newLISP®

For Mac OS X, GNU Linux, Unix and Windows

User Manual and Reference v.10.6.2

Contents

User Manual

Function Reference

Appendix

(∂)

newLISP User 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 350 built-in functions. Not much over 200k in size on BSD systems, 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.

Both built-in and user-defined functions, along with variables, share the same global symbol tree and are manipulated by the same functions. Lambda expressions and user-defined functions can be handled like any other list expression.

newLISP is dynamically scoped inside lexically separated contexts (namespaces). Contexts in newLISP are used for multiple purposes. They allow (1) partitioning of programs into modules, (2) the definition of *Classes* in FOOP (Functional Object Oriented Programming), (3) the definition of functions with state and (4) the creation of Hash trees for associative key → value storage.

newLISP's efficient *red-black* tree implementation can handle millions of symbols in namespaces or hashes without degrading performance.

newLISP allocates and reclaims memory automatically, without using traditional asynchronous garbage collection. All objects — except for contexts, built-in primitives, and symbols — are passed by value and are referenced only once. Upon creation objects are scheduled for delayed deletion and Lisp cells are recycled for newly created objects. 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. More than any other Lisp, it implements the *data equals program* paradigm and full self reflection.

Many of newLISP's built-in functions are polymorphic and accept a variety of data types and optional parameters. This greatly reduces the number of functions and syntactic forms necessary to learn and implement. High-level functions are available for string and list

processing, financial math, statistics, and Artificial Intelligence applications.

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

Because strings can contain null characters in newLISP, they can be used to process binary data with most string manipulating functions.

newLISP can also be extended with a shared library interface to import functions that access data in foreign binary data structures. The distribution contains modules for importing popular C-library 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 examples of HTML forms processing. newLISP can be run a as a CGI capable web server using its built-in http mode option.

newLISP has built-in support for distributed processing on networks and parallel, concurrent processing on the same CPU with one or more processing cores.

The source distribution can be compiled for Linux, Mac OS X/Darwin, BSDs, many other Unix flavors and Win32. newLISP can be compiled as a 64-bit LP64 application for full 64-bit memory addressing.

Since version 10.5.7, newLISP also can be compiled to JavaScript and run in a web browser.

newLISP-GS

newLISP-GS comprises a graphical user interface (GUI) and library server. The GUI front-end is written in newLISP, whereas the library server is Java based and uses the standard Java runtime environment installed on all Windows and Mac OS X platforms. Applications built with newLISP-GS can have the host operating system's native look and feel. Interfaces to GTK, Tcl/Tk and OpenGL graphics libraries are also available.

newLISP and Java are available for most operating systems. This makes newLISP-GS a platform-independent solution for writing GUI applications.

For more information on newLISP-GS, see newLISP-GS.

Licensing

newLISP and newLISP-GS are licensed under version 3 of the GPL (General Public License). The newLISP documentation as well as other documentation packaged with newLISP are licensed under the GNU Free Documentation License.

(§)

2. Deprecated functions and future changes

Since version 10.3.0 newLISP can switch between IPv4 and IPv6 modes during run-time using the new net-ipv function. The `-6` commandline option can be used to start newLISP in IPv6 mode. After transition to IPv6 the `-6` commandline switch will be changed to `-4` for starting up in IPv4 mode.

The old writing `parse-date` of date-parse is still recognized but deprecated since version 10.3.0. The old writing will be removed in a future version.

Since version 10.4.2 `if-not` is deprecated and will be removed in a future version.

Since version 10.4.6 newLISP has a built-in function json-parse for translating JSON data into S-expressions. The module file `json.lsp` is removed from the distribution.

Since version 10.4.8 newLISP has built-in support for unlimited precision integers. This makes the GNU GMP module `gmp.lsp` obsolete.

(§)

3. Interactive Lisp mode

The best way to experience Lisp and experiment with it, is using interactive mode in a terminal window or operating system command shell. Since version 10.3, newLISP's read-eval-print-loop (REPL) accepts multi-line statements.

To enter a multi-line statement hit the [enter] key on an empty line after the system prompt. To exit multi-line mode, hit the [enter] key again on an empty line. In the following example computer output is shown in bold letters:

```
>
(define (foo x y)
    (+ x y))

(lambda (x y) (+ x y))
> (foo 3 4)
7
>
```

Note, that multi-line mode is only possible in an OS command terminal window or command shell. The monitor window in the Java based newLISP-GS IDE will not accept multi-line statements.

Interactive Lisp mode can accept operating system shell commands. To hit an OS command enter the '`!`' character right after the prompt, immediately followed by the shell command:

```
> !ls *.html
CodePatterns.html              MemoryManagement.html   newLISPdoc.html
```

```
ExpressionEvaluation.html       manual_frame.html              newlisp_index.html
License.html                    newLISP-10.3-Release.html      newlisp_manual.html
>
```

In the example a `ls` shell command is entered to show HTML files in the current directory. On MS Windows a `dir` command could be used in the same fashion.

The mode can also be used to call an editor or any other program:

```
> !vi foo.lsp
```

The Vi editor will open to edit the program "foo.lsp". After leaving the editor the program could be run using a load statement:

```
> (load "foo.lsp")
```

The program `foo.lsp` is now run. This mode using '`!`' can also be used from the newLISP-GS IDE.

When using a Unix terminal or command shell, tab-expansion for built-in newLISP functions can be used:

```
> (pri
print       println     primitive?
> (pri
```

After entering the characters `(pri` hit the [tab] key once to show all the built-in functions starting with the same characters. When hitting [tab] twice before a function name has started, all built-in function names will be displayed.

On most Unix, parenthesis matching can be enabled on the commandline by including the following line in the file `.inputrc` in the home directory:

```
set blink-matching-paren on
```

Not all systems have a version of `libreadline` advanced enough for this to work.

(§)

4. Command-line options, startup and directories

Command line help summary

When starting newLISP from the command-line several switches and options and source files can be specified. Executing:

```
newlisp -h
```

in a command shell will produce the following summary of options and switches:

```
 -h this help
 -n no init.lsp (must be first)
 -x <source> <target> link
```

```
 -v version
 -s <stacksize>
 -m <max-mem-MB> cell memory
 -e <quoted lisp expression>
 -l <path-file> log connections
 -L <path-file> log all
 -w <working dir>
 -c no prompts, HTTP
 -C force prompts
 -t <usec-server-timeout>
 -p <port-no>
 -d <port-no> demon mode
 -http only
 -6 IPv6 mode
```

Before or after the command-line switches, files to load and execute can be specified. If a newLISP executable program is followed by parameters, the program must finish with and `(exit)` statement, else newLISP will take command-line parameters as additional newLISP scripts to be loaded and executed.

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.

Specifying files as URLs

newLISP will load and execute files specified on the command-line. Files are specified with either their pathname or a `file://` URL on the local file system or with a `http://` URL on remote file systems running an HTTP server. That HTTP server can be newLISP running in HTTP server mode.

```
newlisp aprog.lsp bprog.lsp prog.lsp
newlisp http://newlisp.org/example.lsp
newlisp file:///usr/home/newlisp/demo.lsp
```

No loading of init.lsp

This option suppresses loading of any present initialization file `init.lsp` or `.init.lsp`. In order to work, this must be the first option specified:

```
newlisp -n
```

More about initialization files.

Stack size

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

```
newlisp -s 6000 myprog
newlisp -s 6000 http://asite.com/example.lsp
```

The above examples show starting newLISP with different stack sizes using the `-s` option, as well as loading one or more newLISP source files and loading files specified by an URL. When no stack size is specified, the stack defaults to 2048. Per stack position about 80 bytes of memory are preallocated.

Maximum memory usage

```
newlisp -m 128
```

This example limits newLISP cell memory to 128 megabytes. In 32-bit newLISP, each Lisp cell consumes 16 bytes, so the argument `128` would represent a maximum of 8,388,608 newLISP 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 dynamic memory usage.

Direct execution mode

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

```
newlisp -e "(+ 3 4)"  → 7 ; On Win32 and Unix

newlisp -e '(append "abc" "def")'  → "abcdef" ; On Unix
```

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 string delimiters. Note that there is a space between `-e` and the quoted command string.

Logging I/O

In any mode, newLISP can write a log when started with the `-l` or `-L` option. Depending on the mode newLISP is running, different output is written to the log file. Both options always must specify the path of a log-file. The path may be a relative path and can be either attached or detached to the `-l` or `-L` option. If the file does not exist, it is created when the first logging output is written.

```
newlisp -l./logfile.txt -c

newlisp -L /usr/home/www/log.txt -http -w /usr/home/www/htpdocs
```

The following table shows the items logged in different situations:

logging mode	**command-line and net-eval with `-c`**	**HTTP server with `-http`**
`newlisp -l`	log only input and network connections	log only network connections

`newlisp -L`	log also newLISP output (w/o prompts)	log also HTTP requests

All logging output is written to the file specified after the `-l` or `-L` option.

Specifying the working directory

The `-w` option specifies the initial working directory for newLISP after startup:

```
newlisp -w /usr/home/newlisp
```

All file requests without a directory path will now be directed to the path specified with the `-w` option.

Suppressing the prompt and HTTP processing

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

```
newlisp -c
```

Listen and connection messages 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 to answer `HTTP GET`, `PUT`, `POST` and `DELETE` requests, as well as perform CGI processing. Using the `-c` option, together with the `-w` and `-d` options, newLISP can serve as a standalone `httpd` webserver:

```
newlisp -c -d 8080 -w /usr/home/www
```

When running newLISP as a `inetd` or `xinetd` enabled server on Unix machines, use:

```
newlisp -c -w /usr/home/www
```

In `-c` mode, newLISP processes command-line requests as well as HTTP and net-eval requests. Running newLISP in this mode is only recommended on a machine behind a firewall. This mode should not be run on machines open and accessible through the Internet. To suppress the processing of net-eval and command-line–like requests, use the safer `-http` option.

Forcing prompts in pipe I/O mode

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

```
newlisp -C
```

To suppress console output from 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 mainly used to control newLISP from another application, such as a newLISP GUI front-end or a program written in another language. As soon as the controlling client closes the connection, newLISP will exit.

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. On Windows, start the server process without the `&` in the foreground and open a second command window for the telnet application. Now connect with a telnet:

```
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`.

After each transaction, when a connection closes, newLISP will go through a reset process, reinitialize stack and signals and go to the `MAIN` context. Only the contents of program and variable symbols will be preserved when running a stateful server.

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.

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 will also answer `HTTP GET`, `PUT` and `DELETE` requests. This can be used to retrieve and store files with get-url, put-url, delete-url, read-file, write-file and append-file, or to load and save programs using load and save from and to remote server nodes. See the chapters for the `-c` and `-http` options for more details.

HTTP-only server mode

newLISP can be limited to HTTP processing using the `-http` option. With this mode, a secure `httpd` web server daemon can be configured:

```
newlisp -http -d 8080 -w /usr/home/www
```

When running newLISP as an `inetd` or `xinetd`-enabled server on Unix machines, use:

```
newlisp -http -w /usr/home/www
```

To further enhance security and HTTP processing, load a program during startup when using this mode:

```
newlisp httpd-conf.lsp -http -w /usr/home/www
```

The file `httpd-conf.lsp` contains a command-event function configuring a user-defined function to analyze, filter and translate requests. See the reference for this function for a working example.

In the HTTP modes enabled by either `-c` or `-http`, the following file types are recognized, and a correctly formatted `Content-Type:` header is sent back:

file extension	media type
.avi	video/x-msvideo
.css	text/css
.gif	image/gif
.htm	text/htm
.html	text/html
.jpg	image/jpg
.js	application/javascript
.mov	video/quicktime
.mp3	audio/mpeg
.mpg	video/mpeg
.pdf	application/pdf
.png	image/png
.wav	audio/x-wav
.zip	application/zip

any other text/plain

To serve CGI, HTTP server mode needs a `/tmp` directory on Unix-like platforms or a `C:\tmp` directory on Win32. newLISP can process GET, PUT, POST and DELETE requests and create custom response headers. CGI files must have the extension `.cgi` and have executable permission on Unix. More information about CGI processing for newLISP server modes can be found in the document Code Patterns in newLISP.

In both server modes `-c` and `-http` the environment variables DOCUMENT_ROOT, HTTP_HOST, REMOTE_ADDR, REQUEST_METHOD, SERVER_SOFTWARE and QUERY_STRING are set. The variables CONTENT_TYPE, CONTENT_LENGTH, HTTP_HOST, HTTP_USER_AGENT and HTTP_COOKIE are also set, if present in the HTTP header sent by the client.

Local domain Unix socket server

Instead of a port, a local domain Unix socket path can be specified in the `-d` or `-p` server modes.

```
newlisp -c -d /tmp/mysocket &
```

Test the server using another newLISP process:

```
newlisp -e '(net-eval "/tmp/mysocket" 0 "(symbols)")'
```

A list of all built-in symbols will be printed to the terminal

This mode will work together with local domain socket modes of net-connect, net-listen, and net-eval. Local domain sockets opened with `net-connect` and `net-listen` can be served using net-accept, net-receive, and net-send. Local domain socket connections can be monitored using net-peek and net-select.

Local domain socket connections are much faster than normal TCP/IP network connections and preferred for communications between processes on the same local file system in distributed applications. This mode is not available on Win32.

Connection timeout

Specifies a connection timeout when running in `-p` or `-d` demon mode. A newLISP Server will disconnect when no further input is read after accepting a client connection. The timeout is specified in micro seconds:

```
newlisp -c -t 3000000 -d 4711 &
```

The example specifies a timeout of three seconds.

`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
```

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 and file permissions of the www document directory adjusted accordingly. 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 answer `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.

Linking a source file with newLISP for a new executable

Source code and the newLISP executable can be linked together to build a self-contained application by using the `-x` command line flag.

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

The program `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 steps:

```
# on OSX, Linux and other UNIX

newlisp -x uppercase.lsp uppercase

chmod 755 uppercase # give executable permission

# on Windows the target needs .exe extension

newlisp -x uppercase.lsp uppercase.exe
```

newLISP will find a newLISP executable in the execution path of the environment and link a copy of the source code.

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

The console should print:

```
CONVERT ME TO UPPERCASE
```

Note that neither one of the initialization files `init.lsp` nor `.init.lsp` is loaded during startup of linked programs.

(§)

5. Startup, directories, environment

Environment variable NEWLISPDIR

During startup, newLISP sets the environment variable NEWLISPDIR, if it is not set already. On Linux, BSDs, Mac OS X and other Unixes the variable is set to /usr/share/newlisp. On Win32 the variable is set to %PROGRAMFILES%/newlisp.

The environment variable NEWLISPDIR is useful when loading files installed with newLISP:

```
(load (append (env "NEWLISPDIR") "/guiserver.lsp"))

(load (append (env "NEWLISPDIR") "/modules/mysql.lsp"))
```

A predefined function module can be used to shorten the second statement loading from the modules/ directory:

```
(module "mysql.lsp")
```

The initialization file init.lsp

Before loading any files specified on the command-line, and before the banner and prompt are shown. newLISP tries to load a file .init.lsp from the home directory of the user starting newLISP. On Mac OS X, Linux and other Unix the home directory is found in the HOME environment variable. On Win32 the directory name is contained in the USERPROFILE or DOCUMENT_ROOT environment variable.

If a .init.lsp cannot be found in the home directory newLISP tries to load the file init.lsp from the directory found in the environment variable NEWLISPDIR.

When newLISP is run as a shared library, an initialization file is looked for in the environment variable NEWLISPLIB_INIT. The full path-name of the initialization file must be specified. If NEWLISPLIB_INIT is not defined, no initialization file will be loaded by the library module.

Although newLISP does not require init.lsp to run, it is convenient for defining functions and system-wide variables.

Note that neither one of the initialization files init.lsp nor .init.lsp is loaded during startup of linked programs.

Directories on Linux, BSD, Mac OS X and other Unix

The directory /usr/share/newlisp/modules contains modules with useful functions POP3 mail, etc. The directory /usr/share/newlisp/guiserver contains sample programs for writing GUI applications with newLISP-GS. The directory /usr/share/doc/newlisp/ contains documentation in HTML format.

Directories on Win32

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. The subdirectories `%PROGRAMFILES%\newlisp\modules` and `%PROGRAMFILES%\newlisp\guiserver` contain modules for interfacing to external libraries and sample programs written for newLISP-GS.

(§)

6. Extending newLISP with shared libraries

Many shared libraries on Unix and Win32 systems can be used to extend newLISP's functionality. Examples are libraries for writing graphical user interfaces, libraries for encryption or decryption and libraries for accessing databases.

The function import is used to import functions from external libraries. The function callback is used to register callback functions in external libraries. Other functions like pack, unpack, get-string, get-int and get-long exist to facilitate formatting input and output to and from imported library functions.

See also the chapter 23. Extending newLISP in the Code Patterns in newLISP document.

(§)

7. newLISP as a shared library

newLISP as C library

newLISP can be compiled as a shared C library. On Linux, BSDs and other Unix flavors the library is called `newlisp.so`. On Windows it is called `newlisp.dll` and `newlisp.dylib` on Mac OS X. A newLISP shared library is used like any other shared library.

The main function to import is `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 retrieved using get-string. The returned string is formatted like output from a command-line session. It contains terminating line-feed characters, but not the prompt string.

When calling `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.

When passing multi-line source to `newlispEvalStr`, that source should be bracketed by `[cmd]`, `[/cmd]` tags, each on a different line:

```
(set 'code [text][cmd]
```

```
...
...
...
[/cmd][/text])
```

Since v.10.3.3 callbacks can also be registered using `newlispCallback`. For more information read the chapter 24. newLISP compiled as a shared library in the Code Patterns in newLISP document.

newLISP as a JavaScript library

Since version 10.5.7, newLISP can be compile to JavaScript using the Emscripten toolset. The library can be used to run newLISP clientr-side in a web browser, just like JavaScript or HTML. An HTML page can host both, newLISP code and JavaScript code together. Both languages can call each other. For more information see the `newlisp-js-x.x.x.zip` distribution package which contains the library `newlisp-js-lib.js`, documentaion and example applications. A small newLISP development environment hosted in a browser can also be accessed here: newlisp-js The application contains links to another example application, documentation and a download link for the whole package.

newLISP compiled as a JavaScript library adds new functions linked from API for newLISP in a web browser.

(§)

8. Evaluating newLISP expressions

The following is a short introduction to newLISP 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.

Interactive multiline expressions

Multiline expressions can be entered by entering an empty line first. Once in multiline mode, another empty line returns from entry mode and evaluates the statement(s) entered (ouput in boldface):

```
>
(define (foo x y)
    (+ x y))

(lambda (x y) (+ x y))
> (foo 3 4)
7
> _
```

Entering multiline mode by hitting the enter key on an empty line suppresses the prompt.

Entering another empty line will leave the multiline mode and evaluate expressions.

As an alternativo to entering empty lines, the [cmd] and [/cmd] tags are used, each entered on separate lines. This mode is used by some interactive IDEs controlling newLISP and internally by the net-eval function. The [cmd] and [/cmd] tags must also be used in the console part of the newLISP-GS Java IDE.

Integer, 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.

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, whereas 64 bit double floating point numbers offer only 52 bits of resolution in the integer part of the number.

Big integer, multiple precision arithmetic

The following operators, functions and predicates work on big integers:

function	description
+ - * / ++ -- %	arithmetic operators
< > = <= >= !=	logical operators
abs	returns the absolute value of a number
gcd	calculates the greatest common divisor of a group of integers
even?	checks the parity of an integer number
odd?	checks the parity of an integer number
number?	checks if an expression is a float or an integer
zero?	checks if an expression is 0 or 0.0

If the first argument in any of these operators and functions is a big integer, the calculation performed will be in big integer mode. In the Function Reference section of this manual these are marked with a bigint suffix.

Literal integer values greater than 9223372036854775807 or smaller than -9223372036854775808, or integers with an appended letter L, will be converted and processed in big integer mode. The function bigint can be used to convert from integer, float or string format to big integer. The predicate bigint? checks for big integer type.

```
; first argument triggers big integer mode because it's big enough

(+ 12345678901234567890123456789O 12345) → 12345678901234567890123458023 5L

; first small literal put in big integer format by
; appending L to guarantee big integer mode

(+ 12345L 12345678901234567890123456789O) → 12345678901234567890123458023 5L

(setq x 12345678901234567890123 45)
(* x x) → 152415787532388367504953347995733866912056239902 5L

; conversion from bigint to float introduces rounding errors

(bigint (float (* x x))) → 152415787532388372534400000000000000000000000000 0L

; sequence itself does not take big integers, before using
; apply, the sequence is converted with bigint

(apply * (map bigint (sequence 1 100))) ; calculate 100!
→ 93326215443944152681699238856266700490715968264381
  62146859296389521759999322991560894146397615651828
  62536979208272237582511852109168640000000000000000
```

```
  00000000L

; only the first operand needs to be bigint for apply
; to work. The following gives the same result

(apply * (cons 1L (sequence 2 100)))

; length on big integers returns the number of decimal digits
(length (apply * (map bigint (sequence 1 100))))
→ 158 ; decimal digits

; all fibonacci numbers up to 200, only the first number
; needs to be formatted as big integer, the rest follows
; automatically - when executed from the command line in
; a 120 char wide terminal, this shows a beautiful pattern

(let (x 1L) (series x (fn (y) (+ x (swap y x))) 200))
```

When doing mixed integer / big integer arithmetic, the first argument should be a big integer to avoid erratic behaviour.

```
; because the first argument is 64-bit, no big integer arithmetic
; will be done, although the second argument is big integer

(+ 123 12345L)
→ 12468

; the second argument is recognized as a big integer
; and overflows the capacity of a 64-bit integer

(+ 123 12345345673564563456546356 3546)
→ ERR: number overflows in function +

; now the first argument converts to big integer and the
; whole expression evaluates in big integer mode

(+ 123L 12345345673564563456546356 3546)
→ 12345345673564563456546356 3669L
```

Under most circumstances mixing float, integers and big integers is transparent. Functions automatically do conversions when needed on the second argument. The overflow behavior when using normal integers and floats only, has not changed from newLISP versions previous to 10.5.0.

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, **big integers** and **floating point** numbers evaluate to themselves:

```
123      → 123    ; decimal integer
0xE8     → 232    ; hexadecimal prefixed by 0x
055      → 45     ; octal prefixed by 0 (zero)
0b101010 → 42     ; binary prefixed by 0b
1.23     → 1.23   ; float
123e-3   → 0.123  ; float in scientific notation

12345678901234567890123456789O
→ 12345678901234567890123456789OL ; parses to big integer
```

Integers are 64-bit including the sign bit. 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.

Big integers are of unlimited precision and only limited in size by memory. The memory requirement of a big integer is:

bytes = 4 * ceil(digits / 9) + 4.

Where *digits* are decimal digits, *bytes* are 8 bits and *ceil* is the ceiling function rounding up to the next integer.

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 2047 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):

character	description
\"	for a double quote inside a quoted string
\n	for a line-feed character (ASCII 10)
\r	for a return character (ASCII 13)
\b	for a backspace BS character (ASCII 8)
\t	for a TAB character (ASCII 9)
\f	for a formfeed FF character (ASCII 12)
\nnn	for a three-digit ASCII number (nnn format between 000 and 255)
\xnn	for a two-digit-hex ASCII number (xnn format between x00 and xff)
\unnnn	for a unicode character encoded in the four nnnn hexadecimal digits. newLISP will translate this to a UTF8 character in the UTF8 enabled versions of newLISP.
\\	for the backslash character (ASCII 92) itself

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-macro (a b) (set (eval a) b))
(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).

Lists are evaluated by first evaluating the first list element 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.

(§)

9. 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 because 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)))  → (lambda (x) (+ x x))
(double 123)                        → 246
```

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:

```
(setf (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.

(§)

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

In newLISP, `nil` and `true` represent both the symbols and the Boolean values *false* and *true*. 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 flow control 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 Lisps. Only in conditional expressions are they treated as a Boolean false, as in `and`, `or`, `if`, `while`, `unless`, `until`, and `cond`.

Evaluation of `(cons 'x '())` yields `(x)`, but `(cons 'x nil)` yields `(x nil)` because `nil` is treated as a Boolean value when evaluated, not 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 newLISP 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*.

(§)

11. 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. Most 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
apply	apply a function or operator to a list of arguments.
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
corr	calculates the *product-moment correlation* coefficient
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
map	applies a function to vector(s) of arguments and returns results in a list.
mat	perform scalar operations on matrices
multiply	multiplies two matrices
nth	returns an element of and array
rest	returns all but the first row of an array
reverse	reverses the elements or rows in an array
setf	sets contents of an array reference
slice	returns a slice of an array
sort	sort the elements in an array
stats	calculates some basic statistics for a data vector
t-test	compares means of data samples using the *Student's t* statistic
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, 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 or list.

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.

(§)

12. 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` or less then -N causes an out-of-bounds error in lists and arrays.

Implicit indexing for `nth`

Implicit indexing can be used instead of nth to retrieve the elements of a list or array or the characters of a string:

```
(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

; an empty index vector yields the original list or array

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

Note that implicit indexing is not breaking newLISP syntax rules but is merely an expansion of existing rules to other data types in the functor position of an s-expression. In original Lisp, the first element in an s-expression list is applied as a function to the rest elements as arguments. In newLISP, a list in the functor position of an s-expression assumes self-indexing functionality using the index arguments following it.

Implicit indexing is faster than the explicit forms, but the explicit forms may be more readable depending on context.

Note that in the UTF-8–enabled version of newLISP, implicit indexing of strings or using the nth function work on character rather than single-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 Variables holding contexts.

The indexed default functor can also be used with setf as shown in the following example:

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

(setf (MyList 3) 999)   → 999
(MyList 3)              → 999

MyList:MyList           → (a b c 999 e f g)
```

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. If the length is negative it counts from the end of the list or string:

```
(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)
(2 -2 lst)   → (c d e)

(set 'str "abcdefg")

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

The functions rest, first and last work on multi-byte character boundaries in UTF-8 enabled versions of newLISP. But the implicit indexing forms for slicing and resting will always work on single-byte boundaries and can be used for binary content. Offset and length results from the regular expression functions find and regex are also in single-byte counts and can be further processed with slice or it's implicit form.

Modify references in lists, arrays and strings

Parts in lists, arrays and strings referenced by indices can be modified using setf:

```
; lists

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

(lst 1) → b

(setf (lst 1) 'z) → z

lst → (a z c d (e f g))

(setf (lst -1) '(E F G)) → (E F G)

lst → (a z c d (E F G))

; arrays

(set 'myarray (array 2 3 (sequence 1 6))) → ((1 2 3) (4 5 6))

(setf (myarray 1 2) 66) → 66

myarray → ((1 2 3) (4 5 66))

; strings

(set 's "NewLISP")

(setf (s 0) "n") → "n"

s → "newLISP"
```

Note that only full elements or nested lists or arrays can be changed this way. Slices or rest parts of lists or arrays as used in implicit resting or slicing cannot be substituted at once using setf, but would have to be substituted element by element. In strings only one character can be replaced at a time, but that character can be replaced by a multi-character string.

(§)

13. 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 variable, list, array, or string:

function	description
++	increments numbers in integer mode
--	decrements numbers in integer mode
bind	binds variable associations in a list
constant	sets the contents of a variable and protects it
extend	extends a list or string
dec	decrements a number referenced by a variable, list or array
define	sets the contents of a variable
define-macro	sets the contents of a variable
inc	increments a number referenced by a variable, list or array
let	declares and initializes local variables
letn	initializes local variables incrementally, like nested lets
letex	expands local variables into an expression, then evaluates
net-receive	reads into a buffer variable
pop	pops an element from a list or string
pop-assoc	removes an association from an association list
push	pushes a new element onto a list or string
read	reads into a buffer variable
receive	receives a message from a parent or child process
replace	replaces elements in a list or string
reverse	reverses a list or string
rotate	rotates the elements of a list or characters of a string
set	sets the contents of a variable
setf setq	sets the contents of a variable, list, array or string
set-ref	searches for an element in a nested list and replaces it
set-ref-all	searches for an element in a nested list and replaces all instances
sort	sorts the elements of a list or array
swap	swaps two elements inside a list or string

Make a destructive function non-destructive

Some destructive functions can be made non-destructive by wrapping the target object into the copy function.

```
(set 'aList '(a b c d e f))

(replace 'c (copy aList)) → (a b d e f)

aList → (a b c d e f)
```

The list in `aList` is left unchanged.

(§)

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 error-event function can be used to catch errors caused by faulty code or user-initiated exceptions.

The throw-error function may be used to throw user-defined errors.

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. Dynamic and lexical scoping

newLISP uses dynamic scoping *inside* contexts. A context is a lexically closed namespace. In this way, parts of a newLISP program can live in different namespaces taking advantage of *lexical scoping*.

When the parameter symbols of a lambda expression are bound to its arguments, the old bindings are pushed onto 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:

```
> (set 'x 1)
1
> (define (f) x)
(lambda () x)
> (f)
1
> (define (g x) (f))
(lambda (x) (f))
> (g 0)
0
> (f)
1
> _
```

The variable `x` is first set to `1`. But when `(g 0)` is called, `x` is bound to `0` and `x` is reported by `(f)` as `0` during execution of `(g 0)`. After execution of `(g 0)`, the call to `(f)` will report `x` as `1` again.

This is different from the *lexical scoping* mechanisms found in languages like C or Java, where the binding of local parameters occurs inside the function only. In lexically scoped languages like C, `(f)` would always print the global bindings of the symbol `x` with `1`.

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

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

- The function args can be used when passing symbols.
- One or more user-defined functions can be placed in their own namespace called 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 default functors for more information.

(§)

16. Contexts

In newLISP, symbols can be separated into namespaces called *contexts*. Each context has a private symbol table 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. Although 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)
```

Symbol creation in contexts

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

1. newLISP first parses and translates each top level expression. The symbols are created during this phase. After the expression is translated, it gets evaluated.
2. A symbol is created when newLISP first *sees* it, while calling the load, sym, or eval-string functions. When newLISP reads a source file, symbols are created *before* evaluation occurs.
3. 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.
4. Once a symbol is created and assigned to a specific context, it will belong to that context permanently.
5. When a user-defined function is evaluated, the context is switched to the name-space which owns that symbol.

6. A context switch only influences symbol creation during load, sym, or eval-string. load by default loads into MAIN except when context switches occur on the top level of the file loaded. The context should always be specified when the functions sym and eval-string are used. When this rule is followed, a context switch should only occur on the top level of a program, never inside a function.

Creating contexts

Contexts can be created either by using the context function or via implicit creation. The first method is used when writing larger portions of code belonging the same context:

```
(context 'FOO)

(set 'var 123)

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

(context MAIN)
```

If the context does not exist yet, the context symbol must be quoted. If the symbol is not quoted, newLISP assumes the symbol is a variable holding the symbol of the context to create. Because a context evaluates to itself, existing contexts like MAIN do not require quoting.

When newLISP reads the above code, it will read, then evaluate the first statement: `(context 'FOO)`. This causes newLISP to switch the namespace to FOO and the following symbols `var`, `x`, `y` and `z` will all be created in the FOO context when reading and evaluating the remaining expressions.

To refer to `var` or `func` from anywhere else outside the FOO namespace they need to be prefixed with the context name:

```
FOO:var → 123

(FOO:func p q r)
```

The symbols function is used to show all symbols belonging to a context:

```
(symbols FOO) → (FOO:func FOO:var FOO:x FOO:y FOO:z)
```

Implicitly creating contexts

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"
> _
```

Note that only the symbols prefixed with their context name will be part of the context:

```
(define (ACTX:foo x y)
    (+ x y))
```

When above code is loaded in MAIN only `foo` will be part of `ACTX`. The symbols `x` and `y` will still be part of `MAIN`.

Loading module files

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` (function `foo`'s context).

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

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

Global scope

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)
ERR: 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 useful when writing functions, which need to switch contexts or use contexts which do not exist yet:

```
(define (update ctx val)
    (set 'ctx:sum val)
    (ctx:func 999)
)

(context 'FOO)
(define (func x)
    (println "=>" x))
(context MAIN)
```

The following shows a terminal session using above definitions. The program output is shown in bold-face:

```
> (update FOO 123)
=> 999

> FOO:sum
123
>
```

The same one function `update` can display different behavior depending on the context passed as first parameter.

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")
ERR: 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.

Contexts as programming modules

Contexts in newLISP are mainly used for partitioning source into modules. Because each module lives in a different namespace, modules are lexically separated and the names of symbols cannot clash with identical names in other modules.

The modules, which are part of the newLISP distribution, are a good example of how to put related functions into a module file, and how to document modules using the newLISPdoc utility.

For best programming practice, a file should only contain one module and the filename should be similar if not identical to the context name used:

```
;; file db.lsp, commonly used database functions
```

```
(context 'db)

;; Variables used throughout this namespace

(define db:handle)
(define db:host "http://localhost")

;; Constants

(constant 'Max_N 1000000)
(constant 'Path "/usr/data/")

;; Functions

(define (db:open ... )
    ... )

(define (db:close ... )
    ... )

(define (db:update ... )
    ... )
```

The example shows a good practice of predefining variables, which are global inside the namespace, and defining as constants the variables that will not change.

If a file contains more than one context, then the end of the context should be marked with a switch back to `MAIN`:

```
;; Multi context file multi.lsp

(context 'A-ctx)
...
(context MAIN)

(context 'B-ctx)
...
(context MAIN)

(context 'C-ctx)
...
(context MAIN)
```

Contexts as data containers

Contexts are frequently uses as data containers, e.g. for hash-like dictionaries and configuration data:

```
;; Config.lsp - configuration setup

(context 'Config)

(set 'user-name "admin")
(set 'password "secret")
(set 'db-name "/usr/data/db.lsp")
...

;; eof
```

Loading the `Config` namespace will now load a whole variable set into memory at once:

```
(load "Config.lsp")

(set 'file (open Config:db-name "read"))
...
...
```

In a similar fashion a whole data set can be saved:

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

Read more about this in the section Serializing contexts.

Loading and declaring contexts

Module files are loaded using the load function. If a programming project contains numerous modules that refer to each other, they should be pre-declared to avoid problems due to context forward references that can occur before the loading of that context.

```
;; pre-declaring contexts, finish with Main to return
(map context '(Utilities Config Acquisition Analysis SysLog MAIN))

;; loading context module files
(load "Utilities.lsp" "Acquisition.lsp")
(load "http://192.168.1.34/Config.lsp") ; load module from remote location
(load "Analysis.lsp" "SysLog.lsp")

(define (run)
    ... )

(run)

;; end of file
```

When pre-declaring and loading modules as shown in the example, the sequence of declaration or loading can be neglected. All forward references to variables and definitions in modules not loaded yet will be translated correctly.

Modules not starting with a context switch are always loaded into `MAIN` except when the load statement specifies a target context as the last parameter. The load function can take `URL`s to load modules from remote locations, via `HTTP`.

The current context after the load statement will always be the same as before the load.

Serializing contexts

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, anything referenced by a symbol 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. The context default functor

A *default functor* or *default function* is a symbol or user-defined function or macro with the same name as its namespace. When the context is used as the name of a function or in the functor position of an s-expression, newLISP executes the default function.

```
;; the default function

(define (Foo:Foo a b c) (+ a b c))

(Foo 1 2 3)  → 6
```

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 a 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.

Like a lambda or lambda-macro function, default functions can be used with map or apply.

Functions with memory

A default function can update the lexically isolated static variables contained inside its namespace:

```
;; a function with memory

(define (Gen:Gen x)
    (if Gen:acc
        (inc Gen:acc x)
        (setq 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.

The definition of `Gen:Gen` shows, how a function is put in its own namespace without using the surrounding `(context 'Gen)` and `(context MAIN)` statements. In that case only symbols qualified by the namespace prefix will end up in the `Gen` context. In the above example the variable `x` is still part of `MAIN`.

Hash functions and dictionaries

There are several functions that can be used to place symbols into namespace contexts. When using dictionaries as simple hash-like collections of variable → value pairs, use the uninitialized default functor:

```
(define Myhash:Myhash) ; create namespace and default functor

; or as a safer alternative

(new Tree 'MyHash) ; create from built-in template
```

Either method can be used to make the `MyHash` dictionary space and default functor. The second method is safer, as it will protect the default functor `MyHash:MyHash` from change. The *default functor* in a namespace must contain `nil` to be used as a dictionary. Creating key-value pairs and retrieving a value is easy:

```
(Myhash "var" 123) ; create and set variable/value pair

(Myhash "var")  → 123 ; retrieve value

; keys can be integers and will be converted to strings internally

(Myhash 456 "hello")

(Myhash 456)    → "hello"
```

Symbol variables created this way can contain spaces or other characters normally not allowed in newLISP symbol names:

```
(define Foo:Foo)

(Foo "John Doe" 123)         → 123
(Foo "#1234" "hello world")  → "hello world"
(Foo "var" '(a b c d))       → (a b c d)

(Foo "John Doe")  → 123
(Foo "#1234")     → "hello world"
(Foo "var")     → (a b c d)
```

An entry which doesn't exist will return `nil`:

```
(Foo "bar")    → nil
```

Setting an entry to `nil` will effectively delete it from the namespace.

An association list can be generated from the contents of the namespace:

```
(Foo) → (("#1234" "hello world") ("John Doe" 123) ("var" (a b c d)))
```

Entries in the dictionary can also be created from a list:

```
(Foo '(("#1234" "hello world") ("John Doe" 123) ("var" (a b c d))) → Foo
```

The list can also be used to iterate through the sorted key -> value pairs:

```
(dolist (item (Foo)) (println (item 0) " -> " (item 1)))

#1234 -> hello world
John Doe -> 123
var -> (a b c d)
```

Like many built-in functions hash expressions return a reference to their content which can be modified directly:

```
(pop (Foo "var")) → a

(Foo "var") → (b c d)

(push 'z (Foo "var")) → (z b c d)

(Foo "var") → (z b c d)
```

When setting hash values the anaphoric system variable `$it` can be used to refer to the old value when setting the new:

```
(Foo "bar" "hello world")

(Foo "bar" (upper-case $it))

(Foo "bar") → "HELLO WORLD"
```

Hash values also can be modified using setf:

```
(Foo "bar" 123)        → 123

(setf (Foo "bar") 456) → 456

(Foo "bar")            → 456
```

But supplying the value as a second parameter to the hash functions is shorter to write and faster.

Dictionaries can easily be saved to a file and reloaded later:

```
; save dictionary
(save "Foo.lsp" 'Foo)

; load dictionary
(load "Foo.lsp")
```

Internally the key strings are created and stored as symbols in the hash context. All key strings are prepended with an _ underscore character. This protects against overwriting the default symbol and symbols like `set` and `sym`, which are needed when loading a hash namespace from disk or over `HTTP`. Note the following difference:

```
(Foo) → (("#1234" "hello world") ("John Doe" 123) ("var" (a b c d)))

(symbols Foo) → (Foo:Foo Foo:_#1234 Foo:_John Doe Foo:_var)
```

In the first line hash symbols are shown as strings without the preceding underscore characters. The second line shows the internal form of the symbols with prepended underscore characters.

For a more detailed introduction to *namespaces*, see the chapter on Contexts.

Passing data by reference

A default functor can also be used to hold data. If this data contains a list or string, the context name can be used as a reference to the data:

```
;; the default functor for holding data

(define Mylist:Mylist '(a b c d e f g))

(Mylist 3) → d

(setf (Mylist 3) 'D) → D

Mylist:Mylist → (a b c D e f g)

;; access list or string data from a default functor

(first Mylist) → a

(reverse Mylist) → (g f e D c b a)

(set 'Str:Str "acdefghijklmnop")

(upper-case Str) → "ACDEFGHIJKLMNOP"
```

Most of the time, newLISP passes parameters by *value copy*. This poses a potential problem when passing large lists or strings to user-defined functions or macros. Strings and lists, which are packed in a namespace using default functors, are passed automatically by reference:

```
;; use a default functor to hold a list

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

(define (change-db obj idx value)
    (setf (obj idx) value))

; pass by context reference
(change-db Mydb 1234 "abcdefg")

(Mydb 1234)  → "abcdefg"
```

Any argument of a built-in function calling for either a list or a string — but no other data type — can receive data passed by reference. Any user-defined function can take either normal variables, or can take a context name for passing a reference to the default functor containing a list or string.

Note that on lists with less than about 100 elements or strings of less than about 50000 characters, the speed difference between reference and value passing is negligible. But on bigger data objects, differences in both speed and memory usage between reference and value passing can be significant.

Built-in and user-defined functions are suitable for both types of arguments, but when passing context names, data will be passed by reference.

Quoted symbols can also be used to pass data by reference, but this method has disadvantages:

```
(define (change-list aList) (push 999 (eval aList)))

(set 'data '(1 2 3 4 5))

; note the quote ' in front of data
(change-list 'data)  → (999 1 2 3 4 5)

data  →  (999 1 2 3 4 5)
```

Although this method is simple to understand and use, it poses the potential problem of *variable capture* when passing the same symbol as used as a function parameter:

```
;; pass data by symbol reference

> (set 'aList '(a b c d))
(a b c d)
> (change-list 'aList)

ERR: list or string expected : (eval aList)
called from user defined function change-list
>
```

At the beginning of the chapter it was shown how to package data in a name-space using a default functor. Not only the default functor but any symbol in context can be used to hold data. The disadvantage is that the calling function must have knowledge about the symbol being used:

```
;; pass data by context reference

(set 'Mydb:data (sequence 1 100000))
```

```
(define (change-db obj idx value)
    (setf (obj:data idx) value))

(change-db Mydb 1234 "abcdefg")

(nth 1234 Mydb:data)   → "abcdefg"
; or
(Mydb:data 1234)   → "abcdefg"
```

The function receives the namespace in the variable `obj`, but it must have the knowledge that the list to access is contained in the `data` symbol of that namespace (context).

(§)

18. Functional object-oriented programming

Functional-object oriented programming (FOOP) is based on the following five principles:

- Class attributes and methods are stored in the namespace of the object class.
- The namespace default functor holds the object constructor method.
- An object is constructed using a list, the first element of which is the context symbol describing the class of the object.
- Polymorphism is implemented using the : (colon) operator, which selects the appropriate class from the object.
- A target object inside a class-method function is accessed via the self function.

The following paragraphs are a short introduction to FOOP as designed by *Michael Michaels* from neglook.com.

FOOP classes and constructors

Class attributes and methods are stored in the namespace of the object class. No object instance data is stored in this namespace/context. Data variables in the class namespace only describe the class of objects as a whole but don't contain any object specific information. A generic FOOP object constructor can be used as a template for specific object constructors when creating new object classes with `new`:

```
; built-in generic FOOP object constructor
(define (Class:Class)
    (cons (context) (args)))

; create some new classes

(new Class 'Rectangle)   → Rectangle
(new Class 'Circle)      → Circle
```

```
; create some objects using the default constructor

(set 'rect (Rectangle 10 20))   → (Rectangle 10 20)
(set 'circ (Circle 10 10 20))   → (Circle 10 10 20)

; create a list of objects
; building the list using the list function instead of assigning
; a quoted list ensures that the object constructors are executed

(set 'shapes (list (Circle 5 8 12) (Rectangle 4 8) (Circle 7 7 15)))
→ ((Circle 5 8 12) (Rectangle 4 8) (Circle 7 7 15))
```

The generic FOOP constructor is already pre-defined, and FOOP code can start with `(new Class ...)` statements right away.

As a matter of style, new classes should only be created in the MAIN context. If creating a new class while in a different namespace, the new class name must be prefixed with MAIN and the statement should be on the top-level:

```
(context 'Geometry)

(new Class 'MAIN:Rectangle)
(new Class 'MAIN:Circle)

...
```

Creating the namespace classes using new reserves the class name as a context in newLISP and facilitates forward references. At the same time, a simple constructor is defined for the new class for instantiating new objects. As a convention, it is recommended to start class names in upper-case to signal that the name stands for a namespace.

In some cases, it may be useful to overwrite the simple constructor, that was created during class creation, with `new`:

```
; overwrite simple constructor
(define (Circle:Circle x y radius)
    (list Circle x y radius))
```

A constructor can also specify defaults:

```
; constructor with defaults
(define (Circle:Circle (x 10) (y 10) (radius 3))
    (list Circle x y radius))

(Circle) → (Circle 10 10 3)
```

In many cases the constructor as created when using `new` is sufficient and overwriting it is not necessary.

Objects and associations

FOOP represents objects as lists. The first element of the list indicates the object's kind or class, while the remaining elements contain the data. The following statements define two *objects* using any of the constructors defined previously:

```
(set 'myrect (Rectangle 5 5 10 20)) → (Rectangle 5 5 10 20)
(set 'mycircle (Circle 1 2 10)) → (Circle 1 2 10)
```

An object created is identical to the function necessary to create it (hence FOOP). Nested objects can be created in a similar manner:

```
; create classes
(new Class 'Person)
(new Class 'Address)
(new Class 'City)
(new Class 'Street)

; create an object containing other objects
(set 'JohnDoe (Person (Address (City "Boston") (Street 123 "Main Street"))))
→ (Person (Address (City "Boston") (Street 123 "Main Street")))
```

Objects in FOOP not only resemble functions they also resemble associations. The assoc function can be used to access object data by name:

```
(assoc Address JohnDoe) → (Address (City "Boston") (Street 123 "Main Street"))

(assoc (list Address Street) JohnDoe) → (Street 123 "Main Street")
```

In a similar manner setf together with assoc can be used to modify object data:

```
(setf (assoc (list Address Street) JohnDoe) '(Street 456 "Main Street"))
→ (Street 456 "Main Street")
```

The street number has been changed from `123` to `456`.

Note that in none of the `assoc` statements `Address` and `Street` need to carry quotes. The same is true in the set statement: `(set 'JohnDoe (Person ...))` for the data part assigned. In both cases we do not deal with symbols or lists of symbols but rather with contexts and FOOP objects which evaluate to themselves. Quoting would not make a difference.

The colon : operator and polymorphism

In newLISP, the colon character : is primarily used to connect the context symbol with the symbol it is qualifying. Secondly, the colon function is used in FOOP to resolve a function's application *polymorphously*.

The following code defines two functions called `area`, each belonging to a different namespace / class. Both functions could have been defined in different modules for better separation, but in this case they are defined in the same file and without bracketing context statements. Here, only the symbols `rectangle:area` and `circle:area` belong to different namespaces. The local parameters `p`, `c`, `dx`, and `dy` are all part of `MAIN`, but this is of no concern.

```
;; class methods for rectangles

(define (Rectangle:area)
    (mul (self 3) (self 4)))

(define (Rectangle:move dx dy)
    (inc (self 1) dx)
```

```
    (inc (self 2) dy))

;; class methods for circles

(define (Circle:area)
    (mul (pow (self 3) 2) (acos 0) 2))

(define (Circle:move dx dy)
    (inc (self 1) dx)
    (inc (self 2) dy))
```

By prefixing the `area` or `move` symbol with the : (colon), we can call these functions for each class of object. Although there is no space between the colon and the symbol following it, newLISP parses them as distinct entities. The colon works as a function that processes parameters:

```
(:area myrect) → 200 ; same as (: area myrect)
(:area mycircle) → 314.1592654 ; same as (: area mycircle)

;; map class methods uses curry to enclose the colon operator and class function

(map (curry :area) (list myrect mycircle)) → (200 314.1592654)

(map (curry :area) '((Rectangle 5 5 10 20) (Circle 1 2 10))) → (200 314.1592654)


;; objects are mutable (since v10.1.8)

(:move myrect 2 3)
(:move mycircle 4 5)

myrect    → (Rectangle 7 8 10 20)
mycircle  → (Circle 5 7 10)
```

In this example, the correct qualified symbol (`rectangle:area` or `circle:area`) is constructed and applied to the object data based on the symbol following the colon and the context name (the first element of the object list).

Note, that although the caller specifies the called target object of the call, the method definition does not include the object as a parameter. When writing functions to modify FOOP objects, instead the function self is used to access and index the object.

Structuring a larger FOOP program

In all the previous examples, class function methods where directly written into the MAIN context namespace. This works and is adequate for smaller programs written by just one programmer. When writing larger systems, all the methods for one class should be surrounded by context statements to provide better isolation of parameter variables used and to create an isolated location for potential class variables.

Class variables could be used in this example as a container for lists of objects, counters or other information specific to a class but not to a specific object. The following code segment rewrites the example from above in this fashion.

Each context / namespace could go into an extra file with the same name as the class

contained. Class creation, startup code and the main control code is in a file `MAIN.lsp`:

```
; file MAIN.lsp - declare all classes used in MAIN

(new Class 'Rectangle)
(new Class 'Circle)

; start up code

(load "Rectangle.lsp")
(load "Circle.lsp")

; main control code

; end of file
```

Each class is in a separate file:

```
; file Rectangle.lsp - class methods for rectangles

(context Rectangle)

(define (Rectangle:area)
(mul (self 3) (self 4)))

(define (Rectangle:move dx dy)
(inc (self 1) dx)
(inc (self 2) dy))

; end of file
```

And the `Circle` class file follows:

```
; file Circle.lsp - class methods for circles

(context Circle)

(define (Circle:area)
    (mul (pow (self 3) 2) (acos 0) 2))

(define (Circle:move dx dy)
    (inc (self 1) dx)
    (inc (self 2) dy))

; end of file
```

All sets of class functions are now lexically separated from each other.

(§)

19. Concurrent processing and distributed computing

newLISP has high-level APIs to control multiple processes on the same CPU or distributed onto different computer nodes on a TCP/IP network.

Cilk API

newLISP implements a Cilk- like API to launch and control concurrent processes. The API can take advantage of multi-core computer architectures. Only three functions, spawn, sync and abort, are necessary to start multiple processes and collect the results in a synchronized fashion. The underlying operating system distributes processes onto different cores inside the CPU or executes them on the same core in parallel if there are not enough cores present. Note that newLISP only implements the API; optimized scheduling of spawned procedures is not performed as in Cilk. Functions are started in the order they appear in `spawn` statements and are distributed and scheduled onto different cores in the CPU by the operating system.

When multiple cores are present, this can increase overall processing speed by evaluating functions in parallel. But even when running on single core CPUs, the Cilk API makes concurrent processing much easier for the programmer and may speed up processing if subtasks include waiting for I/O or sleeping.

Since version 10.1 send and receive message functions are available for communications between parent and child processes. The functions can be used in blocking and non blocking communications and can transfer any kind of newLISP data or expressions. Transmitted expressions can be evaluated in the recipients environment.

Internally, newLISP uses the lower level fork, wait-pid, destroy, and share functionalities to control processes and synchronize the passing of computed results via a shared memory interface.

Only on Mac OS X and other Unixes will the Cilk API parallelize tasks. On Win32, the API partly simulates the behavior on Unix but executes tasks sequentially. This way, code can be written that runs on all platforms.

Distributed network computing

With only one function, net-eval, newLISP implements distributed computing. Using `net-eval`, different tasks can be mapped and evaluated on different nodes running on a TCP/IP network or local domain Unix sockets network when running on the same computer. `net-eval` does all the housekeeping required to connect to remote nodes, transfer functions to execute, and collect the results. `net-eval` can also use a call-back function to further structure consolidation of incoming results from remote nodes.

The functions read-file, write-file, append-file and delete-file all can take URLs instead of path-file names. Server side newLISP running in demon mode or an other HTTP server like Apache, receive standard HTTP requests and translate them into the corresponding actions on files.

(§)

20. JSON, XML, S-XML, and XML-RPC

JSON support

JSON-encoded data can be parsed into S-expressions using the json-parse function. Error information for failed JSON translations can be retrieved using json-error.

For a description of the JSON format (JavaScript Object Notation) consult json.org. Examples for correct formatted JSON text can be seen at json.org/examples.html.

To retrieve data in nested lists resulting from JSON translation, use the assoc, lookup and ref functions.

See the description of json-parse for a complete example of parsing and processing JSON data.

XML support

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")))
("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 newLISP *S-expressions*. The `@` (at symbol) denotes an XML attribute specification.

To retrieve data in nested lists resulting from S-XML translation, use the assoc, lookup and ref functions.

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 network and XML support. A stateless XML-RPC server implemented as a CGI service can be found in the file `examples/xmlrpc.cgi`. This script can be used together with a web server, like Apache. This XML-RPC service script implements the following methods:

method	**description**
`system.listMethods`	Returns a list of all method names
`system.methodHelp`	Returns help for a specific method
`system.methodSignature`	Returns a list of return/calling signatures for a specific method
`newLISP.evalString`	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 script 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.

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

(XMLRPC:newLISP.evalString
"http://localhost:8080/xmlrpc.cgi"
"(+ 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, the expression `(XMLRPC:error)` can be evaluated to return an error message.

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

(§)

21. Customization, localization, and UTF-8

Customizing function names

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

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

Now, `plus` is functionally equivalent to `+` and runs at the same speed.

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.

It is possible to redefine all integer arithmetic operators to their floating point equivalents:

```
(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'

; make the new symbol global at the same time
(constant (global 'imprime) print)  ; Spanish for 'print'
…
```

The new symbol can be made global at the same time using global.

Switching the locale

newLISP can switch locales based on the platform and operating system. On startup, non-UTF-8 enabled newLISP attempts to set the ISO C standard default POSIX locale, available for most platforms and locales. On UTF-8 enabled newLISP the default locale for the platform is set. The set-locale function can also be used 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) works 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 "es_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  or OSX system
(set-locale "de_DE") → ("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, Switzerland, 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, 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 multi-byte character sets as described in this chapter.

newLISP can be compiled as a UTF-8–enabled application. UTF-8 is a multi-byte 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. The U.S. version of WinXP's `notepad.exe` can display Unicode UTF-8–encoded characters, but the command shell 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
(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 multi-byte 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.

Functions working on UTF-8 characters

When UTF-8–enabled newLISP is used, the following string functions work on one- or multi-byte characters rather than one 8-bit byte boundaries:

function	description
char	translates between characters and ASCII/Unicode
chop	chops characters from the end of a string
date	converts date number to string (when used with the third argument)
dostring	evaluates once for each character in a string
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 *nth* 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
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 8-bit bytes. When positions are returned, as in find or regex, they are single 8-bit byte positions rather than character positions which may be multi-byte. The get-char and slice functions do not take multi-byte character offsets, but single-byte offsets, even in UTF-8 enabled versions of newLISP. The reverse function reverses a byte vector, not a character vector. The last three functions can still be used to manipulate binary non-textual data in the UTF-8–enabled version of newLISP. To make slice and reverse work with UTF-8 strings, combine them with explode and join.

To enable UTF-8 in Perl Compatible Regular Expressions (PCRE) — used by directory, find, member, parse, regex, regex-comp and replace — set the option number accordingly (2048). Note that offset and lengths in regex results are always in single byte counts. 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.

Functions only available on UTF-8 enabled versions

function	description
unicode	converts UTF-8 or ASCII strings into USC-4 Unicode
utf8	converts UCS-4 Unicode strings to UTF-8
utf8len	returns the number of UTF-8 characters in a string

The first two 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*.

(§)

22. 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 (the comma symbol, `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 and letn.

(§)

(∂)

newLISP Function Reference

1. Syntax of symbol variables and numbers

Source code in newLISP is parsed according to 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 should 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.

5. A symbol name starting with `$` (dollar sign) is global. There are several of these symbols already built into newLISP and set and changed internally. This type of global symbol can also be created by the user.

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

```
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 ; the symbol '(#:L*'

(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

When parsing binary, hex, decimal, float and integer numbers, up to 1000 digits are parsed when present. The rest will be read as new token(s). Note that IEEE 754 64-bit doubles distinguish only up to 16 significant digits. If more than 308 digits are present before the decimal point, the number will convert to `inf` (infinity). For big integers the 1000 limitation exists only when parsing source. There is no limit when a result of big integers math exceeds 1000 digits.

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).

```
123
+4567
-999
```

Big integers can be of unlimited precision and are processed differently from normal 64-bi integers internally.

```
12345678901234567890123456789O ; will automatically be converted to big int
-123L                          ; appended L forces conversion
0L
```

when parsing the command line or programming source, newLISP will recognise, integers bigger than 64-bit and convert the to big integers. Smaller numbers can be forced to big integer format by appending the letter L.

Hexadecimals start with a `0x` (or `0X`), followed by any combination of the hexadecimal digits: `0123456789abcdefABCDEF`. Any other character ends the hexadecimal number. Only up to 16 hexadecimal digits are valid and any more digits are ignored.

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

Binaries start with a `0b` (or `0B`), followed by up to 64 bits coded with 1's or 0s. Any other character ends the binary number. Only up to 64 bits are valid and any more bits are ignored.

```
0b101010   → 42
```

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. Only up to 21 octal digits are valid and any more digits are ignored.

```
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); 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.

Only 16 digits are siginificant and any more digits are ignored.

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

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

```
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.

Arguments enclosed in brackets [and] are optional. When arguments are separated by a vertical | then one of them must be chosen.

array

An array (constructed with the array function).

body

One or more expressions for evaluation. 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
```

bool

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

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

context

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

```
MyContext, aCtx, TheCTX
```

exp

Any data type described in this chapter.

func

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

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

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)
```

list

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

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

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. Any data type can be element of a matrix, but when using specific matrix operations like 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
```

place

A place referenced by a symbol or a place defined in a list, array or string by indexing with nth or implicit indexing or a place referenced by functions like first, last, assoc or lookup.

str

A string or an expression that evaluates to a string.

Depending on the length and processing of special characters, strings are delimited by either quotes "", braces {} or [text][/text] tags

```
"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:

character	description
\"	for a double quote inside a quoted string
\n	the line-feed character (ASCII 10)
\r	the carriage return character (ASCII 13)
\b	for a backspace BS character (ASCII 8)
\t	for a TAB character (ASCII 9)
\f	for a formfeed FF character (ASCII 12)
\nnn	a decimal ASCII code where nnn is between 000 and 255
\xnn	a hexadecimal code where nn is between 00 and FF
\unnnn	a unicode character encoded in the four nnnn hexadecimal digits. When reading a quoted string, newLISP will translate this to a UTF8 character in the UTF8 enabled versions of newLISP.
\\	the backslash character itself

Decimals start with a digit. Hexadecimals start with x:

```
"\065\066\067" → "ABC"
"\x41\x42\x43" → "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


; path names on MS Windows

(set 'path "C:\\MyDir\\example.lsp")

; no escaping when using braces

(set 'path {C:\MyDir\example.lsp})

; on MS Windows the forward slash can be used in path names

(set 'path "C:/MyDir/example.lsp")

; inner braces 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,
```

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 quoted symbols (e.g., context). For other functions, such as context?, the distinction is critical.

(§)

3. Functions in groups

Some functions appear in more than one group.

List processing, flow control, and integer arithmetic

+, -, *, /, %	integer arithmetic
++	increment integer numbers
--	decrement integer numbers
<, >, =	compares any data type: less, greater, equal
<=, >=, !=	compares any data type: less-equal, greater-equal, not-equal
:	constructs a context symbol and applies it to an object
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 function or macro expression
assoc	searches for keyword associations in a list
begin	begins a block of functions
bigint	convert a number to big integer format
bind	binds variable associations in a list
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
collect	repeat evaluating an expression and collect results in a list
cond	branches conditionally to expressions
cons	prepends an element to a list, making a new list
constant	defines a constant symbol
count	counts elements of one list that occur in another list
curry	transforms a function f(x, y) into a function fx(y)
define	defines a new function or lambda expression
define-macro	defines a macro or lambda-macro expression
def-new	copies a symbol to a different context (namespace)
difference	returns the difference between two lists
doargs	iterates through the arguments of a function
dolist	evaluates once for each element in a list
dostring	evaluates once for each character in a string
dotimes	evaluates once for each number in a range
dotree	iterates through the symbols of a context
do-until	repeats evaluation of an expression until the condition is met
do-while	repeats evaluation of an expression while the condition is true
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
eval	evaluates an expression

exists	checks for the existence of a condition in a list
expand	replaces a symbol in a nested list
explode	explodes a list or string
extend	extends a list or string
first	gets the first element of a list or string
filter	filters a list
find	searches for an element in a list or string
flat	returns the flattened list
fn	defines a new function or lambda expression
for	evaluates once for each number in a range
for-all	checks if all elements in a list meet a condition
if	evaluates an expression conditionally
index	filters elements from a list and returns their indices
intersect	returns the intersection of two lists
lambda	defines a new function or lambda expression
last	returns the last element of a list or string
length	calculates the length of a list or string
let	declares and initializes local variables
letex	expands local variables into an expression, then evaluates
letn	initializes local variables incrementally, like nested lets
list	makes a list
local	declares local variables
lookup	looks up members in an association list
map	maps a function over members of a list, collecting the results
match	matches patterns against lists; for matching against strings, see find and regex
member	finds a member of a list or string
not	logical `not`
nth	gets the *nth* element of a list or string
or	logical `or`
pop	deletes and returns an element from a list or string
pop-assoc	removes an association from an association list
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-all	returns a list of index vectors of elements inside a nested list
rest	returns all but the first element of a list or string
replace	replaces elements inside a list or string
reverse	reverses a list or string
rotate	rotates a list or string
select	selects and permutes elements from a list or string
self	Accesses the target object inside a FOOP method

set	sets the binding or contents of a symbol
setf setq	sets contents of a symbol or list, array or string reference
set-ref	searches for an element in a nested list and replaces it
set-ref-all	searches for an element in a nested list and replaces all instances
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
union	returns a unique list of elements found in two or more lists.
unless	evaluates an expression conditionally
until	repeats evaluation of an expression until the condition is met
when	evaluates a block of statements conditionally
while	repeats evaluation of an expression while the condition is true

String and conversion functions

address	gets the memory address of a number or string
bigint	convert a number to big integer format
bits	translates a number into binary representation
char	translates between characters and ASCII codes
chop	chops off characters from the end of a string
dostring	evaluates once for each character in 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
extend	extends a list or string
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 a 32-bit integer from a memory address
get-long	gets a long 64-bit 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
last	returns the last element of a list or string
lower-case	converts a string to lowercase characters
member	finds a list or string member
name	returns the name of a symbol or its context as a string
nth	gets the *nth* element in a list or string
pack	packs newLISP expressions into a binary structure
parse	breaks a string into tokens
pop	pops from a string
push	pushes onto a string
regex	performs a Perl-compatible regular expression search
regex-comp	pre-compiles a regular expression pattern
replace	replaces elements in a list or string
rest	gets all but the first element of a list or string
reverse	reverses a list or string
rotate	rotates a list or string
select	selects and permutes elements from a list or string
setf setq	sets contents of a string reference
slice	extracts a substring or sublist
source	returns the source required to bind a symbol as a string
starts-with	checks the start of the string or list against a key string or list
string	transforms anything into a string
sym	translates a string into a symbol
title-case	converts the first character of a string to uppercase
trim	trims a string on one or both sides
unicode	converts ASCII or UTF-8 to UCS-4 Unicode
utf8	converts UCS-4 Unicode to UTF-8
utf8len	returns length of an UTF-8 string in UTF-8 characters
unpack	unpacks a binary structure into newLISP expressions
upper-case	converts a string to uppercase characters

Floating point math and special functions

abs	returns the absolute value of a number
acos	calculates the arc-cosine of a number
acosh	calculates the inverse hyperbolic cosine of a number
add	adds floating point or integer numbers and returns a floating point number
array	creates an array
array-list	returns a list conversion from an array

asin	calculates the arcsine of a number
asinh	calculates the inverse hyperbolic sine of a number
atan	calculates the arctangent of a number
atanh	calculates the inverse hyperbolic tangent of a number
atan2	computes the principal value of the arctangent of Y / X in radians
beta	calculates the beta function
betai	calculates the incomplete beta function
binomial	calculates the binomial function
ceil	rounds up to the next integer
cos	calculates the cosine of a number
cosh	calculates the hyperbolic cosine of a number
crc32	calculates a 32-bit CRC for a data buffer
dec	decrements a number in a variable, list or array
div	divides floating point or integer numbers
erf	calculates the error function of a number
exp	calculates the exponential *e* of a number
factor	factors a number into primes
fft	performs a fast Fourier transform (FFT)
floor	rounds down to the next integer
flt	converts a number to a 32-bit integer representing a float
gammai	calculates the incomplete Gamma function
gammaln	calculates the log Gamma function
gcd	calculates the greatest common divisor of a group of integers
ifft	performs an inverse fast Fourier transform (IFFT)
inc	increments a number in a variable, list or array
inf?	checks if a floating point value is infinite
log	calculates the natural or other logarithm of a number
min	finds the smallest value in a series of values
max	finds the largest value in a series of values
mod	calculates the modulo of two numbers
mul	multiplies floating point or integer numbers
NaN?	checks if a float is NaN (not a number)
round	rounds a number
pow	calculates *x* to the power of *y*
sequence	generates a list sequence of numbers
series	creates a geometric sequence of numbers
sgn	calculates the signum function of a number
sin	calculates the sine of a number
sinh	calculates the hyperbolic sine of a number
sqrt	calculates the square root of a number
sub	subtracts floating point or integer numbers
tan	calculates the tangent of a number

tanh	calculates the hyperbolic tangent of a number
uuid	returns a UUID (Universal Unique IDentifier)

Matrix functions

det	returns the determinant of a matrix
invert	returns the inversion of a matrix
mat	performs scalar operations on matrices
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
mat	performs scalar operations on matrices
multiply	multiplies two matrices
nth	returns an element of an array
rest	returns all but the first row of an array
setf	sets contents of an array reference
slice	returns a slice of an array
transpose	transposes a matrix

Bit operators

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

Predicates

atom?	checks if an expression is an atom

array?	checks if an expression is an array
bigint?	checks if a number is a big integer
context?	checks if an expression is a context
directory?	checks if a disk node is a directory
empty?	checks if a list or string is empty
even?	checks the parity of an integer number
file?	checks if a file exists
float?	checks if an expression is a float
global?	checks if a symbol is global
inf?	checks if a floating point value is infinite
integer?	checks if an expression is an integer
lambda?	checks if an expression is a lambda expression
legal?	checks if a string contains a legal symbol
list?	checks if an expression is a list
macro?	checks if an expression is a lambda-macro expression
NaN?	checks if a float is NaN (not a number)
nil?	checks if an expression is `nil`
null?	checks if an expression is `nil`, `""`, `()`, `0` or `0.0`
number?	checks if an expression is a float or an integer
odd?	checks the parity of an integer number
protected?	checks if a symbol is protected
primitive?	checks if an expression is a primitive
quote?	checks if an expression is quoted
string?	checks if an expression is a string
symbol?	checks if an expression is a symbol
true?	checks if an expression is not `nil`
zero?	checks if an expression is `0` or `0.0`

Date and time functions

date	converts a date-time value to a string
date-list	returns a list of year, month, day, hours, minutes, seconds from a time value in seconds
date-parse	parses a date string and returns the number of seconds passed since January 1, 1970, (formerly `parse-date`)
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

Statistics, simulation and modeling functions

amb	randomly selects 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
corr	calculates the *product-moment correlation* coefficient
crit-chi2	calculates the *Chi²* statistic for a given probability
crit-f	calculates the *F* statistic for a given probability
crit-t	calculates the *Student's t* statistic for a given probability
crit-z	calculates the normal distributed *Z* for a given probability
kmeans-query	calculates distances to cluster centroids or other data points
kmeans-train	partitions a data set into clusters
normal	makes a list of normal distributed floating point numbers
prob-chi2	calculates the tail probability of a *Chi²* distribution value
prob-f	calculates the tail probability of a *F* distribution value
prob-t	calculates the tail probability of a *Student's t* distribution value
prob-z	calculates the cumulated probability of a *Z* distribution 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
stats	calculates some basic statistics for a data vector
t-test	compares means of data samples using the *Student's t* statistic

Pattern matching

ends-with	tests if a list or string ends with a pattern
find	searches for a pattern in a list or string
find-all	finds all occurrences of a pattern in a string
match	matches list patterns
parse	breaks a string along around patterns
ref	returns the position of an element inside a nested list
ref-all	returns a list of index vectors of elements inside a nested list
regex	finds patterns in a string
replace	replaces patterns in a string
search	searches for a pattern in a file
starts-with	tests if a list or string starts with a pattern
unify	performs a logical unification of patterns

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

Input/output and file operations

append-file	appends data to a file
close	closes a file
current-line	retrieves contents of last read-line buffer
device	sets or inquires about current print device
exec	launches another program, then reads from or writes to it
load	loads and evaluates a file of newLISP code
open	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	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
read-utf8	reads UTF-8 character from a 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	writes binary data to a file
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 and the Cilk API

!	shells out to the operating system
abort	aborts a child process started with `spawn`
destroy	destroys a process created with `fork` or `process`
exec	runs a process, then reads from or writes to it
fork	launches a newLISP child process
pipe	creates a pipe for interprocess communication
process	launches a child process, remapping standard I/O and standard error
receive	receive a message from another process

semaphore	creates and controls semaphores
send	send a message to another process
share	shares memory with other processes
spawn	launches a child process for Cilk process management
sync	waits for child processes launched with `spawn` and collects results
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

HTTP networking API

base64-enc	encodes a string into BASE64 format
base64-dec	decodes a string from BASE64 format
delete-url	deletes a file or page from the web
get-url	reads a file or page from the web
json-error	returns error information from a failed JSON translation.
json-parse	parses JSON formatted data
post-url	posts info to a URL address
put-url	uploads a page to a URL address
xfer-event	registers an event handler for HTTP byte transfers
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, UDP and ICMP 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-interface	Sets the default interface IP address on multihomed computers.
net-ipv	Switches between IPv4 and IPv6 internet protocol versions.
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-packet	send a custom configured IP packet over raw sockets
net-peek	returns the number of characters ready to be read from a network socket
net-peer	returns the remote IP and port for a net connect
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

API for newLISP in a web browser

display-html	display an HTML page in a web browser
eval-string-js	evaluate JavaScript in the current web browser page

Reflection and customization

command-event	pre-processes the command-line and HTTP requests
error-event	defines an error handler
last-error	report the last error number and text
macro	create a reader expansion macro
ostype	contains a string describing the OS platform
prefix	Returns the context prefix of a symbol
prompt-event	customizes the interactive newLISP shell prompt
read-expr	reads and translates s-expressions from source
reader-event	preprocess expressions before evaluation event-driven
set-locale	switches to a different locale
source	returns the source required to bind a symbol to a string
sys-error	reports OS system error numbers
sys-info	gives information about system resources

term	returns the term part of a symbol or its context as a string

System functions

$	accesses system variables $0 -> $15
callback	registers a callback function for an imported library
catch	evaluates an expression, catching errors and early returns
context	creates or switches to a different namespace
copy	copies the result of an evaluation
debug	debugs a user-defined function
delete	deletes symbols from the symbol table
default	returns the contents of a default functor from a context
env	gets or sets the operating system's environment
exit	exits newLISP, setting the exit value
global	makes a symbol accessible outside MAIN
import	imports a function from a shared library
main-args	gets command-line arguments
new	creates a copy of a context
pretty-print	changes the pretty-printing characteristics
read-expr	translates a string to an s-expression without evaluating it
reset	goes to the top level
signal	sets a signal handler
sleep	suspends processing for specified milliseconds
sym	creates a symbol from a string
symbols	returns a list of all symbols in the system
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

Importing libraries

address	returns the memory address of a number or string
callback	registers a callback function for an imported library
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 a 32-bit integer from a memory address

get-long	gets a long 64-bit 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 newLISP expressions into a binary structure
struct	Defines a data structure with C types
unpack	unpacks a binary structure into newLISP expressions

newLISP internals API

command-event	pre-processes the command-line and HTTP requests
cpymem	copies memory between addresses
dump	shows memory address and contents of newLISP cells
prompt-event	customizes the interactive newLISP shell prompt
read-expr	reads and translates s-expressions from source
reader-event	preprocess expressions before evaluation event-driven

(§)

4. Functions in alphabetical order

!

syntax: (! *str-shell-command* [*int-flags*])

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.

```
(! "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.

On Ms Windows the optional *int-flags* parameter takes process creation flags as defined for the Windows `CreateProcessA` function to control various parameters of process creation. The inclusion of this parameter – which also can be `0` – forces a different creation of the process without a command shell window. This parameter is ignored on Unix.

```
; on MS Windows
; close the console of the currently running newLISP process
(apply (import "kernel32" "FreeConsole"))
```

```
; start another process and wait for it to finish
(! "notepad.exe" 0)

(exit)
```

Without the additional parameter, the ! call would create a new command window replacing the closed one.

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

```
> !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, ends-with, find, find-all, parse, regex, search, starts-with and replace) all bind their results to the predefined system variables $0, $1, $2–$15 after or during the function's execution. System variables can be treated the same as any other symbol. As an alternative, the contents of these variables may also be accessed by using ($ 0), ($ 1), ($ 2), etc. This method allows indexed access (i.e., ($ i), where i is an integer).

```
(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"
```

+, -, *, / ,% bigint

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.

```
(+ 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 the integer value closest to `0` (zero).

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. This includes the values for `+Inf` and `-Inf`.

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.

Floating point values that evaluate to `NaN` (Not a Number), ar treated as `0` (zero).

++ ! bigint

syntax: (++ *place* [*num* ...])

The `++` operator works like inc, but performs integer arithmetic. Without the optional argument in *num*, `++` increments the number in *place* by `1`.

If floating point numbers are passed as arguments, their fractional part gets truncated first.

Calculations 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.

place is either a symbol or a place in a list structure holding a number, or a number returned by an expression.

```
(set 'x 1)
(++ x)        → 2
(set 'x 3.8)
(++ x)        → 4
(++ x 1.3)    → 5
(set 'lst '(1 2 3))
(++ (lst 1) 2))  → 4
lst              → (1 4 3)
```

If the symbol for *place* contains `nil`, it is treated as if containing `0`.

See -- for decrementing numbers in integer mode. See inc for incrementing numbers in floating point mode.

-- ! bigint

syntax: (-- *place* [*num ...*])

The `--` operator works like dec, but performs integer arithmetic. Without the optional argument in *num-2*, `--` decrements the number in *place* by `1`.

If floating point numbers are passed as arguments, their fractional part gets truncated first.

Calculations 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.

place is either a symbol or a place in a list structure holding a number, or a number returned by an expression.

```
(set 'x 1)
(-- x)        → 0
(set 'x 3.8)
(-- x)        → 2
(-- x 1.3)    → 1

(set 'lst '(1 2 3))
(-- (lst 1) 2))  → 0
lst              → (1 0 3)
```

If the symbol for *place* contains `nil`, it is treated as if containing `0`.

See ++ for incrementing numbers in integer mode. See dec for decrementing numbers in

floating point mode.

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

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

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.

If only one argument is supplied, all comparison operators assume `0` (zero) as a second argument. This can be used to check if a number is negative, positive, zero or not zero.

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

```
(< 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

; with single argument compares against 0

(> 1)    → true ; checks for positive
(> -1)   → nil ; checks for negative
(= 123)  → nil ; checks for zero
```

```
(map > '(1 3 -4 -3 1 2))   → (true true nil nil true 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!


(<< 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.

```
(& 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.

```
(| 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.

```
(^ 0xAA 0x55)  → 255
```

~

syntax: (~ *int*)

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

```
(format "%X" (~ 0xFFFFFFAA))  → "55"
(~ 0xFFFFFFFF)                → 0
```

:

syntax: (: *sym-function list-object* [...])

The colon is used not only as a syntactic separator between namespace prefix and the term inside but also as an operator. When used as an operator, the colon : constructs a context symbol from the context name in the object list and the symbol following the colon. The object list in *list-object* can be followed by other parameters.

The : operator implements *polymorphism* of object methods, which are part of different object classes represented by contexts (namespaces). In newLISP, an object is represented by a list, the first element of which is the symbol (name) of its class context. The class context implements the functions applicable to the object. No space is required between the colon and the symbol following it.

```
(define (Rectangle:area)
    (mul (self 3) (self 4)))

(define (Circle:area)
    (mul (pow (self 3) 2) (acos 0) 2))

(define (Rectangle:move dx dy)
    (inc (self 1) dx)
        (inc (self 2) dy))

(define (Circle:move p dx dy)
```

```
    (inc (self 1) dx) (inc (self 2) dy))

(set 'myrect '(Rectangle 5 5 10 20)) ; x y width height
(set 'mycircle '(Circle 1 2 10)) ; x y radius

;; using the : (colon) operator to resolve to a specific context

(:area myrect)     → 200
(:area mycircle)   → 314.1592654

;; map class methods uses curry to enclose the colon operator and class function

(map (curry :area) (list myrect mycircle)) → (200 314.1592654)

(map (curry :area) '((Rectangle 5 5 10 20) (Circle 1 2 10))) → (200 314.1592654)

;; change object attributes using a function and re-assigning
;; to the objects name

(:move myrect 2 3)
myrect   → (Rectangle 7 8 10 20)

(:move mycircle 4 5)
mycircle → (Circle 5 7 10)
```

Inside the FOOP methods the self function is used to access the target object of the method.

abort

syntax: (abort *int-pid*)
syntax: (abort)

In the first form, `abort` aborts a specific child process of the current parent process giving the process id in *int-pid*. The process must have been started using spawn. For processes started using fork, use destroy instead.

The function `abort` is not available on Win32.

```
(abort 2245)  → true
```

To abort all child processes spawned from the current process use `abort` without any parameters:

```
(abort)  → true ; abort all
```

The function `abort` is part of the Cilk API for synchronizing child processes and process parallelization. See the reference for the function spawn for a full discussion of the Cilk API.

abs **bigint**

syntax: (abs *num*)

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

```
(abs -3.5)  → 3.5
```

acos

syntax: (acos *num-radians*)

The arc-cosine function is calculated from the number in *num-radians*.

```
(acos 1)  → 0
(cos (acos 1)) → 1
```

acosh

syntax: (acosh *num-radians*)

Calculates the inverse hyperbolic cosine of *num-radians*, the value whose hyperbolic cosine is *num-radians*. If *num-radians* is less than 1, `acosh` returns `NaN`.

```
(acosh 2)  → 1.316957897
(cosh (acosh 2)) → 2
(acosh 0.5) → NaN
```

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`.

```
(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.

```
(set 's "\001\002\003\004")

(get-char (+ (address s) 3))   → 4

(set 'x 12345) ; x is a 64-bit long int

; on a big-endian CPU, i.e. PPC or SPARC
(get-long (address x))         → 12345
; the 32-bit int is in high 32-bit part of the long int
(get-int (+ (address x) 4))    → 12345

; on a little-endian CPU, i.e. Intel i386
; the 32-bit int is in the low 32-bit part of the long int
(get-int (address x))          → 12345

; on both architectures (integers are 64 bit in newLISP)
(set 'x 1234567890)
(get-long (address x))         →  1234567890
```

When a string is passed to C library function the address of the string is used automatically, and it is not necessary to use the `address` function in that case. As the example shows, `address` can be used to do pointer arithmetic on the string's address.

`address` should only be used on persistent addresses from data objects referred to by a variable symbol, not from volatile intermediate expression objects.

See also the get-char, get-int, get-long and get-float functions.

amb

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

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

```
(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* ...])

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` or the empty list `()`, evaluation is terminated and `nil` or the empty list `()` is returned.

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

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.

```
(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:

```
(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.

```
(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. The string function would cut off strings at zero bytes.

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 extend and push to append to an existing list or string modifying the target.

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.

On failure the function returns `nil`. For error information, use sys-error when used on files. When used on URLs net-error gives more error information.

```
(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 case of the `http://` prefix , `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.

```
(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*])
syntax: (apply *func*)

Applies the contents of *func* (primitive, user-defined function, or lambda expression) to the arguments in *list*. Only functions and operators with standard evaluation of their arguments can be applied.

In the second syntax `apply` is used on functions without any arguments.

```
(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
;; note that newLISP already has a gcd function

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

(define-macro (my-gcd)
    (apply gcd_ (map eval (args)) 2))

(my-gcd 12 18 6)    → 6
(my-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. Note, that a built-in gcd is available.

`apply` should only be used on functions and operators that evaluate all of their arguments, not on *special forms* like dotimes 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.

```
(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*).

```
(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:

```
(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-n1* [*int-n2 ...*] [*list-init*])

Creates an array with *int-n1* 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. Not all list functions may be used on arrays. For a more detailed discussion, see the chapter on arrays.

```
(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 most of the same functions used for modifying lists:

```
(setf (myarray 2 3) 99) → 99)
myarray → ((1 2 3 4) (5 6 7 8) (9 10 11 99))

(setf (myarray 1 1) "hello")  → "hello"

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

(setf (myarray 1) '(a b c d)) → (a b c d)
myarray → ((1 2 3 4) (a b c d) (9 10 11 99))

(nth 1 myarray)     → (a b c d)  ; access a whole row

;; use implicit indexing and slicing on arrays

(myarray 1)     → (a b c d)

(myarray 0 -1)  → 4

(2 myarray)     → ((9 10 11 99))

(-3 2 myarray)  → ((1 2 3 4) (a b c d))
```

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 generated 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 arraylsp ;;

(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:

```
(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? *exp*)

Checks if *exp* is an array:

```
(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*)

Calculates the arcsine function from the number in *num-radians* and returns the result.

```
(asin 1)  → 1.570796327
(sin (asin 1)) → 1
```

asinh

syntax: (asinh *num-radians*)

Calculates the inverse hyperbolic sine of *num-radians*, the value whose hyperbolic sine is *num-radians*.

```
(asinh 2)         → 1.443635475
(sinh (asinh 2))  → 2
```

assoc

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

In the first syntax 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`.

```
(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
```

Together with setf `assoc` can be used to change an association.

```
(setf (assoc 'pears data) '(pears 8))

data  → ((apples 123) (bananas 123 45) (pears 8))
```

In the second syntax more then one key expressions can be specified to search in nested, multilevel association lists:

```
(set 'persons '(
    (id001 (name "Anne") (address (country "USA") (city "New York")))
    (id002 (name "Jean") (address (country "France") (city "Paris")))
))

(assoc '(id001 address) persons) → (address (country "USA") (city "New York"))
(assoc '(id001 address city) persons) → (city "New York")
```

The list in *list-aList* can be a context which will be interpreted as its *default functor*. This

way very big lists can be passed by reference for speedier access and less memory usage:

```
(set 'persons:persons '(
    (id001 (name "Anne") (address (country "USA") (city "New York")))
    (id002 (name "Jean") (address (country "France") (city "Paris")))
))

(define (get-city db id)
    (last (assoc (list id 'address 'city) db ))
)

(get-city persons 'id001) → "New York"
```

For making replacements in association lists, use the setf together with the `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.

```
(atan 1)        → 0.7853981634
(tan (atan 1))  → 1
```

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.

```
(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
```

atanh

syntax: (atanh *num-radians*)

Calculates the inverse hyperbolic tangent of *num-radians*, the value whose hyperbolic

tangent is *num-radians*. If the absolute value of *num-radians* is greater than 1, `atanh` returns `NaN`; if it is equal to 1, `atanh` returns infinity.

```
(atanh 0.5) → 0.5493061443
(tanh (atanh 0.5)) → 0.5
(atanh 1.1) → NaN
(atanh 1) → inf
```

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.

```
(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.

```
(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 and conforms to the RFC 4648 standard.

base64-enc

syntax: (base64-enc *str* [*bool-flag*])

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.

Without the optional *bool-flag* parameter the empty string `""` is encoded into `"===="`. If *bool-flag* evaluates to `true`, the empty string `""` is translated into `""`. Both translations result in `""` when using base64-dec.

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).

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

(base64-enc "")             → "===="
(base64-enc "" true)        → ""
```

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 and conforms to the RFC 4648 standard.

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*) or against all other categories (*Mi*). All tokens in *list-L* should occur in *context-D*. When using the default *R.A. Fisher inverse 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 inverse Chi² method of compounding probabilities or the Chain Bayesian method. By default, when no flag or `nil` is specified in *bool-probs*, the inverse Chi² method of

compounding probabilities is used. When specifying `true` in *bool-probs*, the Chain Bayesian method is used.

If the inverse 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, the formula can be generalized to:

p(Mc|tkn) = p(tkn|Mc) * p(Mc) / 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(Mc|tkn)* and *p(A|tkn)* are the *posterior Bayesian* probabilities of a category or model. This *naive* Bayes formula does nor take into account dependencies between different categories.

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

R.A. Fisher inverse 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)
→ (0.0359554723158327 0.964044527684167)

(bayes-query (parse "immense faculties and extraordinary powers of observation")
    'DoyleDowson)
→ (0.983569359827141 0.0164306401728594)
```

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:

```
(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?

```
(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 Chi² 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:

```
(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 ...*] *sym-context-D*)

Takes one 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.

```
(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)
→ (4 3 3)

S:_one    → (1 1 1)
S:_two    → (2 0 1)
S:_three  → (1 2 1)
S:total   → (4 3 3)
```

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

`bayes-train` creates symbols from strings prepending an underscore character. This is the same way hashes are created and contexts populates with symbols by `bayes-train` can be used like hashes:

```
; use a bayes-trained context namespace like a hash dictionary

(S "two")   → (2 0 1)
(S "three") → (1 2 1)

(S) → (("one" (1 1 1)) ("three" (1 2 1)) ("two" (2 0 1)))
```

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 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 shakepeare-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.

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

Some built-in functions like cond, define, doargs, dolist, dostring, dotimes, when 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* `gammaln` function as follows:

beta = exp(gammaln(a) + gammaln(b) - gammaln(a + b))

```
(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:

pev = Betai(p, k, N - k + 1)

```
(betai 0.5 3 8)  → 0.9453125
```

The 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. See also the binomial function.

bigint

syntax: (bigint *number*)
syntax: (bigint *string*)

A floating point or integer number gets converted to big integer format. When converting from floating point, rounding errors occur going back and forth between decimal and binary arithmetic.

A string argument gets parsed to a number and converted to a big integer.

```
(bigint 12345)          → 12345L

(bigint 1.234567890e30) → 1234567889999999957361000000000L

(set 'num 567890)
(bigint num)            → 567890L

(bigint "-54321")       → -54321L
(bigint "123.45")       → 123L
(bigint "123hello")     → 123L
```

See also the manual chapter Big integer, unlimited precision arithmetic

bigint?

syntax: (bigint? *number*)

Check if a number is formatted as a big integer.

```
(set 'x 12345)
(set 'y 12345L)
(set 'z 12345678901234567890123456789O)
(set 'p 1.2345e20)
(set 'q (bigint p))
```

```
(bigint? x)  → nil
(bigint? y)  → true
(bigint? z)  → true
(bigint? p)  → nil
(bigint? q)  → true
```

See also the manual chapter Big integer, unlimited precision arithmetic

bind !

syntax: (bind *list-variable-associations* [*bool-eval*])

list-variable-associations contains an association list of symbols and their values. `bind` sets all symbols to their associated values.

The associated values are evaluated if the *bool-eval* flag is `true`:

```
(set 'lst '((a (+ 3 4)) (b "hello")))

(bind lst)         → "hello"

a    → (+ 3 4)
b    → "hello"

(bind lst true)    → "hello"

a    → 7
```

The return value of bind is the value of the last association.

`bind` is often used to bind association lists returned by unify.

```
(bind (unify '(p X Y a) '(p Y X X)))    → a

X    → a
Y    → a
```

This can be used for de-structuring:

```
(set 'structure '((one "two") 3 (four (x y z))))
(set 'pattern '((A B) C (D E)))
(bind (unify pattern structure))

A → one
B → "two"
C → 3
D → four
E → (x y z)
```

unify returns an association list and `bind` binds the associations.

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:

binomial = pow(p, k) * pow(1.0 - p, n - k) * n! / (k! * (n - k)!)

where *x!* is the factorial of *x* and *pow(x, y)* is *x* raised to the power of *y*.

```
(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.

bits

syntax: (bits *int* [*bool*])

Transforms a number in *int* to a string of 1's and 0's or a list, if *bool* evaluates to anything not `nil`.

In string representation bits are in high to low order. In list presentation 1's and 0's are represented as `true` and `nil` and in order from the lowest to the highest bit. This allows direct indexing and program control switching on the result.

```
(bits 1234)      → "10011010010"

(int (bits 1234) 0 2) → 1234

(bits 1234 true)     → (nil true nil nil true nil true true nil nil true)

((bits 1234 true) 0) → nil ; indexing of the result
```

int with a base of 2 is the inverse function to `bits`.

callback

syntax: (callback *int-index sym-function*)
syntax: (callback *sym-function str-return-type* [*str_param_type* ...])
syntax: (callback *sym-function*)

In the first **simple `callback` syntax** up to sixteen (0 to 15) *callback* functions for up to eight parameters can be registered with imported libraries. The `callback` function returns a procedure address that invokes a user-defined function in *sym-function*. The following example shows the usage of callback functions when importing the OpenGL graphics library:

If more than sixteen callback functions are required, slots must be reassigned to a different callback function.

```
...
(define (draw)
    (glClear GL_COLOR_BUFFER_BIT )
    (glRotated rotx 0.0 1.0 0.0)
    (glRotated roty 1.0 0.0 0.0)
    (glutWireTeapot 0.5)
    (glutSwapBuffers))

(define (keyboard key x y)
    (if (= (& key 0xFF) 27) (exit)) ; exit program with ESC
    (println "key:" (& key 0xFF) " x:" x  " y:" y))

(define (mouse button state x y)
    (if (= state 0)
        (glutIdleFunc 0) ; stop rotation on button press
        (glutIdleFunc (callback 4 'rotation)))
    (println "button: " button " state:" state " x:" x " y:" y))

(glutDisplayFunc (callback 0 'draw))
(glutKeyboardFunc (callback 1 'keyboard))
(glutMouseFunc (callback 2 'mouse))
...
```

The address returned by `callback` is registered with the Glut library. The above code is a snippet from the file `opengl-demo.lsp`, in the `examples/` directory of the source distribution of newLISP and can also be downloaded from newlisp.org/downloads/OpenGL.

In the second **extended `callback` syntax** type specifiers are used to describe the functions return and parameter value types when the function is called. An unlimited number of callback functions can be registered with the second syntax, and return values are passed back to the calling function. The symbol in *sym-function* contains a newLISP defined function used as a callback function callable from a C program.

In the third syntax `callback` returns a previously returned C-callable address for that symbol.

While the first simple `callback` syntax only handles integers and pointer values, `callback` in the expanded syntax can also handle simple and double precision floating point numbers passed in an out of the `callback` function.

Both the simple and extended syntax can be mixed inside the same program.

The following example shows the import of the `qsort` C library function, which takes as one of it's arguments the address of a comparison function. The comparison function in this case is written in newLISP and called into by the imported `qsort` function:

```
; C void qsort(...) takes an integer array with number and width
; of array elements and a pointer to the comparison function
(import "libc.dylib" "qsort" "void" "void*" "int" "int" "void*")
```

```
(set 'rlist '(2 3 1 2 4 4 3 3 0 3))
; pack the list into an C readable 32-bit integer array
(set 'carray (pack (dup "ld " 10) rlist))

; the comparison callback function receives pointers to integers
(define (cmp a b)
    (- (get-int a) (get-int b)))

; generate a C callable address for cmp
(set 'func (callback 'cmp "int" "void*" "void*"))

; sort the carray
(qsort carray 10 4 func)

; unpack the sorted array into a LISP list
(unpack (dup "ld" 10) carray)  →  (0 1 2 2 3 3 3 3 4 4)
```

As type specifiers the same string tags can be used as in the import function. All pointer types are passed as numbers in and out of the `callback` function. The functions get-char, get-int, get-long and get-string can be used to extract numbers of different precision from parameters. Use pack and unpack to extract data from binary buffers and structures.

Note that newLISP as already a fast built-in sort 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 body expression is returned as the result for the entire `case` expression.

```
(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*:

```
(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.

```
(catch (func 3 4) 'result)  → nil
result
→ "ERR: invalid function in function catch : (func 3 4)"

(constant 'func +)          → + <4068A6>
(catch (func 3 4) 'result)  → true
result                      → 7
```

When a throw is executed during the evaluation of *exp*, `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.

```
(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.

```
(change-dir "/etc")
```

Makes `/etc` the current directory.

char utf8

syntax: (char *str* [*int-index* [true]])
syntax: (char *int*)

Given a string argument, extracts the character at *int-index* from *str*, returning either the ASCII value of that character or the Unicode value on UTF-8 enabled versions of newLISP.

If *int-index* is omitted, 0 (zero) is assumed. If *int-idx* is followed by a boolean `true` value, than the index treats *str* as an 8-bit byte array instead of an array of multi-byte UTF-8 characters.

The empty string returns `nil`. Both `(char 0)` and `(char nil)` will return `"\000"`.

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.

```
(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 "Ω")            → 937 ; UTF-8 code for "Ω"
(char "Ω" 1 true)     → 169 ; byte value at offset 1

(char 65)  → "A"
(char 66)  → "B"
```

```
(char (char 65))  → 65      ; two inverse applications

(map char (sequence 1 255))  ; returns current character set

; The Zen of UTF-8
(char (& (char "生") (char "死"))) → 愛 ; by @kosh_bot
```

chop utf8

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).

```
(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.

```
(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 <= '(3 4 -6 0 2 -3 0))  → (3 4 2)

(clean (curry match '(a *)) '((a 10) (b 5) (a 3) (c 8) (a 9)))
→  ((b 5) (c 8))
```

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 mode 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.

```
(close (device))  → true
(close 7)         → true
(close aHandle)   → true
```

Note that using `close` on device automatically resets it to 0 (zero, the screen device).

collect

syntax: (collect *exp* [*int-max-count*])

Evaluates the expression in *exp* and collects the results in a list until evaluation of *exp* returns `nil`.

Optionally a maximum count of elements can be specified in *int-max-count*.

```
; collect results until nil is returned
(set 'x 0)
(collect (if (<= (inc x) 10) x)) → (1 2 3 4 5 6 7 8 9 10)

; collect results until nil is returned or 6 results are collected
(set 'x 0)
(collect (if (<= (inc x) 10) x) 6) → (1 2 3 4 5 6)
```

command-event

syntax: (command-event *sym-event-handler* | *func-event-handler*)

Specifies a user defined function for pre-processing the newLISP command-line before it gets evaluated. This can be used to write customized interactive newLISP shells and to transform HTTP requests when running in server mode.

`command-event` takes either a symbol of a user-defined function or a lambda function. The event-handler function must return a string or the command-line will be passed untranslated to newLISP.

To only force a prompt, the function should return the empty string `""`.

The following example makes the newLISP shell work like a normal Unix shell when the command starts with a letter. But starting the line with an open parenthesis or a space initiates a newLISP evaluation.

```
(command-event (fn (s)
        (if (starts-with s "[a-zA-Z]" 0) (append "!" s) s)))
```

See also the related prompt-event which can be used for further customizing interactive mode by modifying the newLISP prompt.

The following program can be used either stand-alone or included in newLISP's `init.lsp` startup file:

```
#!/usr/bin/newlisp

; set the prompt to the current directory name
(prompt-event (fn (ctx) (append (real-path) "> ")))

; pre-process the command-line
(command-event (fn (s)
    (if
        (starts-with s "cd")
        (string " " (true? (change-dir (last (parse s " "))))))

        (starts-with s "[a-zA-Z]" 0)
        (append "!" s)

        true s)))
```

In the definition of the command-line translation function the Unix command `cd` gets a special treatment, to make sure that the directory is changed for newLISP process too. This way when shelling out with `!` and coming back, newLISP will maintain the changed directory.

Command lines for newLISP must start either with a space or an opening parenthesis. Unix commands must start at the beginning of the line.

When newLISP is running in server mode either using the `-c` or `-http` option, it receives HTTP requests similar to the following:

```
GET /index.html
```

Or if a query is involved:

```
GET /index.cgi?userid=joe&password=secret
```

A function specified by `command-event` could filter and transform these request lines, e.g.: discovering all queries trying to perform CGI using a file ending in `.exe`. Such a request would be translated into a request for an error page:

```
;; httpd-conf.lsp
;;
;; filter and translate HTTP requests for newLISP
;; -c or -http server modes
;; reject query commands using CGI with .exe files

(command-event (fn (s)
    (let (request s)
        (when (find "?" s) ; is this a query
            (set 'request (first (parse s "?")))
            ; discover illegal extension in queries
            (when (ends-with request ".exe")
                (set 'request "GET /errorpage.html")) )
        request)
))
```

When starting the server mode with `newlisp httpd-conf.lsp -c -d80 -w ./httpdoc` newLISP will load the definition for `command-event` for filtering incoming requests, and the query:

```
GET /cmd.exe?dir
```

Would be translated into:

```
GET /errorpage.html
```

The example shows a technique frequently used in the past by spammers on Win32 based, bad configured web servers to gain control over servers.

`httpd-conf.lsp` files can easily be debugged loading the file into an interactive newLISP session and entering the HTTP requests manually. newLISP will translate the command line and dispatch it to the built-in web server. The server output will appear in the shell window.

Note, that the command line length as well as the line length in HTTP headers is limited to 512 characters for newLISP.

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-condition*s 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*.

```
(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: *cons*ing two atoms constructs a list, not a dotted pair.

```
(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 newLISP 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. The last *exp-n* initializer is always optional.

Symbols initialized with set, define, or define-macro can still be protected by using the `constant` function:

```
(constant 'aVar 123)  → 123
(set 'aVar 999)
ERR: 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 'squareroot sqrt)  → sqrt <406C2E>
(constant '+ add)            → add <4068A6>
```

`squareroot` 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 | sym* [*exp-value*])

In the first syntax, `context` is used to switch to a different context namespace. Subsequent

loads of newLISP source or functions like eval-string and sym 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.

```
(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` can be used to create symbols in a namespace. Note that this should not be used for creating hashes or dictionaries. For a shorter, more convenient method to use namespaces as hash-like dictionaries, see the chapter Hash functions and dictionaries.

```
;; create a symbol and store data in it
(context 'Ctx "abc" 123)   → 123
(context 'Ctx 'xyz 999)    → 999

;; retrieve contents from  symbol
(context 'Ctx "abc")       → 123
(context 'Ctx 'xyz)        → 999
Ctx:abc                    → 123
Ctx:xyz                    → 999
```

The first three statements create a symbol and store a value of any data type inside. The first statement also creates the context named `Ctx`. When a symbol is specified for the name, the name is taken from the symbol and creates a symbol with the same name in the context `Ctx`.

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.

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`.

```
(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

syntax: (copy *exp*)
syntax: (copy *int-addr* [*bool-flag*])

The first syntax makes a copy from evaluating expression in *exp*. Some built-in functions are destructive, changing the original contents of a list, array or string they are working on. With `copy` their behavior can be made non-destructive.

```
(set 'aList '(a b c d e f))

(replace 'c (copy aList)) → (a b d e f)

aList → (a b c d e f)

(set 'str "newLISP") → "newLISP"

(rotate (copy str)) → "PnewLIS"

str → "newLISP"
```

Using `copy` the functions replace and rotate are prevented from changing the data. A modified version of the data is returned.

The second syntax, marked by the `true` in *bool-flag*, copies a newLISP expression from a memory address.The following two expressions are equivalent:

```
(set 'x "hello world")
(copy x)  → "hello world"
(copy (first (dump x)) true) → "hello world"
```

The second syntax can be useful when interfacing with C-code generating newLISP expressions.

copy-file

syntax: (copy-file *str-from-name str-to-name*)

Copies a file from a path-filename given in *str-from-name* to a path-filename given in *str-to-name*. Returns `true` if the copy was successful or `nil`, if the copy was unsuccessful.

```
(copy-file "/home/me/newlisp/data.lsp" "/tmp/data.lsp")
```

corr

syntax: (corr *list-vector-X list-vector-Y*)

Calculates the *Pearson* product-moment correlation coefficient as a measure of the linear relationship between the two variables in *list-vector-X* and *list-vector-Y*. Both lists must be of same length.

`corr` returns a list containing the following values:

name	description
r	Correlation coefficient
b0	Regression coefficient offset
b1	Regression coefficient slope
t	t - statistic for significance testing
df	Degrees of freedom for t
p	Two tailed probability of t under the null hypothesis

```
(set 'study-time '(90 100 130 150 180 200 220 300 350 400))
(set 'test-errors '(25 28 20 20 15 12 13 10 8 6))

(corr study-time test-errors) → (-0.926 29.241 -0.064 -6.944 8 0.0001190)
```

The negative correlation of `-0.926` between study time and test errors is highly significant with a two-tailed `p` of about `0.0001` under the null hypothesis.

The regression coefficients `b0 = 29.241` and `b1 = -0.064` can be used to estimate values of the Y variable (test errors) from values in X (study time) using the equation ***Y = b0 + b1 * X***.

cos

syntax: (cos *num-radians*)

Calculates the cosine of *num-radians* and returns the result.

```
(cos 1)                     → 0.5403023059
(set 'pi (mul 2 (acos 0)))  → 3.141592654
(cos pi)                    → -1
```

cosh

syntax: (cosh *num-radians*)

Calculates the hyperbolic cosine of *num-radians*. The hyperbolic cosine is defined mathematically as: *(exp (x) + exp (-x)) / 2*. An overflow to `inf` may occur if *num-radians* is too large.

```
(cosh 1)     → 1.543080635
(cosh 10)    → 11013.23292
(cosh 1000)  → inf
(= (cosh 1) (div (add (exp 1) (exp -1)) 2))  → true
```

count

syntax: (count *list-1 list-2*)

Counts elements of *list-1* in *list-2* and returns a list of those counts.

```
(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 newLISP 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.

```
(set 's "0123456789")

(cpymem "xxx" (+ (address s) 5) 3)

s  → "01234xxx89")
```

The 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 (Intel CPU) 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

 ; for Win32/stdcall change last line
 C2 08 00 ret
```

The binary representation is attached to a new function created in newLISP:

```
; set up 32-bit version of machine code
(set 'foo-code (append
     (pack "bbbbbbbbbb" 0x55 0x8B 0xEC 0x8B 0x45 0x08 0x03 0x45 0x0C 0x5D)
     (if (= ostype "Win32") (pack "bbb" 0xC2 0x08 0x00) (pack "b" 0xC3))))

; put a function cell template into foo, protect symbol from deletion
(constant 'foo print)

; put the correct type, either 'stdcall' or 'cdecl'
(cpymem (pack "ld" (if (= ostype "Win32") 8456 4360)) (first (dump foo)) 4)

; put the address of foo-code into the new function cell
(cpymem (pack "ld" (address foo-code)) (+ (first (dump foo)) 12) 4)

; take the name address from the foo symbol, copy into function cell
(set 'sym-name (first (unpack "lu" (+ (address 'foo) 8))))
(cpymem (pack "ld" sym-name) (+ (first (dump foo)) 8) 4)

; test the new function
(println "3 * 4 -> " (foo 3 4))
```

The last example will not work on all hardware platforms and OSs.

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.

```
(crc32 "abcdefghijklmnopqrstuvwxyz")  → 1277644989
```

`crc32` is often used to verify data integrity in unsafe data transmissions.

crit-chi2

syntax: (crit-chi2 *num-probability int-df*)

Calculates the critical minimum Chi^2 for a given confidence probability *num-probability* under the null hypothesis and the degrees of freedom in *int-df* for testing the significance of a statistical null hypothesis.

Note that versions prior to 10.2.0 took *(1.0 - p)* for the probability instead of *p*.

```
(crit-chi2 0.01 4)  → 13.27670443
```

See also the inverse function prob-chi2.

crit-f

syntax: (crit-f *num-probability int-df1 int-df2*)

Calculates the critical minimum *F* for a given confidence probability *num-probability* under the null hypothesis and the degrees of freedom given in *int-df1* and *int-df2* for testing the significance of a statistical null hypothesis using the *F-test*.

```
(crit-f 0.05 10 12)  → 2.753386727
```

See also the inverse function prob-f.

crit-t

syntax: (crit-t *num-probability int-df*)

Calculates the critical minimum *Student's t* for a given confidence probability *num-probability* under the null hypothesis and the degrees of freedom in *int-df* for testing the significance of a statistical null hypothesis.

```
(crit-t 0.05 14)  → 1.761310142
```

See also the inverse function prob-t.

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.

```
(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:

```
#!/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`.

curry

syntax: (curry *func exp*)

Transforms *func* from a function *f(x, y)* that takes two arguments into a function *fx(y)* that takes a single argument. `curry` works like a macro in that it does not evaluate its arguments. Instead, they are evaluated during the application of *func*.

```
(set 'f (curry + 10))  → (lambda ($x) (+ 10 $x))

(f 7)  → 17

(filter (curry match '(a *)) '((a 10) (b 5) (a 3) (c 8) (a 9)))
→  ((a 10) (a 3) (a 9))

(clean (curry match '(a *)) '((a 10) (b 5) (a 3) (c 8) (a 9)))
→  ((b 5) (c 8))

(map (curry list 'x) (sequence 1 5))
→  ((x 1) (x 2) (x 3) (x 4) (x 5))
```

`curry` can be used on all functions taking two arguments.

date utf8

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. If *int-secs* is out of range, `nil` is returned.

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. If *int-secs* is out of range or an invalid *str-format* is specified, an empty string `""` is returned.

```
(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"
(date 0 (now 0 -2))     → "Thu Jan  1 00:00:00 1970"  ; Unix epoch
```

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 (UTC; 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:

```
(set-locale "german") → "de_DE"

; on Linux - no leading 0 on day with %-d
(date (date-value) 0 "%A %-d. %B %Y")  → "Montag  7. März 2005"

(set-locale "C")  ; default POSIX

(date (date-value) 0 "%A %B %d %Y")    → "Monday March 07 2005"

; suppressing leading 0 on Win32 using #
(date (date-value) 0 "%a %#d %b %Y")   → "Mon 7 Mar 2005"

(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 manual page for the C function `strftime()` of the individual platform's C library.

format	**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 Unix)

%Z	time zone or name or abbreviation (same as %z on Win32, different on Unix)
%%	a literal '%' character

Leading zeroes in the display of decimal day numbers can be suppressed using `"%-d"` on Linux and FreeBSD and using `"%e"` on OpenBSD, SunOS/Solaris and Mac OS X. On Win32 use `"%#d"`.

See also date-value, date-list, date-parse, time-of-day, time, and now.

date-list

syntax: (date-list *int-seconds* [*int-index*])

Returns a list of year, month, date, hours, minutes, seconds, day of year and day of week from a time value given in seconds after January 1st, 1970 00:00:00. The date and time values aren given as UTC, which may differ from the local timezone.

The week-day value ranges from 1 to 7 for Monday thru Sunday.

```
(date-list 1282479244)      → (2010 8 22 12 14 4 234 1)
(date-list 1282479244 0)    → 2010 ; year
(date-list 1282479244 -2)   → 234  ; day of year

(apply date-value (date-list 1282479244)) → 1282479244

(date-list 0)   → (1970 1 1 0 0 0 1 4) ; Thursday 1st, Jan 1970
```

A second optional *int-index* parameter can be used to return a specific member of the list.

`date-list` is the inverse operation of date-value.

date-parse

syntax: (date-parse *str-date str-format*)

Parses a date from a text string in *str-date* using a format as defined in *str-format*, which uses the same formatting rules found in date. The function `date-parse` returns the number of UTC seconds passed since January 1st, 1970 UTC starting with 0 and up to 2147472000 for a date of January 19th, 2038.

This function is not available on Win32 platforms. The function was named `parse-date` in previous versions. The old form is deprecated.

```
(date-parse "2007.1.3" "%Y.%m.%d")    → 1167782400
(date-parse "January 10, 07" "%B %d, %y")    → 1168387200
```

```
; output of date-parse as input value to date-list produces the same date

(date-list (date-parse "2010.10.18 7:00" "%Y.%m.%d %H:%M"))
→ (2010 10 18 7 0 0 290 1)
```

See the date function for all possible format descriptors.

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 (UTC), not adjusted for the current time zone.

In the second syntax, `date-value` returns the time value in seconds for the current time.

```
(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 function date-list can be used to transform a `date-value` back into a list:

```
(date-list 1014854400)  → (2002 2 28 0 0 0)
(apply date-value (date-list 1014854400)) → 1014854400
```

See also date, date-list, date-parse, 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.

```
;; instead of doing
(trace true)
(my-func a b c)
(trace nil)

;; use debug as a shortcut
(debug (my-func a b c))
```

When in `debug` or trace mode, error messages will be printed. The function causing the

exception will return either `0` or `nil` and processing will continue. This way, variables and the current state of the program can still be inspected while debugging.

See also the trace function.

dec !

syntax: (dec *place* [*num*])

The number in *place* is decremented by `1.0` or the optional number *num* and returned. `dec` performs float arithmetic and converts integer numbers passed into floating point type.

place is either a symbol or a place in a list structure holding a number, or a number returned by an expression.

```
(set x 10)    → 10
(dec x)       → 9
x             → 9
(dec x 0.25)  → 8.75
x             → 8.75
```

If the symbol for *place* contains `nil`, it is treated as if containing `0.0`:

```
z             → nil
(dec z)       → -1

(set z nil)
(dec z 0.01)  → -0.01
```

Places in a list structure or a number returned by another expression can be updated too:

```
(set 'l '(1 2 3 4))

(dec (l 3) 0.1) → 3.9

(dec (first l)) → 0

l → (0 2 3 3.9)

(dec (+ 3 4)) → 6
```

Use the -- function to decrement in integer mode. Use the inc function to increment numbers floating point mode.

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, which must not be MAIN.

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 in *sym-source* with the same context as *sym-source* will be translated into symbol references of the target context.

`def-new` returns the symbol created:

```
> (set 'foo:var '(foo:x foo:y))
(foo:x foo:y)

> (def-new 'foo:var 'ct:myvar)
ct:myvar

> ct:myvar
(ct:x ct:y)

> (context 'K)

K> (def-new 'foo:var)
var

K> var
(x y)
```

The following example shows how a statically scoped function can be created by moving it its own namespace:

```
> (set 'temp (lambda (x) (+ x x)))
(lambda (x) (+ x x))
> (def-new 'temp 'double:double)
double:double
> (double 10)
20
> double:double
(lambda (double:x) (+ double:x double:x))
```

The following definition of `def-static` can be used to create functions living in their own lexically protected name-space:

```
(define (def-static s body)
      (def-new 'body (sym s s)))

(def-static 'acc (lambda (x)
          (inc sum x)))

> (acc 1)
1
> (acc 1)
2
> (acc 8)
10
>
```

The function `def-new` can also be used to configure contexts or context objects in a more

granular fashion than is possible with new, which copies a whole context.

default

syntax: (default *context*)

Return the contents of the default functor in *context*.

```
(define Foo:Foo 123)

(default Foo) → 123

(setf (default Foo) 456)
(set 'ctx Foo)

(default ctx) → 456
Foo:Foo       → 456
```

In many situations newLISP defaults automatically to the default functor when seeing a context name. In circumstances where this is not the case, the `default` function can be used.

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.

```
(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 (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.

```
(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 exp-default*) ...]) *body*)

Functions defined using `define-macro` are called *fexpr* in other LISPs as they don't do variable expansion. In newLISP they are still called macros, because they are written with the same purpose of creating special syntax forms with non-standard evaluation patterns of arguments. Functions created using `define-macro` can be combined with template expansion using expand or letex.

Since v.10.5.8, newLISP also has expansion macros using macro.

Defines the new fexpr *sym-name*, with optional arguments *sym-param-1*. `define-macro` is equivalent to assigning a lambda-macro expression to a symbol. When a `define-macro` function is called, unevaluated arguments are assigned to the variables in *sym-param-1 ...*. Then the *body* expressions are evaluated. When evaluating the `define-macro` function, the lambda-macro expression is returned.

```
(define-macro (my-setq p1 p2) (set p1 (eval p2)))
→ (lambda-macro (p1 p2) (set p1 (eval p2)))

(my-setq x 123)  → 123
x                → 123
```

New functions can be created to behave like built-in functions that delay the evaluation of certain arguments. Because fexprs can access the arguments inside a parameter list, they can be used to create flow-control functions like those already built-in to 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 *fexprs*, the danger exists of passing a parameter with the same variable name as used in the `define-macro` definition. In this case, the *fexpr's* 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*, by writing *hygienic* `define-macro`s:

- Put the definition 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*. This is the preferred method in newLISP to write `define-macro`s.

- Use args to access arguments passed by the function.

```
;; a define-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
x                → 123
```

The definition in the example is lexically isolated, and no variable capture can occur. Instead of the function being called using `(my-setq:my-setq …)`, it can be called with just `(my-setq …)` because it is a *default function*.

The second possibility is to refer to passed parameters using args:

```
;; avoid variable capture in macros using the args function

(define-macro (my-setq) (set (args 0) (eval (args 1))))
```

See also the macro expansion function not susceptible to variable capture.

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`, symbols are only deleted when they are not referenced.

When the expression in *bool* evaluates to `nil`, symbols will be deleted without any reference checking. Note that this mode should only be used, if no references to the symbol exist outside it's namespace. If external references exist, this mode can lead to system crashes, as the external reference is not set to `nil` when using this mode. This mode can be used to delete namespace hashes and to delete namespaces in object systems, where variables are strictly treated as private.

Protected symbols of built-in functions and special symbols like `nil` and `true` cannot be deleted.

`delete` returns `true` if the symbol was deleted successfully or `nil` if the symbol was not deleted.

When deleting a context symbol, the first `delete` removes the context namespace contents

and demotes the context symbol to a normal mono-variable symbol. A second `delete` will remove the symbol from the symbol table.

```
(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)

;; delete all symbols in a context
(set 'foo:x 123)
(set 'foo:y "hello")

(delete 'foo)  → foo:x, foo:y deleted
```

In the last example only the symbols inside context `foo` will be deleted but not the context symbol `foo` itself. It will be converted to a normal unprotected symbol and contain `nil`.

Note that deleting a symbol that is part of an expression 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` if the file was deleted successfully.

On failure the function returns `nil`. For error information, use sys-error when used on files. When used on URLs net-error gives more error information.

The file name can be given as a URL.

```
(delete-file "junk")

(delete-file "http://asite.com/example.html")

(delete-file "file://aFile.txt")
```

The first example deletes the file `junk` in the current directory. The second example shows how to use a URL to specify the file. In this form, additional parameters can be given. See delete-url for details.

delete-url

syntax: (delete-file *str-url*)

This function deletes the file on a remote HTTP server specified in *str-url*. The HTTP DELETE protocol must be enabled on the target web server, or an error message string may be returned. The target file must also have access permissions set accordingly. Additional parameters such as timeout and custom headers are available exactly as in the get-url function.

If *str-url* starts with `file://` a file on the local file system is deleted.

This feature is also available when the delete-file function is used and a URL is specified for the filename.

```
(delete-url "http://www.aserver.com/somefile.txt")
(delete-url "http://site.org:8080/page.html" 5000)

; delete on the local file system
(delete-url "file:///home/joe/somefile.txt")
```

The second example configures a timeout option of five seconds. Other options such as special HTTP protocol headers can be specified, as well. See the get-url function for details.

destroy

syntax: (destroy *int-pid*)
syntax: (destroy *int-pid int-signal*)

Destroys a process with process id in *int-pid* and returns `true` on success or `nil` on failure. The process id is normally obtained from a previous call to fork on Mac OS X and other Unix or process on all platforms. On Unix, `destroy` works like the system utility *kill* using the SIGKILL signal.

CAUTION! If *int-pid* is `0` the signal is sent to all processes whose group ID is equal to the process group ID of the sender. If *int-pid* is `-1` all processes with the current user id will be killed, if newLISP is started with super user privileges, all processes except system processes are destroyed.

When specifying *int-signal*, `destroy` works like a Unix `kill` command sending the specified Unix signal to the process in *int-pid*. This second syntax is not available on Win32.

```
(set 'pid (process "/usr/bin/bc" bcin bcout))
(destroy pid)

(set 'pid (fork (dotimes (i 1000) (println i) (sleep 10))))
(sleep 100) (destroy pid)
```

det

syntax: (det *matrix* [*float-pivot*])

Returns the determinant of a square matrix. A matrix can either be a nested list or an array.

Optionally `0.0` or a very small value can be specified in *float-pivot*. This value substitutes pivot elements in the LU-decomposition algorithm, which result in zero when the algorithm deals with a singular matrix.

```
(set 'A '((-1 1 1) (1 4 -5) (1 -2 0)))
(det A)  → -1

; treatment of singular matrices
(det '((2 -1) (4 -2)))        → nil
(det '((2 -1) (4 -2)) 0)      → -0
(det '((2 -1) (4 -2)) 1e-20)  → -4e-20
```

If the matrix is singular and *float-pivot* is not specified, `nil` is returned.

See also the other matrix operations invert, mat, multiply and transpose.

device

syntax: (device [*int-io-handle*])

int-io-handle is an I/O device number, which is set to 0 (zero) for the default STD I/O pair of handles, 0 for *stdin* and 1 for *stdout*. *int-io-handle* may also be a file handle previously obtained using open. In this case both, input and output are channeled through this handle. 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, println, write, write-line and read-char, read-line. When the current I/O device is 0 or 1, 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.

Note, that on Unix like operating systems, stdin channel 0 can also be used for output and stdout channel 1 can also be used for reading input. This is not the case on Windows, where 0 is strictly for input and stdout 1 strictly for output.

```
(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.

```
(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.

```
(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, unique and union.

directory

syntax: (directory [*str-path*])
syntax: (directory *str-path str-pattern* [*regex-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.

```
(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.

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 *regex-options*, special regular expression options can be specified; see regex for details.

```
(directory "." "\\.c")  → ("foo.c" "bar.c")
;; or using braces as string pattern delimiters
(directory "." {\.c})  → ("foo.c" "bar.c")

; show only hidden files (starting with dot)
(directory "." "^[.]")   → ("." ".." ".profile" ".rnd" ".ssh")
```

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
```

display-html JS

syntax: (display-html *str-html*)
syntax: (display-html *str-html bool-flag*)

Using the first syntax, the function replaces the current page in the browser with the HTML page found in *str-html*.

If *bool-flag* evaluates to `true`, the page gets opened in a new browser tab and the current page is not affected.

This function is only available on newLISP compiled to JavaScript.

```
(set 'page [text]
<html>
<head>
<title>Hello App</title>
</head>
<body>
<h2>Hello World</h2>
</body>
</html>
[/text])

; open the page in a new browser tab
(display-html page true) → "92"
```

The function returns the length of the HTML document displayed as a string.

See also the function eval-string-js for evaluation of JavaScript in the current page.

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`.

```
(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*.

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. If *body* is empty, the last result of *exp-condition* is returned.

`do-until` also updates the system iterator symbol `$idx`.

```
(set 'x 1)
(do-until (> x 0) (inc x))
x  → 2

(set 'x 1)
(until (> x 0) (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.

`do-while` also updates the system iterator symbol `$idx`.

```
(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.

doargs

syntax: (doargs (*sym* [*exp-break*]) *body*)

Iterates through all members of the argument list inside a user-defined function or macro. This function or macro can be defined using define, define-macro, lambda, or lambda-macro. The variable in *sym* is set sequentially to all members in the argument list until the list is exhausted or an optional break expression (defined in *exp-break*) evaluates to `true` or a logical true value. The `doargs` expression always returns the result of the last evaluation.

`doargs` also updates the system iterator symbol `$idx`.

```
(define (foo)
    (doargs (i) (println i)))

> (foo 1 2 3 4)
1
2
3
4
```

The optional break expression causes `doargs` to interrupt processing of the arguments:

```
(define-macro (foo)
    (doargs (i (= i 'x))
        (println i)))

> (foo a b x c d e)
a
b
true
```

Use the args function to access the entire argument list at once.

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*.

```
(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 dolist
;; 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.

dostring [utf8]

syntax: (dostring (*sym string* [*exp-break*]) *body*)

The expressions in *body* are evaluated for each character in *string*. The variable in *sym* is set to each ASCII or UTF-8 integer value of the characters 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*.

```
; ASCII example
(set 'str "abcdefg")
(dostring (c str) (println c " - " (char c)))

97 - a
98 - b
99 - c
100 - d
101 - e
102 - f
103 - g

; UTF8 example
(set 'utf8str "我能吞下玻璃而不伤身体。")
(dostring (c utf8str) (println c " - " (char c)))

25105 - 我
33021 - 能
21534 - 吞
 ...
20307 - 体
12290 - 。
```

This example prints the value of each character in the console window. In UTF-8 enabled versions of newLISP, individual characters may be longer than one byte and the number in the loop variable may exceed 255. The return value of `dostring` is the result of the last evaluated expression.

The internal system variable `$idx` keeps track of the current offset into the string passed to `dostring`, and it can be accessed during its execution.

dotimes

syntax: (dotimes (*sym-var int-count* [*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*. After evaluation of the `dotimes` statement *sym* assumes its previous value.

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*.

```
(dotimes (x 10)
  (print x))  → 9  ; return value
```

This prints `0123456789` to the console window.

dotree

syntax: (dotree (*sym sym-context* [*bool*]) *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.

When the optional *bool* expression evaluates to not `nil`, only symbols starting with an underscore character _ are accessed. Symbol names starting with an _ underscore are used for hash keys and symbols created by bayes-train.

`dotree` also updates the system iterator symbol `$idx`.

```
;; faster and less memory overhead
(dotree (s SomeCTX) (print s " "))

;; slower and higher memory usage
(dolist (s (symbols SomeCTX)) (print s " "))
```

This example prints the names of all symbols inside SomeCTX to the console window.

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.

```
(dump 'a)   → (9586996 5 9578692 9578692 9759280)

(dump 999)  → (9586996 130 9578692 9578692 999)
```

The list contains the following memory addresses and information:

offset	description
0	memory address of the newLISP 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*])
syntax: (dup *exp*)

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*.

Without the repetition parameter, `dup` assumes 2.

```
(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"

(dup "*")         → "**"
```

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.

```
(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 (OTP) 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:

```
(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* [*num-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.

If a regular expression *option* number is specified, *str-key* contains a regular expression pattern. See regex for valid numbers for *option*.

```
(ends-with "newLISP" "LISP")         → true
(ends-with "newLISP" "lisp")         → nil
;; 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.

```
(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 an association list in which each entry is a key-value pair of environment variable and value.

```
(env)
→ (("PATH" "/bin:/usr/bin:/sbin") ("TERM" "xterm-color") ... ))
```

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.

```
(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. If *value-str* is the empty string `""`, then the variable is completely removed from the environment except when running on Solaris, where the variable stays with an empty string.

```
(env "NEWLISPBIN" "/usr/bin/")  → true
(env "NEWLISPBIN")              → "/usr/bin/"
(env "NEWLISPBIN" "")           → true
(env "NEWLISPBIN")              → nil
```

erf

syntax: (erf *num*)

`erf` calculates the error function of a number in *num*. The error function is defined as:

erf (x) = 2/sqrt(pi) * integral from 0 to x of exp(-t^2) dt

```
(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-event-handler | func-event-handler*)

sym-event-handler 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 last-error to retrieve the number and text of the error. The event handler is specified as either a quoted symbol or a lambda function.

```
(define (my-handler)
  (print "error # " (first (last-error)) " has occurred\n") )

(error-event 'my-handler)  → my-handler

;; specify a function directly

(error-event my-handler)  → $error-event

(error-event
  (fn () (print "error # " (first (last-error)) " 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.

eval

syntax: (eval *exp*)

eval evaluates the result of evaluating *exp* in the current variable environment.

```
(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
```

As usual, evaluation of variables happens in the current variable environment:

```
; eval in global (top level) environment
(set 'x 3 'y 4)
(eval '(+ x y))          → 7
```

```
; eval in local environment
(let ( (x 33) (y 44) )
    (eval '(+ x y)))     → 77

; old environment after leaving local let environment
(eval '(+ x y))          → 7
```

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.

There is a safer method to pass arguments by reference in newLISP by enclosing the data inside context objects. See the chapter Passing data by reference. Passing references into user defined function using namespace ids avoids *variable capture* of the passed symbol, in case the symbol passed is the same used as a parameter in the function.

eval-string

syntax: (eval-string *str-source* [*sym-context* [*exp-error* [*int-offset*]]])

The string in *str-source* is compiled into newLISP's internal format and then evaluated. The evaluation result is returned. If the string contains more than one expression, the result of the last evaluation is returned.

An optional second argument can be used to specify the context to which the string should be parsed and translated.

If an error occurs while parsing and evaluating *str-source* then *exp-error* will be evaluated and the result returned.

int-offset specifies an optional offset into *str-source*, where to start evaluation.

```
(eval-string "(+ 3 4)")  → 7
(set 'X 123)             → 123
(eval-string "X")        → 123

(define (repl) ; read print eval loop
  (while true
    (println "=> " (eval-string (read-line) MAIN (last-error)))
  )
)

(set 'a 10)
(set 'b 20)
(set 'foo:a 11)
```

```
(set 'foo:b 22)

(eval-string "(+ a b)")       → 30
(eval-string "(+ a b)" 'foo)  → 33
```

The second example shows a simple newLISP interpreter eval loop.

The last example shows how to specify a target context for translation. The symbols `a` and `b` now refer to symbols and their values in context `foo` instead of `MAIN`.

See also the function read-expr which translates a string without evaluating it.

eval-string-js JS

syntax: (eval-string-js *str-JavaScript-source*)

The function takes a program source in *str-JavaScript-source* and returns the result in a string.

This function is only available on newLISP compiled to JavaScript.

```
(eval-string-js "window.prompt('Enter some text:', '')")

; for single and double quotes inside a string passed to a
; JavaScropt function, single and double quotes must be
; preceded by a backslash \ and the whole string passed
; to eval-string-js limited by [text], [/text] tags.

(eval-string-js [text]alert('A double quote: \" and a single quote: \' ')[/text])

(eval-string-js "6 * 7")
```

The first expression will pop up a dialog box to enter text. The function will return the text string entered. The second expression will return the string `42`.

See also the function display-html for displaying an HTML page in the browser.

even? bigint

syntax: (even? *int-number*)

Checks if an integer number is *even divisible* by `2`, without remainder. When a floating point number is passed for *int-number*, it will be converted to an integer by cutting off its fractional part.

```
(even? 123)  → nil
(even? 8)    → true
(even? 8.7)  → true
```

Use odd? to check if an integer is not divisible by 2.

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 a list of strings (one for each line in standard out (STDOUT)). exec returns nil if the process could not be launched. If the process could be launched but only returns and error and no valid output, the empty list will be returned.

```
(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.

```
(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.

exists

syntax: (exists *func-condition list*)

Successively applies *func-condition* to the elements of *list* and returns the first element that meets the condition in *func-condition*. If no element meets the condition, nil is returned.

```
(exists string? '(2 3 4 6 "hello" 7))       → "hello"

(exists string? '(3 4 2 -7 3 0))            → nil

(exists zero? '(3 4 2 -7 3 0))              → 0 ; check for 0 or 0.0

(exists < '(3 4 2 -7 3 0))                  → -7 ; check for negative

(exists (fn (x) (> x 3)) '(3 4 2 -7 3 0))   → 4

(exists (fn (x) (= x 10)) '(3 4 2 -7 3 0))  → nil
```

If *func-condition* is nil?, the result nil is ambiguous. In this case index or find are the better method when looking for nil.

Use the for-all function to check if a condition is met for all elements in a list.

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.

```
(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.

```
(exp 1)        → 2.718281828
(exp (log 1))  → 1
```

expand

syntax: (expand *exp sym-1* [*sym-2 ...*])
syntax: (expand *exp list-assoc* [*bool*])
syntax: (expand *exp*)

In the first syntax, one symbol in *sym* (or more in *sym-2* through *sym-n*) is looked up in a simple or nested expression *exp*. They are then expanded to the current binding of the symbol and the expanded expression is returned. The original list remains unchanged.

```
(set 'x 2 'a '(d e))
(set 'foo 'a)
(expand foo '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 and doing variable expansion as in rewrite 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* [*bool*])

The second syntax of expand allows expansion bindings to be specified on the fly, without performing a set on the participating variables:

If the *bool* evaluates to true, the value parts in the association list are evaluated.

```
(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))
(expand '(a b) '((a (+ 1 2)) (b (+ 3 4))))     → ((+ 1 2) (+ 3 4))
(expand '(a b) '((a (+ 1 2)) (b (+ 3 4))) true) → (3 7)
```

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:

```
(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 because 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 utf8

syntax: (explode *str* [*int-chunk* [*bool*]])
syntax: (explode *list* [*int-chunk* [*bool*]])

In the first syntax, `explode` transforms the string (*str*) into a list of single-character strings. Optionally, a chunk size can be specified in *int-chunk* to break the string into multi-character chunks. When specifying a value for *bool* other than `nil`, the last chunk will be omitted if it does not have the full length specified in *int-chunk*.

```
(explode "newLISP")  → ("n" "e" "w" "L" "I" "S" "P")

(join (explode "keep it together"))  → "keep it together"

(explode "newLISP" 2)    → ("ne" "wL" "IS" "P")

(explode "newLISP" 3)    → ("new" "LIS" "P")

; omit last chunk if too short
(explode "newLISP" 3 true)    → ("new" "LIS")
```

Only on non UTF8– enabled versions, `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. Processing will stop when a zero byte character is found.

To explode binary contents on UTF-8–enabled versions of newLISP use unpack as shown in the following example:

```
(set 'str "\001\002\003\004") → "\001\002\003\004"

(unpack (dup "c" (length str)) str) → (1 2 3 4)
(unpack (dup "s" (length str)) str) → ("\001" "\002" "\003" "\004")
```

In the second syntax, `explode` explodes a list (*list*) into sublists of chunk size *int-chunk*, which is 1 (one) by default.

The following shows an example of the last chunk being omitted when the value for *bool* is other than `nil`, and the chunk does not have the full length specified in *int-chunk*.

```
(explode '(a b c d e f g h))    → ((a) (b) (c) (d) (e) (f) (g) (h))
(explode '(a b c d e f g) 2)  → ((a b) (c d) (e f) (g))

; omit last chunk if too short
(explode '(a b c d e f g) 2 true)  → ((a b) (c d) (e f))

(transpose (explode '(a b c d e f g h) 2))
→ ((a c e g) (b d f h))
```

The join and append functions are inverse operations of `explode`.

extend !

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

The list in *list-1* is extended by appending *list-2*. More than one list may be appended.

The string in *string-1* is extended by appending *string-2*. More than one string may be appended. The string can contain binary `0` (zero) characters.

The first parameter can be an un-initialized variable.

The extended list or string is returned.

```
; extending lists

(extend lst '(a b) '(c d)) → (a b c d)
(extend lst '(e f g)) → (a b c d e f)
lst → (a b c d e f g)

; extending strings

(extend str "ab" "cd") → "abcd"
(extend str "efg") → "abcdefg"
str → "abcdefg"

; extending in place

(set 'L '(a b "CD" (e f)))
(extend (L 2) "E")
L → (a b "CDE" (e f))

(extend (L 3) '(g))
L → (a b "CDE" (e f g))
```

For a non-destructive list or string extension see append.

factor

syntax: (factor *int*)

Factors the number in *int* into its prime components. When floating point numbers are passed, they are truncated to their integer part first.

```
(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)
  (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.

```
(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
;; plain numbers and 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* [*int-index* [*bool-flag*]])

Returns a list of information about the file or directory in *str_name*. The optional index specifies the list member to return. When no *bool-flag* is specified or when *bool-flag* evaluates to `nil` information about the link is returned if the file is a link to an original file. If *bool-flag* evaluates to anything else than `nil`, information about the original file referenced by the link is returned.

offset	contents
0	size
1	mode (differs with `true` flag)
2	device mode
3	user ID
4	group ID
5	access time
6	modification time
7	status change time

Depending on *bool-flag* set, the function reports on either the link (no flag or `nil` flag) or on the original linked file (`true` flag).

```
(file-info ".bashrc")
→ (124 33188 0 500 0 920951022 920951022 920953074)

(file-info ".bashrc" 0)  → 124

(date (file-info "/etc" -1))  → "Mon Mar 8 18:23:17 2005"
```

In the second example, the last status change date for the directory */etc* is retrieved.

`file-info` gives file statistics (size) for a linked file, not the link, except for the *mode* field.

file?

syntax: (file? *str-path-name* [*bool*])

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. If the optional *bool* value is `true`, the file must not be a directory and *str-path-name* is returned or `nil` if the file is a directory. The existence of a file does not imply anything about its read or write permissions for the current user.

```
(if (file? "afile") (set 'fileNo (open "afile" "read")))
```

```
(file? "/usr/bin/newlisp" true) → "/usr/bin/newlisp"
(file? "/usr/bin/foo" true)     → nil
```

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.

```
(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)

; filter with comparison functor
(set 'L '((a 10 2 7) (b 5) (a 8 3) (c 8) (a 9)))

(filter (curry match '(a *)) L)   → ((a 10 2 7) (a 8 3) (a 9))

(filter (curry match '(? ?)) L)   → ((b 5) (c 8) (a 9))

(filter (curry match '(* 8 *)) L) → ((a 8 3) (c 8))
```

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* [*func-compare* | *regex-option*])
syntax: (find *str-key str-data* [*regex-option* [*int-offset*]])

Find an expression in a list

If the second argument evaluates to a *list*, then `find` returns the index position (offset) of the element derived from evaluating *exp-key*.

Optionally, an operator or user-defined function can be specified in *func-compare*. If the *exp-key* is a string, a regular expression option can be specified with the *regex-option* parameter.

When using regular expressions or comparison functors the system variable `$0` is set to the last element found.

```
; find an expression in a list
(find '(1 2) '((1 4) 5 6 (1 2) (8 9)))  → 3

(find "world" '("hello" "world"))       → 1
(find "hi" '("hello" "world"))          → nil

(find "newlisp" '("Perl" "Python" "newLISP") 1)  → 2
; same with string option
(find "newlisp" '("Perl" "Python" "newLISP") "i")  → 2

; use the comparison functor
(find 3 '(8 4 3  7 2 6) >)  → 4
$0 → 2

(find "newlisp" '("Perl" "Python" "newLISP")
                (fn (x y) (regex x y 1))) → 2
$0 → "newLISP"

(find 5 '((l 3) (k 5) (a 10) (z 22))
         (fn (x y) (= x (last y))))  → 1
$0 → (k 5)

(find '(a ?) '((l 3) (k 5) (a 10) (z 22)) match)  → 2
$0 → (a 10)

(find '(X X) '((a b) (c d) (e e) (f g)) unify)  → 2
$0 → (e e)

; define the comparison functor first for better readability
(define (has-it-as-last x y) (= x (last y)))

(find 22 '((l 3) (k 5) (a 10) (z 22)) has-it-as-last)  → 3
$0 → (z 22)
```

Using match and unify, list searches can be formulated which are as powerful as regular expression searches are for strings.

Find a string in a string

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 offset position returned is always based on counting single byte characters even when running the UTF-8 enabled version of newLISP.

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 *regex-option* (i.e., 1 (one) for case-insensitive search or `0` (zero) for no special options). If *regex-option* is specified an optional *int-offset* argument can be specified too to start the search not at the beginning but at the offset given. In any case the position returned by `find` is calculated relative to the beginning of the string.

To specify *int-offset* in a simple string search without regular expressions, specify `nil` for *regex-option*.

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.

```
; simple string search
(find "world" "Hello world")  → 6
(find "WORLD" "Hello woRLd")  → nil

; case-insensitive regex

(find "WorlD" "Hello woRLd" 1)  → 6
; or
(find "WorlD" "Hello woRLd" "i")  → 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"

; use an optional offset
(find "cat|dog" "I have a cat and a dog" 0)    → 9
(find "cat|dog" "I have a cat and a dog" 0 12) → 19

;; 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: (find-all *str-regex-pattern str-text* [*exp* [*regex-option*]])
syntax: (find-all *list-match-pattern list* [*exp*])
syntax: (find-all *exp-key list* [*exp* [*func-compare*]])

In the first syntax, `find-all` finds all occurrences of *str-regex-pattern* in the text *str-text*, returning a list containing all matching strings. The empty list `()` is returned if no matches are found. In the first syntax string searches are always done using regular expression patterns, even if no *regex-option* is specified. The system variable `$count` is updated with the number of matches 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, *regex-option*, specifies special regular expression options (see regex for further details).

```
(find-all {\d+} "lkjhkljh34ghfdhgfd678gfdhfgd9")
→ ("34" "678" "9")

$count → 3

(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")

; use $count in evaluated expr
(find-all "a" "ababab" (string $count $it)) → ("1a" "2a" "3a")
```

The first example discovers all numbers in a text. The second example shows how an optional expression in *exp* 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` with strings always performs a regular expression search, even if the option in *regex-option* is omitted.

In the second syntax, `find-all` searches for all list match patterns *list-match-pattern* in *list*. As in `find-all` for strings, an expression can be specified in *exp* to process further the matched sublist found in *list*. The system variable `$count` is updated with the number of matches found.

```
(find-all '(? 2) '((a 1) (b 2) (a 2) (c 4))) → ((b 2) (a 2))

(find-all '(? 2) '((a 1) (b 2) (a 2) (c 4)) (first $it)) → (b a)

$count → 2
```

`find-all` for list matches always uses match to compare when searching for sublists and always needs a list for the pattern expression.

In the third syntax, `find-all` can specify a built-in or user-defined function used for

comparing list elements with the key expression in *exp-key*:

```
(find-all 5 '(2 7 4 5 9 2 4 9 7 4 8) $it <) → (7 9 9 7 8)

; process the found element available in $it

(find-all 5 '(2 7 4 5 9 2 4 9 7 4 8) (* 3 $it) <) → (21 27 27 21 24)
; same as
(find-all 5 '(2 7 4 5 9 2 4 9 7 4 8) (* 3 $it) (fn (x y) (< x y))) → (21 27 27 21 24)


(find-all 5 '(2 7 4 5 9 2 4 9 7 4 8) ("abcdefghijk" $it) <) → ("h" "j" "j" "h" "i")

$count → 5

; use $count
(find-all 'a '(a b a b a b) (list $count $it)) → ((1 a) (2 a) (3 a))
```

Any type of expression can be searched for or can be contained in the list. `find-all` in this syntax works similar to filter but with the added benefit of being able to define a processing expression for the found element.

If no *func-compare* is defined and *exp-key* is a list, then match will be used for comparison, as in the second syntax.

first utf8

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.

```
(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)

(first '())                → ERR: list is empty
```

In the third syntax, the first character is returned from the string in *str* as a string.

```
(first "newLISP")         → "n"
(first (rest "newLISP"))  → "e"
```

Note that first works on character boundaries rather than byte boundaries when the UTF-8–enabled version of newLISP is used. See also the functions last and rest.

flat

syntax: (flat *list* [*int-level*])

Returns a flattened list from a list:

```
(set 'lst '(a (b (c d))))
(flat lst)  → (a b c d)

; extract a list of index vectors of all elements

(map (fn (x) (ref x lst)) (flat lst))
→ ((0) (1 0) (1 1 0) (1 1 1))
```

The optional *int-level* parameter can be used to limit the recursion level when flattening the list:

```
(flat '(a b (c d (e f)) (g h (i j))) )   → (a b c d e f g h i j)

(flat '(a b (c d (e f)) (g h (i j))) 1)  → (a b c d (e f) g h (i j))

(flat '(a b (c d (e f)) (g h (i j))) 2)  → (a b c d e f g h i j)
```

If *int-level* is `0`, no flattening will occur.

`flat` can be used to iterate through nested lists.

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 *exp* 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.

```
(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.

```
(set 'num 1.23)
(float? num)  → true
```

floor

syntax: (floor *number*)

Returns the next lowest integer below *number* as a floating point.

```
(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.

```
(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.

fn

syntax: (fn (*list-parameters*) *exp-body*)

`fn` or `lambda` are used to define anonymous functions, which are frequently used in map, sort, and all other expressions where functions can be used as arguments. The `fn` or `lambda` word does not exist on its own as a symbol, but indicates a special list type: the *lambda list*. Together with `fn-macro` and `lambda-macro` these terms are recognized during source parsing. They indicate a specialized type of list which can be used and applied like a function or operator.

Using an anonymous function eliminates the need to define a new function with define. Instead, a function is defined on the fly:

```
(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 of writing) can be used in place of `fn`.

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` 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. After evaluation of the `for` statement *sym* assumes its previous value.

```
> (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.

for-all

syntax: (for-all *func-condition list*)

Applies the function in *func-condition* to all elements in *list*. If all elements meet the condition in *func-condition*, the result is `true`; otherwise, `nil` is returned.

```
(for-all number? '(2 3 4 6 7))                → true

(for-all number? '(2 3 4 6 "hello" 7))        → nil

(for-all (fn (x) (= x 10)) '(10 10 10 10 10))  → true
```

Use the exists function to check if at least one element in a list meets a condition.

fork

syntax: (fork *exp*)

The expression in *exp* is launched as a newLISP child process-thread of the platforms OS. The new process inherits the entire address space, but runs independently so symbol or variable contents changed in the child process 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.

A much simpler automated method to launch processes and collect results is available with spawn and the Cilk API.

```
> (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 processes.

```
#!/usr/bin/newlisp

(define (count-down-proc x channel)
  (while (!= x 0)
      (write-line channel (string x))
      (dec x)))

(define (observer-proc channel)
  (do-until (= i "1")
    (println "process " (setq i (read-line channel)))))

(map set '(in out) (pipe))
(set 'observer (fork (observer-proc in)))
(set 'counter (fork (count-down-proc 5 out)))

; avoid zombies
(wait-pid observer)
(wait-pid counter)

(exit)
```

The following output is generated by observer-proc

```
process 5
process 4
process 3
process 2
process 1
```

The `count-down-proc` writes numbers to the communication pipe, where they are picked up by the `observer-process` and displayed.

A forked process can either exit by itself or it can be destroyed using the destroy function.

```
(define (fork-destroy-demo)
    (set 'pid (fork (dotimes (i 1000) (println i) (sleep 10))))
    (sleep 50)
    (destroy pid)
)

> (fork-destroy-demo)
0
1
2
3
4
true
>
```

The process started by `fork-destroy-demo` will not finish but is destroyed 50 milli-seconds after start by a call to destroy.

Use the semaphore function for synchronizing processes and share for sharing memory between processes.

See spawn for a much simpler and automated way to synchronize processes and collect results.

format

syntax: (format *str-format exp-data-1* [*exp-data-2 ...*])
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: `%%`

On Linux the percent sign can be followed by a single quote `%'` to insert thousand's separators in number formats.

The `w` represents the width field. Data is right-aligned, except when preceded by a minus sign, in which case it is left-aligned. If preceded by a `+` (plus sign), positive numbers are displayed with a `+`. When preceded by a `0` (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. 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`:

format	description
s	text string
c	character (value 1 - 255)
d	decimal (32-bit)
u	unsigned decimal (32-bit)
x	hexadecimal lowercase
X	hexadecimal uppercase
o	octal (32-bits) (not supported on all of newLISP flavors)
f	floating point
e	scientific floating point
E	scientific floating point
g	general floating point

Formatting 64-bit numbers using the 32-bit format specifiers from above table will truncate and format the lower 32 bits of the number on 64-bit systerms and overflow to `0xFFFFFFFF` on 32-bit systems.

For 32-bit and 64-bit numbers use the following format strings. 64-bit numbers will be truncated to 32-bit on 32-bit platforms:

format	description
ld	decimal (32/64-bit)
lu	unsigned decimal (32/64-bit)
lx	hexadecimal (32/64-bit)
lX	hexadecimal uppercase (32/64-bit)

For 64-bit numbers use the following format strings on Unix-like operating systems and on MS Windows (not supported on TRU64):

format	description
lld	decimal (64-bit)
llu	unsigned decimal (64-bit)
llx	hexadecimal (64-bit)
llX	hexadecimal uppercase(64-bit)

On Windows platforms only the following characters apply for 64 bit numbers:

format	description
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`.

```
(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 "%14.2f" 12345678.12)   → "   12345678.12"
; on UNIX glibc compatible platforms only (Linux, MAC OS X 10.9) on some locales
(format "%'14.2f" 12345678.12) → " 12,345,678.12"

(format "%8d" 12345)   → "   12345"
; on UNIX glibc compatible platforms only (Linux, MAC OS X 10.9) on some locales
(format "%'8d" 12345)  → "  12,345"

(format "%-15s" "hello")        → "hello          "
(format "%15s %d" "hello" 123)  → "          hello 123"
(format "%5.2s" "hello")        → "   he"
(format "%-5.2s" "hello")       → "he   "
```

```
(format "%o" 80)    → "120"

(format "%x %X" -1 -1)  → "ffffffff FFFFFFFF"

; 64 bit numbers on Windows
(format "%I64X" 123456789012345678)  → "1B69B4BA630F34E"

; 64 bit numbers on Unix (except TRU64)
(format "%llX" 123456789012345678)   → "1B69B4BA630F34E"

(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) or *int-type* is omitted; for payment at the beginning of each period, *int-type* is 1.

```
(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.

```
(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(Chi²|df) = Q(df/2, Chi²/2) = gammai(df/2, Chi²/2)

See also the prob-chi2 function.

gammaln

syntax: (gammaln *num-x*)

Calculates the log Gamma function of the value *x* in *num-x*.

```
(exp (gammaln 6))  → 120
```

The example uses the equality of *n! = 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:

Beta(z,w) = Exp(Gammaln(z) + Gammaln(w) - Gammaln(z+w))

gcd bigint

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.

```
(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 an 8-bit character from an address specified in *int-address*. This function is useful when using imported shared library functions with import.

```
char * foo(void)
{
char * result;
result = "ABCDEFG";
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 ; ASCII "A"
(print (get-char (+ (foo) 1)))  →  66 ; ASCII "B"
```

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-long, 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.

```
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-long, 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.

```
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-long, get-string, pack, and unpack functions.

get-long

syntax: (get-long *int-address*)

Gets a 64-bit integer from the address specified in *int-address*. This function is handy when using `import` to import shared library functions, a function returning an address pointer to a long integer, or a pointer to a structure containing long integers.

```
long long int * foo(void)
{
int * result;
…
*result = 123;
return(result);
}

long long 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-long` 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-int, get-string, pack, and unpack functions.

get-string

syntax: (get-string *int-address* [*int-bytes* [*str-limit*])

Copies a character string from the address specified in *int-address*. This function is helpful when using imported shared library functions with import and a C-function returns the address to a memory buffer.

```
char * foo(void)
{
char * result;
result = "ABCDEFG";
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)))  → "ABCDEFG"
```

When a string is passed as an argument, `get-string` will take its address as the argument. Without the optional *int-bytes* argument `get-string` breaks off at the first first `\000` (null character) it encounters. This works for retrieving ASCII strings from raw memory addresses:

```
(set 'buff "ABC\000\000\000DEF")  → "ABC\000\000\000DEF"

(length buff)  → 9

(get-string buff)  → "ABC"

(length (get-string buff))  → 3

; get a string from offset into a buffer
```

```
(get-string (+ (address buff) 6)) → "DEF"
```

When specifyung the number of bytes in the optional *int-bytes* parameter, reading does not stpop at the first zero byte found, but copies exactly *int-bytes* number of bytes from the address or string buffer:

```
(set 'buff "ABC\000\000\000DEF")  → "ABC\000\000\000DEF"

; without specifying the number of bytes
; buff is equivalent to (address buff)
(get-string buff)  → "ABC"

; specifying the number of bytes to get
(get-string buff 9) → "ABC\000\000\000DEF"
```

The addtional *str-limit* parameter can be used to limit reading the buffer at a certain string. If *int-bytes* are read before *str-limit* is found, only *int-bytes* are read:

```
(set 'buff "ABC\000\000EFG\000DQW") → "ABC\000\000EFG\000DQW"

; buff is eqivalent to (address buff)
(get-string buff 4 "FG") → "ABC\000"

(get-string buff 10) → "ABC\000\000EFG\000D"

(get-string buff 10 "FG") → "ABC\000\000E"
```

Although UTF-16 and UTF-32 encoding does not specify string termination characters, the sequences "\000\000" and "\000\000\000\000" are used often to terminate UTF-16 and UTF-32 encodings. The additional optional *str-limit* can be used to limit the string when reading from the buffer address:

```
(set 'utf32 (unicode "我能吞下玻璃而不伤身体。"))

(set 'addr (address utf32)) → 140592856255712

; get-string automatically takes the address when a buffer is passed
; utf32 is equivalent to (address utf32) for get-string

(get-string utf32 80 "\000\000\000\000")
→ "\017b\000\000??\000\000\030T\000\000\011N\ 000\000?s\
000\000?t\000\000\f?\000\000\rN\000\000$O\000\000??\000\000SO\000\000\0020\000\000"
```

When using "\000\000" or "\000\000\000\000" as limit strings, the search for these limits is aligned to a 2-byte or 4-byte border.

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. Both `http://` and `file://` URLs are handled. `"header"` can be specified in the optional argument *str-option* to retrieve only the header. The option `"list"` causes header and page information to be returned as separate strings in a list.

A `"debug"` option can be specified either alone or after the `"header"` or `"list"` option separated by one character, i.e. `"header debug"` or `"list debug"`. Including "debug" outputs all outgoing information to the console window.

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` handles redirection if it detects a `Location:` spec in the received header and automatically does a second request. `get-url` also understands the `Transfer-Encoding: chunked` format and will unpack data into an unchunked format.

`get-url` requests are also understood by newLISP server nodes.

```
(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")

(get-url "file:///home/db/data.txt") ; access local file system

(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 shows usage of a `file://` URL to access `/home/db/data.txt` on the local file system.

The third 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.o
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.

```
(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.

global?

syntax: (global? *sym*)

Checks if symbol in *sym* is global. Built-in functions, context symbols, and all symbols made global using the function global are global:

```
global? 'print)   → true
(global 'var)     → var
(global? 'var)    → true

(constant (global 'foo))

(global? 'foo)    → true
```

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 neither `nil` nor 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.

`if` also sets the anaphoric system variable `$it` to the value of the conditional expression in `if`.

```
(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

; more than one statement in the true or false
; part must be blocked with (begin ...)
(if (= x y)
(begin
(some-func x)
(some-func y))
(begin
(do-this x y)
(do-that x y))
)

; if also sets the anaphoric system variable $it
(set 'lst '(A B C))
(if lst (println (last $it)))  → C
```

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.

```
(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 `(>= 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 when and unless functions.

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.

```
(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"])
syntax: (import *str-lib-name str-function-name str-return-type* [*str-param-type . . .*])
syntax: (import *str-lib-name*)

Imports the function specified in *str-function-name* from a shared library named in *str-lib-name*. Depending on the syntax used, string labels for return and parameter types can be specified

If the library in *str-lib-name* is not in the system's library path, the full path name should be

specified.

A function can be imported only once. A repeated import of the same function will simply return the same - already allocated - function address.

Note, that the first simple syntax is available on all versions of newLISP, even those compiled without *libffi* support. On *libffi* enabled versions - capable of the second extended syntax - imported symbols are protected against change and can only be modified using constant.

The third syntax - on OSX, Linux and other Unix only - allows pre-loading libraries without importing functions. This is necessary when other library imports need access internally to other functions from pre-loaded libraries.

Incorrectly using `import` can cause a system bus error or a segfault can occur and crash newLISP or leave it in an unstable state.

The simple `import` syntax

Most library functions can be imported using the simpler first syntax. This form is present on all compile flavors of newLISP. The API expects all function arguments to be passed on the stack in either *cdecl* or *stdcall* conventions. On 32-bit platforms, integers, pointers to strings and buffers sometimes floating point values can be passed as parameters. On 64-bit platforms only integers can be passed but no floating point values. As return values only 32-bit or 64-bit values and pointers are allowed. No floating point numbers can be returned. Strings must be retrieved with the get-string helper function. Regardless of these limitations, most modules included in the distribution use this simple import API.

If pointers are returned to strings or structures the following helper functions can be used extract data: get-char, get-int, get-float, get-string, unpack

To pass pointers for data structures the following functions help to pack data and calculate addresses: address, pack.

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 (pointers in C). Floats can only be used with 32-bit versions of newLISP and libraries. To use floating point numbers in a 64-bit environment use the extended `import` syntax.

```
;; define LIBC platform independent

(define LIBC (lookup ostype '(
("Win32" "msvcrt.dll")
("OSX" "libc.dylib")

(printf "%g %s %d %c\n" 1.23 "hello" 999 65)
1.23 hello 999 A
→ 17 ; return value

;; import Win32 DLLs in Win32 versions

(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.

```
(import "kernel32.dll" "GetWindowsDirectoryA")

;; allocating space for a string return value
(set 'str (dup "\000" 64))  ; reserve space and initialize

(GetWindowsDirectoryA str (length str))

str  → "C:\\WINDOWS\000\000\000 ... "

;; use trim or get-string to cut of trailing binary zeros
(get-string str)  → "C:\\WINDOWS"
(trim str)        → "C:\\WINDOWS"

(import "kernel32.dll" "GetComputerNameA")

;; allocate memory and initialize to zeros
(set 'str (dup "\000" 64))
(set 'len (length str)

;; call the function
;; the length of the string is passed as address reference
;; string str is automatically past by address (C pointer)
(GetComputerNameA str (address len))

str  → "LUTZ-PC\000\000 ... "

(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"
```

The Win32 and Cygwin versions of newLISP uses standard call *stdcall* conventions to call DLL library routines by default. This is necessary for calling DLLs that belong to the Win32 operating system. 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 Mac OS X, 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).

The extended `import` syntax

The extended import API works with the second syntax. It is based on the popular `libffi` library which is pre-installed on most OS platforms. The startup banner of newLISP should show the word `libffi` indicating the running version of newLISP is compiled to use the extended `import` API. The function sys-info can also be used to check for `libffi`-support.

The API works with all atomic C data types for passed parameters and return values. The extended API requires that parameter types are specified in the `import` statement as string type labels. Programs written with extended import API will run without change on 32-bit and 64-bit newLISP and libraries. Integers, floating point values and strings can be returned without using helper functions.

The following types can be specified for the return value in *str-return-type* and for function parameters in *str-param-type*:

label	C type for return value and arguments	newLISP return and argument type
"void"	void	`nil` is returned for return type
"byte"	byte unsigned 8 bit	integer
"char"	char signed 8 bit	integer
"unsigned short int"	unsigned short int 16 bit	integer
"short int"	short int signed 16 bit	integer
"unsigned int"	unsigned int 32 bit	integer
"int"	int signed 32 bit	integer
"long"	long signed 32 or 64 bit depending on platform	integer
"long long"	long long signed 64 bit	integer
"float"	float 32 bit	IEEE-754 64 bit float cut to 32-bit precision
"double"	double 64 bit	IEEE-754 64 bit float
"char*"	char* 32 or 64 bit ptr depending on platform	displayable string return (zero terminated) string buffer arg (no addr. since 10.4.2)
"void*"	void* 32 or 64 bit ptr depending on platform	integer address return either string buffer or integer

address arg

The types `"char*"` and `"void*` can be interchanged and are treated identical inside `libffi`. Depending on the type of arguments passed and the type of return values, one or the other is used.

Aggregate types can be composed using the struct function and can be used for arguments and return values.

The following examples show how the extended `import` syntax can handle return values of floating point values and strings:

```
;; return a float value, LIBC was defined earlier
;             name   return   arg
(import LIBC "atof" "double" "char*")
(atof "3.141") → 3.141

;; return a copied string
;             name     return  arg-1   arg-2
(import LIBC "strcpy" "char*" "char*" "char*")
(set 'from "Hello World")

(set 'to (dup "\000" (length from))) ; reserve memory
(strcpy to from) → "Hello World"
```

The `char*` type takes a string buffer only. The `"void*` type can take either a string buffer or a memory address number as input. When using `"void*"` as a return type the address number of the result buffer will be returned. This is useful when returning pointers to data structures. These pointers can then be used with unpack and struct for destructuring. In the following example the return type is changed to `void*`:

```
(import LIBC "strcpy" "void*" "char*" "char*")
(set 'from "Hello World")
(set 'to (dup "\000" (length from)))

(strcpy to from)       → 2449424
(address to)           → 2449424
(unpack "s11" 2449424) → "Hello World"
(get-string 2449424)   → "Hello World"
to                     → "Hello World"
```

A newLISP string is always passed by it's address reference.

For a more complex example see this OpenGL demo.

Memory management

Any allocation performed by imported foreign functions has to be de-allocated manually if there's no call in the imported API to do so. See the Code Patterns in newLISP document for an example.

In case of calling foreign functions with passing by reference, memory for variables needs to be allocated beforehand by newLISP — see import of `GetWindowsDirectoryA` above — and hence, memory needs not be deallocated manually, because it is managed automatically by newLISP.

inc !

syntax: (inc *place* [*num*])

Increments the number in *place* by `1.0` or by the optional number *num* and returns the result. `inc` performs float arithmetic and converts integer numbers passed into floating point type.

place is either a symbol or a place in a list structure holding a number, or a number returned by an expression.

```
(set 'x 0)    → 0
(inc x)       → 1
x             → 1
(inc x 0.25)  → 1.25
x             → 1.25
(inc x)       → 2.25
```

If a symbol for *place* contains `nil`, it is treated as if containing `0.0`:

```
z             → nil
(inc z)       → 1

(set 'z nil)
(inc z 0.01)  → 0.01
```

Places in a list structure or a number returned by another expression can be updated too:

```
(set 'l '(1 2 3 4))

(inc (l 3) 0.1) → 4.1

(inc (first l)) → 2

l → (2 2 3 4.1)

(inc (+ 3 4)) → 8
```

Use the ++ function for incrementing numbers in integer mode. Use dec to decrement numbers in floating point mode.

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.

```
(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)

(select '(1 10 3 6 4 5 11) '(1 3 6)) → (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.

inf?

syntax: (inf? *float*)

If the value in *float* is infinite the function returns `true` else `nil`.

```
(inf? (div 1 0)) → true

(div 0 0) → NaN
```

Note that an integer division by zero e.g. `(/ 1 0)` will throw an "division by zero" error and not yield infinity. See also NaN? to check if a floating point number is valid.

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 *exp* is invalid, `int` 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 `int`), 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).

```
(int "123")          → 123
(int " 123")         → 123
(int "a123" 0)       → 0
(int (trim " 123"))  → 123
(int "0xFF")         → 255
(int "0b11111")      → 31
(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 "0FF" 0 16)  → 255  ; base 16 conversion

(int 'xyz)     → nil
(int 'xyz 0)   → 0
(int nil 123)  → 123

(int "abc" (throw-error "not a number"))
→ ERR: user error : not a number

(print "Enter a num:")
(set 'num (int (read-line)))

(int (bits 12345) 0 2) → 12345
```

The inverse function to `int` with base `2` is bits.

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`.

```
(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*.

```
(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`.

```
(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, unique and union.

invert

syntax: (invert *matrix* [*float-pivot*])

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.

Optionally `0.0` or a very small value can be specified in *float-pivot*. This value substitutes pivot elements in the LU-decomposition algorithm, which result in zero when the algorithm deals with a singular matrix.

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.

```
(set 'A '((-1 1 1) (1 4 -5) (1 -2 0)))
(invert A)  → ((10 2 9) (5 1 4) (6 1 5))
(invert (invert A)) → ((-1 1 1) (1 4 -5) (1 -2 0))

; solve Ax = b for x
(multiply (invert A) '((1) (2) (3))) → ((41) (19) (23))

; treatment of singular matrices
(invert '((2 -1) (4 -2)))        → nil
(invert '((2 -1) (4 -2)) 0.0)    → ((inf -inf) (inf -inf))
(invert '((2 -1) (4 -2)) 1e-20)  → ((5e+19 -2.5e+19) (1e+20 -5e+19))
```

`invert` will return `nil` if the matrix is *singular* and cannot be inverted, and *float-pivot* is not specified.

All operations shown here on lists can be performed on arrays, as well.

See also the matrix functions det, mat, multiply and transpose.

irr

syntax: (irr *list-amounts* [*list-times* [*num-guess*]])

Calculates 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.

```
(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.

json-error

syntax: (json-error)

When json-parse returns `nil` due to a failed JSON data translation, this function retrieves an error description and the last scan position of the parser.

```
; failed parse returns nil
(json-parse [text]{"address" "http://example.com"}[/text]) → nil

; inspect the error information
(json-error) → ("missing : colon" 11)
```

json-parse

syntax: (json-parse *str-json-data*)

This function parses JSON formatted text and translates it to newLISP S-expressions. All data types conforming to the ECMA-262 standard are translated. The JSON values `false` and `null` will be represented by the symbols `false` and `null` in the symbolic newLISP expressions. Arrays in JSON will be represented by lists in newLISP. The resulting lists from JSON object data can be processed using assoc, lookup and ref.

For JSON attribute values not recognized or wrong JSON syntax, `json-parse` returns `nil` and json-error can be used to retrieve the error text.

The following example shows a nested JSON object from a file `person.json`:

```
{
"name": "John Smith",
"age": 32,
"employed": true,
"address": {
"street": "701 First Ave.",
"city": "Sunnyvale, CA 95125",
"country": "United States"
},
"children": [
{
    "name": "Richard",
    "age": 7
},
{
    "name": "Susan",
    "age": 4
},
{
    "name": "James",
    "age": 3
}
]
}
```

The file is read, parsed and the resulting S-expression stored in `jsp`:

```
(set 'jsp (json-parse (read-file "person.json")))
→
( ("name" "John Smith")
("age" 32)
("employed" true)
("address" ( ("street" "701 First Ave.")
       ("city" "Sunnyvale, CA 95125")
       ("country" "United States")) )
("children" (
(("name" "Richard") ("age" 7))
(("name" "Susan") ("age" 4))
(("name" "James") ("age" 3))) )
)
```

Data can be extracted using assoc, lookup or ref:

```
; the address
(lookup "address" jsp)
→ (("street" "701 First Ave.") ("city" "Sunnyvale, CA 95125") ("country" "United States"))

; the city of the address
(lookup "city" (lookup "address" jsp))
→ "Sunnyvale, CA 95125"

; a child named Susan
(ref '(( * "Susan") *) jsp match true)
→ (("name" "Susan") ("age" 4))

; all names
(map last (ref-all '("name" *) jsp match true))
→ ("John Smith" "Richard" "Susan" "James")

; only names of children
(map last (ref-all '("name" *) (lookup "children" jsp) match true))
→
("Richard" "Susan" "James")

; names of children other method
(map last (map first (lookup "children" jsp)))
→
("Richard" "Susan" "James")
```

Although most of the time JSON object types are parsed, all JSON data types can be parsed directly, without occurring as part of a JSON object. The following examples show parsing of a JSON array:

```
; parse a JSON array data type

(json-parse "[1, 2, 3, 4, 5]") → (1 2 3 4 5)
```

When the UTF-8 capable version of newLISP is used, JSON formatted Unicode gets translated into UTF-8:

```
; parse a JSON object data type ands Unicode
; the outer {,} are newLISP string delimiters [text],[/text] tags could also be used
; the inner {,} are JSON object delimiters

(json-parse { {"greek letters" : "\u03b1\u03b2\u03b3\u03b4"} }) → (("greek letters" "αβγδ"))

; strings longer than 2047 bytes should be delimted with [text], [/text] tags

(json-parse [text]{"greek letters" : "\u03b1\u03b2\u03b3\u03b4"}[/text]) → (("greek letters" "αβγδ"))
```

The hex-code representation of Unicoder characters in JSON is the same as can be used in UTF-8 enabled newLISP.

Because JSON objects contain `{,},"` characters, quotes should not be used to limit JSON data, or all quotes inside the JSON data would need a preceding backslash `\`. `{,}` braces can be used as long as braces inside the JSON data are balanced. The safest delimiter are `[text]`, `[/text]` tags — they suppress all special processing of the string when read by newLISP and are suitable to delimit large data sizes greater 2047 bytes.

join

syntax: (join *list-of-strings* [*str-joint* [*bool-trail-joint*]])

Concatenates the given list of strings in *list-of-strings*. If *str-joint* is present, it is inserted between each string in the join. If *bool-trail-joint* is `true` then a joint string is also appended to the last string.

```
(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"

(join '("A" "B" "C") "-")         → "A-B-C"
(join '("A" "B" "C") "-" true)    → "A-B-C-"
```

See also the append, string, and explode functions, which are the inverse of the `join` operation.

kmeans-query

syntax: (kmeans-query *list-data matrix-centroids*)
syntax: (kmeans-query *list-data matrix-data)*

In the first usage, `kmeans-query` calculates the Euclidian distances from the data vector given in *list-data* to the centroids given in *matrix-centroids*. The data vector in *list-data* has *m* elements. The 2-dimensional list in *matrix-centroids*, result from a previous kmeans-train clustering, has *k* rows and *m* columns for *k* centroids measuring *m* features.

```
; centroids from previous kmeans-train
K:centroids →
( (6.39 7.188333333 5.935)
(7.925714286 3.845714286 9.198571429)
(2.207142857 2.881428571 0.8885714286) )

(kmeans-query '(1 2 3) K:centroids) →
(8.036487279 9.475994267 2.58693657) ; distances to cluster 1, 2 and 3
```

The data record `(1 2 3)` shows the smallest distance to the 3rd cluster centroid and would be classified as belonging to that cluster.

In the second application `kmeans-query` calculates Euclidian distances to a list of other data points which are not centroids. The following example calculates distances of the `(1 2 3)` data vector to all original points from the original kmeans-train data analysis.

The data in *matrix-data* can be either a nested list or a 2-dimensional array.

This vector could be sorted for a subsequent kNN (k Nearest Neighbor) analysis:

```
(kmeans-query '(1 2 3) data) →
(10.91671196 3.190626898 9.19723328 3.014415366 9.079763213
6.83130295 8.533111976 9.624816881 6.444261013 2.013107051
3.186549858 9.475199206 9.32936761 2.874786949 7.084638311
10.96221237 10.50080473 3.162419959 2.423674896 9.526436899)

; show distances to members in each cluster

; for cluster labeled 1
(select (kmeans-query '(1 2 3) data) (K:clusters 0)) →
(9.079763213 6.83130295 9.624816881 6.444261013 7.084638311 10.50080473)

; for cluster labeled 2
(select (kmeans-query '(1 2 3) data) (K:clusters 1)) →
(10.91671196 9.19723328 8.533111976 9.475199206 9.32936761 10.96221237 9.526436899)

; for cluster labeled 3
(select (kmeans-query '(1 2 3) data) (K:clusters 2)) →
(3.190626898 3.014415366 2.013107051 3.186549858 2.874786949 3.162419959 2.423674896)
```

We see that the smallest distances are shown for the data points in the 3rd cluster at offset 2.

If the numbers of elements - features - in records of *list-data* is different from the number of columns in the data or centroid matrix, then the smaller is taken for calculating the Euclidian distances. This is useful when the last column of the data matrix does not contain feature data, but labels identifying the cluster membership of a data point.

kmeans-train

syntax: (kmeans-train *matrix-data int-k context* [*matrix-centroids*])

The function performs Kmeans cluster analysis on *matrix-data*. All *n* data records in *matrix-data* are partitioned into a number of *int-k* different groups.

Both, the *n* * *m matrix-data* and the optional *k* * *m matrix-centroids* can be either nested lists or 2-dimensional arrays.

The Kmeans algorithm tries to minimize the sum of squared inner cluster distances (SSQ) from the cluster centroid. With each iteration the centroids get moved closer to their final position. On some data sets, the end result can depend on the starting centroid points. The right choice of initial centroids can speed up the process and avoid not wanted local minima.

When no optional *matrix-centroids* are given, `kmeans-train` will assign an initial random cluster membership to each data row and calculate starting centroids.

`kmeans-train` returns a vector of total SSQs, the sum of squared inner distances from the centroid inside the cluster for all clusters. The Iterating algorithm stops when the change of SSQ from one to the next iteration is less than 1e-10.

Other results of the analysis are stored as lists in variables of *context*.

The following example analyses 20 data records measuring *m = 3* features and tries to partition data into *k = 3* clusters. Other numbers than *k = 3* could be tried. The target is a result with few clusters of high density measured by the average inner cluster distances.

```
(set 'data '(
(6.57 4.96 11.91)
(2.29 4.18 1.06)
(8.63 2.51 8.11)
(1.85 1.89 0.11)
(7.56 7.93 5.06)
(3.61 7.95 5.11)
(7.18 3.46 8.7)
(8.17 6.59 7.49)
(5.44 5.9 5.57)
(2.43 2.14 1.59)
(2.48 2.26 0.19)
(8.16 3.83 8.93)
(8.49 5.31 7.47)
(3.12 3.1 1.4)
(6.77 6.04 3.76)
(7.01 4.2 11.9)
(6.79 8.72 8.62)
(1.17 4.46 1.02)
(2.11 2.14 0.85)
(9.44 2.65 7.37)))

(kmeans-train data 3 'MAIN:K) →
(439.7949357 90.7474276 85.06633163 82.74597619)

; cluster membership
K:labels → (2 3 2 3 1 1 2 1 1 3 3 2 2 3 1 2 1 3 3 2)

; the centroid for each cluster
K:centroids →
( (6.39 7.188333333 5.935)
(7.925714286 3.845714286 9.198571429)
(2.207142857 2.881428571 0.8885714286) )
```

The returned list of SSQs shows how in each iteration the sum of inner squared distances decreases. The list in `K:labels` shows the membership fo each data point in the same order as in the data.

The centroids in `K:centroids` can be used for later classification of new data records using kmeans-query. When the number of clusters specified in *int-k* is too big, `kmeans-train` will produce unused centroids with `nan` or `NaN` data. When unused cluster centroids are present, the number in *int-k* should be reduced.

The average inner `K:deviations` from cluster members to their centroid show how dense a cluster is packed. Formally, deviations are calculated similarly to Euclidian distances and to standard deviations in conventional statistics. Squaring the deviations and multiplying each with their cluster size (number of members in the cluster) shows the inner SSQ of each cluster:

```
; average inner deviations of cluster members to the centroid
; deviation = sqrt(ssq-of-cluster / n-of-cluster)
K:deviations  → (2.457052209 2.260089397 1.240236975)
```

```
; calculating inner SSQs from cluster deviations
(map mul '(6 7 7) (map mul K:deviations K:deviations)) →
(36.22263333 35.75602857 10.76731429) ; inner SSQs

; SSQ from last iteration as sum of inner SSQs
(apply add '(36.22263333 35.75602857 10.76731429)) → 82.74597619
```

`K:clusters` gives indices of data records into the original data for each cluster. With these, individual clusters can be extracted from the data for further analysis:

```
; ceach of the result clusters with indices into the data set
K:clusters →
( (4 5 7 8 14 16)
(0 2 6 11 12 15 19)
(1 3 9 10 13 17 18) )

; cluster of data records labeled 1 at offset 0
(select data (K:clusters 0)) →
( (7.56 7.93 5.06)
(3.61 7.95 5.11)
(8.17 6.59 7.49)
(5.44 5.9 5.57)
(6.77 6.04 3.76)
(6.79 8.72 8.62) )

; cluster of data records labeled 2 at offset 1
(select data (K:clusters 1)) →
( (6.57 4.96 11.91)
(8.63 2.51 8.11)
(7.18 3.46 8.7)
(8.16 3.83 8.93)
(8.49 5.31 7.47)
(7.01 4.2 11.9)
(9.44 2.65 7.37) )

; cluster of data records labeled 3 at offset 2
(select data (K:clusters 2)) →
( (2.29 4.18 1.06)
(1.85 1.89 0.11)
(2.43 2.14 1.59)
(2.48 2.26 0.19)
(3.12 3.1 1.4)
(1.17 4.46 1.02)
(2.11 2.14 0.85) )
```

In the last example the cluster labels (from 1 to 3) are added to the data:

```
; append a cluster label to each data record
(set 'labeled-data (transpose (push K:labels (transpose data) -1)))

labeled-data: →
( (6.57 4.96 11.91 2)
(2.29 4.18 1.06 3)
(8.63 2.51 8.11 2)
(1.85 1.89 0.11 3)
(7.56 7.93 5.06 1)
(3.61 7.95 5.11 1)
... ...
(2.11 2.14 0.85 3)
(9.44 2.65 7.37 2) )
```

The result context should be prefixed with `MAIN` when code is written in a namespace

context. If the context does not exists already, it will be created.

Results in `K:labels`, `K:clusters`, `K:centroids` and `K:deviations` will be overwritten, if already present from previous runs of `kmeans-train`.

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; otherwise, returns `nil`.

```
(define (square x) (* x x)) → (lambda (x) (* x x))

square → (lambda (x) (* x x))

(lambda? square)  → true
```

See define and define-macro for more information about *lambda* expressions.

last utf8

syntax: (last *list*)
syntax: (last *array*)
syntax: (last *str*)

Returns the last element of a list or a string.

```
(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)

(last '())          → ERR: list is empty
```

In the second version the last character in the string *str* is returned as a string.

```
(last "newLISP")  → "P"
```

Note that last works on character boundaries rather than byte boundaries when the UTF-8–enabled version of newLISP is used. See also first, rest and nth.

last-error

syntax: (last-error)
syntax: (last-error *int-error*)

Reports the last error generated by newLISP due to syntax errors or exhaustion of some resource. For a summary of all possible errors see the chapter Error codes in the appendix.

If no error has occurred since the newLISP session was started, `nil` is returned.

When *int-error* is specified, a list of the number and the error text is returned.

```
(last-error)  → nil

(abc)

ERR: invalid function : (abc)

(last-error) → (24 "ERR: invalid function : (abc)")

(last-error 24) → (24 "invalid function")
(last-error 1) → (1 "not enough memory")
(last-error 12345) → (12345 "Unknown error")
```

For error numbers out of range the string `"Unknown error"` is given for the error text.

Errors can be trapped by error-event and user defined error handlers.

See also net-error for errors generated by networking conditions and sys-error for errors generated by the operating system.

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:

```
(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 [bigint]

syntax: (length *exp*)

Returns the number of elements in a list, the number of rows in an array and the number of bytes in a string or in a symbol name.

Applied to a number, `length` returns the number of digits for normal and big integers and the number of digits before the decimal separator for floats.

`length` returns `0` on all other types.

Before version 10.5.6 `length` returned the storage size in bytes for integers (4 or 8) and floats (8).

```
; number of top level elements in a list
(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

; number of top level elements in an array
(set 'ary (array 2 4 '(0)))  → ((1 2 3 4) (5 6 7 8))
(length ary)                 → 2

; number of bytes in a string or byte buffer
(length "Hello World")  → 11
(length "")             → 0
(length "\000\001\003") → 3

; number of bytes in a symbol name string
(length 'someVar)  → 7

; number of int digits in a number
(length 0)         → 0
(length 123)       → 3
(length 1.23)      → 1
(length 12345678901234567890123456L) → 25
```

Use utf8len to calculate the number of UTF-8 characters in a string.

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. In the fully parenthesized first syntax, initializers are optional and assumed `nil` if missing.

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 parentheses around the variable expression pairs but functions identically.

```
(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` and local for initializing to `nil`. See local for automatic initialization of variables to `nil`.

letex

syntax: (letex ((*sym1* [*exp-init1*]) [(*sym2* [*exp-init2*]) ...]) *body*)
syntax: (letex (*sym1 exp-init1* [*sym2 exp-init2* ...]) *body*)

This function combines let and expand to expand local variables into an expression before evaluating it. In the fully parenthesized first syntax initializers are optional and assumed `nil` if missing.

Both forms provide the same functionality, but in the second form the parentheses around

the initializers can be omitted:

```
(letex (x 1 y 2 z 3) '(x y z))    → (1 2 3)

(letex ( (x 1) (y '(a b c)) (z "hello") ) '(x y z))

→ (1 (a b c) "hello")
```

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)`.

In the second example a function `make-adder` is defined for making adder functions:

```
(define (make-adder n)
    (letex (c n) (lambda (x) (+ x c))))

(define add3 (make-adder 3)) → (lambda (x) (+ x 3))

(add3 10) → 13

; letex can expand symbols into themselves
; the following form also works

(define (make-adder n)
     (letex (n n) (lambda (x) (+ x n))))
```

`letex` evaluates `n` to the constant `3` and replaces `c` with it in the lambda expression. The second examples shows, how a `letex` variable can be expanded into itself.

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 similarly to let, but will incrementally use the new symbol bindings when evaluating the initializer expressions as if several let were nested. In the fully parenthesized first syntax, initializers are optional and assumed `nil` if missing.

The following comparison of let and `letn` show the difference:

```
(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 parentheses 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. Note that arguments of array type are converted to lists.

```
(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.

```
(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-1* [*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.

```
(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/mysql.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 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 a 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 into 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.

```
(log 1)         → 0
(log (exp 1))   → 1
```

In the second syntax, an arbitrary base can be specified in *num-base*.

```
(log 1024 2)             → 10
(log (exp 1) (exp 1))    →  1
```

See also exp, which is the inverse function to `log` with base ***e*** (2.718281828).

lookup

syntax: (lookup *exp-key list-assoc* [*int-index* [*exp-default*]])

Finds in *list-assoc* an association, the key element of which has the same value as *exp-key*, and returns the *int-index* element of association (or the last element if *int-index* is absent).

Optionally, *exp-default* can be specified, which is returned if an association matching *exp-key* cannot be found. If the *exp-default* is absent and no association has been found, `nil` is returned.

See also Indexing elements of strings and lists.

`lookup` is similar to assoc but goes one step further by extracting a specific element found in the list.

```
(set 'params '(
    (name "John Doe")
    (age 35)
    (gender "M")
    (balance 12.34)
))

(lookup 'age params)             → 35

; use together with setf to modify and association list
(setf (lookup 'age params) 42)   → 42
(lookup 'age params)             → 42

(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"

(lookup "Jane Doe" persons 1 "N/A")  → "N/A"
```

See also assoc

lower-case utf8

syntax: (lower-case *str*)

Converts the characters of the string in *str* to lowercase. A new string is created, and the original is left unaltered.

```
(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 (*sym-name* [*sym-param-1* ...]) [*body-1* ...])

The `macro` function is used to define expansion macros. The syntax of `macro` is identical to the syntax of define-macro. But while `define-macro` defines are *fexprs* functions to be evaluated at run-time, `macro` defines a function to be used during the source loading and reading process to transform certain expression call patterns into different call patterns.

Symbols defined with `macro` are protected from re-definition.

```
(macro (double X) (+ X X)) → (lambda-macro (X) (expand '(+ X X)))

(double 123) → 246

(protected? 'double) → true
```

Internally all `macro` defined symbol call patterns are translated using the expand expression during source reading. This can be shown using the read-expr function:

```
(read-expr "(double 123)") → (+ 123 123)
```

All variable names to be expanded must start in upper-case. Macros can be nested containing other macros defined earlier. But `macro` definitions cannot be repeated for the same symbol during the same newLISP session. To redefine a macro, e.g. for reading source with a different definition of an exisiting `macro` definition, use the define function in the following way:

```
; change existing macro 'double' to allow floating point parameters
; use upper-case for variables for expansion

(define double (lambda-macro (X) (expand '(add X X))))
→ (lambda-macro (X) (expand '(add X X)))

(double 1.23) → 2.46
```

Note, that define can be used only to re-define macros, not to create new macros. Internally newLISP knows that `macro` defined symbols are executed during source reading, not evaluation.

Using `map` and `apply` with `macro`

When mapping macros using map or apply the expansion function is mapped:

```
> (macro (double X) (+ X X))
(lambda-macro (X) (expand '(+ X X)))

> (map double '(1 2 3 4 5))
((+ 1 1) (+ 2 2) (+ 3 3) (+ 4 4) (+ 5 5))

> (map eval (map double '(1 2 3 4 5)))
(2 4 6 8 10)

> (apply double '(10))
(+ 10 10)
>
```

This is useful to find out how the expansion mechanism of our `macro` definition works during source load time.

Differences between `macro` and `define-macro` and potential problems.

`macro` definitions are not susceptible to *variable capture* as are fexprs made with define-macro:

```
(define-macro (fexpr-add A B)
    (+ (eval A) (eval B)))

(macro (mac-add A B)
    (+ A B))

(set 'A 11 'B 22)

; variable capture when using the same symbols
; used as locals in define-macro for callling

(fexpr-add A B) →
; or
(fexpr-add B A) →
ERR: value expected : A
called from user defined function fexpr-add

; no variable capture when doing the same with
; expansion macros

(mac-add A B) → 33
```

```
(mac-add B A) → 33
```

But expansion macros using `macro` are susceptible to unwanted double evaluation, just like `define-macro` is:

```
(define-macro (fexpr-double X)
    (+ (eval X) (eval X)))

(macro (mac-double X)
    (+ X X))

(set 'a 10)
(fexpr-double (inc a)) → 23 ; not 22 as expected

(set 'a 10)
(mac-double (inc a)) → 23 ; not 22 as expected
```

In both cases the incoming expression `(inc a)` gets evaulated twice. This must be considered when writing both, `macro` or `define-macro` expressions and symbols occur more than once in the body of the definition.

macro?

syntax: (macro? *exp*)

Returns `true` if *exp* evaluates to a lambda-macro expression. If *exp* evaluates to a symbol and the symbol contains a macro-expansion expression made with the macro function, `true` is also returned. In all other cases `nil` is returned.

```
(define-macro (mysetq lv rv) (set lv (eval rv)))

(macro? mysetq)  → true

(macro (my-setq Lv Rv) (set 'Lv Rv))
→ (lambda-macro (Lv Rv) (expand '(set 'Lv Rv)))

; my-setq contains a lambda-macro expression
(macro? my-setq)   → true

; my-setq symbol was created with macro function
(macro? 'my-setq)   → 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.

```
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. Note that an index out of range will cause `nil` to be returned, not the last element of the list like in list-indexing.

```
newlisp a b c

> (main-args 0)
"/usr/bin/newlisp"
> (main-args -1)
"c"
> (main-args 2)
"b"
> (main-args 10)
nil
```

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.

```
;; 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. Since version 10.5.5 *list-args* can also be array vectors, but the returned result will always be a list.

```
(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)

(map (foo mul 3) '(1 2 3 4 5))  → (3 6 9 12 15)
```

Note that the quote before the operand can be omitted because 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)
```

`map` also sets the internal list index `$idx`.

```
(map (fn (x) (list $idx x)) '(a b c)) → ((0 a) (1 b) (2 c))
```

The number of arguments used is determined by the length of the first argument list. Arguments missing in other argument lists cause map to stop collecting parameters for that level of arguments. This ensures that the nth parameter list gets converted to the nth column during the transposition occurring. If an argument list contains too many elements, the extra ones will be ignored.

Special forms which use parentheses as syntax cannot be mapped (i.e. case).

mat

syntax: (mat + | - | * | / *matrix-A matrix-B*)
syntax: (mat + | - | * | / *matrix-A number*)

Using the first syntax, this function performs fast floating point scalar operations on two-dimensional matrices in *matrix-A* or *matrix-B*. The type of operation is specified by one of the four arithmetic operators `+`, `-`, `*`, or `/`. This type of arithmetic operator is typically used for integer operations in newLISP. In the case of `mat`, however, all operations will be performed as floating point operations (`add`, `sub`, `mul`, `div`).

Matrices in newLISP are two-dimensional lists or arrays. Internally, newLISP translates lists and arrays into fast, accessible C-language data objects. This makes matrix operations in newLISP as fast as those coded directly in C. The same is true for the matrix operations multiply and invert.

```
(set 'A '((1 2 3) (4 5 6)))
(set 'B A)

(mat + A B)    → ((2 4 6) (8 10 12))
(mat - A B)    → ((0 0 0) (0 0 0))
(mat * A B)    → ((1 4 9) (16 25 36))
(mat / A B)    → ((1 1 1) (1 1 1))

; specify the operator in a variable

(set 'op +)
(mat op A B)    → ((2 4 6) (8 10 12))
```

Using the second syntax, all cells in *matrix-A* are operated on with a scalar in *number*:

```
(mat + A 5)    → ((6 7 8) (9 10 11))
(mat - A 2)    → ((-1 0 1) (2 3 4))
(mat * A 3)    → ((3 6 9) (12 15 18))
(mat / A 10)   → ((.1 .2 .3) (.4 .5 .6))
```

See also the other matrix operations det, invert, multiply, and transpose.

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`. If no wildcard characters are present in the pattern an empty list is returned.

Optionally, the Boolean value `true` (or any other expression not evaluating to `nil`) can be supplied as a third argument. This causes `match` to show all elements in the returned result.

`match` is frequently employed as a functor parameter in find, ref, ref-all and replace and is internally used by find-all for lists.

```
(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) true)  → (a (x y z) c)

(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.

unify is another function for matching expressions in a PROLOG like manner.

max

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

Evaluates the expressions *num-1*— and returns the largest number.

```
(max 4 6 2 3.54 7.1)  → 7.1
```

See also the min function.

member

syntax: (member *exp list*)
syntax: (member *str-key str* [*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.

```
(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.

```
(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.

```
(min 4 6 2 3.54 7.1)  → 2
```

See also the max function.

mod

syntax: (mod *num-1 num-2* [*num-3* ...])
syntax: (mod *num-1*)

Calculates the modular value of the numbers in *num-1* and *num-2*. `mod` computes the remainder from the division of the numerator *num-i* by the denominator *num-i + 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.

In the second syntax `1` is assumed for *num-2* and the result is the fractional part of *num-1*.

```
(mod 10.5 3.3)   →  0.6
(mod -10.5 3.3)  → -0.6
(mod -10.5)      → -0.5
```

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`.

```
(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.

```
(set 'A '((1 2 3) (4 5 6)))
(set 'B '((1 2) (1 2) (1 2)))
(multiply A B)  → ((6 12) (15 30))

(set 'v '(10 20 30))
(multiply A (transpose (list v))) → ((140) (320))
```

When multiplying a matrix with a vector of `n` elements, the vector must be transformed into `n` rows by `1` column matrix using transpose.

All operations shown here on lists can be performed on arrays, as well.

See also the matrix operations det, invert, mat and transpose.

name

This function is deprecated, use term instead.

NaN?

syntax: (NaN? *float*)

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'.

```
; floating point operation on NaN yield NaN
(set 'x (sqrt -1))  → NaN
(NaN? x)            → true
(add x 123)         → NaN
(mul x 123)         → NaN

; integer operations treat NaN as zero
(+ x 123)  → 123
(* x 123)  → 0

; comparisons with NaN values yield nil
(> x 0)   → nil
(<= x 0)  → nil
(= x x)   → nil
```

```
(set 'infinity (mul 1.0e200 1.0e200)) → inf
(NaN? (sub infinity infinity)) → 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.

See also inf? for testing a floating point value for infinity.

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.

```
(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 files `server` and `client` examples in the `examples/` directory of the source distribution.

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.

```
(net-close aSock)
```

The optional *true* flag suppresses immediate shutdown of sockets by waiting for pending data transmissions to finish.

net-connect

syntax: (net-connect *str-remote-host int-port* [*int-timeout-ms*])
syntax: (net-connect *str-remote-host int-port* [*str-mode* [*int-ttl*]])
syntax: (net-connect *str-file-path*)

In the first syntax, 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; otherwise, returns `nil`.

```
(set 'socket (net-connect "example.com" 80))
(net-send socket "GET /\r\n\r\n")
(net-receive socket buffer 10000)
(println buffer)
(exit)
```

If successful, the `net-connect` function returns a socket number which can be used to send and receive information from the host. In the example a HTTP GET request is sent and subsequently a web page received. Note that newLISP has already a built-in function get-url offering the same functionality.

Optionally a timeout value *int-timeout* in milliseconds can be specified. Without a timeout value the function will wait up to 10 seconds for an open port. With a timeout value the function can be made to return on an unavailable port much earlier or later. The following example shows a port scanner looking for open ports:

```
(set 'host (main-args 2))
(println "Scanning: " host)
(for (port 1 1024)
    (if (set 'socket (net-connect host port 500))
        (println "open port: " port " " (or (net-service port "tcp") ""))
        (print port "\r"))
)
```

The programs takes the host string from the shell command line as either a domain name or an IP number in dot notation then tries to open each port from 1 to 1024. For each open port the port number and the service description string is printed. If no description is available, an empty string "" is output. For closed ports the function outputs numbers in the shell window staying on the same line.

On Unix `net-connect` may return with `nil` before the timeout expires, when the port is not available. On Win32 `net-connect` will always wait for the timeout to expire before failing with `nil`.

UDP communications

In the second syntax, 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 creates the socket and binds it to the remote address, if an address is specified. A subsequent net-send will send a UDP packet containing that target address. When using net-send-to, only one of the two functions `net-connect` or `net-send-to` should provide a target address. The other function should specify and empty string `""` as the target address.

```
;; example server
(net-listen 4096 "226.0.0.1" "udp")  → 5
(net-receive-from 5 20)

;; example client I
(net-connect "226.0.0.1" 4096 "udp") → 3
(net-send 3 "hello")
```

```
;; example client II
(net-connect "" 4096 "udp") → 3
(net-send-to "226.0.0.1" 4096 "hello" 3)
```

The functions net-receive and net-receive-from can both be used and will perform UDP communications when the `"udp"` option as been used in `net-listen` or `net-connect`. net-select and net-peek can be used to check for received data in a non-blocking fashion.

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 client I
(net-connect "" 4096 "multi")  → 3
(net-send-to "226.0.0.1" 4096 "hello" 3)

;; example client II
(net-connect "226.0.0.1" 4096 "multi")  → 3
(net-send 3 "hello")

;; 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 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.

Local domain Unix sockets

In the third syntax, `net-connect` connects to a server on the local file system via a *local domain Unix socket* named using *str-file-path*. Returns a socket handle after having connected successfully; otherwise, returns `nil`.

```
(net-connect "/tmp/mysocket")  → 3

; on OS/2 use "\\socket\\" prefix

(net-connect "\\socket\\mysocket")
```

A *local domain* file system socket is created and returned. On the server side, *local domain* sockets have been created using net-listen and net-accept. After the connection has been established the functions net-select, net-send and net-receive can be used as usual for TCP/IP stream communications. This type of connection can be used as a fast bi-directional communications channel between processes on the same file system. This type of connection is not available on Win32 platforms.

net-error

syntax: (net-error)
syntax: (net-error *int-error*)

Retrieves the last error that occurred when calling a any of the following functions: 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. Whenever one of these functions fails, it returns `nil` and `net-error` can be used to retrieve more information.

Functions that communicate using sockets close the socket automatically and remove it from the net-sessions list.

Each successful termination of a net-* function clears the error number.

The following messages are returned:

no	**description**
1	Cannot open socket
2	DNS resolution failed
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
17	Cannot unblock socket
18	Operation timed out
19	HTTP bad formed URL
20	HTTP file operation failed
21	HTTP transfer failed
22	HTTP invalid response from server
23	HTTP no response from server
24	HTTP document empty
25	HTTP error in header
26	HTTP error in chunked format

```
(net-error) → nil

(net-connect "jhghjgkjhg" 80)  →  nil

(net-error)  →  (2 "ERR: "DNS resolution failed")
```

When *int-error* is specified the number and error text for that error number is returned.

```
(net-error 10) → (10 "Cannot bind socket")
```

See also last-error and sys-error.

net-eval

syntax: (net-eval *str-host int-port exp* [*int-timeout* [*func-handler*]])
syntax: (net-eval '((*str-host int-port exp*) ...) [*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.

The expression in *exp* evaluates to either a string or an expression which will be evaluated remotely in the environment of the target node.

The remote TCP/IP servers are started in the following way:

```
newlisp -c -d 4711 &

; or with logging connections

newlisp -l -c -d 4711 &

; communicate via Uix local domain socket

newlisp -c /tmp/mysocket
```

The `-c` option is necessary to suppress newLISP emitting prompts.

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.

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.

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 *exp*. If *int-timeout* is not given, `net-eval` will wait up to 60 seconds for a response after a connection is made. Otherwise, if the timeout in milliseconds has expired, `nil` is returned; else, the evaluation result of *exp* is returned.

```
; the code to be evaluated is given in a quoted expression
(net-eval "192.168.1.94" 4711 '(+ 3 4))       → 7

; expression as a string
(net-eval "192.168.1.94" 4711 "(+ 3 4)")      → 7

; with timeout
(net-eval "192.168.1.94" 4711 '(+ 3 4) 1)     → nil  ; timeout to short
(net-error)                                   → (17 "ERR: Operation timed out")

(net-eval "192.168.1.94" 4711 '(+ 3 4) 1000)  → 7
```

```
; program contained in a variable
(set 'prog '(+ 3 4))
(net-eval "192.168.1.94" 4711 prog)           → 7

; specify a local-domain Unix socket (not available on Win32)
(net-eval "/tmp/mysocket" 0 '(+ 3 4))         → 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`. The last example line shows how to specify a local-domain Unix socket specifying the socket path and a port number of `0`. 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.

```
; two different remote nodes different IPs
(net-eval '(
    ("192.168.1.94" 4711 '(+ 3 4))
    ("192.168.1.95" 4711 '(+ 5 6))
    ) 5000)
→ (7 11)

; two persistent nodes on the same CPU different ports
(net-eval '(
    ("localhost" 8081 '(foo "abc"))
    ("localhost" 8082 '(myfunc 123)')
    ) 3000)

; inetd or xinetd nodes on the same server and port
; nodes are loaded on demand
(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 specifying `0` for the port number , `net-eval` takes the host name as the file path to the local-domain Unix socket.

Note that definitions of `foo` and `myfunc` must both exist in the target environment. This can be done using a `net-eval` sending `define` statements before. It also can be done by preloading code when starting remote nodes.

When nodes are inetd or xinetd-controlled, several nodes may have the same IP address and port number. In this case, the Unix daemon inetd or xinetd will start multiple newLISP servers on demand. This is useful when testing distributed programs on just one machine. The last example illustrates this case. It is also useful on multi core CPUs, where the platform OS can distribute different processes on to different CPU cores.

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. The a `net-eval` statement could load these programs.

Optionally, a handler function can be specified. This function will be repeatedly called while waiting and once for every remote evaluation completion.

```
(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.

```
(net-eval (list
  (list (Nodes 0 0) (Nodes 0 1) (Progs 0))
  (list (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.

For a longer example see this program: mapreduce. The example shows how a word counting task gets distributed to three remote nodes. The three nodes count words in different texts and the master node receives and consolidates the results.

net-interface

syntax: (net-interface *str-ip-addr*)
syntax: (net-interface)

Sets the default local interface address to be used for network connections. If not set then network functions will default to an internal default address, except when overwritten by an optional interface address given in net-listen.

When no *str-ip-addr* is specified, the current default is returned. If the `net-interface` has not been used yet to specify an IP address, the address `0.0.0.0` is returned. This means that all network routines will use the default address preconfigured by the underlying operating

system.

This function has only usage on multihomed servers with either multiple network interface hardware or otherwise supplied multiple IP numbers. On all other machines network functions will automatically select the single network interface installed.

On error the function returns `nil` and net-error can be used to report the error.

```
(net-interface "192.168.1.95")  → "192.168.1.95"
(net-interface "localhost")     → "127.0.0.1"
```

An interface address can be defined as either an IP address or a name. The return value is the address given in *str-ip-addr*

net-ipv

syntax: (net-ipv *int-version*)
syntax: (net-ipv)

Switches between IPv4 and IPv6 internet protocol versions. *int-version* contains either a 4 for IPv4 or a 6 for IPv6. When no parameter is given, `net-ipv` returns the current setting.

```
(net-ipv)      → 4
(net-ipv 6)    → 6
```

By default newLISP starts up in IPv4 mode. The IPv6 protocol mode can also be specified from the commandline when starting newlisp:

```
newlisp -6
```

Once a socket is connected with either net-connect or listened on with net-listen, the net-accept, net-select, net-send, net-receive and net-receive-from functions automatically adjust to the address protocol used when creating the sockets. Different connections with different IPv4/6 settings can be open at the same time.

Note, that currently net-packet does not support IPv6 and will work in IPv4 mode regardless of settings.

net-listen

syntax: (net-listen *int-port* [*str-ip-addr* [*str-mode*]])
syntax: (net-listen *str-file-path*)

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.

```
(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")
```

Local domain Unix sockets

In the second syntax, `net-listen` listens for a client on the local file system via a *local domain Unix socket* named using *str-file-path*. If successful, returns a socket handle that can be used with net-accept to accept a client connection; otherwise, returns `nil`.

```
(net-listen "/tmp/mysocket")  → 5

; on OS/2 use "\\socket\\" prefix

(net-listen "\\socket\\mysocket")

(net-accept 5)
```

A *local domain* file system socket is created and listened on. A client will try to connect using the same *str-file-path*. After a connection has been accepted the functions net-select, net-send and net-receive can be used as usual for TCP/IP stream communications. This type of connection can be used as a fast bi-directional communications channel between processes on the same file system. This type of connection is not available on Win32 platforms.

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 10002 "192.168.1.120" "udp")

(net-listen 10002 "" "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 the files `examples/client` and `examples/server` in the newLISP source distribution.

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 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.

Packet divert sockets and ports

If *str-mode* is specified as `"divert"` or `"d"`, a divert socket can be created for a divert port in *int-port* on BSD like platforms. The content of IP address in *str-ip-addr* is ignored and can be specified as an empty string. Only the *int-port* is relevant and will be bound to the raw socket returned.

To use the divert option in `net-listen`, newLISP must run in super-user mode. This option is only available on Unix like platforms.

The divert socket will receive all raw packets diverted to the divert port. Packets may also be written back to a divert socket, in which case they re-enter OS kernel IP packet processing.

Rules for packet diversion to the divert port must be defined using either the *ipfw* BSD or *ipchains* Linux configuration utilities.

The net-receive-from and net-send-to functions are used to read and write raw packets on the divert socket created and returned by the `net-listen` statement. The same address received by net-receive-from is used in the net-send-to call when re-injecting the packet:

```
; rules have been previously configured for a divert port
(set 'divertSocket (net-listen divertPort "" "divert"))

(until (net-error)
    (set 'rlist (net-receive-from divertSocket maxBytes))
    (set 'buffer (rlist 1))
    ; buffer can be processed here before reinjecting
    (net-send-to (rlist 0) divertPort buffer divertSocket)
)
```

For more information see the Unix man pages for *divert* and the *ipfw* (BSDs) or *ipchains* (Linux) configuration utilities.

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*.

```
(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*:

```
(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)
→ "216.16.84.66"
```

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-packet

syntax: (net-packet *str-packet*)

The function allows custom configured network packets to be sent via a *raw sockets* interface. The packet in *str-packet* must start with an IP (Internet Protocol) header followed by either a TCP, UDP or ICMP header and optional data. newLISP must be run with super user privileges, and this function is only available on Mac OS X, Linux and other Unix operating systems and only for IPv4. Currently `net-packet` is IPv4 only and has been tested on Mac OS X, Linux and OpenBSD.

On success the function returns the number of bytes sent. On failure the function returns `nil` and both, net-error and sys-error, should be inspected.

When custom configured packets contain zeros in the checksum fields, `net-packet` will calculate and insert the correct checksums. Already existing checksums stay untouched.

The following example injects a UDP packet for IP number `192.168.1.92`. The IP header consists of 20 bytes ending with the target IP number. The following UDP header has a length of 8 bytes and is followed by the data string `Hello World`. The checksum bytes in both headers are left as `0x00 0x00` and will be recalculated internally.

```
; packet as generated by: (net-send-udp "192.168.1.92" 12345 "Hello World")

(set 'udp-packet (pack (dup "b" 39) '(
    0x45 0x00 0x00 0x27 0x4b 0x8f 0x00 0x00 0x40 0x11 0x00 0x00 192  168  1    95
    192  168  1    92   0xf2 0xc8 0x30 0x39 0x00 0x13 0x00 0x00 0x48 0x65 0x6c 0x6c
    0x6f 0x20 0x57 0x6f 0x72 0x6c 0x64)))

(unless (net-packet udp-packet)
    (println "net-error: " (net-error))
    (println "sys-error: " (sys-error)))
```

The `net-packet` function is used when testing net security. Its wrong application can upset the correct functioning of network routers and other devices connected to a network. For this

reason the function should only be used on well isolated, private intra-nets and only by network professionals.

For other examples of packet configuration, see the file `qa-specific-tests/qa-packet` in the newLISP source distribution.

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.

```
(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 number of the remote computer for a connection on *int-socket*.

```
(net-peer 16)  → ("192.100.81.100" 13)
```

Use the net-local function to access the local computer's IP number and port number.

net-ping

syntax: (net-ping *str-address* [*int-timeout* [*int-count bool*]]])
syntax: (net-ping *list-addresses* [*int-timeout* [*int-count bool*]]])

This function is only available on Unix-based systems and must be run in superuser mode, i.e. using: `sudo newlisp` to start newLISP on Mac OS X or other BSD's, or as the root user on

Linux. Broadcast mode and specifying ranges with the - (hyphen) or * (star) are not available on IPv6 address mode.

Superuser mode is not required on Mac OS X.

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 lists of IP strings and round-trip time in microseconds 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.

```
(net-ping "newlisp.org")     → (("66.235.209.72" 634080))
(net-ping "127.0.0.1")       → (("127.0.0.1" 115))
(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 specified in a list or specified by range. Packets are sent out as fast as possible. In this case, multiple answers can be received. If the same address is specified multiple times, the receiving IP address will be flooded with ICMP packets.

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-count*. 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. Not specifying *int-count* or specifying 0 assumes an *int-count* equal to the number of packets sent out.

As third optional parameter, a true value can be specified. This setting will return an error string instead of the response time, if the host does not answer.

```
(net-ping '("newlisp.org" "192.168.1.255") 2000 20)
→ (("66.235.209.72" 826420) ("192.168.1.1" 124) ("192.168.1.254" 210))

(net-ping "192.168.1.*" 500) ; from 1 to 254
→ (("192.168.1.1" 120) ("192.168.1.2" 245) ("192.168.2.3" 180) ("192.168.2.254" 234))

(net-ping "192.168.1.*" 500 2) ; returns after 2 responses
→ (("192.168.1.3" 115) ("192.168.1.1" 145))

(net-ping "192.168.1.1-10" 1000) ; returns after 1 second
→ (("192.168.1.3" 196) ("192.168.1.1" 205))
```

```
(net-ping '("192.168.1.100-120" "192.168.1.124-132") 2000) ; returns after 2 seconds
→ ()
```

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 lines 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. The fourth and fifth lines show how an IP range can be specified for the last subnet octet in the IP number. `net-ping` will iterate through all numbers from either 1 to 254 for the star `*` or the range specified, 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 specifying broadcast mode, *int-count* should be specified, too.

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 an incomplete list or the empty list `()`. In this case, net-error will return a timeout error. On error, `nil` is returned and net-error can be used to retrieve an error message.

On some systems only lists up to a specific length can be handled regardless of the timeout specified. In this case, the range should be broken up into sub-ranges and used with multiple `net-ping` invocations. In any case, `net-ping` will send out packages as quickly as possible.

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*. *sym-buffer* can also be a default functor specified by a context symbol for reference passing in and out of user-defined functions.

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*.

```
(define (gettime)
    (set 'socket (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` returns `nil` and closes the socket.

On success `net-receive` returns a list of the data string, remote IP number and remote port used. On failure it returns `nil`.

```
;; bind port 1001 and 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.

Both, the sender and the receiver have to issue net-listen commands for UDP mode. Not for

listening as in TCP/IP protocol communications, but to create the socket bound to the port and address. For a complete example see the files `udp-server.lsp` and `udp-client.lsp` in the `newlisp-x.x.x/examples/` directory of the source distribution.

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. `net-receive` can also be used for TCP/IP communications.

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.

```
;; wait for datagram with maximum 20 bytes
(net-receive-udp 10001 20)

;; or
(net-receive-udp 10001 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.

```
(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:

```
(set 'listen-list '(1001 1002))

; accept-connection, read-connection and write-connection
; are user defined functions

(while (not (net-error))
    (dolist (conn (net-select listen-list "r" 1000))
    (accept-connection conn))  ; build an accept-list

    (dolist (conn (net-select accept-list "r" 1000))
    (read-connection conn))    ; read on connected sockets

    (dolist (conn (net-select accept-list "w" 1000))
    (write-connection conn)))  ; write on connected sockets
```

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 str-buffer* [*int-num-bytes*])

Sends the contents of *str-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, the entire contents will be sent. `net-send` returns the number of bytes sent or `nil` on failure.

On failure, use net-error to get more error information.

```
(set 'buf "hello there")

(net-send sock buf)       → 11
(net-send sock buf 5)     → 5

(net-send sock "bye bye") → 7
```

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*)

Can be used for either UDP or TCP/IP communications. The socket in *int-socket* must have

previously been opened with a net-connect or net-listen function. If the opening functions was used with the "udp" option, net-listen or net-connect are not used to listen or to connect but only to create the UDP socket. The host in *str-remotehost* can be specified either as a hostname or as an IP-number string.

When using net-connect together with net-send-to, then only one of the functions should specify the remote host. The other should leave the address as an empty string.

```
;;;;;;;;;;;;;;;;; UDP server
(set 'socket (net-listen 10001 "" "udp"))
(if socket (println "server listening on port " 10001)
       (println (net-error)))
(while (not (net-error))
   (set 'msg (net-receive-from socket 255))
   (println "-> " msg)
   (net-send-to (nth 1 msg) (nth 2 msg)
                (upper-case (first msg)) socket))

;;;;;;;;;;;;;;;;; UDP client
(set 'socket (net-listen 10002 "" "udp"))
(if (not socket) (println (net-error)))
(while (not (net-error))
   (print "> ")
   (net-send-to "127.0.0.1" 10001 (read-line) socket)
   (net-receive socket buff 255)
   (println "-> " buff))
```

In the examples both, the client and the server use net-listen to create the UDP socket for sending and receiving. The server extracts the client address and port from the message received and uses it in the net-send-to statement.

See also the net-receive-from function and the net-listen function with the "udp" option.

For blocking short UDP transactions use 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*.

The theoretical maximum data size of a UDP packet on an IPv4 system is 64K minus IP layer overhead, but much smaller on most Unix flavors. 8k seems to be a safe size on Mac OS X, BSDs and Linux.

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.

```
(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 non-visible 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*)
syntax: (net-service *int-port str-protocol*)

In the first syntax `net-service` makes a lookup in the *services* database and returns the standard port number for this service.

In the second syntax a service port is supplied in *int-port* to look up the service name.

Returns `nil` on failure.

```
; get the port number from the name
(net-service "ftp" "tcp")       → 21
(net-service "http" "tcp")      → 80
(net-service "net-eval" "tcp")  → 4711  ; if configured

; get the service name from the port number
(net-service 22 "tcp")          → "ssh"
```

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*)

The context *context-source* is copied to *sym-context-target*. If the target context does not exist, a new context with the same variable names and user-defined functions as in *context-source* is created. If the target context already exists, then new symbols and definitions are added. Existing symbols are only overwritten when the expression in *bool* evaluates to anything other than `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.

```
(new CTX 'CTX-2)  → CTX-2

;; force overwrite of existing symbols
(new CTX MyCTX true)  → MyCTX
```

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 the same name in `MyCTX` will be overwritten. Because `MyCTX` already exists, the quote before the context symbol can be omitted.

Context symbols need not be mentioned explicitly, but they can be contained in variables:

```
(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? *exp*)

If the expression in *exp* evaluates to `nil`, then `nil?` returns `true`; otherwise, it returns `nil`.

```
(map nil? '(x nil  1 nil "hi" ()))
→ (nil true nil true nil nil)

(nil? nil)  → true
(nil? '())  → nil

; nil? means strictly nil
(nil? (not '()))  → nil
```

The `nil?` predicate is useful for distinguishing between `nil` and the empty list `()`.

Note that `nil?` means *strictly* `nil` while `true?` means everything not `nil` or the empty list `()`.

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.

```
(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.

not

syntax: (not *exp*)

If *exp* evaluates to `nil` or the empty list `()`, then `true` is returned; otherwise, `nil` is returned.

```
(not true)            → nil
(not nil)             → true
(not '())             → true
(not (< 1 10))        → nil
(not (not (< 1 10)))  → true
```

now

syntax: (now [*int-minutes-offset* [*int-index*]])

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-minutes-offset*. This causes the time to be shifted forward or backward in time, before being split into separate date values.

An optional list-index in *int-index* makes `now` return a specific member in the result list.

```
(now)       → (2002 2 27 18 21 30 140000 57 3 -300 0)
(now 0 -2)  → -300 ; minutes west of GMT


(apply date-value (now))  → 1014834090
```

The numbers represent the following date-time fields:

format	**description**
year	Gregorian calendar
month	(1–12)
day	(1–31)
hour	(0–23) UTC
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 Monday
time zone offset in minutes	west of GMT
daylight savings time type	(0–6) on Linux/Unix or bias in minutes on Win32

The second example returns the Coordinated Universal Time (UTC) time value of seconds after January 1, 1970.

Ranging from 0 to 23, hours are given in UTC and are not adjusted for the local time zone. The resolution of the `microseconds` field depends on the operating system and platform. On some platforms, the last three digits of the `microseconds` field are always `0` (zero).

The "day of the week" field starts with 1 on Monday conforming to the ISO 8601 international standard for date and time representation.

On some platforms, the daylight savings time flag is not active and returns `0` (zero) even during daylight savings time (dst).

Depending on the geographical area, the daylight savings time type (dst) has a different value from 1 to 6:

type	area
0	not on dst
1	USA style dst
2	Australian style dst
3	Western European dst
4	Middle European dst
5	Eastern European dst
6	Canada dst

See also the date, date-list, date-parse, date-value, time, and time-of-day functions.

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 payment is at the end of the period, *int-type* is `0` (zero) or *int-type* is omitted; for payment at the beginning of each period, *int-type* is 1.

```
(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*.

```
(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 utf8

syntax: (nth *int-index list*)
syntax: (nth *int-index array*)
syntax: (nth *int-index str*)

syntax: (nth *list-indices list*)
syntax: (nth *list-indices array*)

In the first syntax group `nth` uses *int-index* an index into the *list*, *array* or *str* found and 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 or arrays. If there are more indices than nesting levels, the extra indices are ignored. When multiple indices are used, they must be put in a list as shown in the second syntax group.

```
(set 'L '(a b c))
(nth 0 L)    → a
; or simply
(L 0) → a

(set 'names '(john martha robert alex))
→ (john martha robert alex)

(nth 2 names)    → robert
; or simply
(names 2)        → robert

(names -1)       → alex


; multiple indices
(set 'persons '((john 30) (martha 120) ((john doe) 17)))

(persons 1 1)           → 120

(nth '(2 0 1) persons)  → doe

; or simply
(persons 2 0 1)        → doe

; multiple indices in a vector
(set 'v '(2 0 1))
(persons v)       → doe
(nth v persons)   → doe

; negative indices
(persons -2 0)    → martha

; out-of-bounds indices cause error
(persons 10)  → ERR: list index out of bounds
(person -5)   → ERR: list index out of bounds
```

The list `L` can be the context of the default functor `L:L`. This allows lists passed by reference:

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

(define (second ctx)
        (nth 1 ctx))

(reverse L) → (g f e d c b a)
L:L → (g f e d c b a)

;; passing the list in L:L by reference
(second L)   → b

;; passing the list in L:L by value
(second L:L) → b
```

Reference passing is faster and uses less memory in big lists and should be used on lists with more than a few hundred items.

Note that the *implicit indexing* version of `nth` is not breaking newLISP syntax rules but should be understood as a logical expansion of newLISP syntax rules to other data types than built-in functions or lambda expressions. A list in the functor position of an s-expression assumes self-indexing functionality using the index arguments following.

The implicit indexed syntax forms are faster but the other form with an explicit `nth` may be more readable in some situations.

`nth` works on arrays just like it does on lists:

```
(set 'aArray (array 2 3 '(a b c d e f)))
→ ((a b c) (d e f))
(nth 1 aArray)      →  (d e f)
(aArray 1)          →  (d e f)

(nth '(1 0) aArray)    → d
(aArray 1 0)           → d
(aArray '(1 0))        → d

(set 'vec '(1 0))
(aArray vec)           → d
```

In the String version, `nth` returns the character found at the position *int-index* in *str* and returns it as a string.

```
(nth  0 "newLISP")   → "n"

("newLISP" 0)        → "n"

("newLISP" -1)       → "P"
```

Note that nth works on character boundaries rather than byte boundaries when using the UTF-8–enabled version of newLISP. To access ASCII and binary string buffers on single byte boundaries use slice.

See also setf for modifying multidimensional lists and arrays and push and pop for modifying lists.

null?

syntax: (null? *exp*)

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.

```
(set 'x (sqrt -1)) → NaN ; or nan on UNIX
(null? x) → true

(map null? '(1 0 0.0 2 "hello" "" (a b c) () true))
→ (nil true true nil nil true nil true nil)

(filter null? '(1 0 2 0.0 "hello" "" (a b c) () nil true))
→ (0 0 "" () nil)

(clean null? '(1 0 2 0.0 "hello" "" (a b c) () nil true))
→ (1 2 "hello" (a b c) true)
```

See also the predicates empty?, nil? and zero?.

number? bigint

syntax: (number? *exp*)

`true` is returned only if *exp* evaluates to a floating point number or an integer; otherwise, `nil` is returned.

```
(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.

odd? bigint

syntax: (odd? *int-number*)

Checks the parity of an integer number. If the number is not *even divisible* by `2`, it has *odd* parity. When a floating point number is passed for *int-number*, it will be converted first to an integer by cutting off its fractional part.

```
(odd? 123)  → true
(odd? 8)    → nil
(odd? 8.7)  → nil
```

Use even? to check if an integer is even, divisible by 2.

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

```
(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.

or

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

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.

```
(set 'x 10)
(or (> x 100) (= x 10))          → true
(or "hello" (> x 100) (= x 10))  → "hello"
(or '())                         → ()
(or true)                        → true
(or)                             → nil
```

ostype

syntax: ostype

`ostype` is a built-in system constant containing the name of the operating system newLISP is running on.

```
ostype  → "Win32"
```

One of the following strings is returned: `"Linux", "BSD", "OSX", "Tru64Unix", "Solaris", "SunOS", "Win32", "Cygwin", or "OS/2"`.

`ostype` can be used to write platform-independent code:

```
(if
    (= ostype "Linux") (import "libz.so")
    (= ostype "BSD") (import "libz.so")
    (= ostype "OSX") (import "libz.dylib")
    ...
    (println "cannot import libz on this platform")
)
```

Use sys-info to learn more about the current flavor of newLISP running.

For a table of other built-in system variables and symbols see the chapter System Symbols and Constants in the appendix.

pack

syntax: (pack *str-format exp-1* [*exp-2 ...*])
syntax: (pack *str-format list*)

syntax: (pack *struct exp-1* [*exp-2 ...*])
syntax: (pack *struct list*)

When the first parameter is a string, `pack` packs one or more expressions (*exp-1* to *exp-n*) into a binary format specified in the format string *str-format,* and returning the binary structure in a string buffer. The symmetrical unpack function is used for unpacking. The expression arguments can also be given in a *list*. `pack` and `unpack` are useful when reading and

writing binary files (see read and write) or when unpacking binary structures from return values of imported C functions using `import`.

When the first parameter is the symbol of a struct definition, `pack` uses the format as specified in *struct*. While `pack` with *str-format* literally packs as specified, `pack` with *struct* will insert structure aligning pad-bytes depending on data type, order of elements and CPU architecture. Refer to the description of the struct function for more detail.

The following characters are used in *str-format*:

format	description
`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
`Ld`	a signed 64-bit long number
`Lu`	an unsigned 64-bit long number
`f`	a float in 32-bit representation
`lf`	a double float in 64-bit representation
`sn`	a string of *n* null padded ASCII characters
`nn`	*n* null characters
`>`	switch to big endian byte order
`<`	switch to little endian byte order

`pack` will convert all floats into integers when passed to `b`, `c`, `d`, `ld`, or `lu` formats. It will also convert integers into floats when passing them to `f` and `lf` formats.

```
(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)

(pack "n10") → "\000\000\000\000\000\000\000\000\000\000"

(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 specification.

The expressions to pack can also be given in a list:

```
(set 'lst '("A" "B" "C"))
(set 'adr (pack "lululu" lst))
(map get-string (unpack "lululu" adr))    → ("A" "B" "C")
```

Note that the list should be referenced directly in `pack`, so the pointers passed by `adr` are valid. `adr` would be written as `char * adr[]` in the C-programming language and represents a 32-bit pointer to an array of 32-bit string pointers.

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

(pack ">u <u" 1 1) → "\000\001\001\000" ;; switch twice
```

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.

See also the address, get-int, get-long, get-char, get-string, and unpack functions.

parse

syntax: (parse *str-data* [*str-break* [*regex-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 *regex-option* number or string 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 newLISP source. If *str-break* 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*.

```
(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), double 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 CSV (Comma Separated Values) 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.

```
(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.

```
(pipe)  → (3 4)  ; 3 for read, 4 for writing
```

The pipe handles can be passed to a child process 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 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: (pmt *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`). If payment is at the end of the period, *int-type* is `0` (zero) or *int-type* is omitted; for payment at the beginning of each period, *int-type* is 1.

```
(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 ! utf8

syntax: (pop *list* [*int-index-1* [*int-index-2* ...]])
syntax: (pop *list* [*list-indexes*])

syntax: (pop *str* [*int-index* [*int-length*]])

Using `pop`, elements can be removed from lists and characters from strings.

In the first syntax, pop 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-indexes*. 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.

```
(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 -1)   → 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))
→ (a b (c d (x) e))

pList
→ (a b (c d (x) e))

(ref 'x pList)  → (2 2 0)

(pop pList '(2 2 0))  → x
```

pop can also be used on strings with one index:

```
;; use pop on strings

(set 'str "newLISP")

(pop str -4 4)  → "LISP"

str  → "new"

(pop str 1)  → "e"

str  → "nw"

(set 'str "x")

(pop str)  → "x"
(pop str)  → ""
```

Popping an empty string will return an empty string.

See also the push function, the inverse operation to `pop`.

pop-assoc !

syntax: (pop-assoc *exp-key list-assoc*)
syntax: (pop-assoc *list-keys list-assoc*)

Removes an association referred to by the key in *exp-key* from the association list in *list-assoc* and returns the popped expression.

```
;; simple associations

(set 'L '((a 1) (b 2) (c 3)))
(pop-assoc 'b L) → (b 2)
L → ((a 1) (c 3))

;; nested associations

(set 'L '((a (b 1) (c (d 2)))))
(pop-assoc 'a L) → (a (b 1) (c (d 2)))
L → ()

(set 'L '((a (b 1) (c (d 2)))))
(pop-assoc '(a b) L)  → (b 1)
L →  ((a (c (d 2))))

(set 'L '((a (b 1) (c (d 2)))))
(pop-assoc '(a c) L)  → (c (d 2))
L → ((a (b 1)))
```

See also assoc for retrieving associations and setf for modifying association lists.

post-url

syntax: (post-url *str-url str-content* [*str-content-type* [*str-option*] [*int-timeout* [*str-header*]]])

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.

```
;; specify content type
(post-url "http://somesite.com/form.pl"
          "name=johnDoe&city=New%20York"
          "application/x-www-form-urlencoded")

;; specify content type and timeout
(post-url "http://somesite.com/form.pl"
          "name=johnDoe&city=New%20York"
          "application/x-www-form-urlencoded" 8000)

;; assumes default content type and no timeout
(post-url "http://somesite.com/form.pl"
          "name=johnDoe&city=New%20York"
```

The above example uploads a user name and city 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.

```
(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.

prefix

syntax: (prefix *sym*)

Returns the context of a symbol in *sym*:

```
(setf s 'Foo:bar)      → Foo:bar
(prefix s)             → Foo
(context? (prefix s))  → true

(term s)                          → "bar"
(= s (sym (term s) (prefix s)))  → true

>(context (prefix s))   ; switches to context Foo
Foo
Foo>
```

See also term to extract the term part of a symbol.

pretty-print

syntax: (pretty-print [*int-length* [*str-tab* [*str-fp-format*]]])

Reformats expressions for print, save, or source and when printing in an interactive console. The first parameter, *int-length*, specifies the maximum line length, and *str-tab* specifies the string used to indent lines. The third parameter *str-fp-format* describes the default format for printing floating point numbers. All parameters are optional. `pretty-print` returns the current settings or the new settings when parameters are specified.

```
(pretty-print)  → (80 " " "%1.15g")  ; default setting

(pretty-print 90 "\t")  → (90 "\t" "%1.15g")

(pretty-print 100)  → (100 "\t" "%1.15g")

(sin 1)    → 0.841470984807897
(pretty-print 80 " " "%1.3f")
(sin 1)    → 0.841

(set 'x 0.0)
x   → 0.000
```

The first example reports the default settings of 80 for the maximum line length and a `space` character for indenting. The second example changes the line length to 90 and the indent to a TAB character. The third example changes the line length only. The last example changes the default format for floating point numbers. This is useful when printing unformatted floating point numbers without fractional parts, and these numbers should still be recognizable as floating point numbers. Without the custom format, `x` would be printed as `0` indistinguishable from floating point number. All situations where unformatted floating point numbers are printed, are affected.

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:

```
;; 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.

```
(set 'var define)
(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 `\`:

character	description
`\n`	the line-feed character (ASCII 10)
`\r`	the carriage-return character (ASCII 13)
`\t`	the tab character (ASCII 9)
`\nnn`	where `nnn` is a decimal ASCII code between 000 and 255
`\xnn`	where `nn` is a hexadecimal ASCII code between 00 and FF

```
(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 int-df*)

Returns the probability of an observed *Chi²* statistic in *num-chi2* with *num-df* degrees of freedom to be equal or greater under the null hypothesis. `prob-chi2` is derived from the incomplete Gamma function gammai.

```
(prob-chi2 10 6)  → 0.1246520195
```

See also the inverse function crit-chi2.

prob-f

syntax: (prob-f *num-f int-df1 int-df2*)

Returns the probability of an observed *F* statistic in *num-f* with *int-df1* and *int-df2* degrees of freedom to be equal or greater under the null hypothesis.

```
(prob-f 2.75 10 12)  → 0.0501990804
```

See also the inverse function crit-f.

prob-t

syntax: (prob-t *num-t int-df1*)

Returns the probability of an observed *Student's t* statistic in *num-t* with *int-df* degrees of freedom to be equal or greater under the null hypothesis.

```
(prob-t 1.76 14)  → 0.05011454551
```

See also the inverse function crit-t.

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`.

```
(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-unix-pipe-error*])

In the first syntax, `process` launches a process specified in *str-command* and immediately returns with a process ID or `nil` if a process could not be created. This process will execute the program specified or immediately die if *str-command* could not be executed.

On Mac OS X and other Unixes, the application or script must be specified with its full path-name. The new process inherits the OS environment from the parent process.

Command line arguments are parsed out at spaces. Arguments containing spaces must be delimited using single quotes on Mac OS X and other Unixes. On Win32, double quotes are used. The process id returned can be used to destroy the running process using destroy, if the process does not exit by itself.

```
(process "c:/WINDOWS/system32/notepad.exe")  → 1894 ; on Win32

; find out the path of the program to start using exec,
; if the path is not known

(process (first (exec "which xclock")))  → 22607 ; on Unix
```

If the path of the executable is unknown, `exec` together with the Unix `which` command can be used to start a program. The pid returned can be used to destroy the process.

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 and read statements:

```
;; Linux/Unix
;; create pipes
(map set '(myin bcout) (pipe))
```

```
(map set '(bcin myout) (pipe))

;; launch Unix 'bc' calculator application
(process "/usr/bin/bc" bcin bcout) → 7916

(write-line myout "3 + 4")  ; bc expects a line-feed

(read-line myin)  → "7"


;; bc can use bignums with arbitrary precision

(write-line myout "123456789012345 * 123456789012345")

(read-line myin)  → "15241578753238669120562399025"

;; destroy the process
(destroy 7916)

;; Win32
(map set '(myin cmdout) (pipe))
(map set '(cmdin myout) (pipe))

(process "c:/Program Files/newlisp/newlisp.exe -c" cmdin cmdout)
→ 1284

(write-line myout "(+ 3 4)")

(read-line myin) → "7"

;; destroy the process
(destroy 1284)
```

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 *int-unix-pipe-error*.

The function 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 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 appout)
```

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 processes on Linux/Unix.

prompt-event

syntax: (prompt-event *sym-event-handler* | *func-event-handler*)

Refines the prompt as shown in the interactive newLISP shell. The *sym-event-handler* or *func-event-handler* is either a symbol of a user-defined function or a lambda expression:

```
> (prompt-event (fn (ctx) (string ctx ":" (real-path) "$ ")))
$prompt-event
MAIN:/Users/newlisp$ (+ 3 4)
7
MAIN:/Users/newlisp$
```

The current context before calling the `prompt-event` code is passed as a parameter to the function. Computer output is shown in bold.

The example redefines the `>` prompt to be the current context followed by a colon `:`, followed by the directory name, followed by the dollar symbol. Together with the command-event function this can be used to create fully customized shells or custom command interpreters.

The function in `prompt-event` must return a string of 63 characters maximum. Not returning a string will leave the prompt unchanged.

protected?

syntax: (protected? *sym*)

Checks if a symbol in *sym* is protected. Protected symbols are built-in functions, context symbols, and all symbols made constant using the constant function:

```
(protected? 'println)    → true
```

```
(constant 'aVar 123)
(protected? 'aVar)      → true
```

push ! utf8

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 list changed is returned as a reference on which other built-in functions can work. 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.

```
; inserting in front
(set 'pList '(b c))  → (b c)
(push 'a pList)      → (a b c)
pList                → (a b c)

; insert at index
(push "hello" pList 2)  → (a b "hello" c)

; optimized appending at the end
(push 'z pList -1)  → (a b "hello" c z)

; inserting lists in lists
(push '(f g) pList)  → ((f g) a b "hello" c z)

; inserting at negative index
(push 'x pList -3)  → ((f g) a b "hello" x c z)

; using multiple indices
(push 'h pList 0 -1)  → ((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))  → (a b (c d (x) e))

(ref 'x pList)   → (2 2 0)
```

```
(pop pList '(2 2 0))  → x

; the target list is a place reference
(set 'lst '((a 1) (b 2) (c 3) (d)))

(push 4 (assoc 'd lst) -1) → (d 4)

lst → ((a 1) (b 2) (c 3) (d 4))


; push on un-initialized symbol
aVar  → nil

(push 999 aVar)  → (999)

aVar  → (999)
```

`push` and pop can be combined to model a queue:

```
; pop and push a as a queue
(set 'Q '(a b c d e))

(pop (push 'f Q -1)) → a
(pop (push 'g Q -1)) → b

Q →  (c d e f g)
```

Because `push` returns a reference to the modified list, `pop` can work on it directly.

In the third syntax `push` can be used to change strings. When *int-index* is used, it refers to character positions rather than byte positions. UTF-8 characters may be multi-byte characters.

```
;; push on strings

(set 'str "abcdefg")

(push "hijk" str -1)  → "abcdefghijk"
str                   → "abcdefghijk"

(push "123" str)  → "123abcdefghijk"
(push "4" str 3)  → "1234abcdefghijk"

(set 'str "\u03b1\u03b2\u03b3")  →  "αβγ"

(push "*" str 1)  →  "α*βγ"

;; push on a string reference

(set 'lst '("abc" "xyz"))

(push x (lst 0)) → "xabc"

lst → ("xabc" "xyz")
```

See also the pop function, which is the inverse operation to `push`.

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.

If *str-url* starts with `file://` then *str-content* is written to the local file system.

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.

```
(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"))

; write /home/joe/newfile.txt on the local file system
(puts-url "file:///home/joe/newfile.txt" "Hello World!")
```

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`.

```
<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:

```
#!/usr/home/johndoe/bin/newlisp
#
#
# get PUT method data from CGI STDIN
# and write data to a file specified
# int the PUT request
#
#


(print "Content-Type: text/html\n\n")

(set 'cnt 0)
(set 'result "")
```

```
(if (= "PUT" (env "REQUEST_METHOD"))
    (begin
      (set 'len (int (env "CONTENT_LENGTH")))

      (while (< cnt len)
          (set 'n (read (device) buffer len))
          (if (not n)
            (set 'cnt len)
            (begin
              (inc cnt n)
              (write 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-fv* is assumed to be `0.0` if omitted. If payment is at the end of the period, *int-type* is `0` (zero) or *int-type* is omitted; for payment at the beginning of each period, *int-type* is 1.

```
(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*. The function `quote` is resolved during runtime, the prepended `'` quote is translated into a protective envelope (quote cell) during code translation.

```
(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.

```
(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 C `time()` function. Optionally, a second parameter can be specified to return a list of length *int-N* of random numbers.

```
(dotimes (x 100) (print (rand 2))) =>
11100000110100111100111101 ... 10111101011101111101001100001000

(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.

```
(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.

```
(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 !

syntax: (read *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. *sym-buffer* can also be a default functor specified by a context symbol for reference passing in and out of user-defined functions.

`read` is a shorter writing of `read-buffer`. The longer form still works but is deprecated and should be avoided in new code.

Optionally, a string to be waited for can be specified in *str-wait*. `read` 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 and must not contain binary `0` (zero) characters.

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`.

```
(set 'handle (open "aFile.ext" "read"))
(read handle buff 200)
```

Reads 200 bytes into the symbol `buff` from the file `aFile.ext`.

```
(read 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 function.

read-char

syntax: (read-char [*int-file*])

Reads a byte from a file specified by the file handle in *int-file* or from the current I/O device - e.g. *stdin* - when no file handle is specified. 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.

```
(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-expr

syntax: (read-expr *str-source* [*sym-context* [*exp-error* [*int-offset*]]])

`read-expr` parses the first expressions it finds in *str-source* and returns the translated expression without evaluating it. An optional context in *sym-context* specifies a namespace for the translated expression.

After a call to `read-expr` the system variable `$count` contains the number of characters scanned.

If an error occurs when translating *str-source* the expression in *exp-error* is evaluated and the result returned.

int-offset specifies an optional offset into *str-source* where processing should start. When calling `read-expr` repeatedly this number can be updated using `$count`, the number of characters processed.

```
(set 'code "; a statement\n(define (double x) (+ x x))")

(read-expr code) → (define (double x) (+ x x))

$count → 41
```

`read-expr` behaves similar to eval-string but without the evaluation step:

```
(read-expr "(+ 3 4)")    → (+ 3 4)

(eval-string "(+ 3 4)")  → 7
```

Using `read-expr` a customized code reader can be programmed preprocessing expressions before evaluation.

See also reader-event for preprocessing expressions event-driven.

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.

On failure the function returns `nil`. For error information, use sys-error when used on files. When used on URLs net-error gives more error information.

```
(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*. When the prefix is `http://`, `read-file` works exactly like get-url and can take the same additional parameters.

```
(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.

```
(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 only work when newLISP is running in a Unix shell or Win32 command shell. It will not work in the Java based newLISP-GS or Tcl/Tk based newLISP-Tk frontend. It will also not work when executed by newLISP Unix shared library or newLISP Win32 DLL (Dynamic Link Library).

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 0). 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.

When read-line is reading from a file or from *stdin* in a CGI program or pipe, it will return nil when input is exhausted.

When using read-line on *stdin*, line length is limited to 2048 characters and performance is much faster.

```
(print "Enter a num:")
(set 'num (int (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.

read-utf8

syntax: (read-utf8 *int-file*)

Reads an UTF-8 character from a file specified by the file handle in *int-file*. The file handle is obtained from a previous open operation. Each read-utf8 advances the file pointer by the number of bytes contained in the UTF-8 character. Once the end of the file is reached, nil is returned.

The function returns an integer value which can be converted to a displayable UTF-8 character string using the char function.

```
(set 'fle (open "utf8text.txt" "read"))
(while (setq chr (read-utf8 fle))
        (print (char chr)))
```

The example reads a file containing UTF-8 encoded text and displays it to the terminal screen.

reader-event

syntax: (reader-event [*sym-event-handler* | *func-event-handler*])
syntax: (reader-event 'nil)

An event handler can be specified to hook between newLISP's reader, translation and evaluation process. The function specified in *sym-event-handler* or *func-event-handler* gets called after newLISP translates an expression and before evaluating it. The event handler can do transformation on the expression before it gets evaluated.

Specifying a quoted `nil` for the event will disable it.

The following one-liner `reader-event` could be used to enhance the interactive shell with a tracer:

```
>(reader-event (lambda (ex) (print " => " ex)))
$reader-event
> (+ 1 2 3)
 => (+ 1 2 3)
6
>
```

The expression intercepted passes through unchanged, but output is enhanced.

The reader event function will be called after each reading of an s-expression by the load or eval-string function.

In versions previous to 10.5.8 `reader-event` was used to define a `macro` expansion function in the module file `macro.lsp`. Starting version 10.5.8, newLISP has macro as a built-in function behaving the same, but much faster when loading files and reading source.

real-path

syntax: (real-path [*str-path*])
syntax: (real-path *str-exec-name* true)

In the first syntax `real-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.

```
(real-path)  → "/usr/home/fred"  ; current directory
(real-path "./somefile.txt")
→ "/usr/home/fred/somefile.txt"
```

In the second syntax `real-path` returns the full path for an executable found given in *str-exe-name*. This syntax relies on an environment variable PATH defined on UNIX and Windows systems.

```
(real-path "make" true) → "/usr/bin/make"
```

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.

receive !

syntax: (receive *int-pid sym-message*)
syntax: (receive)

In the first syntax, the function is used for message exchange between child processes launched with spawn and their parent process. The message received replaces the contents in *sym-message.*

The function reads one message from the receiver queue of *int-pid* for each invocation. When the queue is empty, `nil` is returned.

```
; sending process
(send spid "hello")  → true

; receiving process
(receive pid msg)    → true
msg                  → "hello"
```

To make `receive` blocking and wait for arriving messages, use the following form:

```
; wait until a message can be read
(until (receive pid msg))
```

The function will loop until a message can be read from the queue.

In the second syntax, the function returns a list of all child processes with pending messages for the parent process:

```
; read pending messages from child processes
(dolist (pid (receive))
    (receive pid msg)
    (println "received message: " msg " from:" pid)
)
```

The list of child process IDs returned by `(receive)` only contains PIDs of processes which have unread messages in their send queues. The `(receive pid msg)` statement now can be issued non-blocking, because it always is guaranteed to find a pending message in a child's message queue.

The `receive` function is not available on Win32.

For a more detailed discussion of this function and examples, see the send function.

ref

syntax: (ref *exp-key list* [*func-compare* [true]])

`ref` searches for the key expression *exp-key* in *list* and returns a list of integer indices or an empty list if *exp-key* cannot be found. `ref` can work together with push and pop, both of which can also take lists of indices.

By default, `ref` checks if expressions are equal. With *func-compare*, more complex comparison functions can be used. The comparison function can be a previously defined function. Note that this function always takes two arguments, even if only the second argument is used inside the function.

When the optional `true` parameter is present, the element found is returned instead of the index vector.

```
; get index vectors for list elements

(set 'pList '(a b (c d (x) e)))

(ref 'x pList)    → (2 2 0)

(ref '(x) pList)   → (2 2)

; the key expression is in a variable

(set 'p '(c d (x) e))

(ref p pList)   → (2)

; indexing using the vector returned from ref

(set 'v (ref '(x) pList)) → (2 2)

(pList v) → (x)

; if nothing is found, nil is returned

(ref 'foo plist)  → nil

; not specifying a comparison functor assumes =

(set 'L '(a b (c d (e) f)))

(ref 'e L)      → (2 2 0)
(ref 'e L =)    → (2 2 0)

; a is the first symbol where e is greater

(ref 'e L >)  → (0)

; return the element instead of the index

(ref 'e L > true)  → a

; use an anonymous comparison function

(ref 'e L (fn (x y) (or (= x y) (= y 'd))))      → (2 1)

(ref 'e L (fn (x y) (or (= x y) (= y 'd))) true) → d
```

The following example shows the use of match and unify to formulate searches that are as

powerful as regular expressions are for strings:

```
(set 'L '((l 3) (a 12) (k 5) (a 10) (z 22)))

; use match as a comparison function

(ref '(a ?) L match) → (1)

; use unify as a comparison function

(set 'L '( ((a b) (c d)) ((e e) (f g)) ))

(ref '(X X) L unify)      → (1 0)

(ref '(X g) L unify)      → (1 1)

(ref '(X g) L unify true) → (f g)
```

The `'(X X)` pattern with unify searches for a list pair where the two elements are equal. The `unify` pattern `'(X g)` searches for a list pair with the symbol `g` as the second member. The patterns are quoted to protect them from evaluation.

Pass the list as a default functor:

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

(ref 'd C)  → (2 1)
```

This is suitable when passing lists by reference using a context. See also the chapter Passing data by reference.

See also the ref-all function, which searches for all occurrences of a key expression in a nested list.

ref-all

syntax: (ref-all *exp-key list* [*func-compare* [true]])

Works similarly to ref, but returns a list of all index vectors found for *exp-key* in *list*.

When the optional `true` parameter is present, the elements found is returned of the index vectors.

By default, `ref-all` checks if expressions are equal. With *func-compare*, more complex comparison functions can be used.

The system variable `$count` counts the number of elements found.

```
(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))
$count → 6
```

```
; the index vector returned by ref-all can be used to index the list

(L '(3 1)) → a

; mapped implicit indexing of L

(map 'L (ref-all 'a L)) → (a a a a a a)

; with comparison operator

(set 'L '(a b c (d f (h l a)) (k a (m n) (x))))

; not specifying a comparison functor assumes =

(ref-all 'c L)       → ((2))
(ref-all 'c L =)     → ((2))

; look for all elements where c is greater

(ref-all 'c L >)       → ((0) (1) (3 2 2) (4 1))
(ref-all 'c L > true)  → (a b a a)


; use an anonymous function to compare

(ref-all 'a L (fn (x y) (or (= x y) (= y 'k))))
→ ((0) (3 2 2) (4 0) (4 1))

; the key is nil because the comparison function only looks at the second argument

(ref-all nil L (fn (x y) (> (length y) 2)))
→ ((3) (3 2) (4))

; define the comparison functions first

(define (is-long? x y) (> (length y) 2)) ; the x gets occupied by 'nil

(ref-all nil L is-long?)    →  ((3) (3 2) (4))

(define (is-it-or-d x y) (or (= x y) (= y 'd)))

(set 'L '(a b (c d (e) f)) )

(ref-all 'e L is-it-or-d)  → ((2 1) (2 2 0))
```

The comparison function can be a previously defined function. Note that the comparison function always takes two arguments, even if only the second argument is used inside the function (as in the example using `is-long?`).

Using the match and unify functions, list searches can be formulated that are as powerful as regular expression searches are for strings.

```
(set 'L '((l 3) (a 12) (k 5) (a 10) (z 22)) )

; look for all pairs staring with the symbol a

(ref-all '(a ?) L match)      → ((1) (3))
(ref-all '(a ?) L match true) → ((a 12) (a 10))

; look for all pairs where elements are equal
```

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

(ref-all '(X X) L unify)      → ((1 0) (2))
(ref-all '(X X) L unify true) → ((e e) ((z) (z)))

; look for all pairs where the second element is the symbol g

(set 'L '( ((x y z) g) ((a b) (c d)) ((e e) (f g)) ))

(ref-all '(X g) L unify)      → ((0) (2 1))
(ref-all '(X g) L unify true) → (((x y z) g) (f g))
```

See also the ref function.

regex

syntax: (regex *str-pattern str-text* [*regex-option* [*int-offset*]])

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.

Additionally a *regex-option* can be specified to control certain regular expression options explained later. Options can be given either by numbers or letters in a string.

The additional *int-offset* parameter tells `regex` to start searching for a match not at the beginning of the string but at an offset.

When no *regex-option* is present, the offset and length numbers in the `regex` results are given based bytes even when running the UTF-8 enabled version of newLISP. When specifying the PCRE_UTF8 option in *regex-option* only offset and length are reported in UTF8 characters.

`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 newLISP expressions for further processing.

Functions using regular expressions will not reset the `$0`, `$1 ... $15` variables to `nil` when no match is found.

```
(regex "b+" "aaaabbbaaaa")  → ("bbb" 4 3)

; case-insensitive search option 1
(regex "b+" "AAAABBBAAAA" 1)  → ("BBB" 4 3)
; same option given as a string
(regex "b+" "AAAABBBAAAA" "i")  → ("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 that 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 and other special char in regex
(regex "\\(abc\\)" "xyz(abc)xyz")  → ("(abc)" 3 5)
;; double backslash for backslash (special char in regex)
(regex "\\d{1,3}" "qwerty567asdfg")  → ("567" 6 3)

;; one backslash for quotes (special char in newLISP)
(regex "\"" "abc\"def")  → ("\"" 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 *regex-option*. Several options can be combined using a binary or `|` (pipe) operator. E.g. `(| 1 4)` would combine options `1` and `4` or `"is"` when using letters for the two options.

The last two options are specific for newLISP. The REPLACE_ONCE option is only to be used

in replace; it can be combined with other PCRE options.

Multiple options can be combined using a + (plus) or | (or) operator, e.g.: `(| PCRE_CASELESS PCRE_DOTALL)` or `"is"` when using letters as options.

PCRE name	no	description
PCRE_CASELESS	1 or i	treat uppercase like lowercase
PCRE_MULTILINE	2 or m	limit search at a newline like Perl's /m
PCRE_DOTALL	4 or s	. (dot) also matches newline
PCRE_EXTENDED	8 or x	ignore whitespace except inside char class
PCRE_ANCHORED	16 or A	anchor at the start
PCRE_DOLLAR_ENDONLY	32 or D	$ matches at end of string, not before newline
PCRE_EXTRA	64	additional functionality currently not used
PCRE_NOTBOL	128	first ch, not start of line; ^ shouldn't match
PCRE_NOTEOL	256	last char, not end of line; $ shouldn't match
PCRE_UNGREEDY	512i or U	invert greediness of quantifiers
PCRE_NOTEMPTY	1024	empty string considered invalid
PCRE_UTF8	2048 or u	pattern and strings as UTF-8 characters
REPLACE_ONCE	0x8000	replace only one occurrence only for use in replace
PRECOMPILED	0x10000 or p	pattern is pre-compiled, can only be combined with RREPLACE_ONCE 0x8000

The settings of the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED options can be changed from within the pattern by a sequence of option letters enclosed between "(?" and ")". The option letters are:

i for PCRE_CASELESS
m for PCRE_MULTILINE
s for PCRE_DOTALL
x for PCRE_EXTENDED

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.

Regular expression patterns can be precompiled for higher speed when using changing repetitive patterns with regex-comp.

regex-comp

syntax: (regex-comp *str-pattern* [*int-option*])

newLISP automatically compiles regular expression patterns and caches the last compilation to speed up repetitive pattern searches. If patterns change from one to the next, but are repeated over and over again, then the caching of the last pattern is not sufficient. `regex-comp` can be used to pre-compile repetitive patterns to speed up regular expression searches:

```
; slower without pre-compilation

(dolist (line page)
        (replace pattern-str1 line repl1 0)
        (replace pattern-str2 line repl2 512)
)

; fast with pre-compilation and option 0x10000

(set 'p1 (regex-comp pattern-str1))
(set 'p2 (regex-comp pattern-str2 512))

(dolist (line page)
        (replace p1 line repl1 0x10000)
        (replace p2 line repl2 0x10000)
)
```

When using pre-compiled patterns in any of the functions using regular expressions, the option number is set to `0x10000` to signal that pre-compiled patterns are used. Normal pattern options are specified during pre-compilation with `regex-comp` . The `0x10000` option can only be combined with `0x8000`, the option used to specify that only one replacement should be made when using replace.

The function ends-with should not be used with compiled patterns, as it tries to append to an un-compiled pattern internally.

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.

```
(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.

```
(rename-file "data.lisp" "data.backup")
```

replace !

syntax: (replace *exp-key list exp-replacement* [*func-compare*])
syntax: (replace *exp-key list*)

syntax: (replace *str-key str-data exp-replacement*)
syntax: (replace *str-pattern str-data exp-replacement regex-option*)

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*. If *exp-replacement* is missing, all instances of *exp-key* will be deleted from *list*.

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 `$count` when the function returns. During executions of the replacement expression, the anaphoric system variable `$it` is set to the expression to be replaced.

Optionally, *func-compare* can specify a comparison operator or user-defined function. By default, *func-compare* is the = (equals sign).

```
;; list replacement

(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)
$count → 2  ; number of replacements

;; list replacement with special compare functor/function

; replace all numbers where 10 < number
(set 'L '(1 4 22 5 6 89 2 3 24))

(replace 10 L 10 <) → (1 4 10 5 6 10 2 3 10)
$count → 3

; same as:

(replace 10 L 10 (fn (x y) (< x y))) → (1 4 10 5 6 10 2 3 10)

; change name-string to symbol, x is ignored as nil

(set 'AL '((john 5 6 4) ("mary" 3 4 7) (bob 4 2 7 9) ("jane" 3)))

(replace nil AL (cons (sym ($it 0)) (rest $it))
                (fn (x y) (string? (y 0)))) ; parameter x = nil not used
```

```
→ ((john 5 6 4) (mary 3 4 7) (bob 4 2 7 9) (jane 3))

; use $count in the replacement expression
(replace 'a '(a b a b a b) (list $count $it) =) →  ((1 a) b (2 a) b (3 a) b)
```

Using the match and unify functions, list searches can be formulated that are as powerful as regular expression string searches:

```
; calculate the sum in all associations with 'mary

(set 'AL '((john 5 6 4) (mary 3 4 7) (bob 4 2 7 9) (jane 3)))

(replace '(mary *)  AL (list 'mary (apply + (rest $it))) match)
→ ((john 5 6 4) (mary 14) (bob 4 2 7 9) (jane 3))
$count → 1

; make sum in all expressions

(set 'AL '((john 5 6 4) (mary 3 4 7) (bob 4 2 7 9) (jane 3)))

(replace '(*) AL (list ($it 0) (apply + (rest $it))) match)
→ ((john 15) (mary 14) (bob 22) (jane 3))
$count → 4

; using unify, replace only if elements are equal
(replace '(X X) '((3 10) (2 5) (4 4) (6 7) (8 8)) (list ($it 0) 'double ($it 1)) unify)
→ ((3 10) (2 5) (4 double 4) (6 7) (8 double 8))
```

List removal

The last form of `replace` has only two arguments: the expression *exp* and *list*. This form removes all *exp*s found in *list*.

```
;; 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)

$count → 4
```

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 `$count`. This form of `replace` can also process binary `0`s (zeros).

```
;; string replacement
(set 'str "this isa sentence")
(replace "isa" str "is a")  → "this is a sentence"

$count → 1
```

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 *regex-option* (e.g., `1` (one) or "i" for case-insensitive searching or `0` (zero) for a standard Perl Compatible Regular Expression (PCRE) search without options). 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 `$2–` with the contents of the expressions and subexpressions found. The anaphoric system variable `$it` is set to the same value as `$0`. These can be used to perform replacements that depend on the content found during replacement. The symbols `$it, $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).

After all replacements are made, the number of replacements is contained in the system variable `$count`.

```
;; 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 (int (append "0x" $1))) 1)

str    → "xxxAxxxB"

$count → 2
```

The setf function together with nth, first or last can also be used to change elements in a list.

See directory, find, find-all, parse, regex, and search for other functions using regular

expressions.

reset

syntax: (reset)
syntax: (reset true)
syntax: (reset *int-max-cells*)

In the first syntax, `reset` returns to the top level of evaluation, switches the trace mode off, and switches to the MAIN context/namespace. `reset` restores the top-level variable environment using the saved variable environments on the stack. It also throws an error "user reset - no error" which can be reported with user defined error handlers. Since version 10.5.5 `reset` also interrupts command line parameter processing.

`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 clean newLISP process with the same command-line parameters. This mode will only work when newLISP was started using its full path-name, e.g. `/usr/bin/newlisp` instead of only `newlisp`. This mode is not available on Win32.

In the third syntax. `reset` will change the maximum cell count allowed in the system. This number is also reported as the second number in the list by sys-info. On 64-bit newLISP one lisp cell occupies 32 bytes, or 16 bytes on the 32-bit version. This does not include string memory, which may be pointed to by cells.

The minimum cell count is 4095, trying to specify less will set it to 4095. The program will exit when trying to allocate more.

```
(sys-info)  → (437 576460752303423488 409 1 0 2048 0 60391 10602 1411)

; allocate about 1 Mbyte of cell memory on 64-bit newlisp
(reset 32768) → true

(sys-info)  → (437 32768 409 1 0 2048 0 60392 10602 1411)
```

Resetting the maximum cell count will not restart the system and can be done at any point in a program. Cell memory is allocated in blocks of 4095 cells, which is also initial minimum configuration.

rest utf8

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.

```
(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))

(rest '()) → ()
```

In the second version, `rest` returns all but the first character of the string *str* in a string.

```
(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 *array*)
syntax: (reverse *string*)

In the first and second form, `reverse` reverses and returns the *list* or *array*. Note that `reverse` is destructive and changes the original list or array.

```
; reverse a list
(set 'l '(a b c d e f))

(reverse l)  → (f e d c b a)
l            → (f e d c b a)
i
; reverse an array
(set 'a (array 3 2 '(1 2 3 4 5 6))) → ((1 2) (3 4) (5 6))

(reverse a)                      → ((5 6) (3 4) (1 2))
a                                → ((5 6) (3 4) (1 2))
```

In the third form, `reverse` is used to reverse the order of characters in a string.

```
; reverse byte character string

(set 'str "newLISP")

(reverse str)  → "PSILwen"
str            → "PSILwen"

; reverse a multibyte character UTF-8 string, explode is UTF-8 sensitive

(join (reverse (explode "ΑΒΓΔΕΖΗΘ")))  → "ΘΗΖΕΔΓΒΑ"
```

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.

```
(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)

; rotate a byte character string

(set 'str "newLISP")

(rotate str)     → "PnewLIS"
(rotate str 3)   → "LISPnew"
(rotate str -4)  → "newLISP"

; rotate a multibyte character UTF-8 string on character boundaries

(join (rotate (explode "ΑΒΓΔΕΖΗΘ")))  → "ΘΑΒΓΔΕΖΗ"
```

When working on a string, `rotate` works on byte boundaries rather than character boundaries.

round

syntax: (round *number* [*int-digits*])

Rounds the number in *number* to the number of digits given in *int-digits*. When decimals are being rounded, *int-digits* is negative. It is positive when the integer part of a number is being rounded.

If *int-digits* is omitted, the function rounds to `0` decimal digits.

```
(round 123.49 2)    → 100
(round 123.49 1)    → 120
(round 123.49 0)    → 123
(round 123.49)      → 123
(round 123.49 -1)   → 123.5
(round 123.49 -2)   → 123.49
```

Note that rounding for display purposes is better accomplished using format.

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), symbols of built-in functions and symbols containing `nil` 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.

`save` returns `true` on completion.

```
(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)
```

Because all context symbols are part of the context `MAIN`, saving `MAIN` saves all contexts.

Saving to a URL will cause an HTTP PUT request to be sent 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* [*bool-flag* [*regex-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 or the end of the searched string or at the end of the file if nothing is found.

By default, the file pointer is positioned at the beginning of the searched string. If *bool-flag* evaluates to `true`, then the file pointer is positioned at the end of the searched string.

In *regex-options*, the options flags can be specified to perform a PCRE regular expression search. See the function regex for details. If *regex-options* is not specified a faster, plain string search is performed. `search` returns the new file position or `nil` if nothing is found.

When using the regular expression options flag, patterns found are stored in the system variables `$0` to `$15`.

```
(set 'file (open "init.lsp" "read"))
(search file "define")
(print (read-line file) "\n")
(close file)

(set 'file (open "program.c" "r"))
(while (search file "#define (.*)" true 0) (println $1))
(close file)
```

The file `init.lsp` is opened and searched for the string `define` and the line in which the string occurs is printed.

The second example looks for all lines in the file `program.c` which start with the string `#define`

and prints the rest of the line after 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.

```
(seed 12345)

(seed (time-of-day))
```

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.

self

syntax: (self [*int-index* ...])

The function `self` accesses the target object of a FOOP method. One or more *int-index* are used to access the object members. `self` is set by the : colon operator.

Objects referenced with `self` are mutable:

```
(new Class 'Circle)

(define (Circle:move dx dy)
        (inc (self 1) dx)
        (inc (self 2) dy))

(set 'aCircle (Circle 1 2 3))
(:move aCircle 10 20)

aCircle → (Circle 11 22 3)
```

```
; objects can be anonymous
(set 'circles '((Circle 1 2 3) (Circle 4 5 6)))

(:move (circles 0) 10 20)
(:move (circles 1) 10 20)

circles → ((Circle 11 22 3) (Circle 14 25 6))
```

See also the chapter about programming with FOOP: Functional object-oriented programming

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.

`seek` can set the file position past the current end of the file. Subsequent writing to this position will extend the file and fill unused positions with zero's. The blocks of zeros are not actually allocated on disk, so the file takes up less space and is called a *sparse file*.

```
(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 'fle (open "large-file" "read")
(seek file 30000000000)  → 30000000000
```

newLISP supports file position numbers up to 9,223,372,036,854,775,807.

select utf8

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*.

```
(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*.

```
(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 0*)

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 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 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 processes will be released.

On Win32, only parent and child processes can share a semaphore. On Linux/Unix, independent processes can share a semaphore.

On failure the `semaphore` function returns `nil`. sys-error can be used to retrieve the error number and text from the underlying operating system.

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 system-wide (always 0)
(semaphore sid 0)
```

The following example shows semaphores controlling a child process:

```
;; counter process 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 the counter process

> (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-signaled state). When starting the process `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` process, 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` process an opportunity to decrement the semaphore several times before blocking.

More than one process can participate in controlling the semaphore, just as more than one semaphore can be created. The maximum number of semaphores is controlled by a system-wide kernel setting on Unix-like operating systems.

Use the fork function to start a new process and the share function to share information between processes. For a more comprehensive example of using `semaphore` to synchronize processes, see the file `prodcons.lsp` example in the `examples` directory in the source distribution, as well as the examples and modules distributed with newLISP.

send

syntax: (send *int-pid exp*)
syntax: (send)

The `send` function enables communication between parent and child processes started with spawn. Parent processes can send and receive messages to and from their child processes and child processes can send and receive messages to and from their parent process. A proxy technique – shown further down – is employed to communicate between child process peers. `send` and receive do not require locks or semaphores. They work on dual send and receive message queues.

Processes started using fork or process can not use `send` and `receive` message functions. Instead they should use either share with semaphore or pipe to communicate.

The `send` function is not available on Win32.

In the first syntax `send` is used to send a message from a parent to a child process or a child

to a parent process.

The second syntax is only used by parent processes to get a list of all child processes ready to accept message from the parent in their receive queues. If a child's receive queue is full, it will not be part of the list returned by the `(send)` statement.

The content of a message may be any newLISP expression either atomic or list expressions: boolean constants `nil` and `true`, integers, floating point numbers or strings, or any list expression in valid newLISP syntax. The size of a message is unlimited.

The *exp* parameter specifies the data to be sent to the recipient in *int-pid*. The recipient can be either a spawned child process of the current process or the parent process. If a message queue is full, it can be read from the receiving end, but a `send` issued on the other side of the queue will fail and return `nil`.

```
; child process dispatching message to parent

(set 'ppid (sys-info -4)) ; get parent pid

(send ppid "hello") ; send message
```

The targeted recipient of the message is the parent process:

```
; parent process receiving message from child

(receive child-pid msg) → true
msg                     → "hello"
```

When the `send` queue is full, `send` will return `nil` until enough message content is read on the receiving side of the queue and the queue is ready to accept new messages from `send` statements.

Using the until looping function, the message statements can be repeated until they return a value not `nil`. This way, non-blocking `send` and `receive` can be made blocking until they succeed:

```
; blocking sender
(until (send pid msg)) ; true after message is queued up

; blocking receiver
(until (receive pid msg)) ; true after message could be read
```

The sender statement blocks until the message could be deposited in the recipients queue.

The `receive` statement blocks until a new message can be fetched from the queue.

As the `until` statements in this example lack body expressions, the last value of the evaluated conditional expression is the return value of the `until` loop.

Blocking message exchange

The following code shows how a recipient can listen for incoming messages, and in turn how a sender can retry to deposit a message into a queue. The example shows 5 child processes constantly delivering status data to a parent process which will display the data. After three data sets have been read, the parent will abort all child processes and exit:

```
#!/usr/bin/newlisp

; child process transmits random numbers
(define (child-process)
    (set 'ppid (sys-info -4)) ; get parent pid
    (while true
        (until (send ppid (rand 100))))
)

; parent starts 5  child processes, listens and displays
; the true flag is specified to enable send/receive

(dotimes (i 5) (spawn 'result (child-process) true))

(for (i 1 3)
    (dolist (cpid (sync)) ; iterate thru pending child PIDs
        (until (receive cpid msg))
        (print "pid:" cpid "->" (format "%-2d  " msg)))
    (println)
)

(abort) ; cancel child-processes
(exit)
```

Running above example produces the following output:

```
pid:53181->47  pid:53180->61  pid:53179->75  pid:53178->39  pid:53177->3
pid:53181->59  pid:53180->12  pid:53179->20  pid:53178->77  pid:53177->47
pid:53181->6   pid:53180->56  pid:53179->96  pid:53178->78  pid:53177->18
```

The `(sync)` expression returns a list of all child PIDs, and `(until (receive cpid msg))` is used to force a wait until status messages are received for each of the child processes.

A timeout mechanism could be part of an `until` or `while` loop to stop waiting after certain time has expired.

The examples show messages flowing from a child processes to a parent process, in the same fashion messages could flow into the other direction from parent to child processes. In that case the parent process would use `(send)` to obtain a list of child processes with place in their message queues.

Messages containing code for evaluation

The most powerful feature of the message functions is the ability to send any newLISP expression, which then can be evaluated by the recipient. The recipient uses eval to evaluate the received expression. Symbols contained in the expression are evaluated in the receivers environment.

The following example shows how a parent process acts like a message proxy. The parent receives messages from a child process A and routes them to a second child process with ID B. In effect this implements messages between child process peers. The implementation relies on the fact that the recipient can evaluate expressions contained in messages received. These expressions can be any valid newLISP statements:

```
#!/usr/bin/newlisp

; sender child process of the message
```

```
(set 'A (spawn 'result
    (begin
        (dotimes (i 3)
            (set 'ppid (sys-info -4))
            /* the statement in msg will be evaluated in the proxy */
            (set 'msg '(until (send B (string "greetings from " A))))
            (until (send ppid msg)))
        (until (send ppid '(begin
            (sleep 100) ; make sure all else is printed
            (println "parent exiting ...\n")
            (set 'finished true))))) true))

; receiver child process of the message
(set 'B (spawn 'result
    (begin
        (set 'ppid (sys-info -4))
        (while true
            (until (receive ppid msg))
            (println msg)
            (unless (= msg (string "greetings from " A))
                (println "ERROR in proxy message: " msg)))) true))

(until finished (if (receive A msg) (eval msg))) ; proxy loop

(abort)
(exit)
```

Child process `A` sends three messages to `B`. As this cannot be done directly `A` sends `send` statements to the parent for evaluation. The statement:

```
(until (send pidB (string "greetings from " A)))
```

will be evaluated in the environment of the parent process. Even so the variables `A` and `B` are bound to `nil` in the sender process `A`, in the parent process they will be bound to the correct process ID numbers.

After sending the three messages, the statement:

```
(set 'finished true)
```

is sent to the parent process. Once evaluated, it will cause the `until` loop to finish.

For more details on `send` and `receive` and more examples see the Code Patterns document.

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).

```
(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*)
syntax: (series *exp-start func num-count*)

In the first syntax, `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.

When *num-count* is less than `1`, then `series` returns an empty list.

```
(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)
```

In the second syntax, `series` uses a function specified in *func* to transform the previous expression in to the next expression:

```
; embed the function Phi: f(x) = 1 / (1 + x)
; see also http://en.wikipedia.org/wiki/Golden_ratio

(series 1 (fn (x) (div (add 1 x))) 20)  →

(1 0.5 0.6666666 0.6 0.625 0.6153846 0.619047 0.6176470 0.6181818
 0.6179775 0.6180555 0.6180257 0.6180371 0.6180327 0.6180344
 0.6180338 0.6180340 0.6180339 0.6180339 0.6180339)

; pre-define the function

(define (oscillate x)
  (if (< x)
    (+ (- x) 1)
    (- (+ x 1)))
)

(series 1 oscillate 20)  →

(1 -2 3 -4 5 -6 7 -8 9 -10 11 -12 13 -14 15 -16 17 -18 19 -20)

; any data type is accepted as a start expression

(series "a" (fn (c) (char (inc (char c)))) 5) → ("a" "b" "c" "d" "e")

; dependency of the two previous values in this fibonacci generator
```

```
(let (x 1) (series x (fn (y) (+ x (swap y x))) 10))  → 

(1 2 3 5 8 13 21 34 55 89)
```

The first example shows a series converging to the *golden ratio, φ* (for any starting value). The second example shows how *func* can be defined previously for better readability of the `series` statement.

The `series` function also updates the internal list `$idx` index value, which can be used inside *func*.

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.

```
(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 or setf function eliminates the need to quote the variable symbol.

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 locale string and decimal point string in a list or `nil` if the requested change could not be performed.

```
; report current locale

(set-locale)

; set default locale of your platform and country
; return value shown when executing on German MS-Windows

(set-locale "")    → ("German_Germany.1252" ",")
(add 1,234 1,234)  → 2,468
```

By default, newLISP – if not enabled for UTF-8 – starts up with the POSIX C default locale. This guarantees that newLISP's behavior will be identical on any platform locale. On UTF-8 enabled versions of newLISP the locale of the current platform is chosen.

```
; after non-UTF-8 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 options valid on your platform can be found in the C include file `locale.h` and may be different on each platform. When no *int-category* is specified, `LC_ALL` is used to turn on all options for that locale.

Category	**Mac OS X, BSDs & MS Windows**
LC_ALL	0

LC_COLLATE	1
LC_CTYPE	2
LC_MONETARY	3
LC_NUMERIC	4
LC_TIME	5

The default C locale uses the decimal dot, but most others use a decimal comma.

```
; with the current locale "en_US.UTF-8", only change the decimal separator
; to German locale comma on  Mac OS X. LC_NUMERIC is 4 on most platforms

(set-locale) → ("en_US.UTF-8" ".")
(set-locale "de_DE.UTF-8" 4) → ("de_DE.UTF-8" ",")

; mixed locale shows country setting for each category, 4 has changed
(set-locale) → ("en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/de_DE.UTF-8/en_US.UTF-8/en_US.UTF-8" ",")
```

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-ref !

syntax: (set-ref *exp-key list exp-replacement* [*func-compare*])

Searches for *exp-key* in *list* and replaces the found element with *exp-replacement*. The *list* can be nested. The system variables `$it` contains the expression found and can be used in *exp-replacement*. The function returns the new modified *list*.

```
(set 'data '(fruits (apples 123 44) (oranges 1 5 3)))

(set-ref 'apples data 'Apples)  → (fruits (Apples 123 44) (oranges 1 5 3))

data → (fruits (Apples 123 44) (oranges 1 5 3)))
```

`data` could be the context identifier of a default function for passing lists by reference:

```
(set 'db:db '(fruits (apples 123 44) (oranges 1 5 3)))

(define (update ct key value)
        (set-ref key ct value))

(update db 'apples 'Apples)    → (fruits (Apples 123 44) (oranges 1 5 3))
(update db 'oranges 'Oranges)  → (fruits (Apples 123 44) (Oranges 1 5 3))

db:db → (fruits (Apples 123 44) (Oranges 1 5 3))
```

For examples on how to use *func-compare* see set-ref-all

For changing all occurrences of an element in a list use set-ref-all.

set-ref-all !

syntax: (set-ref-all *exp-key list exp-replacement* [*func-compare*])

Searches for *exp-key* in *list* and replaces each instance of the found element with *exp-replacement*. The *list* can be nested. The system variable `$it` contains the expression found and can be used in *exp-replacement*. The system variable `$count` contains the number of replacements made. The function returns the new modified *list*.

```
(set 'data '((monday (apples 20 30) (oranges 2 4 9)) (tuesday (apples 5) (oranges 32 1))))

(set-ref-all 'apples data "Apples")
 → ((monday ("Apples" 20 30) (oranges 2 4 9)) (tuesday ("Apples" 5) (oranges 32 1)))

$count → 2
```

Using the default functor in the `(list key)` pattern allows the list to be passed by reference to a user-defined function containing a `set-ref-all` statement. This would result in less memory usage and higher speeds in when doing replacements in large lists:

```
(set 'db:db '((monday (apples 20 30) (oranges 2 4 9)) (tuesday (apples 5) (oranges 32 1))))

(define (foo ctx)
        (set-ref-all 'apples ctx "Apples")
)

(foo db)
 → ((monday ("Apples" 20 30) (oranges 2 4 9)) (tuesday ("Apples" 5) (oranges 32 1)))
```

When evaluating `(foo db)`, the list in `db:db` will be passed by reference and `set-ref-all` will make the changes on the original, not on a copy of `db:db`.

Like with find, replace, ref and ref-all, complex searches can be expressed using match or unify in *func-compare*:

```
(set 'data '((monday (apples 20 30) (oranges 2 4 9)) (tuesday (apples 5) (oranges 32 1))))

(set-ref-all '(oranges *) data (list (first $it) (apply + (rest $it))) match)
    → ( ... (oranges 15) ... (oranges 33) ... )
```

The example sums all numbers found in records starting with the symbol `oranges`. The found items appear in `$it`

See also set-ref which replaces only the first element found.

setq setf !

syntax: (setq *place-1 exp-1* [*place-2 exp-2 ...*])

`setq` and `setf` work alike in newLISP and set the contents of a symbol, list, array or string or of a list, array or string place reference. Like set, `setq` and `setf` can take multiple argument pairs. Although both `setq` and `setf` point to the same built-in function internally, throughout this manual `setq` is used when setting a symbol reference and `setf` is used when setting list or array references.

```
(setq x 123)  → 123

; multiple arguments

(setq x 1 y 2 z 3)  → 3

x  → 1
y  → 2
z  → 3

; with nth or implicit indices
(setq L '(a b (c d) e f g))

(setf (L 1) 'B)      → B
; or the same
(setf (nth 1 L) 'B)
L                    → (a B (c d) e f g)

(setf (L 2 0) 'C)    → C
L                    → (a B (C d) e f g)

(setf (L 2) 'X)
L                    → (A B X e f g)

; with assoc
(setq L '((a 1) (b 2)))
(setf (assoc 'b L) '(b 3)) → (b 3)
L                          → ((a 1) (b 3))

; with lookup
(setf (lookup 'b L) 30) → 30
L                       → ((a 1) (b 30))

; several list accessors can be nested
(setq L '((a 1) (b 2)))

(push 'b (setf (assoc 'b l) '(b 4))) 'b) → b
L                                        →((a 1) (b b 4)))

; on strings
(set 's "NewISP")

(setf (s 0) "n") → "n"
s → "newISP"

(setf (s 3) "LI") → "LI"
s → "newLISP"
```

Often the new value set is dependent on the old value. `setf` can use the anaphoric system variable `$it` to refer to the old value inside the `setf` expression:

```
(setq L '((apples 4) (oranges 1)))
```

```
(setf (L 1 1) (+ $it 1)) → 2

L                        → ((apples 4) (oranges 2))

(set 's "NewLISP")

(setf (s 0) (lower-case $it)) → "n")

s → "newLISP"
```

sgn

syntax: (sgn *num*)
syntax: (sgn *num exp-1* [*exp-2* [*exp-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:

***x* > `0` : sgn(*x*) = 1**
***x* < `0` : sgn(*x*) = -1**
***x* = `0` : sgn(*x*) = `0`**

```
(sgn -3.5)  → -1
(sgn 0)     → 0
(sgn 123)   → 1
```

In the second syntax, the result of evaluating one of the optional expressions *exp-1*, *exp-2*, or *exp-3* is returned, instead of `-1`, `0`, or `1`. If *exp-n* is missing for the case triggered, then `nil` is returned.

```
(sgn x -1 0 1)         ; works like (sgn x)
(sgn x -1 1 1)         ; -1 for negative x all others 1
(sgn x nil true true)  ; nil for negative else true
(sgn x (abs x) 0)      ; (abs x) for x < 0, 0 for x = 0, else nil
```

Any expression or constant can be used for *exp-1*, *exp-2*, or *exp-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. When called without arguments, `share` requests a page of shared memory 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. This function is not available on OS/2.

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, string expression or any other expression (since v.10.1.0) supplied 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 any constant or expression (since v.10.1.0) written previously into the memory. 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, but not between independent processes.

Since v.10.1.0 size of share objects can exceed the shared memory pagesize of the operating system. For objects bigger than the pagesize, newLISP internally uses files for sharing. This requires a `/tmp` directory on Unix-like operating system and a `temp` directory in the root of the current disk drive on Win32 systems. On Unix-like systems this directory is present, on Win32 it may have to be created.

```
(set 'mem (share))

(share mem 123)  → 123
(share mem)      → 123

(share mem "hello world") → "hello world"
(share mem)               → "hello world"

(share mem true)  → true
(share mem)       → true

(share mem '(+ 1 2 3 4))  → (+ 1 2 3 4)
(share mem)               → (+ 1 2 3 4)

; expressions received can be evaluated (since v.10.1.0)
(eval (share mem))        → 10

(share nil mem)   → true  ; unmap only on Unix
```

Expression read from shared memory and evaluated, will be evaluated in the recipient's process environment.

Note that shared memory access between different processes should be synchronized using a semaphore. Simultaneous access to shared memory can crash the running process.

For a more comprehensive example of using shared memory in a multi process Linux/Unix application, see the file `example/prodcons.lsp` in the newLISP source distribution.

signal

syntax: (signal *int-signal sym-event-handler | func-event-handler*)
syntax: (signal *int-signal* "ignore" | "default" | "reset")
syntax: (signal *int-signal*)

Sets a user-defined handler in *sym-event-handler* for a signal specified in *int-signal* or sets to a function expression in *func-event-handler*.

A parameter following *int-signal* is not evaluated.

If no signal handler is specified any of the string constants `"ignore"`, `"default"` or `"reset"` can be specified in either lower or upper case or simply using the first letter of the option string. When signal setup with any of these three options has been successful, `true` is returned.

Using `"ignore"` will make newLISP ignore the signal. Using `"default"` will set the handler to the default handler of the underlying platform OS. The `"reset"` option will restore the handler to newLISP startup state.

On startup, newLISP either specifies an empty newLISP handler or a Ctrl-C handler for `SIGINT` and a `waitpipd(-1, 0, WNOHANG)` C-call for `SIGCHLD`.

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.

```
(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
; this will only work in an interactive terminal window
; and will not work in the newLISP-GS editor

ctrl-C has been pressed

; reset treatment of signal 2 to startup conditions

(signal SIGINT "reset")
```

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 using the function destroy:

```
(destroy 2035) → true
```

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 currently defined handler function or `nil`.

The user-defined signal handler can pass 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 [*exp-1* [*exp-2* ...]])

Evaluates one or more expressions in *exp-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.

```
(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.

```
(sin 1)                     → 0.8414709838
(set 'pi (mul 2 (acos 0)))  → 3.141592654
(sin (div pi 2))            → 1
```

sinh

syntax: (sinh *num-radians*)

Calculates the hyperbolic sine of *num-radians*. The hyperbolic sine is defined mathematically as: *(exp (x) - exp (-x)) / 2*. An overflow to `inf` may occur if *num-radians* is too large.

```
(sinh 1)     → 1.175201194
(sinh 10)    → 11013.23287
(sinh 1000)  → inf
(sub (tanh 1) (div (sinh 1) (cosh 1))) → 0
```

sleep

syntax: (sleep *num-milliseconds*)

Gives up CPU time to other processes for the amount of milliseconds specified in *num-milli-seconds*.

```
(sleep 1000)  ; sleeps 1 second
```

```
(sleep 0.5)   ; sleeps 500 micro seconds
```

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.

A sleep may be cut short by a finishing child process started with fork or spawn.

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 negative, slice will take the parameter as offset counting from the end and copy up to that offset. 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.

```
(slice '(a b c d e f) 3 2)   → (d e)
(slice '(a b c d e f) 2 -2)  → (c d)
(slice '(a b c d e f) 2)     → (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.

Note that slice always works on single 8-bit byte boundaries for offset and length numbers, even when running the UTF-8 enabled version of newLISP.

```
(slice "Hello World" 6 2)  → "Wo"
(slice "Hello World" 0 5)  → "Hello"
(slice "Hello World" 6)    → "World"
(slice "newLISP" -4 2)     → "LI"

; UTF-8 strings are converted to a list, then joined again

(join (slice (explode "ΩΨΧΦΥΤΣΣΡΠΟΞΝΜΛΚΙ0ΗΖΕΔΓΒΑ") 3 5))  → "ΦΥΤΣΣ"
```

Note that an *implicit slice* is available for lists. See the chapter Implicit rest and slice.

Be aware that slice 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* [*func-compare*])
syntax: (sort *array* [*func-compare*])

All members in *list* or *array* are sorted in ascending order. Anything may be sorted, regardless of the types. When members are themselves lists or arrays, each 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 or array and returning the sorted list or array. It is a stable binary merge-sort with approximately *O(n log2 n)* performance preserving the order of adjacent elements which are equal. When *func-compare* is used it must work with either `<=` or `>=` operators to be stable.

An optional comparison operator, user-defined function, or anonymous function can be supplied. The functor or operator can be given with or without a preceding quote.

```
(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)

(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.

```
(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.

spawn

syntax: (spawn *sym exp* [true])

Launches the evaluation of *exp* as a child process and immediately returns. The symbol in *sym* is quoted and receives the result of the evaluation when the function sync is executed. `spawn` is used to start parallel evaluation of expressions in concurrent processes. If newLISP is running on a multi-core CPU, the underlying operating system will distribute spawned processes onto different cores, thereby evaluating expressions in parallel and speeding up overall processing.

The optional `true` parameter must be set if send or receive is used to communicated with the child process spawned.

The function `spawn` is not available on Win32.

After successfully starting a child process, the `spawn` expression returns the process id of the forked process. The following examples shows how the calculation of a range of prime numbers can be split up in four sub ranges to speed up the calculation of the whole range:

```
; calculate primes in a range
(define (primes from to)
  (local (plist)
      (for (i from to)
          (if (= 1 (length (factor i)))
              (push i plist -1)))
      plist))

; start child processes
(set 'start (time-of-day))

(spawn 'p1 (primes 1 1000000))
(spawn 'p2 (primes 1000001 2000000))
(spawn 'p3 (primes 2000001 3000000))
(spawn 'p4 (primes 3000001 4000000))
```

```
; wait for a maximum of 60 seconds for all tasks to finish
(sync 60000) ; returns true if all finished in time
; p1, p2, p3 and p4 now each contain a lists of primes

(println "time spawn: " (- (time-of-day) start))
(println "time simple: " (time  (primes 1 4000000)))

(exit)
```

On a 1.83 Intel Core 2 Duo processor, the above example will finish after about 13 seconds. Calculating all primes using `(primes 1 4000000)` would take about 20 seconds.

The sync function will wait for all child processes to finish and receive the evaluation results in the symbols `p1` to `p4`. When all results are collected, `sync` will stop waiting and return `true`. When the time specified was insufficient , `sync` will return `nil` and another `sync` statement could be given to further wait and collect results. A short timeout time can be used to do other processing during waiting:

```
(spawn 'p1 (primes 1 1000000))
(spawn 'p2 (primes 1000001 2000000))
(spawn 'p3 (primes 2000001 3000000))
(spawn 'p4 (primes 3000001 4000000))

; print a dot after each 2 seconds of waiting
(until (sync 2000) (println "."))
```

`sync` when used without any parameters, will not wait but immediately return a list of pending child processes. For the `primes` example, the following `sync` expression could be used to watch the progress:

```
(spawn 'p1 (primes 1 1000000))
(spawn 'p2 (primes 1000001 2000000))
(spawn 'p3 (primes 2000001 3000000))
(spawn 'p4 (primes 3000001 4000000))

; show a list of pending process ids after each three-tenths of a second
(until (sync 300) (println (sync)))
```

A parameter of `-1` tells `sync` to wait for a very long time (~ 1193 hours). A better solution would be to wait for a maximum time, then abort all pending child processes:

```
(spawn 'p1 (primes 1 1000000))
(spawn 'p2 (primes 1000001 2000000))
(spawn 'p3 (primes 2000001 3000000))
(spawn 'p4 (primes 3000001 4000000))

; wait for one minute, then abort and
; report unfinished PIDs

(if (not (sync 60000))
    (begin
        (println "aborting unfinished: " (sync))
        (abort))
    (println "all finished successfully")
)
```

The three functions `spawn`, `sync` and `abort` are part of the Cilk API. The original implementation also does sophisticated scheduling and allocation of threaded tasks to multiple CPU cores. The newLISP implementation of the Cilk API lets the operating system of the underlying

platform handle process management. Internally, the API is implemented using the Unix libc functions `fork()`, `waitpid()` and `kill()`. Intercommunications between processes and child processes is done using the send and receive functions.

`spawn` can be called recursively from spawned subtasks:

```
(define (fibo n)
  (local (f1 f2)
    (if(< n 2) 1
       (begin
          (spawn 'f1 (fibo (- n 1)))
          (spawn 'f2 (fibo (- n 2)))
          (sync 10000)
          (+ f1 f2)))))

(fibo 7)  → 21
```

With `(fibo 7)` 41 processes will be generated. Although the above code shows the working of the Cilk API in a recursive application, it would not be practical, as the overhead required to spawn subtasks is much higher than the time saved through parallelization.

Since version 10.1 a send and receive message functions are available for communications between parent and child processes. Using these functions any data or expression of any size can be transferred. Additionally messaged expressions can be evaluated in the recipient's environment.

sqrt

syntax: (sqrt *num*)

Calculates the square root from the expression in *num* and returns the result.

```
(sqrt 10)  → 3.16227766
(sqrt 25)  → 5
```

starts-with

syntax: (starts-with *str str-key* [*num-option*])
syntax: (starts-with *list* [*exp*])

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.

```
(starts-with "this is useful" "this")       → true
(starts-with "this is useful" "THIS")       → nil
```

```
;; use regular expressions
(starts-with "this is useful" "THIS" 1)      → true
(starts-with "this is useful" "this|that" 0) → true
```

In the second version, `starts-with` checks to see if a list starts with the list element in *exp*. `true` or `nil` is returned depending on outcome.

```
(starts-with '(1 2 3 4 5) 1)            → true
(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.

stats

syntax: (stats *list-vector*)

The functions calculates statistical values of central tendency and distribution moments of values in *list-vector*. The following values are returned by `stats` in a list:

name	description
N	Number of values
mean	Mean of values
avdev	Average deviation from mean value
sdev	Standard deviation (population estimate)
var	Variance (population estimate)
skew	Skew of distribution
kurt	Kurtosis of distribution

The following example uses the list output from the `stats` expression as an argument for the format statement:

```
(set 'data '(90 100 130 150 180 200 220 300 350 400))

(println (format [text]
    N        = %5d
    mean     = %8.2f
    avdev    = %8.2f
    sdev     = %8.2f
    var      = %8.2f
    skew     = %8.2f
    kurt     = %8.2f
[/text] (stats data)))

; outputs the following

    N        =    10
    mean     =   212.00
    avdev    =    84.40
```

```
sdev     =   106.12
var      = 11262.22
skew     =     0.49
kurtosis =    -1.34
```

string

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

Translates into a string anything that results from evaluating *exp-1*—. If more than one expression is specified, the resulting strings are concatenated.

```
(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 containing zero bytes. 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.

```
(set 'var "hello")
(string? var)  → true
```

struct

syntax: (struct *symbol* [*str-data-type* ...])

The `struct` function can be used to define aggregate data types for usage with the extended syntax of import, pack and unpack. This allows importing functions which take C-language *struct* data types or pointers to these aggregate data types.

The following example illustrates the usage of `struct` together with the C data functions `localtime` and `asctime`. The `localtime` functions works similar to the built-in now function. The `asctime` function takes the numerical data output by `localtime` and formats these to readable text.

```
/* The C function prototypes for the functions to import */

struct tm * localtime(const time_t *clock);

char * asctime(const struct tm *timeptr);

/* the tm struct aggregating different time related values */

struct tm {
    int tm_sec;      /* seconds after the minute [0-60] */
    int tm_min;      /* minutes after the hour [0-59] */
    int tm_hour;     /* hours since midnight [0-23] */
    int tm_mday;     /* day of the month [1-31] */
    int tm_mon;      /* months since January [0-11] */
    int tm_year;     /* years since 1900 */
    int tm_wday;     /* days since Sunday [0-6] */
    int tm_yday;     /* days since January 1 [0-365] */
    int tm_isdst;    /* Daylight Savings Time flag */
    long tm_gmtoff;  /* offset from CUT in seconds */   /*** not on Windows ***/
    char *tm_zone;   /* timezone abbreviation */        /*** not on Windows ***/
};
```

Function import and definition of the structure data type in newLISP:

```
;; for pointers to structs always use void*
;; as a library use msvcrt.dll on Windows or libc.so on Unix.
;; The tm struct type is configured for Mac OSX and Linux.
;; On other OS the tm structure may be different

(import "libc.dylib" "asctime" "char*" "void*")
(import "libc.dylib" "localtime" "void*" "void*")

; definition of the struct
(struct 'tm "int" "int" "int" "int" "int" "int" "int" "int" "int" "long" "char*")


;; use import and struct

; todays date number (seconds after 1970 also called Unix epoch time)
(set 'today (date-value))  → 1324134913

;; the time value is passed by it's address
;; localtime retirns a pointer to a tm struct

(set 'ptr (localtime (address today))) → 2896219696

; unpack the tm struct  (7:15:13 on the 17th etc.)
(unpack tm ptr) → (13 15 7 17 11 111 6 350 0 -28800 "PST")
```

```
; transform to readable form
(asctime ptr) → "Sat Dec 17 07:15:13 2011\n"

; all in one statement does actually not use struct, pointers are passed directly
(asctime (localtime (address today))) → "Sat Dec 17 07:15:13 2011"

; same as the built-in date function
(date today) → "Sat Dec 17 07:15:13 2011"
```

Care must be taken to pass valid addresses to pointer parameters in imported functions or when passing address pointers to unpack. Invalid address pointers can crash newLISP or make it unstable.

`struct` definitions can be nested:

```
; the pair aggregate type
(struct 'pair "char" "char") → pair

; nested struct type
(struct 'comp "pair" "int")  → comp

; pack data using the extended pack syntax
; note the insertion of structure alignment bytes after the pair
(pack comp (pack pair 1 2) 3) → "\001\002\000\000\003\000\000\000"

; unpack reverses the process
(unpack comp "\001\002\000\000\003\000\000\000") → ((1 2) 3)
```

Nested structures are unpacked recursively.

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`.

```
(sub 10 8 0.25)  → 1.75
(sub 123)        → -123
```

swap !

syntax: (swap *place-1 place-2*)

The contents of the two places *place-1 and place-2* are swapped. A *place* can be the contents of an unquoted symbol or any list or array references expressed with nth, first, last or implicit indexing or places referenced by assoc or lookup.

`swap` is a destructive operation that changes the contents of the lists, arrays, or symbols involved.

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

(swap (first lst) (last lst)) → a
lst                           → (f b c d e a)

(set 'lst-b '(x y z))

(swap (lst 0) (lst-b -1)) → f
lst                       → (z b c d e a)
lst-b                     → (x y f)

(set 'A (array 2 3 (sequence 1 6)) → ((1 2 3) (4 5 6))

(swap (A 0) (A 1)) → (1 2 3)
A                  → ((4 5 6) (1 2 3))

(set 'x 1 'y 2)

(swap x y)  → 1
x  → 2
y  → 1

(set 'lst '((a 1 2 3) (b 10 20 30)))
(swap (lookup 'a lst -1) (lookup 'b lst 1))
lst → ((a 1 2 10) (b 3 20 30))

(swap (assoc 'a lst) (assoc 'b lst))
lst →  ((b 3 20 30) (a 1 2 10))
```

Any two places can be swept in the same or different objects.

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.

```
(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:

```
;; 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:

```
(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.

A syntax of the context function can also be used to create, set and evaluate symbols.

symbol?

syntax: (symbol? *exp*)

Evaluates the *exp* expression and returns `true` if the value is a symbol; otherwise, it returns `nil`.

```
(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.

```
(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.

sync

syntax: (sync *int-timeout* [*func-inlet*])
syntax: (sync)

When *int-timeout* in milliseconds is specified, `sync` waits for child processes launched with spawn to finish. Whenever a child process finishes, `sync` assigns the evaluation result of the spawned subtask to the symbol specified in the spawn statement. The `sync` returns `true` if all child processes have been processed or `nil` if the timeout value has been reached and more child processes are pending.

If `sync` additionally is given with an optional user-defined *inlet* function in *func-inlet*, this function will be called with the child process-id as argument whenever a spawned child process returns. *func-inlet* can contain either a lambda expression or a symbol which defines a function.

Without any parameter, sync returns a list of pending child process PIDs (process identifiers), for which results have not been processed yet.

The function sync is not available on Win32.

```
; wait for 10 seconds and process finished child processes
(sync 10000)

; wait for the maximum time (~ 1193 hours)
(sync -1)

(define (report pid)
    (println "process: " pid " has returned"))

; call the report function, when a child returns
(sync 10000 report) ; wait for 10 seconds max

; return a list of pending child processes
(sync)         → (245 246 247 248)

; wait and do something else
(until (true? (sync 10 report) )
    (println (time-of-day)))
```

When sync is given with a timeout parameter, it will block until timeout or until all child processes have returned, whichever comes earlier. When no parameter is specified or a function is specified, sync returns immediately.

The function sync is part of the Cilk API for synchronizing child processes and process parallelization. See the reference for the function spawn for a full discussion of the Cilk API.

sys-error

syntax: (sys-error)
syntax: (sys-error *int-error*)
syntax: (sys-error 0)

Reports the last error generated by the underlying OS which newLISP is running on. The error reported may differ on the platforms newLISP has been compiled for. Consult the platform's C library information. The error is reported as a list of error number and error text.

If no error has occurred or the system error number has been reset, nil is returned.

When *int-error* is greater 0 (zero) a list of the number and the error text is returned.

To reset the error specify 0 as the error number.

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.

```
;; trying to open a nonexistent file
(open "xyz" "r")  → nil

(sys-error)       → (2 "No such file or directory")

;; reset errno
(sys-error 0)     → (0 "Unknown error: 0")
(sys-error)       → nil
```

See also last-error and net-error.

sys-info

syntax: (sys-info [*int-idx*])

Calling `sys-info` without *int-idx* returns a list of internal resource statistics. Ten integers report the following status:

offset	description
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	Pid of the parent process or 0
7	Pid of running newLISP process
8	Version number as an integer constant
9	Operating system constant: linux=1, bsd=2, osx=3, solaris=4, win32=6, os/2=7, cygwin=8, tru64 unix=9, aix=10, android=11 bit 11 will be set for ffilib (extended import/callback API) versions (add 1024) bit 10 will be set for IPv6 versions (add 512) bit 9 will be set for 64-bit (changeable at runtime) versions (add 256) bit 8 will be set for UTF-8 versions (add 128) bit 7 will be added for library versions (add 64)

The numbers from `0` to `9` indicate the optional offset in the returned list.

It is recommended to use offsets 0 to 5 to address up and including "Maximum call stack constant" and to use negative offsets -1 to -4 to access the last four entries in the system info list. Future new entries will be inserted after offset 5. This way older source code does not need to change.

When using *int-idx*, one element of the list will be returned.

```
(sys-info)     → (429 268435456 402 1 0 2048 0 19453 10406 1155)
(sys-info 3)   → 1
(sys-info -2)  → 10406 ;; version 10.4.6
```

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.

t-test

syntax: (t-test *list-vector number-value*)
syntax: (t-test *list-vector-A list-vector-B* [`true`])
syntax: (t-test *list-vector-A list-vector-B float-probability*)

In the **first syntax** the function uses a one sample *Student's t* test to compare the mean value of *list-vector* to the value in *number-value*:

```
; one sample t-test
(t-test '(3 5 4 2 5 7 4 3) 2.5)
→ '(4.125 2.5 1.552 0.549 2.960 7 0.021)
```

The following data are returned in a list:

name	**description**
mean	mean of data in vector
value	value to compare
sdev	standard deviation in data vector
mean-error	standard error of mean
t	t between mean and value
df	degrees of freedom
p	two tailed probability of t under the null hypothesis

In above example the difference of the mean value `4.125` from `2.5` is moderately significant. With a probability `p = 0.021 (2.1%)` the null hypothesis that the mean is not significantly different, can be rejected.

In the **second syntax**, the function performs a t-test using the *Student's t* statistic for comparing the means values in *list-vector-A* and *list-vector-B*. If the `true` flag is not used, both vectors in A and B can be of different length and groups represented by A and B are not related.

When the optional flag is set to `true`, measurements were taken from the same group twice, e.g. before and after a procedure.

The following results are returned in a list:

name	description
mean-a	mean of group A
mean-b	mean of group B
sdev-a	standard deviation in group A
sdev-b	standard deviation in group B
t	t between mean values
df	degrees of freedom
p	two tailed probability of t under the null hypothesis

The first example studies the effect of different sleep length before a test on the SCAT (Sam's Cognitive Ability Test):

```
; SCAT (Sam's Cognitive Ability Test)
; two independent sample t-test
(set 'hours-sleep-8 '(5 7 5 3 5 3 3 9))
(set 'hours-sleep-4 '(8 1 4 6 6 4 1 2))

(t-test hours-sleep-8 hours-sleep-4)
 → (5 4 2.138 2.563 0.847 14 0.411)
```

The duration of sleeps before the SCAT does not have a significant effect with a probability value of `0.411`.

In the second example, the same group of people get tested twice, before and after a treatment with Prozac depression medication:

```
; Effect of an antidepressant on a group of depressed people
; two related samples t-test
(set 'mood-pre '(3 0 6 7 4 3 2 1 4))
(set 'mood-post '(5 1 5 7 10 9 7 11 8))

(t-test mood-pre mood-post true)
→ (3.333 7 2.236 3.041 -3.143 8 0.0137)
```

The effect of the antidepressant treatment is moderately significant with a `p` of `0.0137`.

In the **third syntax**, a form of the *Student's t* called *Welch's t-test* is performed. This method is used when the variances observed in both samples are significantly different. The threshold can be set using the *float-probability* parameter. When this parameter is used the `t-test` function will perform a F-test to compare the variances in the two data samples. If the probability of the found *F-ratio* is below the *float-probability* parameter, the *Welch's t-test* method will be used. Specifying this value as `1.0` effectively forces a *Welch's t-test*:

```
; two independent sample t-test using the Welch method
(t-test '(10 4 7 1 1 6 1 8 2 4) '(4 6 9 4 6 8 9 3) 1.0)
→ (4.4 6.125 3.239 2.357 -1.307 15 0.211)

; two independent sample t-test using the normal method
(t-test '(10 4 7 1 1 6 1 8 2 4) '(4 6 9 4 6 8 9 3))
→ (4.4 6.125 3.239 2.357 -1.260 16 0.226)
```

There is no significant difference between the means of the two samples. The *Welch* method of the t-test is slightly more sensitive in this case than using the normal t-test method.

Smaller values than `1.0` would trigger the *Welch's t-test* method only when the significance

of variance difference in the samples reaches certain value.

tan

syntax: (tan *num-radians*)

Calculates the tangent function from *num-radians* and returns the result.

```
(tan 1)                     → 1.557407725
(set 'pi (mul 2 (asin 1)))  → 3.141592654
(tan (div pi 4))            → 1
```

tanh

syntax: (tanh *num-radians*)

Calculates the hyperbolic tangent of *num-radians*. The hyperbolic tangent is defined mathematically as: *sinh (x) / cosh (x)*.

```
(tanh 1)     → 0.761594156
(tanh 10)    → 0.9999999959
(tanh 1000)  → 1
(= (tanh 1) (div (sinh 1) (cosh 1)))  → true
```

term

syntax: (term *symbol*)

Returns as a string, the term part of a *symbol* without the context prefix.

```
(set 'ACTX:var 123)
(set 'sm 'ACTX:var)
(string sm)     → "ACTX:var"
(term sm)      → "var"

(set 's 'foo:bar)
(= s (sym (term s) (prefix s)))
```

See also prefix to extract the namespace or context prefix from a symbol.

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`.

```
(define (throw-test)
    (dotimes (x 1000)
        (if (= x 500) (throw "interrupted"))))

(catch (throw-test) 'result)  → true

result  → "interrupted"

(catch (throw-test))  → "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 *exp*)

Causes a user-defined error exception with text provided by evaluating *exp*.

```
(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
ERR: 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 floating point milliseconds. Depending on the platform decimals of milliseconds are shown or not shown.

```
(time (myprog x y z))  → 450.340

(time (myprog x y z) 10)  → 4420.021
```

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* | *func-event-handler num-seconds* [*int-option*])
syntax: (timer *sym-event-handler* | *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-* or *func-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 process owning the timer. Option `2` 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-* or *func-event-handler*, `timer` returns the elapsed time of the timer in

progress. This can be used to program time lines or schedules.

`timer` called without arguments returns the symbol of the current event handler.

```
(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
```

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:

```
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
            (net-receive 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 utf8

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.

```
(title-case "hello")       → "Hello"
(title-case "hELLO" true)  → "Hello"
(title-case "hELLO")       → "HELLO"
```

See also the lower-case and upper-case functions.

trace

syntax: (trace [*bool*])

Tracing is switched on when *bool* evaluates to anything other than `nil`. 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 after entering the next user defined function, displaying the function and highlighting the current expression upon entry and exit.

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.

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.

```
;; 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 newLISP code where execution should switch on the debugger.

Use the debug function as a shortcut for the above example:

```
(debug (my-func a b c))
```

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.

```
;; 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. 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 matrix to transpose can contain any data-type.

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.

```
(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))

; any data type is allowed in the matrix
(transpose '((a b) (c d) (e f)))   → ((a c e) (b d f))

; arrays can be transposed too
(set 'A (array 2 3 (sequence 1 6)))
(set 'M (transpose A))
```

```
M → ((1 4) (2 5) (3 6))
```

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, mat and multiply.

trim utf8

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.

```
(trim "   hello  ")              → "hello"
(trim "----hello-----" "-")      → "hello"
(trim "00012340" "0" "")         → "12340"
(trim "1234000" "" "0")          → "1234"
(trim "----hello=====" "-" "=")  → "hello"
```

true?

syntax: (true? *exp*)

If the expression in *exp* evaluates to anything other than `nil` or the empty list `()`, `true?` returns `true`; otherwise, it returns `nil`.

```
(map true? '(x 1 "hi" (a b c) nil ()))
→ (true true true true nil nil)
```

```
(true? nil)  → nil
(true? '())  → nil
```

`true?` behaves like if and rejects the empty list `()`.

unicode

syntax: (unicode *str-utf8*)

Converts ASCII/UTF-8 character strings in *str* to UCS-4–encoded Unicode of 4-byte integers per character. The string is terminated with a 4-byte integer `0`. This function is only available on UTF-8–enabled versions of newLISP.

```
(unicode "new")
→ "n\000\000\000e\000\000\000w\000\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 *exp-1 exp-2* [*list-env*])

Evaluates and matches *exp-1* and *exp-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. newLISP uses a modified *J. Alan Robinson* unification algorithm with correctly applied *occurs check*. See also *Peter Norvig*'s paper about a common unification algorithm bug, which is not present in this implementation.

Since version 10.4.0 the underscore symbol _ (ASCII 95) matches any atom, list or unbound variable and never binds.

Like match, `unify` is frequently employed as a parameter functor in find, ref, ref-all and replace.

```
(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

(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

(unify '(A b _) '(x G z)) → ((A x) (G b)) ; _ matches atom z

(unify '(A b c _) '(x G _ z)) → ((A x) (G b)) ; _ never binds, matches c and z

(unify '(A b _) '(x G (x y z))) → ((A x) (G b)) ; _ matches list (x y z)

;; some examples taken from http://en.wikipedia.org/wiki/Unification_(computer_science)
```

`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:

```
(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`.

Use `unify` with `expand`

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)
```

Use `unify` with `bind` for de-structuring

The function bind can be used to set unified variables:

```
(bind (unify '(f (g A) A) '(f B xyz)))

A → xyz
B → (g xyz)
```

This can be used for de-structuring:

```
(set 'structure '((one "two") 3 (four (x y z))))
(set 'pattern '((A B) C (D E)))
(bind (unify pattern structure))

A → one
B → "two"
C → 3
D → four
E → (x y z)
```

`unify` returns an association list and `bind` binds the associations.

Model propositional logic with `unify`

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 trm)
    (or  (when (find trm facts) true) (catch (prove-rule trm))))
```

```
(define (prove-rule trm)
    (dolist (r rules)
        (when (list? (set 'e (unify trm (first r))))
            (when (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.

union

syntax: (union *list-1 list-2* [*list-3 ...*])

`union` returns a unique collection list of distinct elements found in two or more lists.

```
(union '(1 3 1 4 4 3) '(2 1 5 6 4))  →  (1 3 4 2 5 6)
```

Like the other set functions difference, intersect and unique, `union` maintains the order of elements as found in the original lists.

unique

syntax: (unique *list*)

Returns a unique version of *list* with all duplicates removed.

```
(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.

Other *set* functions are difference, intersect and union.

unless

syntax: (unless *exp-condition body*)

The statements in *body* are only evaluated if *exp-condition* evaluates to `nil` or the empty list `()`. The result of the last expression in *body* is returned or `nil` or the empty list `()` if *body* was not executed.

Because `unless` does not have an *else* condition as in if, the statements in *body* need not to be grouped with begin:

```
(unless (starts-with (read-line) "quit")
        (process (current-line))
        ...
        (finish)
)
```

See also the function when.

unpack

syntax: (unpack *str-format str-addr-packed*)
syntax: (unpack *str-format num-addr-packed*)

syntax: (unpack *struct num-addr-packed*)
syntax: (unpack *struct str-addr-packed*)

When the first parameter is a string, `unpack` unpacks a binary structure in *str-addr-packed* or pointed to by *num-addr-packed* into newLISP variables using the format in *str-format*. `unpack` is the reverse operation of `pack`. Using *num-addr-packed* facilitates the unpacking of

structures returned from imported, shared library functions.

If the number specified in *num-addr-packed* is not a valid memory address, a system bus error or segfault can occur and crash newLISP or leave it in an unstable state.

When the first parameter is the symbol of a struct definition, `unpack` uses the format as specified in *struct*. While `unpack` with *str-format* literally unpacks as specified, `unpack` with *struct* will skip structure aligning pad-bytes depending on data type, order of elements and CPU architecture. Refer to the description of the struct function for more detail.

The following characters may define a format:

format	**description**
`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
`Ld`	a signed 64-bit long number
`Lu`	an unsigned 64-bit long number
`f`	a float in 32-bit representation
`lf`	a double float in 64-bit representation
`sn`	a string of *n* null padded ASCII characters
`nn`	*n* null characters
`>`	switches to big endian byte order
`<`	switches to little endian byte order

```
(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.

If the buffer's size at a memory address is smaller than the formatting string specifies, some formatting characters may be left unused.

See also the address, get-int, get-long, get-char, get-string, and pack functions.

until

syntax: (until *exp-condition* [*body*])

Evaluates the condition in *exp-condition*. 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 in *body* is the return value of the `until` expression. If *body* is empty, the result of last *exp-condition* is returned. `until` works like (while (not ...)).

`until` also updates the system iterator symbol `$idx`.

```
(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 utf8

syntax: (upper-case *str*)

Returns a copy of the string in *str* converted to uppercase. International characters are converted correctly.

```
(upper-case "hello world")  → "HELLO WORLD"
```

See also the lower-case and title-case functions.

utf8

syntax: (utf8 *str-unicode*)

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.

```
(unicode "new")
→ "n\000\000\000e\000\000\000w\000\000\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.

utf8len

syntax: (utf8len *str*)

Returns the number of characters in a UTF-8–encoded string. UTF-8 characters can be encoded in more than one 8-bit byte. `utf8len` returns the number of UTF-8 characters in a string. This function is only available on UTF-8–enabled versions of newLISP.

```
(utf8len "我能吞下玻璃而不伤身体。")    → 12
(length "我能吞下玻璃而不伤身体。")      → 36
```

See also the unicode and utf8 functions. Above Chinese text from UTF-8 Sampler.

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.

```
;; 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 system-wide 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* | `nil`])

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 list of pid and status value is returned. The status value describes the reason for termination of the child process. 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*, pid and status information of any child process started by the parent are 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*.

An option can be specified in *int-option*. See Linux/Unix documentation for details on integer values for *int-options*. As an alternative, `nil` can be specified. This option causes `wait-pid` to be non-blocking, returning right away with a `0` in the pid of the list returned. This option used together with an *int-pid* parameter of `-1` can be used to continuously loop and act on returned child processes.

This function is only available on Mac OS X, Linux and other Unix-like operating systems.

```
(set 'pid (fork (my-process))) → 8596
```

```
(set 'ret (wait-pid pid))  → (8596 0) ; child has exited

(println "process: " pid " has finished with status: " (last ret))
```

The process `my-process` is started, then the main program blocks in the `wait-pid` call until `my-process` has finished.

when

syntax: (when *exp-condition body*)

The statements in *body* are only evaluated if *exp-condition* evaluates to anything not `nil` and not the empty list `()`. The result of the last expression in *body* is returned or `nil` or the empty list `()` if *body* was not executed.

Because `when` does not have an *else* condition as in if, the statements in *body* need not to be grouped with begin:

```
(when (read-line)
        (set 'result (analyze (current-line)))
        (report result)
        (finish)
)
```

See also the function unless working like `(when (not ...) ...)`.

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 evaluated expression is the return value of the `while` expression.

`while` also updates the system iterator symbol `$idx`.

```
(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 !

syntax: (write)
syntax: (write *int-file str-buffer* [*int-size*])
syntax: (write *str str-buffer* [*int-size*])

In the second syntax `write` writes *int-size* bytes from a buffer in *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` returns the number of bytes written or `nil` on failure.

If all parameters are omitted, `write` writes the contents from the last read-line to standard out (STDOUT).

`write` is a shorter writing of `write-buffer`. The longer form still works but is deprecated and should be avoided in new code.

```
(set 'handle (open "myfile.ext" "write"))
(write handle data 100)
(write handle "a quick message\n")
```

The code in the example writes 100 bytes to the file `myfile.ext` from the contents in `data`.

In the third syntax, `write` can be used for destructive string appending:

```
(set 'str "")
(write str "hello world")

str   → "hello world"
```

See also the read function.

write-char

syntax: (write-char *int-file int-byte1* [*int-byte2 ...*])

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 8-bit byte.

`write-char` returns the number of bytes written.

```
(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 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.

On failure the function returns `nil`. For error information, use sys-error when used on files. When used on URLs net-error gives more error information.

```
(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*. When the prefix `http://` is used, `write-file` works exactly like put-url and can take the same additional parameters:

```
(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 [*int-file* [*str*]])
syntax: (write-line *str-out* [*str*])

The string in *str* and the line termination character(s) are written to the device specified in *int-file*. When the string argument is omitted `write-line` writes the contents of the last read-line to *int-file* If the first argument is omitted too then it writes to to standard out (STDOUT) or to whatever device is set by device.

In the second syntax lines are appended to a string in *str-out*.

`write-line` returns the number of bytes written.

```
(set 'out-file (open "myfile" "write"))
(write-line out-file "hello there")
(close out-file)

(set 'myFile (open "init.lsp" "read")
(while (read-line myFile) (write-line))

(set 'str "")
(write-line str "hello")
(write-line str "world")

str  →  "hello\nworld\n"
```

The first example opens/creates a file, writes a line to it, and closes the file. The second example shows the usage of `write-line` without arguments. The contents of `init.lsp` are written to the console screen.

See also the function write for writing to a device without the line-terminating character.

xfer-event

syntax: (xfer-event *sym-event-handler* | *func-event-handler*)

Registers a function in symbol *sym-event-handler* or in lambda function *func-event-handler* to monitor HTTP byte transfers initiated by get-url, post-url or put-url or initiated by file functions which can take URLs like load, save, read-file, write-file and append-file.

E.g. whenever a block of data requested with get-url arrives, the function in *sym* or *func* will be called with the number of bytes transferred. Likewise when sending data with post-url or any of the other data sending functions, *sym* or *func* will be called with the number of bytes transferred for each block of data transferred.

```
(xfer-event (fn (n) (println "->" n)))
(length (get-url "http://newlisp.org"))

->73
->799
->1452
->351
->1093
->352
->211
->885
->564
->884
->561
->75
->812
->638
->1452
->801
->5
->927
11935
```

The computer output is shown in bold. Whenever a block of data is received its byte size is printed. Instead of defining the handler function directory with a lambda function in *func*, a symbol containing a function definition could have been used:

```
(define (report n) (println "->" n))
(xfer-event 'report)
```

This can be used to monitor the progress of longer lasting byte transfers in HTTP uploads or downloads.

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).

```
(xml-parse "<atag>hello</atag><fin")  → nil

(xml-error)  → ("expected closing tag: >" 18)
```

xml-parse

syntax: (xml-parse *string-xml* [*int-options* [*sym-context* [*func-callback*]]])

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.

```
(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:

option	description
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")))
```

```
    ("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")))
    ("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")))
    ("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, empty attribute lists and comments with option (+ 1 2 4):

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`, `2`, `4`, `8`, and `16`—SXML formatted output can be generated:

```
(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"))
    (FRUIT (NAME "orange") (COLOR "orange") (PRICE "1.00"))
    (FRUIT (NAME "banana") (COLOR "yellow") (PRICE "0.60"))))
```

If the original XML tags contain a namespace part separated by a `:`, that colon will be translated into a `.` dot in the resulting newLISP symbol.

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.

Parsing into a specified context

When parsing XML expressions, XML tags are translated into newLISP symbols, when option 8 is specified. The *sym-context* option specifies the target context for the symbol creation:

```
(xml-type-tags nil nil nil nil)
(xml-parse (read-file "example.xml") (+ 1 2 4 8 16) 'CTX)
→((CTX:DATABASE (@ (CTX:name "example.xml"))
    (CTX:FRUIT (CTX:NAME "apple") (CTX:COLOR "red") (CTX:PRICE "0.80"))
    (CTX:FRUIT (CTX:NAME "orange") (CTX:COLOR "orange") (CTX:PRICE "1.00"))
    (CTX:FRUIT (CTX:NAME "banana") (CTX:COLOR "yellow") (CTX:PRICE "0.60"))))
```

If the context does not exist, it will be created. If it exists, the quote can be omitted or the context can be referred to by a variable.

Using a call back function

Normally, `xml-parse` will not return until all parsing has finished. Using the *func-callback* option, `xml-parse` will call back after each tag closing with the generated S-expression and a

start position and length in the source XML:

```
;; demo callback feature
(define (xml-callback s-expr start size)
    (if (or (= (s-expr 0) 'NAME) (= (s-expr 0) 'COLOR) (= (s-expr 0) 'PRICE))
        (begin
            (print "parsed expression:" s-expr)
            (println ", source:" (start size example-xml))
        )
    )
)

(xml-type-tags nil 'cdata '!-- nil)
(xml-parse  (read-file "example.xml") (+ 1 2 8) MAIN xml-callback)
```

The following output will be generated by the callback function `xml-callback`:

```
parsed expression:(NAME "apple"), source:<NAME>apple</NAME>
parsed expression:(COLOR "red"), source:<COLOR>red</COLOR>
parsed expression:(PRICE "0.80"), source:<PRICE>0.80</PRICE>
parsed expression:(NAME "orange"), source:<NAME>orange</NAME>
parsed expression:(COLOR "orange"), source:<COLOR>orange</COLOR>
parsed expression:(PRICE "1.00"), source:<PRICE>1.00</PRICE>
parsed expression:(NAME "banana"), source:<NAME>banana</NAME>
parsed expression:(COLOR "yellow"), source:<COLOR>yellow</COLOR>
parsed expression:(PRICE "0.60"), source:<PRICE>0.60</PRICE>
```

The example callback handler function filters the tags of interest and processes them as they occur.

xml-type-tags

syntax: (xml-type-tags [*exp-text-tag exp-cdata-tag exp-comment-tag exp-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:

```
(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? bigint

syntax: (zero? *exp*)

Checks the evaluation of *exp* to see if it equals `0` (zero).

```
(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)

(map zero? '(nil true 0 0.0 "" ()))  → (nil nil true true nil nil)
```

`zero?` will return `nil` on data types other than numbers.

(∂)

---

newLISP APPENDIX

Error codes

description	no
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

invalid syntax	57
user error	58
user reset -	59
received SIGINT -	60
function is not reentrant	61
local symbol is protected	62
no reference found	63
list is empty	64
I/O error	65
working directory not found	66
invalid PID	67
cannot open socket pair	68
cannot fork process	69
no comm channel found	70

Error codes extended FFI

description	no
ffi preparation failed	71
invalid ffi type	72
ffi struct expected	73

Big Integers

description	no
bigint type not applicable	74
not a number or infinite	75

TCP/IP and UDP Error Codes

no	description
1	Cannot open socket
2	DNS resolution failed
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
17 Cannot unblock socket
18 Operation timed out
19 HTTP bad formed URL
20 HTTP file operation failed
21 HTTP transfer failed
22 HTTP invalid response from server
23 HTTP no response from server
24 HTTP document empty
25 HTTP error in header
26 HTTP error in chunked format

System Symbols and Constants

Variables changed by the system

newLISP maintains several internal symbol variables. All of them are global and can be used by the programmer. Some have write protection, others are user settable. Some will change when used in a sub-expression of the enclosing expression using it. Others are safe when using reentrant in nested functions or expressions.

All symbols starting with the `$` character will not be serialized when using the save or source functions.

variable name	purpose	protected	reentrant
`$0 - $15`	Used primarily in regular expressions. `$0` is also used to record the last state or count of execution of some functions.	no	no
`$args`	Contains the list parameters not bound to local variables. Normally the function args is used to retrieve the contents of this variable.	yes	yes

`$count`	The count of elements matching when using find-all, replace, ref-all and set-ref-all or the count of characters processed by read-expr.	yes	no
`$idx`	The function dolist maintains this as a list index or offset. The functions map, series, while, until, do-while and do-until maintain this variable as an iteration counter starting with 0 (zero) for the first iteration.	yes	yes
`$it`	The *anaphoric* `$it` refers to the result inside an executing expression, i.e. in self referential assignments. `$it` is only available inside the function expression setting it, and is set to `nil` on exit of that expression. The following functions use it: if, hashes, find-all, replace, set-ref, set-ref-all and setf setq.	yes	no
`$main-args`	Contains the list of command line arguments passed by the OS to newLISP when it was started. Normally the function main-args is used to retrieve the contents.	yes	n/a

Predefined variables and functions.

These are preset symbol constants. Two of them are used as namespace templates, one two write platform independent code.

name	purpose	protected	reentrant
`Class`	Is the predefined general FOOP class constructor which can be used together with `new` to create new FOOP classes, e.g: `(new Class 'Rectangle)` would create a class and object constructor for a user class `Rectangle`. See the FOOP classes and constructors chapter in the users manual for details.	no	n/a
`ostype`	Contains a string identifying the OS-Platform for which the running newLISP version has been compiled. See the reference section for details	yes	n/a
`Tree`	Is a predefined namespace to serve as a hash like dictionary. Instead of writing `(define Foo:Foo)` to create a `Foo` dictionary, the expression `(new Tree 'Foo)` can be used as well. See the chapter Hash functions and dictionaries foe details.	no	n/a
`module`	Is a predefined function to load modules. Instead of using `load` together with the `NEWLISPDIR` environment variable, the `module` function loads automatically from `$NEWLISPDIR/modules/`.	no	n/a

The symbols `Class`, `Tree` and `module` are predefined as follows:

```
; built-in template for FOOP constructors
```

```
; usage: (new Class 'MyClass)
(define (Class:Class)
    (cons (context) (args)))

; built-in template for hashes
; usage: (new Tree 'MyHash)
(context 'Tree)
  (constant 'Tree:Tree)
(context MAIN)"

; load modules from standard path
; usage (module "mymodule.lsp")
(define (module $x)
  (load (append (env "NEWLISPDIR") "/modules/" $x)))

(global 'module)
```

These symbols are not protected and can be redefined by the user. The $x variable is built-in and protected against deletion. This $x variable is also used in curry expressions.

(∂)

GNU Free Documentation License

Version 1.2, November 2002

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. Secondarily, 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 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. http://fsf.org/ Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of

works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. 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 them 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 prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. 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.

Developers that use the GNU GPL protect your rights with two steps:

(1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey 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; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the

material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program.

Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If 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 convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special

requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU 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 that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

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.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS 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.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local

legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

(∂)