Code Patterns in newLISP®

Version 2014 November 3rd
newLISP v.10.6.2

§

1. Introduction

When programming in newLISP, certain functions and usage patterns occur repeatedly. For some problems, an optimal way to solve them evolves over time. The following chapters present example code and explanations for the solution of specific problems when programming in newLISP.

Some content is overlapping with material covered in the newLISP Users Manual and Reference or presented here with a different slant.

Only a subset of newLISP's total function repertoire is used here. Some functions demonstrated have additional calling patterns or applications not mentioned on these pages.

This collection of patterns and solutions is a work in progress. Over time,

material will be added or existing material improved.

§

2. newLISP script files

Command line options

On Linux/Unix, put the following in the first line of the script/program file:

```
#!/usr/bin/newlisp
```

specifying a bigger stack:

```
#!/usr/bin/newlisp -s 100000
```

or

```
#!/usr/bin/newlisp -s100000
```

Operating systems' shells behave differently when parsing the first line and extracting parameters. newLISP takes both attached or detached parameters. Put the following lines in small script to test the behavior of the underlying OS and platform. The script changes the stack size allocated to 100,000 and limits newLISP cell memory to about 10 M bytes.

```
#!/usr/bin/newlisp -s 100000 -m 10

(println (main-args))
(println (sys-info))

(exit) ; important
```

A typical output executing the script from the system shell would be:

```
./arg-test

("/usr/bin/newlisp" "-s" "100000" "-m" "10" "./arg-test")
(308 655360 299 2 0 100000 8410 2)
```

Note that few programs in newLISP need a bigger stack configured; most programs run on the internal default of 2048. Each stack position takes an average of 80 bytes. Other options are available to start newLISP. See the Users Manual for details.

Scripts as pipes

The following example shows how a file can be piped into a newLISP script.

```
#!/usr/bin/newlisp
#
# uppercase - demo filter script as pipe
#
# usage:
#          ./uppercase < file-spec
#
# example:
#          ./uppercase < my-text
#
#

(while (read-line) (println (upper-case (current-line))))

(exit)
```

The file will be printed to `std-out` translated to uppercase.

The following program would also work with binary non-textual information containing `0`'s :

```
#!/usr/bin/newlisp
;
; inout - demo binary pipe
;
; read from stdin into buffer
; then write to stdout
;
; usage: ./inout < inputfile > outputfile
;

(while (read 0 buffer 1024)
    (write 1 buffer 1024))

(exit)
```

Set buffersize to best performance.

File filters

The following script works like a Unix `grep` utility iterating through files and filtering each line in a file using a regular expression pattern.

```
#!/usr/bin/newlisp
#
# nlgrep - grep utility on newLISP
#
# usage:
#          ./nlgrep "regex-pattern" file-spec
#
# file spec can contain globbing characters
```

```
#
# example:
#          ./nlgrep "this|that" *.c
#
# will print all lines containing 'this' or 'that' in *.c files
#

(dolist (fname (3 (main-args)))
    (set 'file (open fname "read"))
    (println "file ---> " fname)
    (while (read-line file)
        (if (find (main-args 2) (current-line) 0)
            (write-line)))
    (close file))

(exit)
```

The expression:

```
(3 (main-args))
```

is a short form of writing:

```
(rest (rest (rest (main-args))))
```

It returns a list of all the filenames. This form of specifying indexes for rest is called implicit indexing. See the Users Manual for implicit indexing with other functions. The expression `(main-args 2)` extracts the 3rd argument from the command line containing the regular expression pattern.

newLISP as a pipe

Pipe one-liners directly into the executable for evaluation of short expressions:

```
~> echo '(+ 1 2 3)' | newlisp
6
~>
```

§

3. Writing software in modules

Structuring an application

When writing bigger applications or when several programmers are working on the same code base, it is necessary to divide the code base into modules.

Modules in newLISP are implemented using contexts, which are namespaces. Namespaces allow lexical isolation between modules. Variables of the same name in one module cannot clash with variables of the same name in another module.

Typically, modules are organized in one context per file. One file module may contain database access routines.

```
; database.lsp
;
(context 'db)


(define (update x y z)
...
)

(define (erase x y z)
...
)
```

Another module may contain various utilities

```
; auxiliary.lsp
;
(context 'aux)

(define (getval a b)
...
)
```

Typically, there will be one MAIN module that loads and controls all others:

```
; application.lsp
;

(load "auxiliary.lsp")
(load "database.lsp")

(define (run)
    (db:update ....)
    (aux:putval ...)
    ...
    ...
)

(run)
```

More than one context per file

When using more than one context per file, each context section should be closed with a `(context MAIN)` statement:

```
; myapp.lsp
;
(context 'A)

(define (foo ...) ...)

(context MAIN)

(context 'B)

(define (bar ...) ...)

(context MAIN)

(define (main-func)
    (A:foo ...)
    (B:bar ...)
)
```

Note that in the namespace statements for contexts `A` and `B` that the context names are quoted because they are newly created, but `MAIN` can stay unquoted because it already exists when newLISP starts up. However, quoting it does not present a problem.

The line `(context MAIN)` that closes a context can be omitted by using the following technique:

```
; myapp.lsp
;
(context 'A)

(define (foo ...) ...)

(context 'MAIN:B)

(define (bar ...) ...)

(context 'MAIN)

(define (main-func)
    (A:foo ...)
    (B:bar ...)
)
```

The line `(context 'MAIN:B)` switches back to MAIN then opens the new context `B`.

The default function

A function in a context may have the same name as the host context itself. This function has special characteristics:

```
(context 'foo)
```

```
(define (foo:foo a b c)
...
)
```

The function `foo:foo` is called the `default function`, because when using the context name `foo` like a function, it will default to `foo:foo`

```
(foo x y z)
; same as
(foo:foo x y z)
```

The default function makes it possible to write functions which look like normal functions but carry their own lexical namespace. We can use this to write functions which keep state:

```
(context 'generator)

(define (generator:generator)
    (inc acc)) ; when acc is nil, assumes 0

(context MAIN)

(generator) → 1
(generator) → 2
(generator) → 3
```

The following is a more complex example for a function generating a Fibonacci sequence:

```
(define (fibo:fibo)
    (if (not fibo:mem) (set 'fibo:mem '(0 1)))
    (last (push (+ (fibo:mem -1) (fibo:mem -2)) fibo:mem -1)))

(fibo) → 1
(fibo) → 2
(fibo) → 3
(fibo) → 5
(fibo) → 8
...
```

This example also shows how a default function is defined `on-the-fly` without the need of explicit `context` statements. As an alternative, the function could also have been written so that the context is created explicitly:

```
(context 'fibo)
(define (fibo:fibo)
        (if (not mem) (set 'mem '(0 1)))
        (last (push (+ (mem -1) (mem -2)) mem -1)))
(context MAIN)

(fibo) → 1
(fibo) → 2
```

```
(fibo) → 3
(fibo) → 5
(fibo) → 8
```

Although the first form is shorter, the second form is more readable.

Packaging data with contexts

The previous examples already presented functions packaged with data in a namespace. In the `generator` example the `acc` variable kept state. In the `fibo` example the variable `mem` kept a growing list. In both cases, functions and data are living together in a namespace. The following example shows how a namespace holds only data in a default functor:

```
(set 'db:db '(a "b" (c d) 1 2 3 x y z))
```

Just like we used the default function to refer to `fibo` and `generator` we can refer to the list in `db:db` by only using `db`. This will work in all situations where we do list indexing:

```
(db 0)    → a
(db 1)    → "b"
(db 2 1)  → d
(db -1)   → z
(db -3)   → x

(3 db)    → (1 2 3 x y z)
(2 1 db)  → ((c d))
(-6 2 db) → (1 2)
```

Passing objects by reference

When the default functor is used as an argument in a user defined function, the default functor is passed by reference. This means that a reference to the original contents is passed, not a copy of the list or string. This is useful when handling large lists or strings:

```
(define (update data idx expr)
    (if (not (or (lambda? expr) (primitive? expr)))
        (setf (data idx) expr)
        (setf (data idx) (expr $it))))

(update db 0 99) → a
db:db → (99 "b" (c d) 1 2 3 x y z)

(update db 1 upper-case) → "b"
db:db → (99 "B" (c d) 1 2 3 x y z)

(update db 4 (fn (x) (mul 1.1 x))) →
db:db → (99 "B" (c d) 1 2.2 3 x y z)
```

The data in `db:db` is passed via the `update` function parameter `data`, which now holds a reference to the context `db`. The `expr` parameter passed is checked to determine if it is a built-in function, operator or a user defined lambda expression and then works on `$it`, the anaphoric system variable containing the old content referenced by `(data idx)`.

Whenever a function in newLISP asks for a string or list in a parameter, a default functor can be passed by its context symbol. Another example:

```
(define (pop-last data)
(pop data -1))

(pop-last db) → z

db:db         → (99 "B" (c d) 1 2.2 3 x y)
```

The function `update` is also a good example of how to pass operators or functions as a function argument (`upper-case` working on `$it`). Read more about this in the chapter Functions as data.

§

4. Local variables

Locals in looping functions

All looping functions like `doargs`, `dolist`, `dostring`, `dotimes`, `dotree` and `for` use local variables. During loop execution, the variable takes different values. But after leaving the looping function, the variable regains its old value. `let`, `define`, and `lambda` expressions are another method for making variables local:

Locals in `let`, `letn`, `local` and `letex`

`let` is the usual way in newLISP to declare symbols as local to a block.

```
(define (sum-sq a b)
    (let ((x (* a a)) (y (* b b)))
        (+ x y)))

(sum-sq 3 4) → 25

; alternative syntax
(define (sum-sq a b)
    (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 ...])
```

When initializing several parameters, a nested `let`, `letn` can be used to reference previously initialized variables in subsequent initializer expressions:

```
(letn ((x 1) (y (+ x 1)))
    (list x y))              → (1 2)
```

`local` works the same way but variables are initialized to `nil`

```
(local (a b c)
   ...          ; expressions using the locale variables a b c
)
```

`letex` works similar to `let` but variables are expanded in the body to values assigned.

```
; assign to local variable and expand in body

(letex ( (x 1) (y '(a b c)) (z "hello") ) '(x y z))
→ (1 (a b c) "hello")

; as in let, parentheses around the initializers can be omitted

(letex (x 1 y 2 z 3) '(x y z))    → (1 2 3)
```

After exiting any of the `let`, `letn`, `local` or `letex` expressions, the variable symbols used as locals get their old values back.

Unused parameters as locals

In newLISP, all parameters in user defined functions are optional. Unused parameters are filled with `nil` and are of local scope to the dynamic scope of the function. Defining a user function with more parameters than required is a convenient method to create local variable symbols:

```
(define (sum-sq a b , x y)
    (set 'x (* a a))
    (set 'y (* b b))
    (+ x y))
```

The comma is not a special syntax feature but only a visual helper to separate normal parameters from local variable symbols. (Technically, the comma, like x and y, is a local variable and is set to nil.)

Default variable values

In the definition of a function default values can be specified:

```
(define (foo (a 1) (b 2))
    (list a b))

    (foo)      →  (1 2)
    (foo 3)    →  (3 2)
    (foo 3 4)  →  (3 4)
```

`args` as local substitute

Using the `args` function no parameter symbols need to be used at all and `args` returns a list of all parameters passed but not taken by declared parameters:

```
(define (foo)
    (args))

(foo 1 2 3)   → (1 2 3)


(define (foo a b)
    (args))

(foo 1 2 3 4 5)   → (3 4 5)
```

The second example shows how `args` only contains the list of arguments not bound by the variable symbols `a` and `b`.

Indices can be used to access members of the `(args)` list:

```
(define (foo)
      (+ (args 0) (args 1)))

(foo 3 4)   → 7
```

`args` and `local` used together for named variables

```
(define-macro (foo)
   (local (len width height)
      (bind (args) true)
      (println "len:" len " width:" width " height:" height)
   ))

(foo (width 20) (height 30) (len 10))
```

```
len:10 width:20 height:30
```

`local` will shadow / protect the values of the variables `len`, `width` and `height` at higher dynamic scoping levels.

§

5. Walking through lists and data

Recursion or iteration?

Although recursion is a powerful feature to express many algorithms in a readable form, it can also be inefficient in some instances. newLISP has many iterative constructs and high level functions like `flat` or the built-in XML functions, which use recursion internally. In many cases this makes defining a recursive algorithm unnecessary.

Some times a non-recursive solution can be much faster and lighter on system resources.

```
; classic recursion
; slow and resource hungry
(define (fib n)
    (if (< n 2) 1
        (+  (fib (- n 1))
            (fib (- n 2)))))
```

The recursive solution is slow because of the frequent calling overhead. Also, the recursive solution uses a lot of memory for holding intermediate and frequently redundant results.

```
; iteration
; fast and also returns the whole list
(define (fibo n , f)
    (set 'f '(1 0))
    (dotimes (i n)
         (push (+ (f 0) (f 1)) f)) )
```

The iterative solution is fast and uses very little memory.

Speed up with memoization

A memoizing function caches results for faster retrieval when called with the same parameters again. The following function makes a memoizing function from any built-in or user defined function with an arbitrary number of arguments. A namespace is created for the memoizing function as a data cache.

```
; speed up a recursive function using memoization
(define-macro (memoize mem-func func)
    (set (sym mem-func mem-func)
```

```
        (letex (f func  c mem-func)
          (lambda ()
              (or (context c (string (args)))
              (context c (string (args)) (apply f (args))))))))

(define (fibo n)
    (if (< n 2)  1
        (+  (fibo (- n 1))
            (fibo (- n 2)))))

(memoize fibo-m fibo)

(time (fibo-m 25)) → 148
(time (fibo-m 25)) → 0
```

The function creates a context and `default` function for the original function with a new name and stores all results in symbols in the same context.

When memoizing recursive functions, include the raw lambda specification of the function so recursive calls are memoized too:

```
(memoize fibo
    (lambda (n)
        (if(< n 2) 1
            (+  (fibo (- n 1))
                (fibo (- n 2))))))

(time (fibo 100)) → 1
(fibo 80)         → 37889062373143906
```

The `fibo` function in the last example would take hours to calculate without memoization. The memoized version takes only about a milli-second for an argument of `100`.

Walking a tree

Tree walks are a typical pattern in traditional LISP and in newLISP as well for walking through a nested list. But many times a tree walk is only used to iterate through all elements of an existing tree or nested list. In this case the built-in `flat` function is much faster than using recursion:

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

; classic car/cdr and recursion
;
(define (walk-tree tree)
    (cond ((= tree '()) true)
          ((atom? (first tree))
             (println (first tree))
             (walk-tree (rest tree)))
          (true
             (walk-tree (first tree))
```

```
             (walk-tree (rest tree)))))

; classic recursion
; 3 times faster
;
(define (walk-tree tree)
    (dolist (elmnt tree)
        (if (list? elmnt)
            (walk-tree elmnt)
            (println elmnt))))

(walk-tree L) →
     a
     b
     c
     d
     e
     ...
```

Using the built-in `flat` in newLISP a nested list can be transformed into a flat list. Now the list can be processed with a `dolist` or `map`:

```
; fast and short using 'flat'
; 30 times faster with map
;
(map println (flat L))

; same as

(dolist (item (flat L)) (println item))
```

Walking a directory tree

Walking a directory tree is a task where recursion works well:

```
; walks a disk directory and prints all path-file names
;
(define (show-tree dir)
    (when (directory? dir)
        (dolist (nde (directory dir))
            (if (and (directory? (append dir "/" nde))
                     (!= nde ".") (!= nde ".."))
                (show-tree (append dir "/" nde))
                (println (append dir "/" nde))))))
```

In this example recursion is the only solution, because the entire nested list of files is not available when the function is called but gets created recursively during function execution.

§

6. Modifying and searching lists

newLISP has facilities for multidimensional indexing into nested lists. There are `destructive` functions like `push`, `pop`, `setf`, `set-ref`, `set-ref-all`, `sort` and `reverse` and many others for `non-destructive` operations, like `nth`, `ref`, `ref-all`, `first`, `last` and `rest` etc.. Many of the list functions in newLISP also work on strings.

Note that any list or string index in newLISP can be negative starting with -1 from the right side of a list:

```
(set 'L '(a b c d))
(L -1)   → d
(L -2)   → c
(-3 2 L) → (b c)

(set 'S  "abcd")

(S -1)   → d
(S -2)   → c
(-3 2 S) → "bc")
```

`push` and `pop`

To add to a list use `push`, to eliminate an element from a list use `pop`. Both functions are destructive, changing the contents of a list:

```
(set 'L '(b c d e f))

(push 'a L) → (a b c d e f)
(push 'g L -1) ; push to the end with negative index
(pop L)        ; pop first a
(pop L -1)     ; pop last g
(pop L -2)     ; pop second to last e
(pop L 1)      ; pop second c

L → (b d f)

; multidimensional push / pop
(set 'L '(a b (c d (e f) g) h i))

(push 'x L 2 1) → (a b (c x d (e f) g) h i)

L → (a b (c x d (e f) g) h i)

(pop L 2 1) → 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))
```

Pushing to the end of a list repeatedly is optimized in newLISP and as fast as pushing in front of a list.

When pushing an element with index vector V it can be popped with the same index vector V:

```
(set 'L '(a b (c d (e f) g) h i))
(set 'V '(2 1))
(push 'x L V)
L → (a b (c x d (e f) g) h i))
(ref 'x L) → (2 1) ; search for a nested member
(pop L V) → 'x
```

Extend using `extend`

Using `extend` lists can be appended destructively. Like `push` and `pop`, `extend` modifies the list in the first argument.

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

L → '(a b c d e f g)

; extending in a place

(set 'L '(a b "CD" (e f)))
(extend (L 3) '(g))
L → (a b "CD" (e f g))
```

Accessing lists

Multiple indexes can be specified to access elements in a nested list structure:

```
(set 'L '(a b (c d (e f) g) h i))

; old syntax only for one index
(nth 2 L) → (c d (e f) g)

; use new syntax for multiple indices
(nth '(2 2 1) L) → f
(nth '(2 2) L) → (e f)

; vector indexing
(set 'vec '(2 2))
(nth vec L) → (e f)

; implicit indexing
(L 2 2 1) → f
```

```
(L 2 2)   → (e f)

; implicit indexing with vector
(set 'vec '(2 2 1))
(L vec)   → f
```

Implicit indexing shown in the last example can make code more readable. Indexes before a list select subsections of a list, which in turn are always lists.

Implicit indexing is also available for rest and slice:

```
(rest '(a b c d e))      → (b c d e)
(rest (rest '(a b c d e) → (c d e)
; same as
(1 '(a b c d e)) → (b c d e)
(2 '(a b c d e)) → (c d e)

; negative indices
(-2 '(a b c d e)) → (d e)

; slicing
(2 2 '(a b c d e f g))  → (c d)
(-5 3 '(a b c d e f g)) → (c d e)
```

Selecting more elements

Sometimes more than one element must be selected from a list. This is done using select:

```
; pick several elements from a list
(set 'L '(a b c d e f g))
(select L 1 2 4 -1) → (b c e g)

; indices can be delivered in an index vector:
(set 'vec '(1 2 4 -1))
(select L vec) → (b c e g)
```

The selecting process can re-arrange or double elements at the same time:

```
(select L 2 2 1 1) → (c c b b)
```

Filtering and differencing lists

Lists can be filtered, returning only those elements that meet a specific condition:

```
(filter (fn(x) (< 5 x)) '(1 6 3 7 8))    → (6 7 8)
(filter symbol? '(a b 3 c 4 "hello" g)) → (a b c g)
(difference '(1 3 2 5 5 7) '(3 7)) → (1 2 5)
```

The first example could be written more concisely, as follows:

```
(filter (curry < 5) '(1 6 3 7 8))
```

The `curry` function makes a one argument function out of a two argument function:

```
(curry < 5) → (lambda (_x) (< 5 _x))
```

With `curry`, a function taking two arguments can be quickly converted into a predicate taking one argument.

Changing list elements

`setf` can be used to change a list element by referencing it with either `nth` or `assoc`:

```
; modify a list at an index
(set 'L '(a b (c d (e f) g) h i))

(setf (L 2 2 1) 'x) → x
L → (a b (c d (e x) g) h i)
(setf (L 2 2) 'z) → z
L → (a b (c d z g) h i)

; modify an association list
(set 'A '((a 1) (b 2) (c 3)))

; using setf with assoc
(setf (assoc 'b A) '(b 22)) → (b 22)
A → ((a 1) (b 22) (c 3))
; using setf with lookup
(setf (lookup 'c A) 33) → 33
A → ((a 1) (b 22) (c 33))
```

The anaphoric variable

The internal *anaphoric* system variable `$it` holds the old list element. This can be used to configure the new one:

```
(set 'L '(0 0 0))
(setf (L 1) (+ $it 1)) → 1 ; the new value
(setf (L 1) (+ $it 1)) → 2
(setf (L 1) (+ $it 1)) → 4
L → '(0 3 0)
```

The following functions use the anaphoric `$it`: `find-all`, `if`, `replace`, `set-ref`, `set-ref-all` and `setf setq`.

Replace in simple lists

Replace, which can also be used on strings, can search for and replace

multiple elements in a list at once. Together with `match` and `unify` complex search patterns can be specified. Like with `setf`, the replacement expression can use the old element contents to form the 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)
```

The function `replace` can take a comparison function for picking list elements:

```
; 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)
```

Using the built-in functions `match` and `unify` more complex selection criteria can be defined:

```
; replace only sublists starting 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))

; 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))

$0 → 4  ; replacements made

; change only sublists where both elements are the same

(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))

$0 → 2  ; replacements made
```

During replacements `$0` and the anaphoric system variable `$it` contain the current found expression.

After a replacement statement is executed the newLISP system variable `$0` contains the number of replacements made.

Replace in nested lists

Sometimes lists are nested, e.g. the SXML results from parsing XML. The functions `ref-set`, `set-ref` and `set-ref-all` can be used to find a single element or

all elements in a nested list, and replace it or all.

```
(set 'data '((monday (apples 20 30) (oranges 2 4 9)) (tuesday (apples 5) (oranges 32 1))))

(set-ref 'monday data tuesday)
→ ((tuesday (apples 20 30) (oranges 2 4 9)) (tuesday (apples 5) (oranges 32 1)))
```

The function `set-ref-all` does a `set-ref` multiple times, replacing all found occurrences of an element.

```
(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)))
```

Like `find`, `replace`, `ref` and `ref-all`, more complex searches can be expressed using `match` or `unify`:

```
(set 'data '((monday (apples 20 30) (oranges 2 4 9)) (tuesday (apples 5) (oranges 32 1))))

(set-ref-all '(oranges *) data (list (first $0) (apply + (rest $it))) match)
→ ((monday (apples 20 30) (oranges 15)) (tuesday (apples 5) (oranges 33)))
```

The last example shows how `$0` can be used to access the old list element in the updating expression. In this case the numbers for `oranges` records have been summed. Instead of `$0` the anaphoric system variable `$it` can also be used.

Passing lists by reference

Sometimes a larger list (more than a few hundred elements) must be passed to a user-defined function for elements in it to be changed. Normally newLISP passes all parameters to user-defined functions by value. But the following snippet shows a technique that can be used to pass a bigger list or string object by reference:

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

(define (change db i value)
    (setf (db i) value))

(change data 3 999) → d

data:data → '(a b c 999 d e f g h)
```

In this example the list is encapsulated in a context named `data` holding a variable `data` with the same name.

Whenever a function in newLISP looks for a string or list parameter, a context can be passed, which will then be interpreted as the default functor.

When returning a list or array or an element belonging to a list or array referenced by a symbol, many built-in functions return a //reference// to the list – not a copy. This can be used to nest built-in functions when modifying a list:

```
(set 'L '(r w j s r b))

(pop (sort L)) → b

L → (j r r s w)
```

Variable expansion

Two functions are available to do macro-expansion: `expand` and `letex`. The `expand` function has three different syntax patterns.

Symbols get expanded to their value:

```
; expand from one or more listed symbols
(set 'x 2 'a '(d e))
(expand '(a x (b c x)) 'x 'a)  → ((d e) 2 (b c 2))
```

`expand` is useful when composing lambda expressions or when doing variable expansion inside functions and function macros (fexpr with `define-macro`):

```
; use expansion inside a function
(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
```

`expand` can take an association list:

```
; expand from an association list
(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))
```

and the value part in associations can be evaluated first:

```
; evaluate the value parts in the association list before expansion
(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)
```

`expand` does its work on variables starting with an uppercase letter when expansion variables have neither been specified stand-alone nor in an association list.

```
; expand from uppercase variables
(set 'A 1 'Bvar 2 'C nil 'd 5 'e 6)
```

```
(expand '(A (Bvar) C d e f))  → (1 (2) C d e f)
```

Using this, a previous function definition can be made even shorter.

```
; use expansion from uppercase variables in function factories
(define (raise-to Power)
    (expand (fn (base) (pow base Power))))
(define cube (raise-to 3)) → (lambda (base) (pow base 3))
(cube 4) → 64
```

The `letex` function works like `expand`, but expansion symbols are local to the `letex` expression.

```
; use letex for variable expansion
(letex ( (x 1) (y '(a b c)) (z "hello") ) '(x y z)) → (1 (a b c) "hello")
```

Note that in the example the body expression in `letex`: `(x y z)` is quoted to prevent evaluation.

Destructuring nested lists

The following method can be used to bind variables to subparts of a nested list:

```
; uses unify together with bind for destructuring
(set 'structure '((one "two") 3 (four (x y z))))
(set 'pattern '((A B) C (D E))) ; unify needs uppercase for binding
(bind (unify pattern structure))
A → one
B → "two"
C → 3
D → four
E → (x y z)
```

§

7. Program flow

Program flow in newLISP is mostly functional but it also has looping and branching constructs and a `catch` and `throw` to break out of the normal flow.

Looping expressions as a whole behave like a function or block returning the last expression evaluated.

Loops

Most of the traditional looping patterns are supported. Whenever there is a

looping variable, it is local in scope to the loop, behaving according the rules of dynamic scoping inside the current name-space or context:

```
; loop a number of times
; i goes from 0 to N - 1
(dotimes (i N)
    ....
)

; demonstrate locality of i
(dotimes (i 3)
    (print i ":")
    (dotimes (i 3) (print i))
    (println)
)

→ ; will output
 0:012
 1:012
 2:012

; loop through a list
; takes the value of each element in aList
(dolist (e aList)
    ...
)

; loop through a string
; takes the ASCII or UTF-8 value of each character in aString
(dostring (e aString)
    ...
)

; loop through the symbols of a context in
; alphabetical order of the symbol name
(dotree (s CTX)
    ...
)

; loop from to with optional step size
; i goes from init to <= N inclusive with step size step
; Note that the sign in step is irrelevant, N can be greater
; or less then init.
(for (i init N step)
    ...
)

; loop while a condition is true
; first test condition then perform body
(while condition
    ...
)

; loop while a condition is false
; first test condition then perform body
```

```
(until condition
    ...
)

; loop while a condition is true
; first perform body then test
; body is performed at least once
(do-while condition
    ...
)

; loop while a condition is false
; first perform body then test
; body is performed at least once
(do-until condition
    ...
)
```

Note that the looping functions `dolist`, `dotimes` and `for` can also take a break condition as an additional argument. When the break condition evaluates to true the loop finishes:

```
(dolist (x '(a b c d e f g) (= x 'e))
    (print x))
→ ; will output
 abcd
```

Blocks

Blocks are collections of s-expressions evaluated sequentially. All looping constructs may have expression blocks after the condition expression as a body.

Blocks can also be constructed by enclosing them in a begin expression:

```
(begin
    s-exp1
    s-exp2
     ...
    s-expN)
```

Looping constructs do not need to use an explicit begin after the looping conditions. `begin` is mostly used to block expressions in `if` and `cond` statements.

The functions `and`, `or`, `let`, `letn` and `local` can also be used to form blocks and do not require `begin` for blocking statements.

Branching

```
(if condition true-expr false-expr)
```

```
;or when no false clause is present
(if condition true-expr)

;or unary if for (filter if '(...))
(if condition)

; 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))
)

; the when form can take several statements without
; using a (begin ...) block
(when condition
    exp-1
    exp-2
    ...
)

; unless works like (when (not ...) ...)
(unless condition
    exp-1
    exp-2
    ...
)
```

Depending on the condition, the `exp-true` or `exp-false` part is evaluated and returned.

More than one `condition/exp-true` pair can occur in an if expression, making it look like a `cond`:

```
(if condition-1 exp-true-1
    condition-2 exp-true-2
    ...
    condition-n exp-true-n
    expr-false
)
```

The first `exp-true-i` for which the `condition-i` is not `nil` is evaluated and returned, or the `exp-false` if none of the `condition-i` is `true`.

`cond`works like the multiple condition form of if but each part of `condition-i` `exp-true-i` must be braced in parentheses:

```
(cond
    (condition-1 exp-true-1 )
```

```
    (condition-2 exp-true-2 )
                ...
    (condition-n exp-true-n )
    (true exp-true)
)
```

Fuzzy flow

Using `amb` the program flow can be regulated in a probabilistic fashion:

```
(amb
    exp-1
    exp-2
    ...
    exp-n
)
```

One of the alternative expressions `exp-1` to `exp-n` is evaluated with a probability of `p = 1/n` and the result is returned from the amb expression.

Flow with `catch` and `throw`

Any loop or other expression block can be enclosed in a catch expression. The moment a throw expression is evaluated, the whole catch expression returns the value of the throw expression.

```
(catch
    (dotimes (i 10)
    (if (= i 5) (throw "The End"))
    (print i " "))
)
; will output
0 1 2 3 4
; and the return value of the catch expression will be
→ "The End"
```

Several `catch` expressions can be nested. The function `catch` can also catch errors. See the chapter on //Error Handling// below.

Leave loops with a break condition

Loops built using `dotimes`, `dolist` or `for` can specify a break condition for leaving the loop early:

```
(dotimes (x 10 (> (* x x) 9))
    (println x))
→
 0
 1
 2
```

```
 3

(dolist (i '(a b c nil d e) (not i))
    (println i))
→
 a
 b
 c
```

Change flow with `and` or `or`

Similar to programming in the Prolog language, the logical `and` and `or` can be used to control program flow depending on the outcome of expressions logically connected:

```
(and
   expr-1
   expr-2
    ...
   expr-n)
```

Expressions are evaluated sequentially until one `expr-i` evaluates to `nil` or the empty list `()` or until all `expr-i` are exhausted. The last expression evaluated is the return value of the whole `and` expression.

```
(or
   expr-1
   expr-2
    ...
   expr-n)
```

Expressions are evaluated sequentially until one `expr-i` evaluates to **not** `nil` and not `()` or until all `expr-i` are exhausted. The last expression evaluated is the return value of the whole `or` expression.

§

8. Error handling

Several conditions during evaluation of a newLISP expression can cause error exceptions. For a complete list of errors see the Appendix in the newLISP Reference Manual.

newLISP errors

newLISP errors are caused by the programmer using the wrong syntax when invoking functions, supplying the wrong number of parameters or parameters with the wrong data type, or by trying to evaluate nonexistent functions.

```
; examples of newLISP errors
;
(foo foo)   → invalid function : (foo foo)
(+ "hello") → value expected in function + : "hello"
```

User defined errors

User errors are error exceptions thrown using the function throw-error:

```
; user defined error
;
(define (double x)
    (if (= x 99) (throw-error "illegal number"))
    (+ x x)
)

(double 8)   → 16
(double 10)  → 20
(double 99)
→
user error : illegal number
called from user defined function double
```

Error event handlers

newLISP and user defined errors can be caught using the function error-event to define an event handler.

```
; define an error event handler
;
(define (MyHandler)
    (println  (last (last-error))  " has occurred"))

(error-event 'MyHandler)

(foo) → ERR: invalid function : (foo) has occurred
```

Catching errors

A finer grained and more specific error exception handling can be achieved using a special syntax of the function catch.

```
(define (double x)
    (if (= x 99) (throw-error "illegal number"))
    (+ x x))
```

catch with a second parameter can be used to catch both system and

user-defined errors:

```
(catch (double 8) 'result) → true
result → 16
(catch (double 99) 'result) → nil
(print result)
 →
user error : illegal number
called from user defined function double

(catch (double "hi") 'result) → nil
(print result)
→
value expected in function + : x
called from user defined function double
```

The catch expression returns true when no error exception occurred, and the result of the expression is found in the symbol `result` specified as a second parameter.

If an error exception occurs, it is caught and the catch clause returns nil. In this case the symbol `result` contains the error message.

Operating system errors

Some errors originating at operating system level are not caught by newLISP, but can be inspected using the function `sys-error`. For example the failure to open a file could have different causes:

```
; trying to open a nonexistent file
(open "blahbla" "r")  →  nil
(sys-error)           →  (2 "No such file or directory")


; to clear errno specify 0
(sys-error 0)         →  (0 "Unknown error: 0")
```

Numbers returned may be different on different Unix platforms. Consult the file `/usr/include/sys/errno.h` on your platform.

§

9. Functions as data

Manipulate after definition

```
(define (double x) (+ x x))
```

```
→ (lambda (x) (+ x x))

(first double) → (x)
(last double)  → (+ x x)

; make a fuzzy double
(setf (nth 1 double) '(mul (normal x (div x 10)) 2))

(double 10) → 20.31445313
(double 10) → 19.60351563
```

lambda in newLISP is not an operator or symbol, but rather a special s-expression or list attribute:

```
(first double) → (x)   ; not lambda
```

The lambda attribute of an s-expression is right-associative in `append`:

```
(append (lambda) '((x) (+ x x))) → (lambda (x) (+ x x))
; or shorter
(append (fn) '((x) (+ x x))) → (lambda (x) (+ x x))

(set 'double (append (lambda) '((x) (+ x x)))

(double 10) → 20
```

and left-associative when using cons:

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

Lambda expressions in newLISP never lose their first class object property.

The word `lambda` can be abbreviated as `fn`, which is convenient when mapping or applying functions to make the expression more readable and shorter to type.

Mapping and applying

Functions or operators can be applied to a list of data at once and all results are returned in a list

```
(define (double (x) (+ x x))

(map double '(1 2 3 4 5)) → (2 4 6 8 10)
```

Functions can be applied to parameters occurring in a list:

```
(apply + (sequence 1 10)) → 55
```

Functions making functions

Here an expression is passed as a parameter:

```
; macro expansion using expand
(define (raise-to power)
    (expand (fn (base) (pow base power)) 'power))

; or as an alternative using letex
(define (raise-to power)
    (letex (p power) (fn (base) (pow base p))))

(define square (raise-to 2))

(define cube (raise-to 3))

(square 5)   → 25
(cube 5)     → 125
```

The built-in function `curry` can be used to make a function taking one argument from a function taking two arguments.

```
(define add-one (curry add 1))  → (lambda () (add 1 ($args 0)))

(define by-ten (curry mul 10))  → (lambda () (mul 10 ($args 0)))

(add-one 5)    → 6

(by-ten 1.23)  → 12.3
```

Note that the 'curried' parameter is always the first parameter of the original function.

Functions with memory

newLISP can create local state variables using a name-space context:

```
; newLISP generator

(define (gen:gen)
    (setq gen:sum
    (if gen:sum (inc gen:sum) 1)))

; this could be written even shorter, because
; 'inc' treats nil as zero

(define (gen:gen)
    (inc gen:sum))

(gen) → 1
(gen) → 2
(gen) → 3
```

The example uses a default functor — functions name equals names-space

name — to give it the appearance of a normal function. Other functions could be added to the namespace, e.g. for initializing the sum.

```
(define (gen:init x)
    (setq gen:sum x))

(gen:init 20) → 20

(gen) → 21
(gen) → 22
```

Functions using self modifying code

The first class nature of lambda expressions in newLISP makes it possible to write self modifying code:

```
;; sum accumulator
(define (sum (x 0)) (inc 0 x))

(sum 1)    → 1
(sum 2)    → 3
(sum 100)  → 103
(sum)      → 103

sum  → (lambda ((x 0)) (inc 103 x))
```

The following example shows a function making a self modifying stream function using `expand` :

```
(define (make-stream lst)
    (letex (stream lst)
        (lambda () (pop 'stream))))

(set 'lst '(a b c d e f g h))
(define mystream (make-stream lst))

(mystream)  → a
(mystream)  → b
(mystream)  → c
```

Because `pop` works on both: lists and strings, the same function factory can be used for a string stream:

```
(set 'str "abcddefgh")
(define mystream (make-stream str))

(mystream)  → "a"
(mystream)  → "c"
```

§

10. Text processing

Regular expressions

Regular expressions in newLISP can be used in a number of functions:

function	function description
directory	Return a list of files whose names match a pattern.
ends-with	Test if a string ends with a key string or pattern.
find	Find the position of a pattern.
find-all	Assemble a list of all patterns found.
parse	Break a string into tokens at patterns found between tokens.
regex	Find patterns and returns a list of all sub patterns found, with offset and length.
replace	Replace found patterns with a user defined function, which can take as input the patterns themselves.
search	Search for a pattern in a file.
starts-with	Test if a string starts with a key string or pattern.

The functions find, regex, replace and search store pattern matching results in the system variables $0 to $15. See the newLISP Users Manual for details.

The following paragraphs show frequently-used algorithms for scanning and tokenizing text.

Scanning text

The replace function, together with a regular expression pattern, can be used to scan text. The pattern in this case describes the tokens scanned for. As each token is found, it is pushed on a list. The work is done in the replacement expression part of replace. This example saves all files linked on a web page:

```
#!/usr/bin/newlisp

; tokenize using replace with regular expressions
; names are of the form <a href="example.lsp">example.lsp</a>

(set 'page (get-url "http://newlisp.digidep.net/scripts/"))
(replace {>(.*lsp)<} page (first (push $1 links)) 0) ; old technique
;(set 'links (find-all {>(.*lsp)<} page $1)) ; new technique

(dolist (fname links)
   (write-file fname (get-url (append "http://newlisp.digidep.net/scripts/" fname)))
   (println "->" fname))
```

```
(exit)
```

Curly braces ({,}) are used in the regular expression pattern to avoid having to escape the quotes (") or other characters that have special meanings in regular expressions.

The following alternative technique is even shorter. The `find-all` function puts all matching strings into a list:

```
(set 'links (find-all {>(.*lsp)<} page $1)) ; new technique
```

In an additional expression `find-all` can be directed to do additional work with the sub expressions found:

```
(find-all {(new)(lisp)} "newLISPisNEWLISP" (append $2 $1) 1)
→ ("LISPnew" "LISPNEW")
```

In the last example `find-all` appends the sub expressions found in reverse order before returning them in the result list.

Another technique for tokenizing text uses `parse`. Whereas with `replace` and `find-all` the regular expression defined the token, when using `parse`, the regex pattern describes the space between the tokens:

```
; tokenize using parse
(set 'str "1 2,3,4 5, 6 7  8")
(parse str {,\ *|\ +,*} 0)
→ ("1" "2" "3" "4" "5" "6" "7" "8")
```

Without the curly braces in the parse pattern, the backslashes would need to be doubled. Note that there is a space after each backslash.

Appending strings

When appending strings append and join can be used to form a new string:

```
(set 'lstr (map string (rand 1000 100)))
→ ("976" "329" ... "692" "425")

; the wrong slowest way
(set 'bigStr "")
(dolist (s lstr)
    (set 'bigStr (append bigStr s)))

; smarter way - 50 times faster
;
(apply append lstr)
```

Sometimes strings are not readily available in a list like in the above

examples. In this case push can be used to push strings on a list while they get produced. The list then can be used as an argument for join, making the fastest method for putting strings together from existing pieces:

```
; smartest way - 300 times faster
; join an existing list of strings
;
(join lstr) → "97632936869242555543 ...."

; join can specify a string between the elements
; to be joined
(join lstr "-") → "976-329-368-692-425-555-43 ...."
```

Growing strings

Often it is best to grow a string in place. The function `extend` can be used to append to a string at the end. The function `push` can be used to insert new content at any place in the string.

.

```
; smartest way - much faster on big strings
; grow a string in place

; using extend
(set 'str "")
(extend str "AB" "CD")
str → "ABCD"

; extending in a place
(set 'L '(a b "CD" (e f)))
(extend (L 2) "E")
L → (a b "CDE" (e f))

; using push
(set 'str "")
(push "AB" str -1)
(push "CD" str -1)
str → "ABCD"
```

Rearranging strings

The function select for selecting elements from lists can also be used to select and re-arrange characters from strings:

```
(set 'str "eilnpsw")
(select str '(3 0 -1 2 1 -2 -3)) → "newlisp"

; alternative syntax
(select str 3 0 -1 2 1 -2 -3) → "newlisp"
```

The second syntax is useful when indexes are specified not as constants but occur as variables.

Modifying strings

newLISP has a variety of functions, which can destructively change a string:

function	description
`extend`	Extend a string with another string.
`push pop`	Insert or extract one or more characters at a specific position.
`replace`	Replace all occurrences of a string or string pattern with a string.
`setf`	Replace a character in a string with one or more characters.

`replace` can also be used to remove all occurrences of string or string pattern when specifying an empty string `""` as replacement.

When indexing strings with either `nth` or implicit indexing, the string is addressed at *character* rather than *byte* boundaries to work correctly on UTF-8 enabled versions of newLISP. A UTF-8 character can contain more than one byte.

§

11. Dictionaries and hashes

Hash-like key → value access

newLISP has functions to create and manipulate symbols using the functions `sym` and a special syntax of the function `context`. In older versions of newLISP, these functions were used to program hash-like creation and access of key-value pairs. Now a shorter and more convenient method is available, using the un-initialized `default functor` of a namespace context:

```
(define Myhash:Myhash) ; establish the namespace and default functor
```

As an alternative to the above methods, the predefined namespace and default functor `Tree` can be used to instantiate a new one:

```
(new Tree 'Myhash)
```

Both methods produce the same result, but the second method also protects the default functor `Myhash:Myhash` from change.

A *default functor* is the symbol with the same name as the namespace (context) it belongs to. If this default functor symbol does not contain anything except `nil`, it works like a hash function:

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

(Myhash "var") → 123 ; retrieve value

(Myhash "foo" "hello")

(Myhash "bar" '(q w e r t y))

(Myhash "!*@$" '(a b c))

; numbers can be used too and will be converted to strings internally

(Myhash 555 42)

(Myhash 555) → 42
```

Setting a hash symbol to `nil` will effectively erase it:

```
(Myhash "bar" nil)
```

The key can be any string; newLISP prevents symbol clashes with built-in newLISP symbols by prepending an underscore character (_) to all key strings internally. The value can be any string, number or any other newLISP s-expression.

The `Myhash` namespace can be transformed in an association list:

```
(Myhash) → (("!*@$" (a b c)) ("foo" "hello") ("var" 123))
```

Or the raw contents of `Myhash` can be shown using the `symbols` function:

```
(symbols Myhash) → (Myhash:Myhash Myhash:_!*@$ Myhash:_foo Myhash:_var)
```

Dictionaries can be built by converting an existing association list:

```
(set 'aList '(("one" 1) ("two" 2) ("three")))

(Myhash aList)

(Myhash) → (("!*@$" (a b c)) ("foo" "hello") ("one" 1) ("three" nil) ("two" 2) ("var" 123))
```

Saving and loading dictionaries

The dictionary can be easily saved to a file by serializing the namespace `Myhash`:

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

The whole namespace is saved to the file `Myhash.lsp` and can be reloaded into newLISP at a later time:

```
(load "Myhash")
```

Note that hashes create contexts similar to the `bayes-train` function. All string keys are prepended with an underscore and then transformed into a symbol. This means that namespaces created using `bayes-train` can be used like hashes to retrieve words and their statistics. See the `bayes-train` function in the manual for more detail.

§

12. TCP/IP client server

Open connection

In this pattern the server keeps the connection open until the client closes the connection, then the server loops into a new `net-accept`:

```
; sender listens
(constant 'max-bytes 1024)
(if (not (set 'listen (net-listen 123)))
    (print (net-error)))
(while (not (net-error))
    (set 'connection (net-accept listen)) ; blocking here
    (while (not (net-error))
         (net-receive connection message-from-client max-bytes)
         .... process message from client ...
         .... configure message to client ...
         (net-send connection message-to-client))
)
```

and the client:

```
; client connects to sender
(if (not (set 'connection (net-connect "host.com" 123)))
    (println (net-error)))
; maximum bytes to receive
(constant 'max-bytes 1024)
; message send-receive loop
(while (not (net-error))
     .... configure message to server ...
     (net-send connection message-to-server)
     (net-receive connection message-from-server max-bytes)
     .... process message-from-server ...
)
```

Closed transaction

In this pattern the server closes the connection after each transaction exchange of messages.

```
; sender
(while (not (net-error))
    (set 'connection (net-accept listen)) ; blocking here
    (net-receive connection message-from-client max-bytes)
        .... process message from client ...
        .... configure message to client ...
    (net-send connection message-to-client)
    (close connection)
)
```

and the client again tries to connect to the sender:

```
; client
(unless (set 'connection (net-connect "host.com" 123))
    (println (net-error))
    (exit))
; maximum bytes to receive
(constant 'max-bytes 1024)
  .... configure message to server ...
(net-send connection message-to-server)
(net-receive connection message-from-server max-bytes)
  .... process message-from-server ...
```

There are many different ways to set up a client/server connection, see also the examples in the newLISP manual.

§

13. UDP communications

They are fast and need less setup than TCP/IP and offer `multi casting`. UDP is also less reliable because the protocol does less checking, i.e. of correct packet sequence or if all packets are received. This is normally no problem when not working on the Internet but in a well controlled local network or when doing machine control. A simple more specific protocol could be made part of the message.

Open connection

In this example the server keeps the connection open. UDP communications

with net-listen, net-receive-from and net-send-to can block on receiving.

Note that both, the client and server use `net-listen` with the `"udp"` option. In this case `net-listen` is used only for binding the socket to the address, it is not used for listening for a connection. The server could receive messages from several clients. The `net-send-to` function extracts the target address from the message received.

The sender:

```
; sender
(set 'socket (net-listen 10001 "localhost" "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 (first (parse (nth 1 msg) ":"))
                 (nth 2 msg) (upper-case (first msg)) socket))
(exit)
```

and the client:

```
(set 'socket (net-listen 10002 "" "udp"))
(if (not socket) (println (net-error)))
(while (not (net-error))
    (print "enter something -> ")
    (net-send-to  "127.0.0.1" 10001 (read-line) socket)
    (net-receive socket buff 255)
    (println "=> " buff))
(exit)
```

Closed transaction

This form is sometimes used for controlling hardware or equipment. No setup is required, just one function for sending, another one for receiving.

```
; wait for data gram with maximum 20 bytes
(net-receive-udp 1001 20)
; or
(net-receive-udp 1001 20 5000000)  ; wait for max 5 seconds
; the sender
(net-send-udp "host.com" 1001 "Hello")
```

Win32 and Unix's show different behavior when sending less or more bytes than specified on the receiving end.

Multi-cast communications

In this scheme the server subscribes to one of a range of multi cast addresses

using the net-listen function.

```
; example server
(net-listen 4096 "226.0.0.1" "multi") → 5
(net-receive-from 5 20)

; example client I
(net-connect "226.0.0.1" 4096 "multi") → 3
(net-send 3 "hello")
; example client II
(net-connect "" 4096 "multi") → 3
(net-send-to "226.0.0.1" 4096 "hello" 3)
```

The connection in the example is blocking on `net-receive` but could be de-blocked using `net-select` or `net-peek`

§

14. Non-blocking communications

Using `net-select`

In all previous patterns the client blocks when in receive. The net-select call can be used to unblock communications:

```
; optionally poll for arriving data with 100ms timeout
(while (not (net-select connection "r" 100000))
    (do-something-while-waiting ...))

(net-receive...)
```

`connection` can be a single number for a connection socket or a list of numbers to wait on various sockets.

Using `net-peek`

`net-peek` returns the number of characters pending to read.

```
(while ( = (net-peek aSock) 0)
    (do-something-while-waiting ...))
(net-receive...)
```

§

15. Controlling other applications

Using exec

This method is only suited for short exchanges, executing one command and receiving the output.

```
(exec "ls *.c") → ("." ".." "util.c" "example.ls")
```

The `exec` function opens a process pipe for the Unix command-line utility `ls` and collects each line of STDOUT into a list of strings.

Most following examples use `process` to launch an application. This function returns immediately after launching the other application and does not block.

In all of the following patterns the server is not independent but controlled by the client, which launches the server and then communicates via a line oriented protocol:

```
→ launch server
→ talk to server
← wait for response from server
→ talk to server
← wait for response from server
     ...
```

Sometimes a sleep time is necessary on the client side to wait for the server to be ready loading. Except for the first example, most of these are condensed snippets from GTK-Server from [http://www.gtk-server.org www.gtk-server.org]. The basic program logic will be the same for any other application.

STD I/O pipes

The `process` function allows specifying 2 pipes for communications with the launched application.

```
; setup communications
(map set '(myin tcout) (pipe))
(map set '(tcin myout) (pipe))
(process "/usr/bin/wish" tcin tcout)

; make GUI
(write myout
[text]
wm geometry . 250x90
wm title . "Tcl/Tk and newLISP"
bind . <Destroy> {puts {(exit)}}
[/text])
```

```
; run event loop
(while (read-line myin)
    (eval-string (current-line))
)
```

This is the preferred way to set up longer lasting, bidirectional communications with Unix command line utilities and languages. For one-command exchanges the `exec` function does the job shorter.

For a more elaborate Tcl/Tk example see the application `examples/tcltk.lsp` in the source distribution.

Communicate via TCP/IP

```
; Define communication function
(define (gtk str , tmp)
    (net-send connection str)
    (net-receive connection tmp 64)
    tmp)

; Start the gtk-server
(process "gtk-server tcp localhost:50000")
(sleep 1000)

; Connect to the GTK-server
(set 'connection (net-connect "localhost" 50000))
(set 'result (gtk "gtk_init NULL NULL"))
(set 'result (gtk "gtk_window_new 0"))
               .....
```

Communicate via named FIFO

Make a FIFO first (looks like a special file node):

```
(exec "mkfifo myfifo")

; or alternatively

(import "/lib/libc.so.6" "mkfifo")
(mkfifo "/tmp/myfifo" 0777)

; Define communication function
(define (gtk str)
        (set 'handle (open "myfifo" "write"))
        (write handle str)
        (close handle)
        (set 'handle (open "myfifo" "read"))
        (read handle tmp 20)
        (close handle)
tmp)
```

Communicate via UDP

Note that the listen function with "udp" option just binds the sockets to a address/hardware but not actually listens as in TCP/IP.

```
; Define communication function
(define (gtk str , tmp)
(net-send-to "localhost" 50000 str socket)
(net-receive socket 'tmp net-buffer)
tmp)

; Start the gtk-server
(define (start)
        (process "gtk-server udp localhost:50000")
        (sleep 500)
        (set 'socket (net-listen 50001 "localhost" "udp")) )

(set 'result (gtk "gtk_init NULL NULL"))

(set 'result (gtk "gtk_window_new 0"))
.....
```

§

16. Launching apps blocking

Shell execution

This is frequently used from newLISP's interactive command line to execute processes in a blocking fashion, which need a shell to run:

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

There is an interesting variant of this form working not inside a newLISP expression, but only on the command line:

```
!ls -ltr
```

The `!` should be the first character on the command line. This form works like a shell escape in the VI editor. It is useful for invoking an editor or doing quick shell work without completely leaving the newLISP console.

Capturing std-out

```
(exec "ls /") → ("bin" "etc" "home" "lib")
```

Feeding std-in

```
(exec "script.cgi" cgi-input)
```

In this example `cgi-input` could contain a string feeding a query input, normally coming from a web server. Note that output in this case is written directly to the screen, and cannot be returned to newLISP. Use process and pipe for two way std i/o communications with other applications.

§

17. Semaphores, shared memory

Shared memory, semaphores and processes work frequently together. Semaphores can synchronize tasks in different process threads and shared memory can be used to communicate between them.

The following is a more complex example showing the working of all three mechanisms at the same time.

The producer loops through all n values from i = 0 to n - 1 and puts each value into shared memory where it is picked up by the consumer thread. Semaphores are used to signal that a data value is ready for reading.

Although controlling processes with semaphores and shared memory is fast, it is also error prone, specially when more the two processes are involved. It is easier to control multiple processes using the Cilk API and messaging between processes. See chapters 18. and 19. for these topics.

```
#!/usr/bin/newlisp
# prodcons.lsp -  Producer/consumer
#
# usage of 'fork', 'wait-pid', 'semaphore' and 'share'

(when (= ostype "Win32")
    (println "this will not run on Win32")
    (exit))

(constant 'wait -1 'sig 1 'release 0)

(define (consumer n)
    (set 'i 0)
    (while (< i n)
        (semaphore cons-sem wait)
        (println (set 'i (share data)) " <-")
        (semaphore prod-sem sig))
    (exit))
```

```
(define (producer n)
    (for (i 1 n)
        (semaphore prod-sem wait)
        (println "-> " (share data i))
        semaphore cons-sem sig))
    (exit))

(define (run n)
    (set 'data (share))
    (share data 0)
    (set 'prod-sem (semaphore)) ; get semaphores
    (set 'cons-sem (semaphore))
    (set 'prod-pid (fork (producer n))) ; start processes
    (set 'cons-pid (fork (consumer n)))
    (semaphore prod-sem sig) ; get producer started
    (wait-pid prod-pid) ; wait for processes to finish
    (wait-pid cons-pid) ;
    (semaphore cons-sem release) ; release semaphores
    (semaphore prod-sem release))

(run 10)

(exit)
```

§

18. Multiprocessing and Cilk

On multiprocessor CPUs the operating system will distribute processes and child processes created on different processor cores in an optimized fashion. newLISP offers a simple API which does all the work of launching processes and does the synchronized collection of evaluation results. The Cilk API consists of only 3 function calls, implemented in newLISP as `spawn`, `sync` and `abort`

Since v.10.1 newLISP's `message` function enables communications between parent and child processes. For more details about this, see the next chapter **19. Message exchange**.

Starting concurrent processes

```
; calculate primes in a range
(define (primes from to)
    (let (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
```

The example shows how the task of generating a range of prime numbers can be organized for parallel processing by splitting the range into sub-ranges. All `spawn` calls will return immediately, but `sync` will block until all child processes have finished and the result lists are available in the four variables `p1` to `p4`.

Watching progress

When the timeout value specified is too short for all processes to finish, `sync` will return `nil`. This can be used to watch progress:

```
; print a dot after each 2 seconds of waiting
(until (sync 2000) (println "."))
```

When `sync` is called without parameters, it returns a list of still active process ids:

```
; show a list of pending process ids after
;each three-tenths of a second
(until (sync 300) (println (sync)))
```

Invoking spawn recursively

```
(define (fibo n)
    (let (f1 nil f2 nil)
        (if (< n 2) 1
            (begin
                (spawn 'f1 (fibo (- n 1)))
                (spawn 'f2 (fibo (- n 2)))
                (sync 10000)
                (+ f1 f2)))))

(fibo 7)  → 21
```

Event driven notification

When processes launched with `spawn` finish, an `inlet` function specified in the `sync` statement can be called.

```
(define (report pid)
    (println "process: " pid " has returned"))

; call the report function, when a child returns
(sync 10000 report)
```

§

19. Message exchange

Parent and child processes started with `spawn` can exchange messages. Messages flow either from the parent to child processes or from child processes to the parent. By means of evaluating messages in the parent process, the parent process can be used as a proxy routing messages between child peers.

Internally newLISP uses UNIX local domain sockets for dual message queues between parent and child processes. When the receiving side of a queue is empty a `receive` call will return `nil`. Likewise when a queue is full, a `send` call will return `nil`. The looping function `until` can be used to make `send` and `receive` statements blocking.

Blocking message sending and receiving

```
; blocking sender
(until (send pid msg))     ; true when a msg queued up

; blocking receiver
(until (receive pid msg))  ; true after a msg is read
```

Blocking messages exchange

The parent process loops through all child process IDs and uses the `(until (receive cpid msg))` form of `receive` to wait for pending messages. `(sync)` returns a list of all child PIDs from processes launched by `spawn`.

```
#!/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 enables usage of send and 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)
```

generates this 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
```

Non blocking message exchange

Neither the sending child process nor the receiving parent process block. Each sends and receives messages as fast as possible. There is no guarantee that all messages will be delivered. It depends on the size of the sending queue and the speed of pick-up of messages by the parent process. If the sending queue for a child process is full, the `(send ppid (rand 100))` call will fail and return `nil`.

```
#!/usr/bin/newlisp

; child process transmits random numbers non-blocking
; not all calls succeed
(set 'start (time-of-day))

(define (child-process)
    (set 'ppid (sys-info -4)) ; get parent pid
    (while true
        (send ppid (rand 100)))
)

; parent starts 5  child processes, listens and displays
(dotimes (i 5) (spawn 'result (child-process) true))

(set 'N 1000)

(until finished
    (if (= (inc counter) N) (set 'finished true))
    (dolist (cpid (receive)) ; iterate thru ready child pids
        (receive cpid msg)
    (if msg (print "pid:" cpid "->" (format "%-2d  \r" msg))))
)

(abort) ; cancel child-processes
```

```
(sleep 300)

(exit)
```

Message timeouts

A messaging statement can be made to block for a certain time:

```
(define (receive-timeout pid msec)
    (let ( (start (time-of-day)) (msg nil))
        (until (receive pid msg)
            (if (> (- (time-of-day) start) 1000) (throw-error "timeout")))
    msg)
)
; use it

(receive-timeout pid 1000)  ; return message or throw error after 1 second
```

In this example blocking will occur for 1000 ms. Many methods exist to implement timeout behavior.

Evaluating messages

Messages sent can contain expressions which can be evaluated in the recipient's environment. This way variables can be set in the evaluator's environment, and messages can be routed to other processes. The following example implements a message router:

```
#!/usr/bin/newlisp

; sender child process of the message
(set 'A (spawn 'result
    (begin
        (dotimes (i 3)
            (set 'ppid (sys-info -4))
            /* the following 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 200) ; 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)
```

Acting as a proxy

In the last example program the expression:

```
; content of message to be evaluated by proxy
(until (send B (string "greetings from " A)))
```

A programming statement sent from child process ID A to the parent, where it is evaluated, causing a message to be sent to child process B. The parent process acts as a proxy agent for the child process A.

```
; the set statement is evaluated in the proxy
(until (send ppid '(set 'finished true)))
```

The expression `(set 'finished true)` is sent to the parent where it gets evaluated and causes the parent's `until` loop to finish.

The `sleep` statement in the A process ensures that the `"parent exiting ..."` message does not appear before all received messages are reported by process identified with B.

§

20. Databases and lookup tables

For smaller tables of not more than a few hundred entries association lists can be used. For larger databases use dictionaries and hashes as described in chapter 11.

Association lists

The association list is a classic LISP data structure for storing information for associative retrieval:

```
; creating association lists
; pushing at the end with -1 is optimized and
; as fast as pushing in front

(push '("John Doe" "123-5555" 1200.00) Persons -1)
(push '("Jane Doe" "456-7777" 2000.00) Persons -1)
```

```
.....

Persons →  (
("John Doe" "123-5555" 1200.00)
("Jane Doe" "456-7777" 2000.00) ...)

; access/lookup data records
(assoc "John Doe" Persons)

→ ("John Doe" "123-5555" 1200.00 male)

(assoc "Jane Doe" Persons)

→ ("Jane Doe" "456-7777" 2000.00 female)
```

newLISP has a lookup function similar to what is used in spreadsheet software. This function which works like a combination of `assoc` and `nth` can find the association and pick a specific member of the data record at the same time:

```
(lookup "John Doe" Persons 0)   → "123-555"
(lookup "John Doe" Persons -1)  → male
(lookup "Jane Doe" Persons 1)   → 2000.00
(lookup "Jane Doe" Persons -2)  → 2000.00

; update data records
(setf (assoc "John Doe" Persons)
    '("John Doe" "123-5555" 900.00 male))

; replace as a function of existing/replaced data
(setf (assoc "John Doe" Persons) (update-person $it))

; delete data records
(replace (assoc "John Doe" Persons) Persons)
```

Nested associations

If the data part of an association is itself an association list, we have a nested association:

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

A different syntax of the `assoc` function can be used to specify multiple keys:

```
; one key
(assoc "Anne" persons) → ("Anne" (address (country "USA") (city "New York")))

; two keys
(assoc '("Anne" address) persons) → (address (country "USA") (city "New York"))
```

```
; three keys
(assoc '("Anne" address city) persons) → (city "New York")

; three keys in a vector
(set 'anne-city '("Anne" address city))
(assoc anne-city persons) → (city "New York")
```

When all keys are symbols, as is in `address`, `country` and `city`, simple and nested associations in newLISP have the same format as newLISP `FOOP` (Functional Object Oriented Programming) objects. See the users manual chapter "18. Functional object-oriented programming" for details.

Updating nested associations

The functions `assoc` and `setf` can be used to update simple or nested associations:

```
(setf (assoc '("Anne" address city) persons) '(city "Boston")) → (city "New York")
```

`setf` always returns the newly set element.

Combining associations and hashes

Hashes and FOOP objects can be combined to form an in-memory database with keyed access.

In the following example, data records are stored in a hash namespace and access is with the name of the person as a key.

`setf` and `lookup` are used to update nested FOOP objects:

```
(new Tree 'Person)
(new Class 'Address)
(new Class 'City)
(new Class 'Telephone)


(Person "John Doe" (Address (City "Small Town") (Telephone 5551234)))

(lookup 'Telephone (Person "John Doe"))
(setf (lookup 'Telephone (Person "John Doe")) 1234567)
(setf (lookup 'City (Person "John Doe")) (lower-case $it))

(Person "John Doe") → (Address (City "small town") (Telephone 1234567))
```

§

21. Distributed computing

Many of todays applications are distributed on to several computers on the network or distributed on to several processes on one CPU. Often both methods of distributing an application are used at the same time.

newLISP has facilities to evaluate many expressions in parallel on different network nodes or processes running newLISP. The `net-eval` function does all the work necessary to communicate to other nodes, distribute expressions for evaluation and collect results in either a blocking or event driven fashion.

The functions `read-file`, `write-file`, `append-file` and `delete-file` can also be used to access with files on remote nodes when using URLs in file specifications. In a similar way the functions `load` and `save` can be used to load and save code from and to remote nodes.

newLISP uses existing HTTP protocols and newLISP command line behavior to implement this functionality. This means that programs can be debugged and tested using standard Unix applications like terminal, telnet or a web browser. This also enables easy integration of other tools and programs into distributed applications built with newLISP. For example the Unix utility `netcat` (`nc`) could be used to evaluate expressions remotely or a web browser could be used to retrieve webpages from nodes running a newLISP server.

Setting up in server mode

A newLISP server node is essentially a newLISP process listening to a network port and behaving like a newLISP command-line console and HTTP server for HTTP `GET`, `PUT`, `POST` and `DELETE` requests. Since version 9.1 newLISP server mode also answers `CGI` queries received by either `GET` or `POST` request.

Two methods are used to start a newLISP server node. One results in a state full server, maintaining state in between communications with different clients, the other method a server with no state, reloading for every new client connection.

Start a state-full server

```
newlisp -c -d 4711 &

newlisp myprog.lsp -c -d 4711 &

newlisp myprog.lsp -c -w /home/node25 -d 4711 &
```

newLISP is now listening on port 4711, the & (ampersand) sign tells newLISP to run in the background (Unix only). The `-c` switch suppresses command line

prompts. newLISP now behaves like a newLISP console without prompts listening on port 4711 for command line like input. Any other available port could have been chosen. Note that on Unix, ports below 1024 need administrator access rights.

The second example also pre-loads code. The third example also specifies a working directory using the `-w` option. If no working directory is specified using `-w`, the startup directory is assumed to be the working directory.

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.

Stateless server with inetd

On Unix the `inetd` or `xindetd` facility can be used to start a stateless server. In this case the TCP/IP net connections are managed by a special Unix utility with the ability to handle multiple requests at the same time. For each connection made by a client the `inetd` or `xinetd` utility will start a fresh newLISP process. After the connection is closed the newLISP process will shut down.

When nodes are not required to keep state, this is the preferred method for a newLISP server node, for handling multiple connections at the same time.

The `inetd` or `xinetd` process needs to be configured using configuration files found in the `/etc` directory of most Unix installations.

For both the `inetd` and `xinetd` configurations add the following line to the `/etc/services` file:

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

Note that any other port than `4711` could be supplied.

When configuring `inetd` add also the following lines to the `/etc/inetd.conf` file:

```
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 myprog.lsp -c

# a working directory can also be specified

net-eval  stream  tcp  nowait  newlisp  /usr/bin/newlisp -c -w /usr/home/newlisp
```

The last line also specified a working directory and a user `newlisp` instead of the `root` user. This is a more secure mode limiting newLISP server node access

to a specific user account with restricted permissions.

On some Unix system a modern flavor of `inetd`: the `xinetd` facility can be used. Add the following configuration to a file `/etc/xinet.d/net-eval`:

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

Note that a variety of parameter combinations are possible to restrict access from different places or limit access to certain users. Consult the man-pages for `inetd` and `xinetd` for details.

After configuring `inetd` or `xinetd` either process must be restarted to re-read the configuration files. This can be accomplished by sending the Unix `HUP` signal to either the `inetd` or `xinetd` process using the Unix `kill` or Unix `nohup` utility.

On Mac OS X the `launchd` facility can be used in a similar fashion. The newLISP source distribution contains the file `org.newlisp.newlisp.plist` in the `util/` directory. This file can be used to launch newlisp server during OS boot time as a persistent server.

Test the server with telnet

A newLISP server node can be tested using the Unix `telnet` utility:

```
telnet localhost 4711

; or when running on a different computer i.e. ip 192.168.1.100

telnet 192.168.1.100 4711
```

Multi-line expressions can be entered by enclosing them in `[cmd]`, `[/cmd]` tags, each tag on a separate line. Both the opening and closing tags should be on separate lines. Although newLISP has a second, new multi-line mode for the interactive shell since version 10.3.0 without tags, when using `netcat` or other Unix utilities, multi-line expressions still have to be enclosed in `[cmd]`, `[/cmd]` tags.

Test with netcat on Unix

```
echo '(symbols) (exit)' | nc localhost 4711
```

Or talking to a remote node:

```
echo '(symbols) (exit)' | nc 192.168.1.100 4711
```

In both examples `netcat` will echo back the result of evaluating the `(symbols)` expression.

Multi-line expressions can be entered by enclosing them in `[cmd]`, `[/cmd]` tags, each tag on a separate line.

Test from the command line

The `net-eval` function as a syntax form for connecting to only one remote server node. This mode is practical for quick testing from the newLISP command line:

```
(net-eval "localhost" 4711 "(+ 3 4)" 1000) → 7

; to a remote node

(net-eval "192.168.1.100" 4711 {(upper-case "newlisp")} 1000) → "NEWLISP"
```

In the second example curly braces `{,}` are used to limit the program string for evaluation. This way quotes can be used to limit a string inside the expression.

No `[cmd]`, `[/cmd]` tags are required when sending multi-line expressions. `net-eval` supplies these tags automatically.

Test HTTP with a browser

A newLISP server also understands simple HTTP `GET` and `PUT` requests. Enter the full path of a file in the address-bar of the browser:

```
http://localhost:4711//usr/share/newlisp/doc/newlisp_manual.html
```

The manual file is almost 800 Kbyte in size and will take a few seconds to load into the browser. Specify the port-number with a colon separated from the host-name or host IP. Note the double slash necessary to specify a file address relative to the root directory.

Evaluating remotely

When testing the correct installation of newLISP server nodes, we were already sending expressions to remote node for evaluation. Many times remote evaluation is used to split up a lengthy task into shorter subtasks for remote evaluation on different nodes.

The first example is trivial, because it only evaluates several very simple expressions remotely, but it demonstrates the principles involved easily:

```
#!/usr/bin/newlisp

(set 'result (net-eval '(
    ("192.168.1.100" 4711 {(+ 3 4)})
    ("192.168.1.101" 4711 {(+ 5 6)})
    ("192.168.1.102" 4711 {(+ 7 8)})
    ("192.168.1.103" 4711 {(+ 9 10)})
    ("192.168.1.104" 4711 {(+ 11 12)})
) 1000))


(println "result: " result)

(exit)
```

Running this program will produce the following output:

```
result: (7 11 15 19 23)
```

When running Unix and using an `inetd` or `xinetd` configured newLISP server, the servers and programs can be run on just one CPU, replacing all IP numbers with `"localhost"` or the same local IP number. The `indetd` or `xinetd` daemon will then start 5 independent newLISP processes. On Win32 5 state-full newLISP servers could be started on different port numbers to accomplish the same.

Instead of collecting all results at once on the return of `net-eval`, a callback function can be used to receive and process results as they become available:

```
#!/usr/bin/newlisp

(define (idle-loop p)
    (if p (println p)))

(set 'result (net-eval '(
    ("192.168.1.100" 4711 {(+ 3 4)})
    ("192.168.1.101" 4711 {(+ 5 6)})
    ("192.168.1.102" 4711 {(+ 7 8)})
    ("192.168.1.103" 4711 {(+ 9 10)})
    ("192.168.1.104" 4711 {(+ 11 12)})
) 1000 idle-loop))

(exit)
```

While `net-eval` is waiting for results, it calls the function `idle-loop` repeatedly with parameter `p`. The parameter `p` is `nil` when no result was received during the last 1000 milli seconds, or `p` contains a list sent back from the remote node. The list contains the remote address and port and the evaluation result.

The example shown would generate the following output:

```
("192.168.1.100" 4711 7)
("192.168.1.101" 4711 11)
("192.168.1.102" 4711 15)
("192.168.1.103" 4711 19)
("192.168.1.104" 4711 23)
```

For testing on just one CPU, replace addresses with `"localhost"`; the Unix `inetd` or `xinetd` daemon will start a separate process for each connection made and all listening on port `4711`. When using a state-full server on the same Win32 CPU specify a different port number for each server.

Setting up the 'net-eval' parameter structure

In a networked environment where an application gets moved around, or server nodes with changing IP numbers are used, it is necessary to set up the node parameters in the `net-eval` parameter list as variables. The following more complex example shows how this can be done. The example also shows how a bigger piece of program text can be transferred to a remote node for evaluation and how this program piece can be customized for each node differently:

```
#!/usr/bin/newlisp

; node parameters
(set 'nodes '(
    ("192.168.1.100" 4711)
    ("192.168.1.101" 4711)
    ("192.168.1.102" 4711)
    ("192.168.1.103" 4711)
    ("192.168.1.104" 4711)
))

; program template for nodes
(set 'program [text]
    (begin
        (map set '(from to node) '(%d %d %d))
        (for (x from to)
                (if (= 1 (length (factor x)))
        (push x primes -1)))
    primes)
[/text])

; call back routine for net-eval
(define (idle-loop p)
    (when p
        (println (p 0) ":" (p 1))
        (push (p 2) primes))
)
```

```
(println "Sending request to nodes, and waiting ...")

; machines could be on different IP addresses.
; For this test 5 nodes are started on localhost
(set 'result (net-eval (list
    (list (nodes 0 0) (nodes 0 1) (format program 0 99999 1))
    (list (nodes 1 0) (nodes 1 1) (format program 100000 199999 2))
    (list (nodes 2 0) (nodes 2 1) (format program 200000 299999 3))
    (list (nodes 3 0) (nodes 3 1) (format program 300000 399999 4))
    (list (nodes 4 0) (nodes 4 1) (format program 400000 499999 5))
) 20000 idle-loop))

(set 'primes (sort (flat primes)))
(save "primes" 'primes)

(exit)
```

At the beginning of the program a `nodes` list structure contains all the relevant node information for hostname and port.

The `program` calculates all prime numbers in a given range. The `from`, `to` and `node` variables are configured into the program text using `format`. All instructions are placed into a `begin` expression block, so only one expression result will be send back from the remote node.

Many other schemes to configure a `net-eval` parameter list are possible. The following scheme without `idle-loop` would give the same results:

```
(set 'node-eval-list (list
    (list (nodes 0 0) (nodes 0 1) (format program 0 99999 1))
    (list (nodes 1 0) (nodes 1 1) (format program 100000 199999 2))
    (list (nodes 2 0) (nodes 2 1) (format program 200000 299999 3))
    (list (nodes 3 0) (nodes 3 1) (format program 300000 399999 4))
    (list (nodes 4 0) (nodes 4 1) (format program 400000 499999 5))
))

(set 'result (net-eval node-eval-list  20000))
```

The function `idle-loop` aggregates all lists of primes received and generates the following output:

```
192.168.1.100:4711
192.168.1.101:4711
192.168.1.102:4711
192.168.1.103:4711
192.168.1.104:4711
```

As with the previous examples all IP numbers could be replaced with `"localhost"` or any other host-name or IP number to test a distributed application on a single host before deployment in a distributed environment with many networked hosts.

Transferring files

Files can be read from or written to remote nodes with the same functions used to read and write files to a local file system. This functionality is currently only available on Unix systems when talking to newLISP servers. As functions are based on standard GET and PUT HTTP protocols they can also be used communicating with web servers. Note that few Apache web-server installations have enabled the PUT protocol by default.

The functions read-file, write-file and append-file can all take URLs in their filename specifications for reading from and writing to remote nodes running a newLISP server or a web-server:

```
(write-file "http://127.0.0.1:4711//Users/newlisp/afile.txt" "The message - ")
→ "14 bytes transferred for /Users/newlisp/afile.txt\r\n"

(append-file "http://127.0.0.1:4711//Users/newlisp/afile.txt" "more text")
→ "9 bytes transferred for /Users/newlisp/afile.txt\r\n"

(read-file "http://127.0.0.1:4711//Users/newlisp/afile.txt")
→ "The message - more text"
```

The first two function return a message starting with the numbers of bytes transferred and the name of the remote file affected. The read-file function returns the contents received.

Under all error conditions an error message starting with the characters ERR: would be returned:

```
(read-file "http://127.0.0.1:4711//Users/newlisp/somefile.txt")
→ "ERR:404 File not found: /Users/newlisp/somefile.txt\r\n"
```

Note the double backslash necessary to reference files relative to root on the server node.

All functions can be used to transfer binary non-ascii contents containing zero characters. Internally newLISP uses the functions get-url and put-url, which could be used instead of the functions read-file, write-file and append-file. Additional options like used with get-url and put-url could be used with the functions read-file, write-file and append-file as well. For more detail see the newLISP function reference for these functions.

Loading and saving data

The same load and save functions used to load program or LISP data from a local file system can be used to load or save programs and LISP data from or to remote nodes.

By using URLs in the file specifications of `load` and `save` these functions can work over the network communicating with a newLISP server node.:

```
(load "http://192.168.1.2:4711//usr/share/newlisp/mysql5.lsp")

(save "http://192.168.1.2:4711//home/newlisp/data.lsp" 'db-data)
```

Although the `load` and `save` functions internally use `get-url` and `put-url` to perform its works they behave exactly as when used on a local file system, but instead of a file path URLs are specified. Both function will timeout after 60 seconds if a connection could not be established. When finer control is necessary use the functions `get-url` and `put-url` together with `eval-string` and `source` to realize a similar result as when using the `load` and `save` in HTTP mode.

Local domain Unix sockets

newLISP supports named local domain sockets in newLISP server mode and using the built-in functions `net-eval`, `net-listen`, `net-connect` together with the functions `net-accept`, `net-receive`, `net-select` and `net-send`.

Using local domain sockets fast communications between processes on the same file system and with newLISP servers is possible. See the Users Manual for more details.

§

22. HTTPD web server only mode

In all previous chapters the `-c` server mode was used. This mode can act as a `net-eval` server and at the same time answer `HTTP` requests for serving web pages or transfer of files and programs. The `-c` mode is the preferred mode for secure operation behind a firewall. newLISP also has a `-http` mode which works like a restricted `-c` mode. In `-http` mode only `HTTP` requests are served and command-line like formatted requests and `net-eval` requests are not answered. In this mode newLISP can act like a web server answering HTTP `GET`, `PUT`, `POST` and `DELETE` requests as well as `CGI` requests, but additional efforts should be made to restrict the access to unauthorized files and directories to secure the server when exposed to the internet.

Environment variables

In both server modes `-c` and `-http` the environment variables DOCUMENT_ROOT, REQUEST_METHOD, SERVER_SOFTWARE and

QUERY_STRING are set. The variables CONTENT_TYPE, CONTENT_LENGTH, HTTP_HOST, HTTP_USER_AGENT and HTTP_COOKIE are set too if present in the HTTP header sent by the client.

Pre-processing the request

When the newLISP server answers any kind of requests (`HTTP` and command line), the newLISP function `command-event` can be used to pre-process the request. The pre-processing function can be loaded from a file `httpd-conf.lsp` when starting the server:

```
server_args = httpd-conf.lsp -http -w /home/node
```

The above snippet shows part of a `xinetd` configuration file. A startup program `httpd-conf.lsp` has been added which will be loaded upon invocation of newLISP. The `-c` options has been replaced with the `-http` option. Now neither `net-eval` nor command-line requests will be answered but only HTTP requests.

The startup file could also have been added the following way when starting the server in the background from a command shell, and `httpd-conf.lsp` is in the current directory:

```
newlisp httpd-conf.lsp -http -d 80 -w /home/www &
```

All requests will be pre-processed with a function specified using `command-event` in `httpd-conf.lsp`:

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

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

All CGI requests files ending with `.exe` would be rejected and the request translated into the request of an error page.

CGI processing in HTTP mode

On http://www.newlisp.org various CGI examples can be found. In the download directory at http://www.newlisp.org/downloads two more complex applications can be found: `newlisp-ide` is a web based IDE and `newlisp-wiki` is a content management system which also runs the [http://www.newlisp.org www.newlisp.org] website.

CGI program files must have the extension `.cgi` and have executable permission on Unix.

The following is a minimum CGI program:

```
#!/usr/bin/newlisp

(print "Content-type: text/html\r\n\r\n")
(println "<h2>Hello World</h2>")
(exit)
```

newLISP normally puts out a standard `HTTP/1.0 200 OK\r\n` response header plus a `Server: newLISP v. ...\r\n` header line. If the first line of CGI program output starts with "Status:" then newLISP's standard header output is suppressed, and the CGI program must supply the full status header by itself. The following example redirects a request to a new location:

```
#!/usr/bin/newlisp
(print "Status: 301 Moved Permanently\r\n")
(print "Location: http://www.newlisp.org/index.cgi\r\n\r\n")
(exit)
```

A newLISP installation contains a module file `cgi.lsp`. This module contains subroutines for extracting parameters from HTTP GET and POST requests, extract or set cookies and other useful routines when writing CGI files. See the modules section at: http://www.newlisp.org/modules/.

Media types in HTTP modes

In both the `-c` and `-http` HTTP modes 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

§

23. Extending newLISP

newLISP has an import function, which allows importing function from DLLs (Dynamic Link Libraries) on Win32 or shared libraries on Linux/Unix (ending in .so, ending in .dylib on Mac OS X).

Simple versus extended FFI interface

In version 10.4.0 newLISP introduced an extended syntax for the `import`, `callback` and `struct` functions and for the `pack` and `unpack` support functions. This extended syntax is only available on newLISP versions built with libffi . All standard binary versions distributed on www.newlisp.org are enabled to use the new extensions additionally to the simpler API. The simpler API is used by all standard extension modules part of the distribution except for the module `gsl.lsp`.

The extended syntax allows specifying C-language types for parameter and return values of imported functions and for functions registered as callbacks. The extended syntax also allows handling of floating point values and C-structures in parameters and returns. Handling of floating point types was either impossible or unreliable using the simple API that depended on pure *cdecl* calling conventions. These are not available on all platforms. The extended API also handles packing and unpacking of C-structures with automatic alignment of C-types on different CPU architectures. See the extended syntax of the `pack` and `unpack` functions in the User Manual and Reference and OpenGL demo.

The following chapters describe the older simple API. Much of it is applicable

to the extended API as well. For details on the new API, consult the User Manual and Reference for the functions `import`, `callback`, `struct`, `pack` and `unpack`.

A shared library in C

This chapter shows how to compile and use libraries on both, Win32 and Linux/Unix platforms. We will compile a DLL and a Linux/Unix shared library from the following 'C' program.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int foo1(char * ptr, int number)
     {
     printf("the string: %s the number: %d\n", ptr, number);
     return(10 * number);
     }

char * foo2(char * ptr, int number)
     {
     char * upper;
     printf("the string: %s the number: %d\n", ptr, number);
     upper = ptr;
     while(*ptr) { *ptr = toupper(*ptr); ptr++; }
     return(upper);
     }

/* eof */
```

Both functions foo1 and foo2 print their arguments, but while foo1 returns the number multiplied 10 times, foo2 returns the uppercase of a string to show how to return strings from 'C' functions.

Compile on Unix

On Mac OS X and Linux/Unix we can compile and link `testlib.so` in one step:

```
gcc testlib.c -shared -o testlib.so
```

Or On Mac OSX/darwin do:

```
gcc testlib.c -bundle -o testlib.dylib
```

The library `testlib.so` will be built with Linux/Unix default `cdecl` conventions. Importing the library is very similar on both Linux and Win32 platforms, but on Win32 the library can be found in the current directory. You may have to specify the full path or put the library in the library path of the os:

```
> (import "/home/newlisp/testlib.so" "foo1")
```

```
foo1 <6710118F>

> (import "/home/newlisp/testlib.so" "foo2")
foo2 <671011B9>

> (foo1 "hello" 123)
the string: hello the number: 123
1230

> (foo2 "hello" 123)
the string: hello the number: 123
4054088

> (get-string (foo2 "hello" 123))
the string: hello the number: 123
"HELLO"
>
```

Again, the number returned from `foo2` is the string address pointer and get-string can be used to access the string. When using get-string only character up to a zero byte are returned. When returning the addresses to binary buffers different techniques using unpack are used to access the information.

Compile a DLL on Win32

DLLs on Win32 can be made using the MinGW, Borland or CYGWIN compilers. This example shows, how to do it using the MinGW compiler.

Compile it:

```
gcc -c testlib.c -o testlib.o
```

Before we can transform `testlib.o` into a DLL we need a `testlib.def` declaring the exported functions:

```
LIBRARY  testlib.dll
EXPORTS
         foo1
         foo2
```

Now wrap the DLL:

```
dllwrap testlib.o --def testlib.def -o testlib.dll -lws2_32
```

The library `testlib.dll` will be built with default Win32 `stdcall` conventions. The following shows an interactive session, importing the library and using the functions:

```
> (import "testlib.dll" "foo1")
foo1 <6710118F>
```

```
> (import "testlib.dll" "foo2")
foo2 <671011B9>

> (foo1 "hello" 123)
the string: hello the number: 123
1230

> (foo2 "hello" 123)
the string: hello the number: 123
4054088

> (get-string (foo2 "hello" 123))
the string: hello the number: 123
"HELLO"

>
; import a library compiled for cdecl
; calling conventions
> (import "foo.dll" "func" "cdecl")
```

Note that the first time using `foo2` the return value 4054088 is the memory address of the string returned. Using get-string the string belonging to it can be accessed. If the library is compiled using `cdecl` calling conventions, the `cdecl` keyword must be used in the import expression.

Importing data structures

Just like 'C' strings are returned using string pointers, 'C' structures can be returned using structure pointers and functions like `get-string`, `get-int` or `get-char` can be used to access the members. The following example illustrates this:

```
typedef struct mystruc
   {
   int number;
   char * ptr;
   } MYSTRUC;

MYSTRUC * foo3(char * ptr, int num )
   {
   MYSTRUC * astruc;
   astruc = malloc(sizeof(MYSTRUC));
   astruc->ptr = malloc(strlen(ptr) + 1);
   strcpy(astruc->ptr, ptr);
   astruc->number = num;
   return(astruc);
   }
```

The newLISP program would access the structure members as follows:

```
> (set 'astruc (foo3 "hello world" 123))
```

```
4054280

> (get-string (get-integer (+ astruc 4)))
"hello world"

> (get-integer astruc)
123
>
```

The return value from `foo3` is the address to the structure `astruc`. To access the string pointer, 4 must be added as the size of an integer type in the 'C' programming language. The string in the string pointer then gets accessed using get-string.

Memory management

Any allocation performed by imported foreign functions has to be deallocated manually if there's no call in the imported API to do so. The `libc` function `free` can be imported and used to free memory allocated inside imported functions:

```
(import "/usr/lib/libc.so" "free")

(free astruc) ; astruc contains memory address of allocated structure
```

In case of calling foreign functions with passing by reference, memory for variables needs to be allocated beforehand by newLISP, and hence, memory needs not be deallocated manually.

Unevenly aligned structures

Sometimes data structures contain data types of different length than the normal CPU register word:

```
struct mystruct
    {
    short int x;
    int z;
    short int y;
    } data;

struct mystruct * foo(void)
    {
    data.x = 123;
    data.y = 456;
    data.z = sizeof(data);
    return(&data);
    }
```

The `x` and `y` variables are 16 bit wide and only z takes 32 bit. When a compiler on a 32-bit CPU packs this structure the variables x and y will each fill up 32

bits instead of the 16 bit each. This is necessary so the 32-bit variable z can be aligned properly. The following code would be necessary to access the structure members:

```
> (import "/usr/home/nuevatec/test.so" "foo")
foo <281A1588>

> (unpack "lu lu lu" (foo))
(123 12 456)
```

The whole structure consumes 3 by 4 = 12 bytes, because all members have to be aligned to 32 bit borders in memory.

The following data structure packs the short 16 bit variables next to each other. This time only 8 bytes are required: 2 each for `x` and `y` and 4 bytes for `z`. Because `x` and `y` are together in one 32-bit word, none of the variables needs to cross a 32-bit boundary:

```
struct mystruct
     {
     short int x;
     short int y;
     int z;
     } data;

 struct mystruct * foo(void)
    {
    data.x = 123;
    data.y = 456;
    data.z = sizeof(data);
    return(&data);
    }
```

This time the access code in newLISP reflects the size of the structure members:

```
> (import "/usr/home/nuevatec/test.so" "foo")
foo <281A1588>

> (unpack "u u lu" (foo))
(123 456 8)
```

Passing parameters

data Type	newLISP call	C function call
integer	(foo 123)	foo(int number)
double float	(foo 1.234)	foo(double number)
float	(foo (flt 1.234))	foo(float number)
string	(foo "Hello World!")	foo(char * string)
integer array	(foo (pack "d d d" 123 456 789))	foo(int numbers[])

float array	(foo (pack "f f f" 1.23 4.56 7.89))	foo(float[])
double array	(foo (pack "lf lf lf) 1.23 4.56 7.89)	foo(double[])
string array	(foo (pack "lu lu lu" "one" "two" "three")))	foo(char * string[])

Note that floats and double floats are only passed correctly on x86 platforms with *cdecl* calling conventions or when passed by pointer reference as in variable argument functions, i.e: printf(). For reliable handling of single and double precision floating point types and for advanced usage of pack and unpack for handling C-structures, see the descriptions of the import, callback and struct functions in the newLISP User Manual and Reference.

pack can receive multiple arguments after the format specifier in a list too:

```
(pack "lu lu lu" '("one" "two" "three"))
```

Extracting return values

data Type	newLISP to extract return value	C return
integer	(set 'number (foo x y z))	return(int number)
double float	n/a - only 32bit returns, use double float pointer instead	not available
double float ptr	(set 'number (get-float (foo x y z)))	return(double * numPtr)
float	not available	not available
string	(set 'string (get-string (foo x y z)	return(char * string)
integer array	(set 'numList (unpack "ld ld ld" (foo x y z)))	return(int numList[])
float array	(set 'numList (unpack "f f f" (foo x y z)))	return(float numList[])
double array	(set 'numList (unpack "lf lf lf") (foo x y z)))	return(double numList[])
string array	(set 'stringList (map get-string (unpack "ld ld ld" (foo x y z))))	return(char * string[])

Floats and doubles can only be returned via address pointer references.

When returning array types the number of elements in the array must be known. The examples always assume 3 elements.

All pack and unpack and formats can also be given without spaces, but are spaced in the examples for better readability.

The formats "ld" and "lu" are interchangeable, but the 16-bit formats "u" and "d" may produce different results, because of sign extension when going from unsigned 16 bits to signed 32 or 64-bits bits of newLISP's internal integer format.

Flags are available for changing endian byte order during `pack` and `unpack`.

Writing library wrappers

Sometimes the simple version of newLISP's built-in import facility cannot be used with a library. This happens whenever a library does not strictly adhere to `cdecl` calling conventions expecting all parameters passed on the stack. E.g. when running Mac OS X on older PPC CPUs instead of Intel CPUs, the OpenGL libraries installed by default on Mac OS X cannot be used.

Since newLISP version 10.4.0, the problem can be solved easiest using the newer extended syntax of `import`, which automatically resolves platform and architectural differences. On very small systems or whenever the needed `libffi` system library is not present on a platform, a special wrapper library can be built to translate `cdecl` conventions expected by newLISP into the calling conventions expected by the target library.

```
/* wrapper.c - demo for wrapping library function

compile:
    gcc -m32 -shared wrapper.c -o wrapper.so
or:
    gcc -m32 -bundle wrapper.c -o wrapper.dylib

usage from newLISP:

    (import "./wrapper.dylib" "wrapperFoo")

    (define (glFoo x y z)
        (get-float (wrapperFoo 5 (float x) (int y) (float z))) )

 (glFoo 1.2 3 1.4) => 7.8

*/

#include <stdio.h>
#include <stdarg.h>


/* the glFoo() function would normally live in the library to be
   wrapped, e.g. libopengl.so or libopengl.dylib, and this
   program would link to it.
   For demo and test purpose the function has been included in this
   file
*/

double glFoo(double x, int y, double z)
    {
    double result;

    result = (x + z) * y;
```

```
    return(result);
    }

/* this is the wrapper for glFoo which gets imported by newLISP
   declaring it as a va-arg function guarantees 'cdecl'
   calling conventions on most architectures
*/

double * wrapperFoo(int argc, ...)
    {
    va_list ap;
    double x, z;
    static double result;
    int y;

    va_start(ap, argc);

    x = va_arg(ap, double);
    y = va_arg(ap, int);
    z = va_arg(ap, double);

    va_end(ap);

    result = glFoo(x, y, z);
    return(&result);
    }

/* eof */
```

Registering callbacks in external libraries

Many shared libraries allow registering callback functions to call back into the controlling program. The function `callback` is used in newLISP to extract the function address from a user-defined newLISP function and pass it to the external library via a registering function:

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

(glutKeyboardFunc (callback 1 'keyboard))
```

The example is a snippet from the file `opengl-demo.lsp` in the `newlisp-x.x.x/examples/` directory of the source distribution. A file `win32demo.lsp` can be found in the same directory demonstrating callbacks on the Windows platform.

For an advanced syntax of `callback` using C-type specifiers see newLISP User Manual and Reference.

§

24. newLISP as a shared library

On all platforms, newLISP can be compiled as a shared library. On Win32, the library is called `newlisp.dll`, on Mac OS X `newlisp.dylib` and on Linux and BSDs, the library is called `newlisp.so`. Makefiles are included in the source distribution for most platforms. Only on Win32, the installer comes with a precompiled `newlisp.dll` and will install it in the `WINDOWS\system32\` directory (since v.10.3.3) and in the `Program Files/newlisp/` directory.

Evaluating code in the shared library

The first example shows how to import `newlispEvalStr` from newLISP itself as the caller:

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

When calling the library function `newlispEvalStr`, output normally directed to the console (e.g. return values or print statements) is returned in the form of an integer string pointer. The output can be accessed by passing this pointer to the `get-string` function. To silence the output from return values, use the `silent` function. All Results, even if they are numbers, are always returned as strings and a trailing linefeed as in interactive console mode. Use the `int` or `float` functions to convert the strings to other data types.

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])
```

The second example shows how to import `newlispEvalStr` into a C-program:

```
/* libdemo.c - demo for importing newlisp.so
 *
 * compile using:
 *    gcc -ldl libdemo.c -o libdemo
 *
 * use:
 *
 *    ./libdemo '(+ 3 4)'
 *    ./libdemo '(symbols)'
```

```
 *
 */
#include <stdio.h>
#include <dlfcn.h>

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

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

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

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

return(0);
}

/* eof */
```

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

Registering callbacks

Like many other share libraries, callbacks can be registered in newLISP library. The function `newlispCallback` must be imported and is used for registering callback functions. The example shows newLISP importing newLISP as a library and registering a callback `callme`:

```
#!/usr/bin/newlisp

; path-name of the library depending on platform
(set 'LIBRARY (if (= ostype "Win32") "newlisp.dll" "newlisp.dylib"))

; import functions from the newLISP shared library
(import LIBRARY "newlispEvalStr")
(import LIBRARY "newlispCallback")

; set calltype platform specific
```

```
(set 'CALLTYPE (if (= ostype "Win32") "stdcall" "cdecl"))

; the callback function
(define (callme p1 p2 p3 result)
    (println "p1 => " p1 " p2 => " p2 " p3 => " p3)
    result)

; register the callback with newLISP library
(newlispCallback "callme" (callback 0 'callme) CALLTYPE)

; the callback returns a string
(println (get-string (newlispEvalStr
    {(get-string (callme 123 456 789 "hello world"))})))

; the callback returns a number
(println (get-string (newlispEvalStr
    {(callme 123 456 789 99999)})))
```

Depending on the type of the return value, different code is used. The program shows the following output:

```
p1 => 123 p2 => 456 p3 => 789
"hello world"

p1 => 123 p2 => 456 p3 => 789
99999
```

Note that Win32 and many Unix flavors will look for `newlisp.dll` in the system library path, but Mac OS X will look for `newlisp.dylib` first in the current directory, if the full file path is not specified. The program above can also be found as `callback` in the source distribution in the `newlisp-x.x.x/examples` directory.

§

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.

∂