using `-1` as the *int-index* is optimized and as fast as pushing to the front of a list with no index at all. This can be used to efficiently grow a list.

example:

```
; inserting in front
(set 'pList '(b c)) ⇒ (b c)
(push 'a pList)    ⇒ a
pList              ⇒ (a b c)

; insert at index
(push "hello" pList 2) ⇒ "hello"
pList                 ⇒ (a b "hello" c)

; optimized appending at the end
(push 'z pList -1) ⇒ z
pList              ⇒ (a b "hello" c z)

; inserting lists in lists
(push '(f g) pList) ⇒ (f g)
pList              ⇒ ((f g) a b "hello" c z)

; inserting at negative index
(push 'x pList -3) ⇒ x
pList              ⇒ ((f g) a b "hello" x c z)

; using multiple indices
(push 'h pList 0 -1) ⇒ h
```

```

pList                ⇒ ((f g h) a b "hello" x c z)

; use indices in a list
(set 'pList '(a b (c d () e)))

(push 'x pList '(2 2 0)) ⇒ x
pList                  ⇒ (a b (c d (x) e))

(ref 'x pList)        ⇒ (2 2 0)

(pop pList '(2 2 0))  ⇒ x

;; push on strings

(set 'str "abcdefg")

(push "hijk" str -1)   ⇒ "hijk"
str                   ⇒ "abcdefghijk"

(push "123" str)       ⇒ "123"
str                   ⇒ "123abcdefghijk"

(push "4" str 3)       ⇒ "4"
str                   ⇒ "1234abcdefghijk"

; push on uninitialized symbol
aVar ⇒ nil

(push 999 aVar)       ⇒ 999

aVar ⇒ (999)

```

See also the [pop](#) function, which is the inverse operation to push, and the [set-nth](#), [nth-set](#), and [nth](#) functions, which can all take multidimensional indices into lists.

put-url

syntax: (put-url *str-url str-content* [*str-option*] [*int-timeout*] [*str-header*])

The HTTP PUT protocol is used to transfer information in *str-content* to a file specified in *str-url*. The lesser-known HTTP PUT mode is frequently used for transferring web pages from HTML editors to Web servers. In order to use PUT mode, the web server's software must be configured correctly. On the Apache web server, use the 'Script PUT' directive in the section where directory access rights are configured.

Optionally, an *int-timeout* value can be specified in milliseconds as the last parameter. put-url will return ERR: timeout when the host gives no response and the timeout expires. On other error conditions, put-url returns a string starting with ERR: and the description of the error.

put-url requests are also understood by newLISP server nodes.

example:

```

(put-url "http://asite.com/myFile.txt" "Hi there")
(put-url "http://asite.com/myFile.txt" "Hi there" 2000)

```

```
(put-url "http://asite.com/webpage.html"
  (read-file "webpage.html"))
```

The first example creates a file called `myFile.txt` on the target server and stores the text string `'Hi there'` in it. In the second example, the local file `webpage.html` is transferred to `asite.com`.

On an Apache web server, the following could be configured in `httpd.conf`.

example:

```
<directory /www/htdocs>
Options All
Script PUT /cgi-bin/put.cgi
</directory>
```

The script `put.cgi` would contain code to receive content from the web server via STDIN. The following is a working `put.cgi` written in newLISP for the Apache web server:

example:

```
#!/usr/home/johndoe/bin/newlisp
#
#
# get PUT method data from CGI STDIN
# and write data to a file specified
# in the PUT request
#
#

(print "Content-type: text/html\n\n")

(set 'cnt 0)
(set 'result "")

(if (= "PUT" (env "REQUEST_METHOD"))
  (begin
    (set 'len (integer (env "CONTENT_LENGTH")))

    (while (< cnt len)
      (set 'n (read-buffer (device) 'buffer len))
      (if (not n)
        (set 'cnt len)
        (begin
          (inc 'cnt n)
          (write-buffer result buffer))))

    (set 'path (append
      "/usr/home/johndoe"
      (env "PATH_TRANSLATED")))

    (write-file path result)
  )
)

(exit)
```

Note that the script appends ".txt" to the path to avoid the CGI execution of uploaded malicious scripts. Note also that the two lines where the file path is composed may work differently in your web server environment. Check environment variables passed by your web server for composition of the right file path.

`put-url` returns content returned by the `put.cgi` script.

Additional parameters

In *str-option*, "header" or "list" can be specified for the returned page. If the *int-timeout* option is specified, the custom header option *str-header* can be specified, as well. See the function [get-url](#) for details on both of these options.

See also the functions [get-url](#) and [post-url](#), which can be used to upload files when formatting form data as `multipart/form-data`.

pv

syntax: (`pv num-int num-nper num-pmt [num-fv int-type]`)

Calculates the present value of a loan with the constant interest rate *num-interest* and the constant payment *num-pmt* after *num-nper* number of payments. The future value *num-pmt* is assumed to be 0.0 if omitted. If payment is at the end of each period, 0 (zero) is assumed for *int-type*; otherwise 1 is assumed.

example:

```
(pv (div 0.07 12) 240 775.30) ⇒ -100000.1373
```

In the example, a loan that would be paid off (future value = 0.0) in 240 payments of \$775.30 at a constant interest rate of 7 percent per year would start out at \$100,000.14.

See also the [fv](#), [irr](#), [nper](#), [npv](#), and [pmt](#) functions.

quote

syntax: (`quote exp`)

Returns *exp* without evaluating it. The same effect can be obtained by prepending a ' (single quote) to *exp*.

example:

```
(quote x)           ⇒ x
(quote 123)          ⇒ 123
(quote (a b c))      ⇒ (a b c)
(= (quote x) 'x)     ⇒ true
```

quote?

syntax: (quote? *exp*)

Evaluates and tests whether *exp* is quoted. Returns `true` or `nil` depending on the result.

example:

```
(set 'var 'x) ⇒ 'x
(quote? var) ⇒ true
```

Note that in the `set` statement, `' x` is quoted twice because the first quote is lost during the evaluation of the `set` assignment.

rand

syntax: (rand *int-range* [*int-N*])

Evaluates the expression in *int-range* and generates a random number in the range of 0 (zero) to (*int-range* - 1). When 0 (zero) is passed, the internal random generator is initialized using the current value returned by the `time` function. Optionally, a second parameter can be specified to return a list of length *int-N* of random numbers.

example:

```
(dotimes (x 100) (print (rand 2)))

(rand 3 100) ⇒ (2 0 1 1 2 0 ...)
```

The first line in the example prints equally distributed 0's and 1's, while the second line produces a list of 100 integers with 0, 1, and 2 equally distributed. Use the [random](#) and [normal](#) functions to generate floating point random numbers, and use [seed](#) to vary the initial seed for random number generation.

random

syntax: (random *float-offset* *float-scale* *int-n*)

syntax: (random *float-offset* *float-scale*)

In the first form, `random` returns a list of *int-n* evenly distributed floating point numbers scaled (multiplied) by *float-scale*, with an added offset of *float-offset*. The starting point of the internal random generator can be seeded using [seed](#).

example:

```
(random 0 1 10)
⇒ (0.10898973 0.69823783 0.56434872 0.041507289 0.16516733
   0.81540917 0.68553784 0.76471068 0.82314585 0.95924564)
```

When used in the second form, `random` returns a single evenly distributed number:

```
(random 10 5) ⇒ 11.0971
```

See also the [normal](#) and [rand](#) functions.

randomize

syntax: (`randomize list [bool]`)

Rearranges the order of elements in *list* into a random order.

example:

```
(randomize '(a b c d e f g)) ⇒ (b a c g d e f)
(randomize (sequence 1 5))   ⇒ (3 5 4 1 2)
```

`randomize` will always return a sequence different from the previous one without the optional *bool* flag. This may require the function to calculate several sets of reordered elements, which in turn may lead to different processing times with different invocations of the function on the same input list length. To allow for the output to be equal to the input, `true` or any expression evaluating to not `nil` must be specified in *bool*.

`randomize` uses an internal *pseudo random sequence* generator that returns the same series of results each time newLISP is started. Use the [seed](#) function to change this sequence.

read-buffer

syntax: (`read-buffer int-file sym-buffer int-size [str-wait]`)

Reads a maximum of *int-size* bytes from a file specified in *int-file* into a buffer in *sym-buffer*. Any data referenced by the symbol *sym-buffer* prior to the reading is deleted. The handle in *int-file* is obtained from a previous [open](#) statement. The symbol *sym-buffer* contains data of type string after the read operation.

Optionally, a string to be waited for can be specified in *str-wait*. `read-buffer` will read a maximum amount of bytes specified in *int-size* or return earlier if *str-wait* was found in the data. The wait-string is part of the returned data.

Returns the number of bytes read or `nil` when the wait-string was not found. In any case, the bytes read are put into the buffer pointed to by *sym-buffer*, and the file pointer of the file read is moved forward. If no new bytes have been read, *sym-buffer* will contain `nil`.

example:

read-buffer

```
(set 'handle (open "aFile.ext" "read"))
(read-buffer handle 'buff 200)
```

Reads 200 bytes into the symbol `buff` from the file `aFile.ext`.

```
(read-buffer handle 'buff 1000 "password:")
```

Reads 1000 bytes or until the string `password:` is encountered. The string `password:` will be part of the data returned.

See also the [write-buffer](#) function.

read-char

syntax: (read-char *int-file*)

Reads a byte from a file specified by the file handle in *int-file*. The file handle is obtained from a previous [open](#) operation. Each `read-char` advances the file pointer by one byte. Once the end of the file is reached, `nil` is returned.

example:

```
(define (slow-file-copy from-file to-file)
  (set 'in-file (open from-file "read"))
  (set 'out-file (open to-file "write"))
  (while (set 'chr (read-char in-file))
    (write-char out-file chr))
  (close in-file)
  (close out-file)
  "finished")
```

Use [read-line](#) and [device](#) to read whole text lines at a time. Note that newLISP supplies a fast built-in function called [copy-file](#) for copying files.

See also the [write-char](#) function.

read-file

syntax: (read-file *str-file-name*)

Reads a file in *str-file-name* in one swoop and returns a string buffer containing the data.

example:

```
(write-file "myfile.enc"
  (encrypt (read-file "/home/lisp/myFile") "secret"))
```

The file `myfile` is read, then encrypted using the password `"secret"` before being written back into a new file titled `"myfile.enc"` in the current directory.

`read-file` can take an `http://` or `file://` URL in *str-file-name*. In this case, `read-file` works exactly like [get-url](#) and can take the same additional parameters.

example:

```
(read-file "http://asite.com/somefile.tgz" 10000)
```

The file `somefile.tgz` is retrieved from the remote location `http://asite.com`. The file transfer will time out after 10 seconds if it is not finished. In this mode, `read-file` can also be used to transfer files from remote newLISP server nodes.

See also the [write-file](#) and [append-file](#) functions.

read-key

syntax: (read-key)

Reads a key from the keyboard and returns an integer value. For navigation keys, more than one `read-key` call must be made. For keys representing ASCII characters, the return value is the same on all OSes, except for navigation keys and other control sequences like function keys, in which case the return values may vary on different OSes and configurations.

example:

```
(read-key) ⇒ 97 ; after hitting the A key
(read-key) ⇒ 65 ; after hitting the shifted A key
(read-key) ⇒ 10 ; after hitting [enter] on Linux
(read-key) ⇒ 13 ; after hitting [enter] on Win32

(while (≠ (set 'c (read-key)) 1) (println c))
```

The last example can be used to check return sequences from navigation and function keys. To break out of the loop, press `Ctrl-A`.

Note that `read-key` will not work from the newLISP-tk front-end or any other application running newLISP over a TCP/IP port connection.

read-line

syntax: (read-line [int-file])

Reads from the current I/O device a string delimited by a line-feed character (ASCII 10). There is no limit to the length of the string that can be read. The line-feed character is not part of the returned string. The line always breaks on a line feed, which is then swallowed. A line breaks on a carriage return (ASCII 13) only if followed by a line feed, in which case both characters are discarded. A carriage return alone only breaks and is swallowed if it is the last character in the stream.

By default, the current [device](#) is the keyboard ([device](#) o). Use the built-in function [device](#) to specify a different I/O device (e.g., a file). Optionally, a file handle can be specified in the *int-file* obtained from a previous [open](#) statement.

The last buffer contents from a read-line operation can be retrieved using [current-line](#).

example:

```
(print "Enter a num:")
(set 'num (integer (read-line)))

(set 'in-file (open "afile.dat" "read"))
(while (read-line in-file)
  (write-line))
(close in-file)
```

The first example reads input from the keyboard and converts it to a number. In the second example, a file is read line by line and displayed on the screen. The `write-line` statement takes advantage of the fact that the result from the last `read-line` operation is stored in a system internal buffer. When [write-line](#) is used without argument, it writes the contents of the last `read-line` buffer to the screen.

See also the [current-line](#) function for retrieving this buffer.

real-path

syntax: (real-path [*str-path*])

Returns the full path from the relative file path given in *str-path*. If a path is not given, "." (the current directory) is assumed.

example:

```
(real-path) ⇒ "/usr/home/fred" ; current directory
(real-path "./somefile.txt")
⇒ "/usr/home/fred/somefile.txt"
```

The output length is limited by the OS's maximum allowed path length. If `real-path` fails (e.g., because of a nonexistent path), `nil` is returned.

ref

syntax: (ref *exp list*)

`ref` searches for the expression *exp* in *list* and returns a list of integer indices or an empty list if *exp* cannot be found. `ref` can work together with [push](#) and [pop](#), both of which can also take lists of indices.

example:

```

(set 'pList '(a b (c d () e)))
(push 'x pList '(2 2 0)) ⇒ x
pList ⇒ (a b (c d (x) e))
(ref 'x pList) ⇒ (2 2 0)
(ref '(x) pList) ⇒ (2 2)
(set 'v (ref '(x) pList)) ⇒ (2 2)
(pList v) ⇒ (x)
(ref '(c d (x) e) pList) ⇒ (2)
(ref 'foo pList) ⇒ ()
(pop pList '(2 2 0)) ⇒ x

```

By using just [push](#), [pop](#), and [ref](#), any list can be constructed or modified at any place with just one statement. See also [ref-all](#).

ref-all

syntax: (ref-all *exp list*)

Works similarly to [ref](#), but returns a list of all index vectors found for *exp* in *list*.

example:

```

(set 'L '(a b c (d a f (a h a)) (k a (m n a) (x))))
(ref-all 'a L) ⇒ ((0) (3 1) (3 3 0) (3 3 2) (4 1) (4 2 2))
(L '(3 1)) ⇒ a
(map 'L (ref-all 'a L)) ⇒ (a a a a a a)

```

See also the [ref](#) function.

regex

syntax: (regex *str-pattern str-text* [*int-option*])

Performs a Perl Compatible Regular Expression (PCRE) search on *str-text* with the pattern specified in *str-pattern*. The same regular expression pattern matching is also supported in the functions [directory](#), [find](#), [find-all](#), [parse](#), [replace](#), and [search](#) when using these functions on strings.

`regex` returns a list with the matched strings and substrings and the beginning and length of each string inside the text. If no match is found, it returns `nil`. The offset numbers can be used for subsequent processing.

`regex` also sets the variables `$0`, `$1`, and `$2`—to the expression and subexpressions found. Just like any other symbol in newLISP, these variables or their equivalent expressions `$0`, `$1`, and `$2`—can be used in other LISP expressions for further processing.

example:

```
(regex "b+" "aaaabbbbaaaa") ⇒ ("bbb" 4 3)

; case-insensitive search option 1
(regex "b+" "AAAABBBAAAA" 1) ⇒ ("BBB" 4 3)

(regex "[bB]+" "AAAABbBAAAA" ) ⇒ ("BbB" 4 3)

(regex "http://(.*):(.*)" "http://nuevatec.com:80")
⇒ ("http://nuevatec.com:80" 0 22 "nuevatec.com" 7 12 "80" 20 2)

$0 ⇒ "http://nuevatec.com:80"
$1 ⇒ "nuevatec.com"
$2 ⇒ "80"

(dotimes (i 3) (println ($ i)))
http://nuevatec.com:80
nuevatec.com
80
⇒ "80"
```

The second example shows the usage of extra options, while the third example demonstrates more complex parsing of two subexpressions, which were marked by parentheses in the search pattern. In the last example, the expression and subexpressions are retrieved using the system variables `$0` to `$2` or their equivalent expression (`$ 0`) to (`$ 2`).

When `"` (quotes) are used to delimit strings that include literal backslashes, the backslash must be doubled in the regular expression pattern. As an alternative, `{ }` (curly brackets) or `[text]` and `[/text]` (text tags) can be used to delimit text strings. In these cases, no extra backslashes are required.

Characters escaped by a backslash in newLISP (e.g., the quote `\` or `\n`) need not to be doubled in a regular expression pattern, which itself is delimited by quotes.

```
;; double backslash for parentheses (special char in regex)
(regex "\\(abc\\)" "xyz(abc)xyz") ⇒ ("(abc)" 3 5)

;; one backslash for quotes (special char in newLISP)
(regex "\"\" \"abc\\\"def\"" "xyz(abc)xyz") ⇒ ("\"\" 3 1)

;; brackets as delimiters
(regex {\(abc\)} "xyz(abc)xyz") ⇒ ("(abc)" 3 5)

;; brackets as delimiters and quote in pattern
(regex {"} "abc\\\"def") ⇒ ("\"\" 3 1)

;; text tags as delimiters, good for multiline text in CGI
(regex [text]\\(abc\\)[/text] "xyz(abc)xyz") ⇒ ("(abc)" 3 5)
(regex [text]"[/text] "abc\\\"def") ⇒ ("\"\" 3 1)
```

When curly brackets or text tags are used to delimit the pattern string instead of quotes, a simple backslash is sufficient. The pattern and string are then passed in raw form to the regular expression routines. When curly brackets are used inside a pattern itself delimited by curly brackets, the inner brackets must be balanced, as follows:

```
;; brackets inside brackets are balanced
(regex {\d{1,3}} "qwerty567asdfg") ⇒ ("567" 6 3)
```

The following constants can be used for *int-option*. Several options can be combined using a binary or | (pipe). The uppercase names are used in the PCRE regex documentation and could be predefined in `init.lsp`. The last option is a newLISP custom option only to be used in [replace](#); it can be combined with PCRE options.

```
PCRE_CASELESS      1    ; treat uppercase like lowercase
PCRE_MULTILINE     2    ; limit search at a newline like Perl's /m
PCRE_DOTALL        4    ; . (dot) also matches newline
PCRE_EXTENDED      8    ; ignore whitespace except inside char class
PCRE_ANCHORED      16   ; anchor at the start
PCRE_DOLLAR_ENDONLY 32  ; $ matches at end of string, not before newline
PCRE_EXTRA         64   ; additional functionality currently not used
PCRE_NOTBOL        128  ; first char, not start of line; ^ shouldn't
match
PCRE_NOTEOL        256  ; last char, not end of line; $ shouldn't match
PCRE_UNGREEDY      512  ; invert greediness of quantifiers
PCRE_NOTEMPTY      1024 ; empty string considered invalid
PCRE_UTF8          2048 ; pattern and strings as UTF-8 characters

REPLACE_ONCE       0x8000 ; replace only one occurrence
                    ; only for use in replace
```

Note that regular expression syntax is very complex and feature-rich with many special characters and forms. Please consult a book or the PCRE manual pages for more detail. Most PERL books or introductions to Linux or UNIX also contain chapters about regular expressions. See also <http://www.pcre.org> for further references and manual pages.

remove-dir

syntax: (remove-dir *str-path*)

Removes the directory whose path name is specified in *str-path*. The directory must be empty for `remove-dir` to succeed. Returns `nil` on failure.

example:

```
(remove-dir "temp")
```

Removes the directory `temp` in the current directory.

rename-file

syntax: (**rename-file** *str-path-old str-path-new*)

Renames a file or directory entry given in the path name *str-path-old* to the name given in *str-path-new*. Returns `nil` or `true` depending on the operation's success.

example:

```
(rename-file "data.lisp" "data.backup")
```

replace

syntax: (**replace** *exp-key list exp-replacement*)

syntax: (**replace** *str-key str-data exp-replacement*)

syntax: (**replace** *str-pattern str-data exp-replacement int-option*)

syntax: (**replace** *exp list*)

List replacement

If the second argument is a list, `replace` replaces all elements in the list *list* that are equal to the expression in *exp-key*. The element is replaced with *exp-replacement*. Note that `replace` is destructive. It changes the list passed to it and returns the changed list. The number of replacements made is contained in the system variable `$0`.

String replacement without regular expression

If all arguments are strings, `replace` replaces all occurrences of *str-key* in *str-data* with the evaluated *exp-replacement*, returning the changed string. The expression in *exp-replacement* is evaluated for every replacement. The number of replacements made is contained in the system variable `$0`.

Regular expression replacement

The presence of a fourth parameter indicates that a regular expression search should be performed with a regular expression pattern specified in *str-pattern* and an option number specified in *int-option* (e.g., 1 (one) for case-insensitive searching or 0 (zero) for a standard Perl Compatible Regular Expression (PCRE) search). See [regex](#) above for details.

By default, `replace` replaces all occurrences of a search string even if a beginning-of-line specification is included in the search pattern. After each replace, a new search is started at a new position in *str-data*. Setting the option bit to `0x8000` in *int-option* will force `replace` to replace only the first occurrence. The changed string is returned.

`replace` with regular expressions also sets the internal variables `$0`, `$1`, and `$2–` with the contents of the expressions and subexpressions found. These can be used to perform replacements that depend on the content found during replacement. The symbols `$0`, `$1`, and `$2–` can be used in expressions just like any other symbols. If the replacement expression evaluates to something other than a string, no replacement is made. As an alternative, the contents of these variables can also be accessed by using `($\$$ 0)`, `($\$$ 1)`, `($\$$ 2)`, and so forth. This method allows indexed access (e.g., `($\$$ i)`, where `i` is an integer).

List removal

The last form of `replace` has only two arguments, the expression *expr* and *list*. This form removes all *exprs* found in *list*.

example:

```
(set 'aList '(a b c d e a b c d))
(replace 'b aList 'B)
⇒ (a B c d e a B c d)
aList ⇒ (a B c d e a B c d)
$0    ⇒ 2 ; number of replacements

(set 'str "this isa sentence")
(replace "isa" str "is a") ⇒ "this is a sentence"

;; using the option parameter to employ regular expressions

(set 'str "ZZZZZxZZZZyy") ⇒ "ZZZZZxZZZZyy"
(replace "[x|y]" str "PP" 0) ⇒ "ZZZZZPPZZZZPPPP"
str ⇒ "ZZZZZPPZZZZPPPP"

;; using system variables for dynamic replacement

(set 'str "---axb---ayb---")
(replace "(a)(.)(b)" str (append $3 $2 $1) 0)
⇒ "---bxa---bya---"

str ⇒ "---bxa---bya---"

;; using the 'replace once' option bit 0x8000

(replace "a" "aaa" "X" 0) ⇒ "XXX"

(replace "a" "aaa" "X" 0x8000) ⇒ "Xaa"

;; URL translation of hex codes with dynamic replacement

(set 'str "xxx%41xxx%42")
(replace "%([0-9A-F][0-9A-F])" str
(char (integer (append "0x" $1))) 1)

str ⇒ "xxxAxxxB"

;; removing elements from a list

(set 'lst '(a b a a c d a f g))
(replace 'a lst) ⇒ (b c d f g)
lst ⇒ (b c d f g)

$0 ⇒ 4
```

The [set-nth](#) and [replace-assoc](#) functions can also be used to change an element in a list.

See [directory](#), [find](#), [find-all](#), [parse](#), [regex](#), and [search](#) for other functions using regular expressions.

replace-assoc

syntax: (replace-assoc *exp-key list-assoc exp-replacement*)

syntax: (replace-assoc *exp-key list-assoc*)

In the first syntax, `replace-assoc` replaces an association element with *exp-key* in the association *list-assoc* with *exp-replacement*. An association list is a list whose elements are in turn lists, the first element serving as a key.

example:

```
(set 'aList '((a 1 2 3)(b 4 5 6)(c 7 8 9)))

(replace-assoc 'b aList '(q "I am the replacement"))
⇒ ((a 1 2 3)(q "I am the replacement")(c 7 8 9))

aList ⇒ ((a 1 2 3)(q "I am the replacement")(c 7 8 9))
```

`replace-assoc` uses the system variable `$0` for the association found. This can be used in the replacement expression:

```
(set 'lst '((a 1)(b 2)(c 3)))

(replace-assoc 'b lst (list 'b (+ 1 (last $0))))

lst ⇒ ((a 1)(b 3)(c 3))
```

`replace-assoc` returns the changed list or `nil` if no association is found. A destructive operation, `replace-assoc` changes the contents of the list.

In the second syntax, `replace-assoc` removes an association from the list and returns it, as follows:

example:

```
(set 'lst '((a 1) (b 2) (c 3)))

(replace-assoc 'c lst)

lst ⇒ ((a 1) (b 3))
$0 ⇒ (c 3)
```

See also the [assoc](#) function for accessing association lists.

reset

syntax: (reset)

syntax: (reset true)

In the first syntax, `reset` returns to the top level of evaluation, switches the [trace](#) mode off, turns the [command-line](#) mode on, and switches to the MAIN context/namespace. `reset` restores the top-level variable environment using the saved variable environments on the stack. It also fires an error "user reset - no error". This behavior can be used when writing error handlers.

`reset` may return memory that was claimed by newLISP to the operating system. `reset` walks through the entire cell space, which may take a few seconds in a heavily loaded system.

`reset` occurs automatically after an error condition.

In the second syntax, `reset` will stop the current process and start a new newLISP process with the same command-line parameters. This mode is not available on Win32.

rest

syntax: (rest *list*)

syntax: (rest *array*)

syntax: (rest *str*)

Returns all of the items in a list or a string, except for the first. `rest` is equivalent to `cdr` or `tail` in other LISP dialects.

example:

```
(rest '(1 2 3 4))           ⇒ (2 3 4)
(rest '((a b) c d))        ⇒ (c d)
(set 'aList '(a b c d e))  ⇒ (a b c d e)
(rest aList)                ⇒ (b c d e)
(first (rest aList))        ⇒ b
(rest (rest aList))         ⇒ (d e)
(rest (first '((a b) c d))) ⇒ (b)

(set 'A (array 2 3 (sequence 1 6)))
⇒ ((1 2) (3 4) (5 6))

(rest A) ⇒ ((3 4) (5 6))
```

In the second version, `rest` returns all but the first character of the string *str* in a string.

example:

```
(rest "newLISP")           ⇒ "ewLISP"
(first (rest "newLISP"))   ⇒ "e"
```

See also the [first](#) and [last](#) functions.

Note that an *implicit rest* is available for lists. See the chapter [Implicit rest and slice](#).

Note that [rest](#) works on character boundaries rather than byte boundaries when the UTF-8-enabled version of newLISP is used.

reverse

syntax: (reverse *list*)

syntax: (reverse *string*)

In the first form, `reverse` reverses and returns the *list*. Note that `reverse` is destructive and changes the original list.

example:

```
(set 'l '(1 2 3 4 5 6 7 8 9))

(reverse l) ⇒ (9 8 7 6 5 4 3 2 1)
l          ⇒ (9 8 7 6 5 4 3 2 1)
```

In the second form, `reverse` is used to reverse the order of characters in a string.

example:

```
(set 'str "newLISP")

(reverse str) ⇒ "PSILwen"
str          ⇒ "PSILwen"
```

See also the [sort](#) function.

rotate

syntax: (rotate *list* [*int-count*])

syntax: (rotate *str* [*int-count*])

Rotates and returns the *list* or string in *str*. A count can be optionally specified in *int-count* to rotate more than one position. If *int-count* is positive, the rotation is to the right; if *int-count* is negative, the rotation is to the left. If no *int-count* is specified, `rotate` rotates 1 to the right. `rotate` is a destructive function that changes the contents of the original list or string.

example:

```
(set 'l '(1 2 3 4 5 6 7 8 9))

(rotate l)      ⇒ (9 1 2 3 4 5 6 7 8)
(rotate l 2)    ⇒ (7 8 9 1 2 3 4 5 6)
l              ⇒ (7 8 9 1 2 3 4 5 6)

(rotate l -3)   ⇒ (1 2 3 4 5 6 7 8 9)

(set 'str "newLISP")
```

```
(rotate str)      ⇒ "PnewLIS"
(rotate str 3)    ⇒ "LISPnew"
(rotate str -4)   ⇒ "newLISP"
```

When working on a string, `rotate` works on byte boundaries rather than character boundaries.

save

syntax: (save *str-file*)

syntax: (save *str-file sym-1 [sym-2 ...]*)

In the first syntax, the `save` function writes the contents of the newLISP workspace (in textual form) to the file *str-file*. `save` is the inverse function of `load`. Using `load` on files created with `save` causes newLISP to return to the same state as when `save` was originally invoked. System symbols starting with the `$` character (e.g., `$0` from regular expressions or `$main-args` from the command line) are not saved.

In the second syntax, symbols can be supplied as arguments. If *sym-n* is supplied, only the definition of that symbol is saved. If *sym-n* evaluates to a context, all symbols in that context are saved. More than one symbol can be specified, and symbols and context symbols can be mixed. When contexts are saved, system variables and symbols starting with the `$` character are not saved. Specifying system symbols explicitly causes them to be saved.

Each symbol is saved by means of a [set](#) statement or—if the symbol contains a lambda or lambda-macro function—by means of [define](#) or [define-macro](#) statements.

Symbols containing `nil` will not be saved.

`save` returns `true` on completion.

example:

```
(save "save.lsp")

(save "/home/myself/myfunc.LSP" 'my-func)
(save "file:///home/myself/myfunc.LSP" 'my-func)

(save "http://asite.com:8080//home/myself/myfunc.LSP" 'my-func)

(save "mycontext.lsp" 'mycontext)

;; multiple args
(save "stuff.lsp" 'aContext 'myFunc '$main-args 'Acontext)
```

Since all context symbols are part of the context `MAIN`, saving `MAIN` saves all contexts.

Saving to a URL will cause an HTTP PUT request send to the URL. In this mode, `save` can also be used to push program source to remote newLISP server nodes. Note that a double backslash is required when path names are specified relative to the root directory. `save` in HTTP mode will observe a 60-second timeout.

Symbols made using [sym](#) that are incompatible with the normal syntax rules for symbols are serialized using a [sym](#) statement instead of a [set](#) statement.

`save` serializes contexts and symbols as if the current context is `MAIN`. Regardless of the current context, `save` will always generate the same output.

See also the functions [load](#) (the inverse operation of `save`) and [source](#), which saves symbols and contexts to a string instead of a file.

search

syntax: (search *int-file* *str-search* [*int-options*])

Searches a file specified by its handle in *int-file* for a string in *str-search*. *int-file* can be obtained from a previous [open](#) file. After the search, the file pointer is positioned at the beginning of the searched string or at the end of the file if nothing is found. In *int-options*, the options flags can be specified to perform a PCRE regular expression search. See the function [regex](#) for details. `search` returns the position of the found string or `nil` if nothing is found.

When using the regular expression options flag, patterns found are stored in the system variables `$0` to `$15`.

example:

```
(set 'file (open "init.lsp" "read"))
(search file "define")
(print (read-line file) "\n")
(close file)
```

The file `init.lsp` is opened and searched for the string `define`.

For other functions using regular expressions, see [directory](#), [find](#), [find-all](#), [parse](#), [regex](#), and [replace](#).

seed

syntax: (seed *int-seed*)

Seeds the internal random generator that generates numbers for [amb](#), [normal](#), [rand](#), and [random](#) with the number specified in *int-seed*. Note that the random generator used in newLISP is the C-library function `rand()`. All randomizing functions in newLISP are based on this function.

Note that the maximum value for *int-seed* is limited to 16 or 32 bits, depending on the operating system used. Internally, only the 32 least significant bits are passed to the random seed function of the OS.

example:

seed

```
(seed 12345)

(seed (date-value))
```

After using `seed` with the same number, the random generator starts the same sequence of numbers. This facilitates debugging when randomized data are involved. Using `seed`, the same random sequences can be generated over and over again.

The second example is useful for guaranteeing a different seed any time the program starts.

seek

syntax: (seek *int-file* [*int-position*])

Sets the file pointer to the new position *int-position* in the file specified by *int-file*. The new position is expressed as an offset from the beginning of the file, 0 (zero) meaning the beginning of the file. If no *int-position* is specified, `seek` returns the current position in the file. If *int-file* is 0 (zero), on BSD, `seek` will return the number of characters printed to STDOUT, and on Linux and Win32, it will return -1. On failure, `seek` returns `nil`. When *int-position* is set to -1, `seek` sets the file pointer to the end of the file.

example:

```
(set 'file (open "myfile" "read")) ⇒ 5
(seek file 100)                    ⇒ 100
(seek file)                        ⇒ 100

(open "newlisp_manual.html" "read")
(seek file -1) ; seek to EOF
⇒ 593816

(set 'file (open "large-file" "read"))
(seek file 30000000000) ⇒ 30000000000
```

newLISP supports file position numbers up to 9,223,372,036,854,775,807.

select

syntax: (select *list list-selection*)

syntax: (select *list* [*int-index_i* ...])

syntax: (select *string list-selection*)

syntax: (select *string* [*int-index_i* ...])

In the first two forms, `select` picks one or more elements from *list* using one or more indices specified in *list-selection* or the *int-index_i*.

example:

```
(set 'lst '(a b c d e f g))
(select lst '(0 3 2 5 3)) ⇒ (a d c f d)
(select lst '(-2 -1 0)) ⇒ (f g a)
(select lst -2 -1 0) ⇒ (f g a)
```

In the second two forms, `select` picks one or more characters from *string* using one or more indices specified in *list-selection* or the *int-index_i*.

example:

```
(set 'str "abcdefg")
(select str '(0 3 2 5 3)) ⇒ "adcfd"
(select str '(-2 -1 0)) ⇒ "fga"
(select str -2 -1 0) ⇒ "fga"
```

Selected elements can be repeated and do not have to appear in order, although this speeds up processing. The order in *list-selection* or *int-index_i* can be changed to rearrange elements.

semaphore

syntax: (semaphore)

syntax: (semaphore *int-id*)

syntax: (semaphore *int-id int-wait*)

syntax: (semaphore *int-id int-signal*)

syntax: (semaphore *int-id o*)

A semaphore is an interprocess synchronization object that maintains a count between 0 (zero) and some maximum value. Useful in controlling access to a shared resource, a semaphore is set to signaled when its count is greater than zero and to non-signaled when its count is zero.

A semaphore is created using the first syntax. This returns the semaphore ID, an integer used subsequently as *int-id* when the *semaphore* function is called. Initially, the semaphore has a value of zero, which represents the non-signaled state.

If calling `semaphore` with a negative value in *int-wait* causes it to be decremented below zero, the function call will block until another process or thread signals the semaphore with a positive value in *int-signal*. Calls to the semaphore with *int-wait* or *int-signal* effectively try to increment or decrement the semaphore value by a positive or negative value specified in *int-signal* or *int-wait*. Because the value of a semaphore must never fall below zero, the function call will block when this is attempted (i.e., a semaphore with a value of zero will block until another process or thread increases the value with a positive *int-signal*).

The second syntax is used to inquire about the value of a semaphore by calling `semaphore` with the *int-id* only. This form is not available on Win32.

Supplying 0 (zero) as the last argument will release system resources for the semaphore, which then becomes unavailable. Any pending waits on this semaphore in other child threads or processes will be released.

On Win32, only parent and child processes can share a semaphore. On Linux/UNIX, independent processes can share a semaphore.

The following code examples summarize the different syntax forms:

```
;; init semaphores
(semaphore)

;; assign a semaphore to sid
(set 'sid (semaphore))

;; inquire the state of a semaphore (not on Win32)
(semaphore sid)

;; put sid semaphore in wait state (-1)
(semaphore sid -1)

;; run sid semaphore previously put in wait (always 1)
(semaphore sid 1)

;; run sid semaphore with X times a skip (backward or forward) on
the function
(semaphore sid X)

;; release sid semaphore systemwide (always 0)
(semaphore sid 0)
```

The following example shows semaphores controlling a child process:

example:

```
;; counter thread output in bold

(define (counter n)
  (println "counter started")
  (dotimes (x n)
    (semaphore sid -1)
    (println x)))

;; hit extra <enter> to make the prompt come back
;; after output to the console from counter thread

> (set 'sid (semaphore))

> (semaphore sid)
0

> (fork (counter 100))

counter started
> (semaphore sid 1)
0
> (semaphore sid 3)
1
2
3
> (semaphore sid 2)
4
```

```
5
> _
```

After the semaphore is acquired in `sid`, it has a value of 0 (the non-sigaled state). When starting the thread `counter`, the semaphore will block after the initial start message and will wait in the semaphore call. The `-1` is trying to decrement the semaphore, which is not possible because its value is already zero. In the interactive, main parent process, the semaphore is signaled by raising its value by 1. This unblocks the semaphore call in the `counter` thread, which can now decrement the semaphore from 1 to 0 and execute the `print` statement. When the semaphore call is reached again, it will block because the semaphore is already in the wait (0) state.

Subsequent calls to `semaphore` with numbers greater than 1 give the `counter` thread an opportunity to decrement the semaphore several times before blocking.

More than one thread can participate in controlling the semaphore, just as more than one semaphore can be created. The maximum number of semaphores is controlled by a systemwide kernel setting on UNIX-like operating systems.

Use the [fork](#) function to start a new thread and the [share](#) function to share information between threads. For a more comprehensive example of using `semaphore` to synchronize threads, see the [prodcons.lsp](#) example in the appendix or the `examples/` directory in the source distribution, as well as the examples and modules distributed with newLISP.

sequence

syntax: `(sequence num-start num-end [num-step])`

Generates a sequence of numbers from *num-start* to *num-end* with an optional step size of *num-step*. When *num-step* is omitted, the value 1 (one) is assumed. The generated numbers are of type integer (when no optional step size is specified) or floating point (when the optional step size is present).

example:

```
(sequence 10 5)      ⇒ (10 9 8 7 6 5)
(sequence 0 1 0.2)  ⇒ (0 0.2 0.4 0.6 0.8 1)
(sequence 2 0 0.3)  ⇒ (2 1.7 1.4 1.1 0.8 0.5 0.2)
```

Note that the step size must be a positive number, even if sequencing from a higher to a lower number.

Use the [series](#) function to generate geometric sequences.

series

syntax: `(series num-start num-factor num-count)`

series

Creates a geometric sequence with *num-count* elements starting with the element in *num-start*. Each subsequent element is multiplied by *num-factor*. The generated numbers are always floating point numbers.

example:

```
(series 2 2 5)      ⇒ (2 4 8 16 32)
(series 1 1.2 6)    ⇒ (1 1.2 1.44 1.728 2.0736 2.48832)
(series 10 0.9 4)   ⇒ (10 9 8.1 7.29)
(series 0 0 10)     ⇒ (0 0 0 0 0 0 0 0 0 0)
(series 99 1 5)     ⇒ (99 99 99 99 99)
```

Use the [sequence](#) function to generate arithmetic sequences.

set

syntax: (set *sym-1 exp-1* [*sym-2 exp-2* ...])

Evaluates both arguments and then assigns the result of *exp* to the symbol found in *sym*. The *set* expression returns the result of the assignment. The assignment is performed by copying the contents of the right side into the symbol. The old contents of the symbol are deleted. An error message results when trying to change the contents of the symbols *nil*, *true*, or a context symbol. *set* can take multiple argument pairs.

example:

```
(set 'x 123)        ⇒ 123
(set 'x 'y)          ⇒ y
(set 'x "hello")     ⇒ "hello"

y ⇒ "hello"

(set 'alist '(1 2 3)) ⇒ (1 2 3)

(set 'x 1 'y "hello") ⇒ "hello" ; multiple arguments

x ⇒ 1
y ⇒ "hello"
```

The symbol for assignment could be the result from another newLISP expression:

```
(set 'lst '(x y z)) ⇒ (x y z)

(set (first lst) 123) ⇒ 123

x ⇒ 123
```

Symbols can be set to lambda or lambda-macro expressions. This operation is equivalent to using [define](#) or [define-macro](#).

```
(set 'double (lambda (x) (+ x x)))
⇒ (lambda (x) (+ x x))
```

is equivalent to:

```
(define (double x) (+ x x))
⇒ (lambda (x) (+ x x))
```

is equivalent to:

```
(define double (lambda (x) (+ x x)))
⇒ (lambda (x) (+ x x))
```

Use the [constant](#) function (which works like `set`) to protect the symbol from subsequent alteration. Using the [setq](#) function eliminates the need to quote the variable symbol.

setq

syntax: (`setq sym-1 exp-1 [sym-2 exp-2 ...]`)

Works just like [set](#), except the symbol in *sym* is not quoted. Like [set](#), `setq` can take multiple arguments.

example:

```
(setq x 123) ⇒ 123

; multiple args

(setq x 1 y 2 z 3) ⇒ 3

x ⇒ 1
y ⇒ 2
z ⇒ 3
```

set-locale

syntax: (`set-locale [str-locale] [int-category]`)

Reports or switches to a different locale on your operating system or platform. When used without arguments, *set-locale* reports the current locale being used. When *str-locale* is specified, *set-locale* switches to the locale with all category options turned on (LC_ALL). Placing an empty string in *str-locale* switches to the default locale used on the current platform. `set-locale` returns either the current settings or `nil` if the requested change could not be performed.

example:

```
(set-locale) ; report current locale

(set-locale "") ; set default locale of your platform
```

By default, newLISP starts up with the POSIX C default locale. This guarantees that newLISP's behavior will be identical on any platform locale:

```
;; after newLISP start up
(set-locale) ⇒ "C"
```

In *int-category*, integer numbers may be specified as *category options* for fine-tuning certain aspects of the locale, such as number display, date display, and so forth. The numbers used vary from system to system. The options valid on your platform can be found in the C include file `locale.h`. This file defines constants like `LC_ALL`, `LC_NUMERIC`, and `LC_MONETARY`. When `set-locale` is used without the option number, it assumes the `LC_ALL` option, which turns on all options for that locale.

Note that the locale also controls the decimal separator in numbers. The default C locale uses the decimal dot, but most others use a decimal comma. Since version 8.4.4, newLISP has been parsing decimal comma numbers correctly.

Note that using `set-locale` does not change the behavior of regular expressions in newLISP. To localize the behavior of PCRE (Perl Compatible Regular Expressions), newLISP must be compiled with different character tables. See the file, `LOCALIZATION`, in the newLISP source distribution for details.

See also the chapter [Switching the locale](#).

set-nth

syntax: `(set-nth int-nth-1 [int-nth-2 ...] list|array exp-replacement)`

syntax: `(set-nth int-nth-1 str str-replacement)`

syntax: `(set-nth (list|array int-nth-1 [int-nth-2 ...]) exp-replacement)`

syntax: `(set-nth (str int-nth-1) str str-replacement)`

`set-nth` works like [nth-set](#), except instead of returning the replaced element, it returns the entire changed expression. For this reason, `set-nth` is slower on larger data objects.

sgn

syntax: `(sgn num)`

syntax: `(sgn num expr-1 [expr-2] [expr-3])`

In the first syntax, the `sgn` function is a logical function that extracts the sign of a real number according to the following rules:

$$\begin{aligned} x > 0 &: \text{sgn}(x) = 1 \\ x < 0 &: \text{sgn}(x) = -1 \\ x = 0 &: \text{sgn}(x) = 0 \end{aligned}$$

example:

```
(sgn -3.5) ⇒ -1
(sgn 0)   ⇒ 0
(sgn 123) ⇒ 1
```

In the second syntax, the result of evaluating one of the optional expressions *expr-1*, *expr-2*, or *expr-3* is returned, instead of -1, 0, or 1. In absence of expression *expr-2* or *expr-3*, nil is returned.

example:

```
(sgn x -1 0 1)      ; works like (sgn x)
(sgn x -1 1 1)      ; return -1 for negative x all others 1
(sgn x nil true true) ; return nil for negative else true
(sgn x (abs x) 0)    ; return (abs x) for negative x, 0 for x =
0, else nil
```

Any expression or constant can be used for *expr-1*, *expr-2*, or *expr-3*.

share

syntax: (share)**syntax: (share *int-address-or-handle*)****syntax: (share *int-address-or-handle exp-value*)****syntax: (share *nil int-address*)**

Accesses shared memory for communicating between several newLISP processes or threads. When called without arguments, *share* requests a page of shared memory (the page is 4k on Win32 but may differ on Linux/UNIX) from the operating system. This returns a memory address on Linux/UNIX and a handle on Win32, which can then be assigned to a variable for later reference.

To set the contents of shared memory, use the third syntax of *share*. Supply a shared memory address on Linux/UNIX or a handle on Win32 in *int-address-or-handle*, along with an integer, float, or string expression in *exp-value*. Using this syntax, the value supplied in *exp-value* is also the return value.

To access the contents of shared memory, use the second syntax of *share*, supplying only the shared memory address or handle. The return value will be an integer or floating point number, a string, or nil or true. If the memory has not been previously set to a value, nil will be returned.

Only available on UNIX-like operating systems, the last syntax unmaps a shared memory address. Note that using a shared address after unmapping it will crash the system.

Memory can be shared between parent and child processes or threads, but not between independent processes.

example:

```
(set 'num (share))
```

```

(set 'str (share))

(share num 123) ⇒ 123

(share str "hello world") ⇒ "hello world"
(share str)                ⇒ "hello world"

(share mVar 123) ⇒ 123
(share mVar)     ⇒ 123

(share mVar true) ⇒ true
(share mVar)     ⇒ true

(share nil mVar) ⇒ true ; unmap only on UNIX

```

For a more comprehensive example of using shared memory in a multithreaded Linux/UNIX application, see the [prodcons.lsp](#) in the Appendix or the examples in the `examples/` directory of the source distribution.

Note that shared memory access between different threads or processes should be synchronized using a [semaphore](#). Simultaneous access to shared memory can crash the running process/thread.

To find out the maximum length of a string buffer that could be stored in a shared memory address, execute the following:

```

(length (share (share) (dup " " 1000000)))
⇒ 4087

```

The statement above tries to initialize a shared memory address to 100,000 bytes, but only 4087 will be initialized as a string buffer. The page size of this platform is 4096 bytes—4087 plus 8 bytes of header information for type and size, as well as 1 terminating byte for displayable strings.

On Linux/UNIX systems, more than one number or string can be stored in one memory page by using offsets added to the main segment address:

example:

```

;; Linux/UNIX only

(set 'num-1 (share))
(set 'num-2 (+ num-1 12))
(set 'num-3 (+ num-2 12))
(set 'str-1 (+ num-3 12))

(share num-1 123)
(share num-2 123.456)
...

(share num-1) ⇒ 123
(share num-3) ⇒ 123.456
...

;; etc.

```

For numbers, reserve 12 bytes; for strings, reserve 12 bytes, plus the length of the string, as well as 1 for the terminating zero-byte. For the boolean values `nil` and `true`, 4 bytes should be reserved.

Note that a shorter string could accidentally be overwritten with a longer one. Therefore, shared strings should be stored after other shared number fields or should reside on their own shared memory page.

The functions [get-int](#), [get-float](#), [get-string](#), and [get-char](#)—as well as [pack](#) and [unpack](#)—could also be used to access contents from a shared memory page. This low-level address requires precise knowledge of the type of information stored, but it allows for very compact storage of information without type/header information in a string buffer.

example:

```
;; Linux/UNIX and Win32

(set 'mem (share))

(mem share (pack "s5 ld lf" "hello" 123 123.456))
(unpack "s10 ls lf" (mem share))
⇒ ("hello" 123 123.456)
```

On Linux/UNIX, supplying a wrong or unmapped share address can cause newLISP to crash.

signal

syntax: (signal *int-signal* *sym-handler*)

syntax: (signal *int-signal* *func-handler*)

syntax: (signal *int-signal* nil)

syntax: (signal *int-signal*)

Sets a user-defined handler in *sym-handler* for a signal specified in *int-signal*. If *nil* is specified, the signal will default to the initialized behavior in newLISP.

Different signals are available on different OS platforms and Linux/UNIX flavors. The numbers to specify in *int-signal* also differ from platform to platform. Valid values can normally be extracted from a file found in `/usr/include/sys/signal.h` or `/usr/include/signal.h`.

Some signals make newLISP exit even after a user-defined handler has been specified and executed (e.g., signal SIGKILL). This behavior may also be different on different platforms.

example:

```
(constant 'SIGINT 2)
(define (ctrlC-handler) (println "ctrl-C has been pressed"))

(signal SIGINT 'ctrlC-handler)

; now press ctrl-C
; the following line will appear

ctrl-C has been pressed
```

On Win32, the above example would execute the handler before exiting newLISP. On most Linux/UNIX systems, newLISP would stay loaded and the prompt would appear after hitting the [enter] key.

Instead of specifying a symbol containing the signal handler, a function can be specified directly. The signal number is passed as a parameter:

```
(signal SIGINT exit) ⇒ $signal-2

(signal SIGINT (fn (s) (println "signal " s " occurred")))
```

Note that the signal SIGKILL (9 on most platforms) will always terminate the application regardless of an existing signal handler.

The signal could have been sent from another shell on the same computer:

```
kill -s SIGINT 2035
```

In this example, 2035 is the process ID of the running newLISP.

The signal could also have been sent from another newLISP application:

```
(constant 'SIGINT 2)
(import "libc.so" "kill")

(kill 2035 SIGINT)
```

When importing `kill`, make sure it always receives an integer for the signal number. If needed, use the [int](#) function to first convert the number.

If newLISP receives a signal while evaluating another function, it will still accept the signal and the handler function will be executed:

```
(constant 'SIGINT 2)
(define (ctrlC-handler) (println "ctrl-C has been pressed"))

(signal SIGINT 'ctrlC-handler)
;; or
(signal SIGINT ctrlC-handler)

(while true (sleep 300) (println "busy"))

;; generates following output
busy
busy
busy
ctrl-C has been pressed
busy
busy
...
```

Specifying only a signal number will return either the name of the current defined handler function or `nil`.

The user-defined signal handler can pass the the signal number as a parameter.

```
(define (signal-handler sig)
  (println "received signal: " sig))
```

```
;; set all signals from 1 to 8 to the same handler
(for (s 1 8)
  (signal s 'signal-handler))
```

In this example, all signals from 1 to 8 are set to the same handler.

silent

syntax: (**silent** [*expr-1*] [*expr-2 ...*])

Evaluates one or more expressions in *expr-1*—. **silent** is similar to [begin](#), but it suppresses console output of the return value and the following prompt. It is often used when communicating from a remote application with newLISP (e.g., GUI front-ends or other applications controlling newLISP), and the return value is of no interest.

Silent mode is reset when returning to a prompt. This way, it can also be used without arguments in a batch of expressions. When in interactive mode, hit [enter] twice after a statement using **silent** to get the prompt back.

example:

```
(silent (my-func)) ; same as next
(silent) (my-func) ; same effect as previous
```

sin

syntax: (**sin** *num-radians*)

Calculates the sine function from *num-radians* and returns the result.

example:

```
(sin 1)                ⇒ 0.8414709838
(set 'pi (mul 2 (acos 0))) ⇒ 3.141592654
(sin (div pi 2))       ⇒ 1
```

sleep

syntax: (**sleep** *int-milli-seconds*)

Gives up CPU time to other processes for the amount of milliseconds specified in *int-milli-seconds*.

example:

sleep


```
(sleep 1000) ; sleeps 1 second
```

On some platforms, `sleep` is only available with a resolution of one second. In this case, the parameter *int-milli-seconds* will be rounded to the nearest full second.

slice

syntax: (slice *list* *int-index* [*int-length*])

syntax: (slice *array* *int-index* [*int-length*])

syntax: (slice *str* *int-index* [*int-length*])

In the first form, `slice` copies a sublist from a *list*. The original list is left unchanged. The sublist extracted starts at index *int-index* and has a length of *int-length*. If *int-length* is -1, or if the parameter is omitted, `slice` copies all of the elements to the end of the list.

See also [Indexing elements of strings and lists](#).

example:

```
(slice '(a b c d e f) 3 2) ⇒ (d e)
(slice '(a b c d e f) 2 -1) ⇒ (c d e f)
(slice '(a b c d e f) -4 3) ⇒ (c d e)

(set 'A (array 3 2 (sequence 1 6))) ⇒ ((1 2) (3 4) (5 6))
(slice A 1 2) ⇒ ((3 4) (5 6))
```

In the second form, a part of the string in *str* is extracted. *int-index* contains the start index and *int-length* contains the length of the substring. If *int-length* is not specified, everything to the end of the string is extracted. `slice` also works on string buffers containing binary data like 0's (zeroes). It operates on byte boundaries rather than character boundaries. See also [Indexing elements of strings and lists](#).

example:

```
(slice "Hello World" 6 2) ⇒ "Wo"
(slice "Hello World" 0 5) ⇒ "Hello"
(slice "Hello World" 6) ⇒ "World"
(slice "newLISP" -4 2) ⇒ "LI"
```

Note that an *implicit slice* is available for lists. See the chapter [Implicit rest and slice](#).

Be aware that [rest](#) always works on byte boundaries rather than character boundaries in the UTF-8-enabled version of newLISP. As a result, [slice](#) can be used to manipulate binary content.

sort

syntax: (sort *list*)

syntax: (sort *list* *func-compare*)

sort

In the first syntax, all members in *list* are sorted in ascending order. Anything may be sorted, regardless of the types. When members are themselves lists, each list element is recursively compared. If two expressions of different types are compared, the lower type is sorted before the higher type in the following order:

Atoms: nil, true, integer or float, string, symbol, primitive

Lists: quoted expression, list, lambda, lambda-macro

The sort is destructive, changing the order of the elements in the original list. The return value of sort is a copy of the sorted list.

In the second syntax, a comparison operator, user-defined function, or anonymous function can be supplied. The functor or operator can be given with or without a preceding quote.

example:

```
(sort '(v f r t h n m j))      ⇒ (f h j m n r t v)
(sort '((3 4) (2 1) (1 10)))   ⇒ ((1 10) (2 1) (3 4))
(sort '((3 4) "hi" 2.8 8 b))   ⇒ (2.8 8 "hi" b (3 4))

(set 's '(k a l s))
(sort s) ⇒ (a k l s)

(sort '(v f r t h n m j) '>)
⇒ (v t r n m j h f)
;; the quote can be omitted beginning with version 8.4.5
(sort '(v f r t h n m j) >)
⇒ (v t r n m j h f)
(sort s <) ⇒ (a k l s)
(sort s >) ⇒ (s l k a)
s          ⇒ (s l k a)

;; define a comparison function
(define (comp x y)
  (> (last x) (last y)))

(set 'db '((a 3) (g 2) (c 5)))

(sort db comp) ⇒ ((c 5) (a 3) (g 2))

;; use an anonymous function
(sort db (fn (x y) (> (last x) (last y))))
```

source

syntax: (source)

syntax: (source *sym-1* [*sym-2* ...])

Works almost identically to [save](#), except symbols and contexts get serialized to a string instead of being written to a file. Multiple variable symbols, definitions, and contexts can be specified. If no argument is given, source serializes the entire newLISP workspace. When context symbols are serialized, any symbols contained within that context will be serialized,

as well. Symbols containing `nil` are not serialized. System symbols beginning with the \$ (dollar sign) character are only serialized when mentioned explicitly.

Symbols not belonging to the current context are written out with their context prefix.

example:

```
(define (double x) (+ x x))

(source 'double) ⇒ "(define (double x)\n  (+ x x))\n\n"
```

As with [save](#), the formatting of line breaks and leading spaces or tabs can be controlled using the [pretty-print](#) function.

sqrt

syntax: (sqrt *num*)

Calculates the square root from the expression in *num* and returns the result.

example:

```
(sqrt 10) ⇒ 3.16227766
(sqrt 25) ⇒ 5
```

starts-with

syntax: (starts-with *str str-key* [*num-option*])

syntax: (starts-with *list* [*expr*])

In the first version, `starts-with` checks if the string *str* starts with a key string in *str-key* and returns `true` or `nil` depending on the outcome.

If a regular expression number is specified in *num-option*, *str-key* contains a regular expression pattern. See [regex](#) for valid *option* numbers.

example:

```
(starts-with "this is useful" "this") ⇒ true
(starts-with "this is useful" "THIS") ⇒ nil
(starts-with "this is useful" "THIS" nil) ⇒ true
;; use regular expressions
(starts-with "this is useful" "this|that" 1) ⇒ true
```

In the second version, `starts-with` checks to see if a list starts with the list element in *expr*. `true` or `nil` is returned depending on outcome.

example:

```
(starts-with '(1 2 3 4 5) 1) ⇒ true
```

starts-with

```
(starts-with '(a b c d e) 'b)      ⇒ nil
(starts-with '(+ 3 4) b c d) '(+ 3 4)) ⇒ true
```

See also the [ends-with](#) function.

string

syntax: (string *exp-1* [*exp-2* ... *exp-n*])

Translates into a string anything that results from evaluating *exp-1*—. If more than one expression is specified, the resulting strings are concatenated.

example:

```
(string 'hello)           ⇒ "hello"
(string 1234)             ⇒ "1234"
(string '(+ 3 4))         ⇒ "(+ 3 4)"
(string (+ 3 4) 8)        ⇒ "78"
(string 'hello " " 123)   ⇒ "hello 123"
```

If a buffer passed to `string` contains `\000`, only the string up to the first terminating zero will be copied:

```
(set 'buff "ABC\000\000\000") ⇒ "ABC\000\000\000"

(length buff) ⇒ 6

(string buff) ⇒ "ABC"

(length (string buff)) ⇒ 3
```

Use the [append](#) and [join](#) (allows the joining string to be specified) functions to concatenate strings. Use the [source](#) function to convert a lambda expression into its newLISP source string representation.

string?

syntax: (string? *exp*)

Evaluates *exp* and tests to see if it is a string. Returns `true` or `nil` depending on the result.

example:

```
(set 'var "hello")
(string? var) ⇒ true
```

sub

syntax: (sub *num-1* [*num-2* ...])

Successively subtracts the expressions in *num-1*, *num-2*—. sub performs mixed-type arithmetic and handles integers or floating points, but it will always return a floating point number. If only one argument is supplied, its sign is reversed. Any floating point calculation with NaN also returns NaN.

example:

```
(sub 10 8 0.25) ⇒ 1.75
(sub 123)       ⇒ -123
```

swap

syntax: (swap *num-1* *num-2* *list*)

syntax: (swap *num-1* *num-2* *str*)

syntax: (swap *sym-1* *sym-2*)

In the first form, swap switches the elements in *list* at indices *num-1* and *num-2* and returns the changed list.

In the second form, the characters in *str* at indices *num-1* and *num-2* are swapped and the changed string is returned.

In the third form, the contents of the two unquoted symbols in *sym-1* and *sym-2* are swapped.

swap is a destructive operation that changes the contents of the list, string, or symbols involved.

example:

```
(set 'lst '(a b c d e f))

(swap 0 5 lst) ⇒ '(f b c d e a)
lst           ⇒ '(f b c d e a)

(swap 0 -1 lst) ⇒ '(a b c d e f)
lst           ⇒ '(a b c d e f)

(swap 3 4 "abcdef") ⇒ "abcedf"

(set 'x 1 'y 2)

(swap x y) ⇒ 1

x ⇒ 2
y ⇒ 1
```

sym

syntax: (sym *string* [*sym-context nil-flag*])
syntax: (sym *number* [*sym-context nil-flag*])
syntax: (sym *symbol* [*sym-context nil-flag*])

Translates the first argument in *string*, *number*, or *symbol* into a symbol and returns it. If the optional context is not specified in *sym-context*, the current context is used when doing symbol lookup or creation. Symbols will be created if they do not already exist. When the context does not exist and the context is specified by a quoted symbol, the symbol also gets created. If the context specification is unquoted, the context is the specified name or the context specification is a variable containing the context.

`sym` can create symbols within the symbol table that are not legal symbols in newLISP source code (e.g., numbers or names containing special characters such as parentheses, colons, etc.). This makes `sym` usable as a function for associative memory access, much like *hash table* access in other scripting languages.

As a third optional argument, `nil` can be specified to suppress symbol creation if the symbol is not found. In this case, `sym` returns `nil` if the symbol looked up does not exist. Using this last form, `sym` can be used to check for the existence of a symbol.

example:

```
(sym "some")           ⇒ some
(set (sym "var") 345)   ⇒ 345
var                    ⇒ 345
(sym "aSym" 'MyCTX)    ⇒ MyCTX:aSym
(sym "aSym" MyCTX)     ⇒ MyCTX:aSym ; unquoted context

(sym "foo" MyCTX nil)  ⇒ nil ; 'foo does not exist
(sym "foo" MyCTX)      ⇒ foo ; 'foo is created
(sym "foo" MyCTX nil)  ⇒ foo ; foo now exists
```

Because the function `sym` returns the symbol looked up or created, expressions with `sym` can be embedded directly in other expressions that use symbols as arguments. The following example shows the use of `sym` as a hash-like function for associative memory access, as well as symbol configurations that are not legal newLISP symbols:

example:

```
;; using sym for simulating hash tables

(set (sym "John Doe" 'MyDB) 1.234)
(set (sym "(" 'MyDB) "parenthesis open")
(set (sym 12 'MyDB) "twelve")

(eval (sym "John Doe" 'MyDB)) ⇒ 1.234
(eval (sym "(" 'MyDB))      ⇒ "parenthesis open"
(eval (sym 12 'MyDB))       ⇒ "twelve"

;; delete a symbol from a symbol table or hash
(delete (sym "John Doe" 'MyDB)) ⇒ true
```

The last statement shows how a symbol can be eliminated using [delete](#).

The third syntax allows symbols to be used instead of strings for the symbol name in the target context. In this case, `sym` will extract the name from the symbol and use it as the name string for the symbol in the target context:

example:

```
(sym 'myVar 'FOO) ⇒ FOO:myVar

(define-macro (def-context)
  (dolist (s (rest (args)))
    (sym s (first (args)))))

(def-context foo x y z)

(symbols foo) ⇒ (foo:x foo:y foo:z)
```

The `def-context` macro shows how this could be used to create a macro that creates contexts and their variables in a dynamic fashion.

Available since version 8.7.4, one syntax of the [context](#) function can be used to create, set, and evaluate symbols in a shorter, faster way.

symbol?

syntax: (symbol? *exp*)

Evaluates the *exp* expression and returns `true` if the value is a symbol; otherwise, it returns `nil`.

example:

```
(set 'x 'y) ⇒ y

(symbol? x) ⇒ true

(symbol? 123) ⇒ nil

(symbol? (first '(var x y z))) ⇒ true
```

The first statement sets the contents of `x` to the symbol `y`. The second statement then checks the contents of `x`. The last example checks the first element of a list.

symbols

syntax: (symbols [*context*])

Returns a sorted list of all symbols in the current context when called without an argument. If a context symbol is specified, symbols defined in that context are returned.

example:

```
(symbols)           ; list of all symbols in current context
(symbols 'CTX)      ; list of symbols in context CTX
(symbols CTX)       ; omitting the quote
(set 'ct CTX)       ; assigning context to a variable
(symbols ct)        ; list of symbols in context CTX
```

The quote can be omitted because contexts evaluate to themselves.

sys-error**syntax: (sys-error)**

Reports error numbers generated by the underlying OS newLISP is running on. The error numbers reported may differ on the platforms newLISP has been compiled for. Consult the platform's C library information, (e.g., the GNU `libc` reference). Most errors reported refer to system resources such as files and semaphores.

Whenever a function in newLISP within the system resources area returns `nil`, `sys-error` can be checked for the underlying reason. For file operations, `sys-error` may be set for nonexistent files or wrong permissions when accessing the resource. Another cause of error could be the exhaustion of certain system resources like file handles or semaphores.

example:

```
;; trying to open a nonexistent file
(open "blahbla" "r") ⇒ nil

(sys-error) ⇒ 2

(sys-error 0) ⇒ 0 ; clear errno
```

The error number can be cleared by giving a 0 (zero) for the optional argument.

sys-info**syntax: (sys-info [*int-idx*])**

Calling `sys-info` without *int-idx* returns a list of internal resource statistics. Eight integers report the following status:

- 0 - Number of LISP cells
- 1 - Maximum number of LISP cells constant
- 2 - Number of symbols
- 3 - Evaluation/recursion level
- 4 - Environment stack level
- 5 - Maximum call stack constant

- 6 - Version number as an integer constant
- 7 - Operating system constant:
 linux=1, bsd=2, osx=3, solaris=4, cygwin=5, win32=6
 the highest bit 7 will be set for UTF-8 versions (add 128)
 bit 6 will be added for library versions (add 64)

The numbers from 0 to 7 indicate the optional offset in the returned list.

When using *int-idx*, one element of the list will be returned.

example:

```
(sys-info)      ⇒ (348 268435456 269 1 0 1024 8404 6)
(sys-info 3)    ⇒ 1
(sys-info -2)   ⇒ 8404
```

The number for the maximum of LISP cells can be changed via the *-m* command-line switch. For each megabyte of LISP cell memory, 64k memory cells can be allocated. The maximum call stack depth can be changed using the *-s* command-line switch.

tan

syntax: (tan *num-radians*)

Calculates the tangent function from *num-radians* and returns the result.

example:

```
(tan 1)                ⇒ 1.557407725
(set 'pi (mul 2 (asin 1))) ⇒ 3.141592654
(tan (div pi 4))       ⇒ 1
```

throw

syntax: (throw *exp*)

Works together with the [catch](#) function. *throw* forces the return of a previous *catch* statement and puts the *exp* into the result symbol of *catch*.

example:

```
(define (throw-test)
  (dotimes (x 1000)
    (if (= x 500) (throw "interrupted"))))

(catch (throw-test) 'result) ⇒ true
result ⇒ "interrupted"
(catch (throw-text)) ⇒ "interrupted"
```

The last example shows a shorter form of [catch](#), which returns the `throw` result directly.

`throw` is useful for breaking out of a loop or for early return from user-defined functions or expression blocks. In the following example, the `begin` block will return `X` if `(foo X)` is `true`; else `Y` will be returned:

```
(catch (begin
  ...
  (if (foo X) (throw X) Y)
  ...
))
```

`throw` will *not* cause an error exception. Use [throw-error](#) to throw user error exceptions.

throw-error

syntax: (throw-error expr)

Causes a user-defined error exception with text provided by evaluating *expr*.

example:

```
(define (foo x y)
  (if (= x 0) (throw-error "first argument cannot be 0"))
  (+ x y))

(foo 1 2) ⇒ 3

(foo 0 2) ; causes a user error exception
user error : first argument cannot be 0
called from user-defined function foo
```

The user error can be handled like any other error exception using user-defined error handlers and the [error-event](#) function, or the form of [catch](#) that can capture error exceptions.

time

syntax: (time exp [int-count])

Evaluates the expression in *exp* and returns the time spent on evaluation in milliseconds.

example:

```
(time (myprog x y z)) ⇒ 450

(time (myprog x y z) 10) ⇒ 4420
```

In first the example, 450 milliseconds elapsed while evaluating `(myprog x y z)`. The second example returns the time for ten evaluations of `(myprog x y z)`. See also [date](#), [date-value](#), [time-of-day](#), and [now](#).

time-of-day

syntax: (time-of-day)

Returns the time in milliseconds since the start of the current day.

See also the [date](#), [date-value](#), [time](#), and [now](#) functions.

timer

syntax: (timer *sym-event-handler* *num-seconds* [*int-option*])

syntax: (timer *func-event-handler* *num-seconds* [*int-option*])

syntax: (timer *sym-event-handler*)

syntax: (timer *func-event-handler*)

syntax: (timer)

Starts a one-shot timer firing off the Unix signal SIGALRM, SIGVTALRM, or SIGPROF after the time in seconds (specified in *num-seconds*) has elapsed. When the timer fires, it calls the user-defined function in *sym-event-handler*.

On Linux/UNIX, an optional 0, 1, or 2 can be specified to control how the timer counts. With default option 0, real time is measured. Option 1 measures the time the CPU spends processing in the thread or process owning the timer. Option 3 is a combination of both called *profiling time*. See the UNIX man page `setitimer()` for details.

The event handler can start the timer again to achieve a continuous flow of events. Starting with version 8.5.9, seconds can be defined as floating point numbers with a fractional part (e.g., 0.25 for 250 milliseconds).

Defining 0 (zero) as time shuts the running timer down and prevents it from firing.

When called with *sym-event-handler*, `timer` returns the elapsed time of the timer in progress. This can be used to program timelines or schedules.

`timer` called without arguments returns the symbol of the current event handler.

example:

```
(define (ticker)
  (println (date)) (timer 'ticker 1.0))

> (ticker)
Tue Apr 12 20:44:48 2005      ; first execution of ticker
⇒ ticker                     ; return value from ticker
> Tue Apr 12 20:44:49 2005    ; first timer event
Tue Apr 12 20:44:50 2005      ; second timer event ...
Tue Apr 12 20:44:51 2005
Tue Apr 12 20:44:52 2005
Tue Apr 12 20:44:53 2005
Tue Apr 12 20:44:54 2005
```

Tue Apr 12 20:44:55 2005

The example shows an event handler, `ticker`, which starts the timer again after each event.

Note that a timer cannot interrupt an ongoing built-in function. The timer interrupt gets registered by newLISP, but a timer handler cannot run until one expression is evaluated and the next one starts. To interrupt an ongoing I/O operation with `timer`, use the following pattern, which calls [net-select](#) to test if a socket is ready for reading:

example:

```
define (interrupt)
  (set 'timeout true))

(set 'listen (net-listen 30001))
(set 'socket (net-accept listen))

(timer 'interrupt 10)
;; or specifying the function directly
(timer (fn () (set 'timeout true)) 10)

(until (or timeout done)
  (if (net-select socket "read" 100000)
    (begin
      (read-buffer socket 'buffer 1024)
      (set 'done true)))
  )

(if timeout
  (println "timeout")
  (println buffer))

(exit)
```

In this example, the `until` loop will run until something can be read from `socket`, or until ten seconds have passed and the `timeout` variable is set.

title-case

syntax: (title-case *str*)

syntax: (title-case *str bool*)

Returns a copy of the string in *str* with the first character converted to uppercase. When the optional *bool* parameter evaluates to any value other than `nil`, the rest of the string is converted to lowercase.

example:

```
(title-case "hello")      ⇒ "Hello"
(title-case "hELLO" true) ⇒ "Hello"
```

See also the [lower-case](#) and [upper-case](#) functions.

trace

syntax: (trace [*exp*])

Tracing is switched on when *exp* evaluates to anything besides `nil` or an empty list `()`. When no argument is supplied, `trace` evaluates to `true` or `nil` depending on the current trace mode. If trace mode is switched on, newLISP goes into debugging mode, displaying the function currently being executed and highlighting the current expression upon entry and exit. The highlighting is done by bracketing the expression between two `#` (number sign) characters. This can be changed to a different character using [trace-highlight](#). Upon exit from the expression, the result of its evaluation is also reported.

If an expression occurs more than once in a function, the first occurrence of the executing function will always be highlighted (bracketed).

newLISP execution stops with a prompt line at each entry and exit of an expression.

```
[ -> 2] s|tep n|ext c|ont q|uit >
```

At the prompt, an `s`, `n`, `c`, or `q` can be entered to step into or merely execute the next expression. Any expression can be entered at the prompt for evaluation. Entering the name of a variable, for example, would evaluate to its contents. In this way, a variable's contents can be checked during debugging or set to different values.

example:

```
;; switches newLISP into debugging mode
(trace true) ⇒ true

;; the debugger will show each step
(my-func a b c)

;; switched newLISP out of debugging mode
(trace nil) ⇒ nil
```

To set break points where newLISP should interrupt normal execution and go into debugging mode, put `(trace true)` statements into the LISP code where execution should switch on the debugger.

Use the [debug](#) function as a shortcut for the above example.

trace-highlight

syntax: (trace-highlight *str-pre str-post* [*str-header str-footer*])

Sets the characters or string of characters used to enclose expressions during [trace](#). By default, the `#` (number sign) is used to enclose the expression highlighted in [trace](#) mode. This can be changed to different characters or strings of up to seven characters. If the console window accepts terminal control characters, this can be used to display the expression in a different color, bold, reverse, and so forth.

Two more strings can optionally be specified for *str-header* and *str-footer*, which control the separator and prompt. A maximum of 15 characters is allowed for the header and 31 for the footer.

example:

```
;; active expressions are enclosed in >> and <<
(trace-highlight ">>" "<<")

;; 'bright' color on a VT100 or similar terminal window
(trace-highlight "\027[1m" "\027[0m")
```

The first example replaces the default # (number sign) with a >> and <<. The second example works on most Linux shells. It may not, however, work in console windows under Win32 or CYGWIN, depending on the configuration of the terminal.

transpose

syntax: (transpose *matrix*)

Transposes a *matrix* by reversing the rows and columns and converting all of the cells to floating point numbers. Any kind of list-matrix can be transposed. Matrices are made rectangular by filling in *nil* for missing elements, omitting elements where appropriate, or expanding atoms in rows into lists. Matrix dimensions are calculated using the number of rows in the original matrix for columns and the number of elements in the first row as number of rows for the transposed matrix.

The dimensions of a matrix are defined by the number of rows and the number of elements in the first row. A matrix can either be a nested list or an [array](#).

example:

```
(set 'A '((1 2 3) (4 5 6)))
(transpose A)           ⇒ ((1 4) (2 5) (3 6))
(transpose (list (sequence 1 5))) ⇒ ((1) (2) (3) (4) (5))

(transpose '((a b) (c d) (e f))) ⇒ ((a c e) (b d f))
```

The number of columns in a matrix is defined by the number of elements in the first row of the matrix. If other rows have fewer elements, *transpose* will assume *nil* for those missing elements. Superfluous elements in a row will be ignored.

```
(set 'A '((1 2 3) (4 5) (7 8 9)))
(transpose A) ⇒ ((1 4 7) (2 5 8) (3 nil 9))
```

If a row is any other data type besides a list, the transposition treats it like an entire row of elements of that data type:

```
(set 'A '((1 2 3) X (7 8 9)))
```

```
(transpose A) ⇒ ((1 X 7) (2 X 8) (3 X 9))
```

All operations shown here on lists can also be performed on arrays.

See also the matrix operations [det](#), [invert](#), and [multiply](#).

trim

syntax: (trim *str* [*str-char*])

syntax: (trim *str* [*str-left-char*] [*str-right-char*])

The first syntax trims the string *str* from both sides, stripping the leading and trailing characters as given in *str-char*. If *str-char* contains no character, the space character is assumed. `trim` returns the new string.

The second syntax can either trim different characters from both sides or trim only one side if an empty string is specified for the other.

example:

```
(trim "  hello ") ⇒ "hello"
(trim "----hello-----" "-") ⇒ "hello"
(trim "00012340" "0" "") ⇒ "12340"
(trim "1234000" "" "0") ⇒ "1234"
(trim "----hello=====" "- " "=") ⇒ "hello"
```

true?

syntax: (true? *expr*)

If the expression in *expr* evaluates to anything other than `nil`, `true?` returns `true`; else it returns `nil`.

example:

```
(map true? '(x nil 1 nil "hi" ()))
⇒ (true nil true nil true true)
(true? nil) ⇒ nil
(true? '()) ⇒ true
```

The `true?` predicate is useful for distinguishing between `nil` and the empty list `()`.

unicode

syntax: (unicode *str*)

Converts ASCII/UTF-8 character strings in *str* to UCS-4–encoded Unicode of 4-byte integers per character. This function is only available on UTF-8–enabled versions of newLISP.

example:

```
(unicode "new")
⇒ "n\000\000\000e\000\000\000w\000\000\000\000\000\000\000"

(utf8 (unicode "new")) ⇒ "new"
```

On *big endian* CPU architectures, the byte order will be reversed from high to low. The `unicode` and `utf8` functions are the inverse of each other. These functions are only necessary if UCS-4 Unicode is in use. Most systems use UTF-8 encoding only.

unify

syntax: (unify *expr-1* *expr-2* [*list-env*])

Evaluates and matches *expr-1* and *expr-2*. Expressions match if they are equal or if one of the expressions is an unbound variable (which would then be bound to the other expression). If expressions are lists, they are matched by comparing subexpressions. Unbound variables start with an uppercase character to distinguish them from symbols. `unify` returns `nil` when the unification process fails, or it returns a list of variable associations on success. When no variables were bound, but the match is still successful, `unify` returns an empty list.

example:

```
(unify 'A 'A) ⇒ () ; tautology

(unify 'A 123) ⇒ ((A 123)) ; A bound to 123

(unify '(A B) '(x y)) ⇒ ((A x) (B y)) ; A bound to x, B bound
to y

(unify '(A B) '(B abc)) ⇒ ((A abc) (B abc)) ; B is alias for A

(unify 'abc 'xyz) ⇒ nil ; fails because symbols are different

(unify '(A A) '(123 456)) ⇒ nil ; fails because A cannot be
bound to different values

(unify '(f A) '(f B)) ⇒ ((A B)) ; A and B are aliases

(unify '(f A) '(g B)) ⇒ nil ; fails because heads of terms are
different

(unify '(f A) '(f A B)) ⇒ nil ; fails because terms are of
different arity

(unify '(f (g A)) '(f B)) ⇒ ((B (g A))) ; B bound to (g A)

(unify '(f (g A) A) '(f B xyz)) ⇒ ((B (g xyz)) (A xyz)) ; B
bound to (g xyz) A to xyz
```


newLISP Users Manual and Reference

```
(unify '(f A) 'A) ⇒ nil ; fails because of infinite
unification (f(f(f ...)))

(unify '(A xyz A) '(abc X X)) ⇒ nil ; indirect alias A to X
doesn't match bound terms

(unify '(p X Y a) '(p Y X X)) ⇒ '((Y a) (X a)) ; X alias Y
and binding to 'a

(unify '(q (p X Y) (p Y X)) '(q Z Z)) ⇒ ((Y X) (Z (p X X))) ;
indirect alias

;; some examples taken from
http://en.wikipedia.org/wiki/Unification
```

`unify` can take an optional binding or association list in *list-env*. This is useful when chaining `unify` expressions and the results of previous `unify` bindings must be included:

example:

```
(unify '(f X) '(f 123)) ⇒ ((X 123))

(unify '(A B) '(X A) '((X 123)))
⇒ ((X 123) (A 123) (B 123))
```

In the previous example, X was bound to 123 earlier and is included in the second statement to pre-bind X.

Note that variables are not actually bound as a newLISP assignment. Rather, an association list is returned showing the logical binding. A special syntax of [expand](#) can be used to actually replace bound variables with their terms:

```
(set 'bindings (unify '(f (g A) A) '(f B xyz)))
⇒ ((B (g xyz)) (A xyz))

(expand '(f (g A) A) bindings) ⇒ (f (g xyz) xyz)

; or in one statement
(expand '(f (g A) A) (unify '(f (g A) A) '(f B xyz)))
⇒ (f (g xyz) xyz)
```

The following example shows how propositional logic can be modeled using `unify` and [expand](#):

```
; if somebody is human, he is mortal -> (X human) :- (X mortal)
; socrates is human -> (socrates human)
; is socrates mortal? -> ? (socrates mortal)

(expand '(X mortal)
  (unify '(X human) '(socrates human)))
⇒ (socrates mortal)
```

The following is a more complex example showing a small, working PROLOG (Programming in Logic) implementation.

```
;; a small PROLOG implementation

(set 'facts '(
  (socrates philosopher)
```

```

(socrates greek)
(socrates human)
(einstein german)
(einstein (studied physics))
(einstein human)
))

(set 'rules '(
  ((X mortal) <- (X human))
  ((X (knows physics)) <- (X physicist))
  ((X physicist) <- (X (studied physics)))
))

(define (query term)
  (or (if (find term facts) true) (catch (prove-rule term))))

(define (prove-rule term)
  (dolist (r rules)
    (if (list? (set 'e (unify term (first r))))
        (if (query (expand (last r) e))
            (throw true))))
  nil
)

; try it

> (query '(socrates human))
true
> (query '(socrates (knows physics)))
nil
> (query '(einstein (knows physics)))
true

```

The program handles a database of facts and a database of simple *A is a fact if B is a fact* rules. A fact is proven true if it either can be found in the facts database or if it can be proven using a rule. Rules can be nested: for example, to prove that somebody (knows physics), it must be proved true that somebody is a physicist. But somebody is only a physicist if that person studied physics. The <- symbol separating the left and right terms of the rules is not required and is only added to make the rules database more readable.

This implementation does not handle multiple terms in the right premise part of the rules, but it does handle backtracking of the rules database to try out different matches. It does not handle backtracking in multiple premises of the rule. For example, if in the following rule A if B and C and D, the premises B and C succeed and D fails, a backtracking mechanism might need to go back and reunify the B or A terms with different facts or rules to make D succeed.

The above algorithm could be written differently by omitting [expand](#) from the definition of prove-rule and by passing the environment, e, as an argument to the unify and query functions.

A *learning* of proven facts can be implemented by appending them to the facts database once they are proven. This would speed up subsequent queries.

Larger PROLOG implementations also allow the evaluation of terms in rules. This makes it possible to implement functions for doing other work while processing rule terms. `prove-rule` could accomplish this testing for the symbol `eval` in each rule term.

unique

syntax: (**unique** *list*)

Returns a unique version of *list* with all duplicates removed.

example:

```
(unique '(2 3 4 4 6 7 8 7)) ⇒ (2 3 4 6 7 8)
```

Note that the list does not need to be sorted, but a sorted list makes `unique` perform faster.

The functions [difference](#) and [intersect](#) work with sets.

unless

syntax: (**unless** *exp-condition exp-1 [exp-2]*)

`unless` is equivalent to (`if` (`not` *exp-condition exp-1 [exp-2]*)). If the value of *exp-condition* is `nil` or the empty list `()`, *exp-1* is evaluated; otherwise, *exp-2* is evaluated.

example:

```
(set 'x 50)                ⇒ 50
(unless (< x 100) "big" "small") ⇒ "small"
(set 'x 1000)              ⇒ 1000
(unless (< x 100) "big" "small") ⇒ "big"
```

unpack

syntax: (**unpack** *str-format str-addr-packed*)

Unpacks a binary structure in *str-addr-packed* into LISP variables using the format in *str-format*. `unpack` is the reverse operation of `pack`. Note that *str-addr-packed* may also be an integer representing a memory address. This facilitates the unpacking of structures returned from imported, shared library functions.

The following characters may define a format:

<i>forma</i> <i>t</i>	<i>description</i>
--------------------------	--------------------

c	a signed 8-bit number
b	an unsigned 8-bit number
d	a signed 16-bit short number
u	an unsigned 16-bit short number
ld	a signed 32-bit long number
lu	an unsigned 32-bit long number
f	a float in 32-bit representation
lf	a double float in 64-bit representation
sn	a string of <i>n</i> null padded ASCII characters
nn	<i>n</i> null characters
>	switches to big endian byte order
<	switches to little endian byte order

example:

```
(pack "c c c" 65 66 67) ⇒ "ABC"
(unpack "c c c" "ABC") ⇒ (65 66 67)

(set 's (pack "c d u" 10 12345 56789))
(unpack "c d u" s) ⇒ (10 12345 56789)

(set 's (pack "s10 f" "result" 1.23))
(unpack "s10 f" s) ⇒ ("result\000\000\000\000" 1.230000019)

(set 's (pack "s3 lf" "result" 1.23))
(unpack "s3 f" s) ⇒ ("res" 1.23)

(set 's (pack "c n7 c" 11 22))
(unpack "c n7 c" s) ⇒ (11 22)
```

The > and < specifiers can be used to switch between *little endian* and *big endian* byte order when packing or unpacking:

```
;; on a little endian system (e.g., Intel CPUs)
(set 'buff (pack "d" 1)) ⇒ "\001\000"

(unpack "d" buff) ⇒ (1)
(unpack ">d" buff) ⇒ (256)
```

Switching the byte order will affect all number formats with 16-, 32-, or 64-bit sizes.

The pack and unpack format need not be the same, as in the following example:

```
(set 's (pack "s3" "ABC"))
(unpack "c c c" s) ⇒ (65 66 67)
```

The examples show spaces between the format specifiers. Although not required, they can improve readability.

See also the [address](#), [get-int](#), [get-char](#), [get-string](#) and [pack](#) functions.

until

syntax: (until *exp-condition body*)

Evaluates the condition in *exp-condition body*. If the result is `nil` or the empty list `()`, the expressions in *body* are evaluated. Evaluation is repeated until the *exp-condition* results in a value other than `nil` or the empty list. The result of the last expression evaluated is the return value of the `until` expression. `until` works like ([while](#) ([not](#) ...)).

example:

```
(device (open "somefile.txt" "read"))
(set 'line-count 0)
(until (not (read-line)) (inc 'line-count))
(close (device))
(print "the file has " line-count " lines\n")
```

Use the [do-until](#) function to test the condition *after* evaluation of the body expressions.

upper-case

syntax: (upper-case *str*)

Returns a copy of the string in *str* converted to uppercase. International characters are converted correctly.

example:

```
(upper-case "hello world") ⇒ "HELLO WORLD"
```

See also the [lower-case](#) and [title-case](#) functions.

utf8

syntax: (utf8 *str*)

Converts a UCS-4, 4-byte, Unicode-encoded string (*str*) into UTF-8. This function is only available on UTF-8-enabled versions of newLISP.

example:

```
(unicode "new")
⇒ "n\000\000\000e\000\000\000w\000\000\000\000\000\000"

(utf8 (unicode "new")) ⇒ "new"
```

The `utf8` function can also be used to test for the presence of UTF-8-enabled newLISP:

```
(if utf8 (do-utf8-version-of-code) (do-ascii-version-of-code))
```

On *big endian* CPU architectures, the byte order will be reversed from highest to lowest. The `utf8` and `unicode` functions are the inverse of each other. These functions are only necessary if UCS-4 Unicode is in use. Most systems use UTF-8 Unicode encoding only.

uuid

syntax: (uuid [*str-node*])

Constructs and returns a UUID (Universally Unique Identifier). Without a node spec in *str-node*, a type 4 UUID random generated byte number is returned. When the optional *str-node* parameter is used, a type 1 UUID is returned. The string in *str-node* specifies a valid MAC (Media Access Code) from a network adapter installed on the node or a random node ID. When a random node ID is specified, the least significant bit of the first node byte should be set to 1 to avoid clashes with real MAC identifiers. UUIDs of type 1 with node ID are generated from a timestamp and other data. See [RFC 4122](#) for details on UUID generation.

example:

```
;; type 4 UUID for any system
(uuid) ⇒ "493AAD61-266F-48A9-B99A-33941BEE3607"

;; type 1 UUID preferred for distributed systems
;; configure node ID for ether 00:14:51:0a:e0:bc
(set 'id (pack "cccccc" 0x00 0x14 0x51 0x0a 0xe0 0xbc))

(uuid id) ⇒ "0749161C-2EC2-11DB-BBB2-0014510AE0BC"
```

Each invocation of the `uuid` function will yield a new unique UUID. The UUIDs are generated without systemwide shared stable store (see RFC 4122). If the system generating the UUIDs is distributed over several nodes, then type 1 generation should be used with a different node ID on each node. For several processes on the same node, valid UUIDs are guaranteed even if requested at the same time. This is because the process ID of the generating newLISP process is part of the seed for the random number generator. When type 4 IDs are used on a distributed system, two identical UUID's are still highly unlikely and impossible for type 1 IDs if real MAC addresses are used.

wait-pid

syntax: (wait-pid *int-pid* [*int-options*])

Waits for a child process specified in *int-pid* to end. The child process was previously started with `process` or `fork`. When the child process specified in *int-pid* ends, a status value describing the reason for termination of the child process or thread is returned. The interpretation of the returned status value differs between Linux and other flavors of UNIX.

Consult the Linux/UNIX man pages for the `waitpid` command (without the hyphen used in newLISP) for further information.

When `-1` is specified for *int-pid*, the status information of any child process started is returned. When `0` is specified, `wait-pid` only watches child processes in the same process group as the calling process. Any other negative value for *int-pid* reports child processes in the same process group as specified with a negative sign in *int-pid*.

This function is only available on Linux and other UNIX-like operating systems or on a CYGWIN compiled version of newLISP on Win32. An option can be specified in *int-option*. See Linux/UNIX documentation for details on options.

example:

```
(set 'pid (fork (my-thread)))
(set 'status (wait-pid pid)) ; wait until my-thread ends
(println "thread: " pid " has finished with status: " status)
```

The process `my-thread` is started, then the main program blocks in the `wait-pid` call until `my-thread` has finished.

while

syntax: (while *exp-condition body*)

Evaluates the condition in *exp-condition*. If the result is not `nil` or the empty list `()`, the expressions in *body* are evaluated. Evaluation is repeated until an *exp-condition* results in `nil` or the empty list `()`. The result of the body's last expression is the return value of the `while` expression.

example:

```
(device (open "somefile.txt" "read"))
(set 'line-count 0)
(while (read-line) (inc 'line-count))
(close (device))
(print "the file has " line-count " lines\n")
```

Use the [do-while](#) function to evaluate the condition *after* evaluating the body of expressions.

write-buffer

syntax: (write-buffer *int-file sym-buffer [int-size]*)

syntax: (write-buffer *int-file str-buffer [int-size]*)

syntax: (write-buffer *str-device* *sym-buffer* [*int-size*])

syntax: (write-buffer *str-device* *str-buffer* [*int-size*])

Using the first syntax, `write-buffer` writes *int-size* bytes from a buffer in *sym-buffer* or *str-buffer* to a file specified in *int-file*, previously obtained from a file open operation. If *int-size* is not specified, all data in *sym-buffer* or *str-buffer* is written. `write-buffer` returns the number of bytes written or `nil` on failure.

The string buffer symbol can be used with or without quoting a symbol.

example:

```
(set 'handle (open "myfile.ext" "write"))
(write-buffer handle 'data 100)

;; string buffer w/o quote
(write-buffer handle data 100)
(write-buffer handle "a quick message\n")
```

The code in the example writes 100 bytes to the file `myfile.ext` from the contents in `data`.

Using the second syntax, `write-buffer` appends contents from a string specified in *sym-buffer* or *str-buffer* to the string specified in *str-device*, which acts like a stream device.

example:

```
;; fast in-place string appending
(set 'str "")
(dotimes (x 5) (write-buffer str "hello"))

str ⇒ "HelloHelloHelloHelloHello")

;; much slower method of string concatenation
(dotimes (x 5) (set 'str (append str "hello")))
```

The above example appends a string to `str` five times. This method is much faster than using [append](#) when concatenating to a string in place.

See also the [read-buffer](#) function.

write-char

syntax: (write-char *int-file* *int-byte*)

Writes a byte specified in *int-byte* to a file specified by the file handle in *int-file*. The file handle is obtained from a previous open operation. Each `write-char` advances the file pointer by one byte.

example:

```
(define (slow-file-copy from-file to-file)
  (set 'in-file (open from-file "read"))
  (set 'out-file (open to-file "write"))
  (while (set 'chr (read-file in-file))
    (write-char out-file chr))
  (close in-file))
```



```
(close out-file)
"finished")
```

Use the [print](#) and [device](#) functions to write larger portions of data at a time. Note that newLISP already supplies a faster built-in function called [copy-file](#).

See also the [read-char](#) function.

write-file

syntax: (write-file *str-file-name* *str-buffer*)

Writes a file in *str-file-name* with contents in *str-buffer* in one swoop and returns the number of bytes written.

example:

```
(write-file "myfile.enc"
  (encrypt (read-file "/home/lisp/myFile") "secret"))
```

The file `myfile` is read, [encrypted](#) using the password `secret`, and written back into the new file `myfile.enc` in the current directory.

`write-file` can take an `http://` or `file://` URL in *str-file-name*. In this case, `write-file` works exactly like [put-url](#) and can take the same additional parameters:

example:

```
(write-file "http://asite.com/message.txt" "This is a message" )
```

The file `message.txt` is created and written at a remote location, `http://asite.com`, with the contents of *str-buffer*. In this mode, `write-file` can also be used to transfer files to remote newLISP server nodes.

See also the [append-file](#) and [read-file](#) functions.

write-line

syntax: (write-line [*str*] [*int-file*])

syntax: (write-line [*str*] [*str-device*])

The string in *str* and the line termination character(s) are written to the console or a file. If no file handle is specified in *int-file*, `write-line` writes to the current device, normally the console screen. When all arguments are omitted, `write-line` writes the contents of the last [read-line](#) to the screen.

example:

```
(write-line "hello there")
```

```

(set 'out-file (open "myfile" "write"))
(write-line "hello there" out-file)
(close out-file)

(set 'myFile (open "init.lsp" "read"))
(while (read-line myFile) (write-line))

;; using a string device:

(set 'str "")
(dotimes (x 4) (write-line "hello" str))

str ⇒ "hello\r\nhello\r\nhello\r\nhello\r\n" ; on Win32

str ⇒ "hello\nhello\nhello\nhello\n" ; on Linux/UNIX

```

The first example puts a string out on the current [device](#), which is probably the console window (`device 0`). The second example opens/creates a file, writes a line to it, and closes the file. The third example shows the usage of `write-line` without arguments. The contents of `init.lsp` are written to the console screen.

In the second syntax, a string can be specified as a device in *str-device* (like the [write-buffer](#) function). When a string device is written to, the string in *str-device* gets appended with *str* and the line termination character(s).

xml-error

syntax: (xml-error)

Returns a list of error information from the last [xml-parse](#) operation; otherwise, returns `nil` if no error occurred. The first element contains text describing the error, and the second element is a number indicating the last scan position in the source XML text, starting at 0 (zero).

example:

```

(xml-parse "<atag>hello</atag><fin") ⇒ nil

(xml-error) ⇒ ("expected closing tag: >" 18)

```

xml-parse

syntax: (xml-parse *string-xml* [*int-options sym-ontext*])

Parses a string containing XML 1.0 compliant, *well-formed* XML. `xml-parse` does not perform DTD validation. It skips DTDs (Document Type Declarations) and processing instructions. Nodes of type ELEMENT, TEXT, CDATA, and COMMENT are parsed, and a newLISP list structure is returned. When an element node does not have attributes or child nodes, it instead contains an empty list. Attributes are returned as association lists, which

can be accessed using [assoc](#). When `xml-parse` fails due to malformed XML, `nil` is returned and [xml-error](#) can be used to access error information.

example:

```
(set 'xml
  "<person name='John Doe' tel='555-1212'>nice guy</person>")

(xml-parse xml)
⇒ ((("ELEMENT" "person"
      (("name" "John Doe")
       ("tel" "555-1212"))
      ("TEXT" "nice guy"))))
```

Modifying the translation process.

Optionally, the *int-options* parameter can be specified to suppress whitespace, empty attribute lists, and comments. It can also be used to transform tags from strings into symbols. Another function, [xml-type-tags](#), serves for translating the XML tags. The following option numbers can be used:

<i>option</i>	<i>description</i>
1	suppress whitespace text tags
2	suppress empty attribute lists
4	suppress comment tags
8	translate string tags into symbols
16	add SXML (S-expression XML) attribute tags

Options can be combined by adding the numbers (e.g., 3 would combine the options for suppressing whitespace text tags/info and empty attribute lists).

The following examples show how the different options can be used:

XML source:

```
<?xml version="1.0" ?>
<DATABASE name="example.xml">
<!--This is a database of fruits-->
  <FRUIT>
    <NAME>apple</NAME>
    <COLOR>red</COLOR>
    <PRICE>0.80</PRICE>
  </FRUIT>

  <FRUIT>
    <NAME>orange</NAME>
    <COLOR>orange</COLOR>
    <PRICE>1.00</PRICE>
  </FRUIT>

  <FRUIT>
    <NAME>banana</NAME>
    <COLOR>yellow</COLOR>
    <PRICE>0.60</PRICE>
  </FRUIT>
</DATABASE>
```

Parsing without any options:

```
(xml-parse (read-file "example.xml"))
⇒ ((("ELEMENT" "DATABASE" (("name" "example.xml")) ("TEXT"
"\r\n\t")
("COMMENT" "This is a database of fruits")
("TEXT" "\r\n\t")
("ELEMENT" "FRUIT" () (("TEXT" "\r\n\t\t") ("ELEMENT" "NAME"
()
(("TEXT" "apple")))
("TEXT" "\r\n\t\t")
("ELEMENT" "COLOR" () (("TEXT" "red")))
("TEXT" "\r\n\t\t")
("ELEMENT" "PRICE" () (("TEXT" "0.80")))
("TEXT" "\r\n\t\t"))
("TEXT" "\r\n\r\n\t")
("ELEMENT" "FRUIT" () (("TEXT" "\r\n\t\t") ("ELEMENT" "NAME"
()
(("TEXT" "orange")))
("TEXT" "\r\n\t\t")
("ELEMENT" "COLOR" () (("TEXT" "orange")))
("TEXT" "\r\n\t\t")
("ELEMENT" "PRICE" () (("TEXT" "1.00")))
("TEXT" "\r\n\t\t"))
("TEXT" "\r\n\r\n\t")
("ELEMENT" "FRUIT" () (("TEXT" "\r\n\t\t") ("ELEMENT" "NAME"
()
(("TEXT" "banana")))
("TEXT" "\r\n\t\t")
("ELEMENT" "COLOR" () (("TEXT" "yellow")))
("TEXT" "\r\n\t\t")
("ELEMENT" "PRICE" () (("TEXT" "0.60")))
("TEXT" "\r\n\t\t"))
("TEXT" "\r\n"))))
```

The TEXT elements containing only whitespace make the output very confusing. As the database in `example.xml` only contains data, we can suppress whitespace and comments with option (+ 1 3):

Filtering whitespace TEXT, COMMENT tags, and empty attribute lists:

```
(xml-parse (read-file "example.xml") (+ 1 2 4))
⇒ ((("ELEMENT" "DATABASE" (("name" "example.xml")) (
("ELEMENT" "FRUIT" () (
("ELEMENT" "NAME" (("TEXT" "apple")))
("ELEMENT" "COLOR" (("TEXT" "red")))
("ELEMENT" "PRICE" (("TEXT" "0.80")))))
("ELEMENT" "FRUIT" (
("ELEMENT" "NAME" (("TEXT" "orange")))
("ELEMENT" "COLOR" (("TEXT" "orange")))
("ELEMENT" "PRICE" (("TEXT" "1.00")))))
("ELEMENT" "FRUIT" (
("ELEMENT" "NAME" (("TEXT" "banana")))
("ELEMENT" "COLOR" (("TEXT" "yellow")))
("ELEMENT" "PRICE" (("TEXT" "0.60"))))))))
```

The resulting output looks much more readable, but it can still be improved by using symbols instead of strings for the tags "FRUIT", "NAME", "COLOR", and "PRICE", as well as by

suppressing the XML type tags "ELEMENT" and "TEXT" completely using the [xml-type-tags](#) directive.

Suppressing XML type tags with [xml-type-tags](#) and translating string tags into symbol tags:

```
;; suppress all XML type tags for TEXT and ELEMENT
;; instead of "CDATA", use cdata and instead of "COMMENT", use
!--

(xml-type-tags nil 'cdata '!-- nil)

;; turn on all options for suppressing whitespace and empty
;; attributes, translate tags to symbols

(xml-parse (read-file "example.xml") (+ 1 2 8))
⇒ ((DATABASE (("name" "example.xml"))
  (!-- "This is a database of fruits")
  (FRUIT (NAME "apple") (COLOR "red") (PRICE "0.80"))
  (FRUIT (NAME "orange") (COLOR "orange") (PRICE "1.00"))
  (FRUIT (NAME "banana") (COLOR "yellow") (PRICE "0.60"))))
```

When tags are translated into symbols by using option 8, a context can be specified in *sym-context*. If no context is specified, all symbols will be created inside the current context.

```
(xml-type-tags nil nil nil nil)
(xml-parse "<msg>Hello World</msg>" (+ 1 2 4 8 16) 'CTX)
⇒ ((CTX:msg "Hello World"))
```

Specifying nil for the XML type tags TEXT and ELEMENT makes them disappear. At the same time, parentheses of the child node list are removed so that child nodes now appear as members of the list, starting with the tag symbol translated from the string tags "FRUIT", "NAME", etcetera.

Parsing into SXML (S-expressions XML) format:

Using [xml-type-tags](#) to suppress all XML-type tags—along with the option numbers 1, 4, 8, and 16—SXML formatted output can be generated:

```
(xml-type-tags nil nil nil nil)
(xml-parse (read-file "example.xml") (+ 1 4 8 16))
⇒ ((DATABASE (@ (name "example.xml"))
  (FRUIT (NAME "apple") (COLOR "red") (PRICE "0.80"))
  (FRUIT (NAME "orange") (COLOR "orange") (PRICE "1.00"))
  (FRUIT (NAME "banana") (COLOR "yellow") (PRICE "0.60"))))
```

Note that using option number 16 causes an @ (at symbol) to be added to attribute lists.

See also the [xml-type-tags](#) function for further information on XML parsing.

xml-type-tags

syntax: (xml-type-tags [*expr-text-tag expr-cdata-tag expr-comment-tag expr-element-tags*])

Can suppress completely or replace the XML type tags "TEXT", "CDATA", "COMMENT", and "ELEMENT" with something else specified in the parameters.

Note that `xml-type-tags` only suppresses or translates the tags themselves but does not suppress or modify the tagged information. The latter would be done using option numbers in [xml-parse](#).

Using `xml-type-tags` without arguments returns the current type tags:

example:

```
(xml-type-tags) ⇒ ("TEXT" "CDATA" "COMMENT" "ELEMENT")
(xml-type-tags nil 'cdata '!-- nil)
```

The first example just shows the currently used type tags. The second example specifies suppression of the "TEXT" and "ELEMENT" tags and shows `cdata` and `!--` instead of "CDATA" and "COMMENT".

zero?

syntax: (zero? *expr*)

Checks the evaluation of *expr* to see if it equals 0 (zero).

example:

```
(set 'value 1.2)
(set 'var 0)
(zero? value) ⇒ nil
(zero? var)   ⇒ true

(map zero? '(0 0.0 3.4 4)) ⇒ (true true nil nil)
```

`zero?` will return `nil` on data types other than numbers.

(0)

newLISP APPENDIX

Error codes

not enough memory	1
environment stack overflow	2
call stack overflow	3
problem accessing file	4
not an expression	5
missing parenthesis	6
string token too long	7
missing argument	8
number or string expected	9
value expected	10
string expected	11
symbol expected	12
context expected	13
symbol or context expected	14
list expected	15
list or symbol expected	16
list or string expected	17
list or number expected	18
array expected	19
array, list or string expected	20
lambda expected	21
lambda-macro expected	22
invalid function	23
invalid lambda expression	24
invalid macro expression	25
invalid let parameter list	26
problem saving file	27
division by zero	28
matrix expected	29
wrong dimensions	30
matrix is singular	31
syntax in regular expression	32
throw without catch	33
problem loading library	34
import function not found	35
symbol is protected	36
error number too high	37
regular expression	38
missing end of text [/text]	39
mismatch in number of arguments	40
problem in format string	41
data type and format don't match	42
invalid parameter: 0.0	43
invalid parameter: NaN	44
illegal parameter type	45
symbol not in MAIN context	46
symbol not in current context	47
array index out of bounds	48
target cannot be MAIN	49
user error	50
user reset -	51

TCP/IP and UDP Error Codes

- 0: No error
- 1: Cannot open socket
- 2: Host name not known
- 3: Not a valid service
- 4: Connection failed
- 5: Accept failed
- 6: Connection closed
- 7: Connection broken
- 8: Socket send() failed
- 9: Socket recv() failed
- 10: Cannot bind socket
- 11: Too much sockets in net-select
- 12: Listen failed
- 13: Badly formed IP
- 14: Select failed
- 15: Peek failed
- 16: Not a valid socket

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could

be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic

equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA. Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance

of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two

goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

(0)

