newLISP

INTRODUCTION

August 29, 2008

Copyright notices.

This document was created on: 2008-08-29 22:45:12 and built using TextWrangler, Scrivener, newLISP, and ConTeXt.
newLISP version: 9.4.05

Contents

1 Introduction to newLISP

Welcome to newLISP

Welcome to this introduction to newLISP. If you've done some basic programming or scripting, you'll find newLISP easy to learn and powerful, combining the power and elegance of classic LISP with the facilities of a modern scripting language, such as regular expressions, network functions, Unicode support, multitasking, and many others.

This is intended to be a straightforward and simple description of the basics of the language, for people who haven't used LISP before but who have probably dabbled in other programming and scripting languages. I hope it's enough to get you started with the language, and ready to explore the real power of newLISP.

I'm writing this on a MacOS X system, but it shouldn't make any difference if you're using Linux, Windows, or one of the many other platforms that newLISP supports. The examples are designed to be run in newLISP version 9.4.

Resources

- the main newLISP web site, `http://www.newlisp.org`, which provides a helpful forum, the excellent documentation, and the newLISP software
- the newLISP on Noodles wiki, `newlisp-on-noodles.org/`
- John Small's excellent 21 minute introduction, newLISP in 21 minutes, at `http://newlisp.org/newlisp-in-21-minutes.html`
- the fine newLISP dragonfly logo, fashioned from 11 pairs of parentheses, is designed by Brian Grayless (`http://fudnik.com`)
- If you have comments, criticisms, or suggestions about this document, please send them to me, at **cormullion - at - mac.com**.

Now is a good time to say thanks to everyone who has contributed to earlier versions of this document, by suggesting additions or finding errors.

2 The basics

Getting started

Once you've installed newLISP, there are various ways to run it. See the newLISP documentation for full details. The most direct way is to run the newLISP interpreter from the command line – in a console or terminal window – by typing the `newlisp` command.

```
$ newlisp
newLISP v.9.4.5 on OSX IPv4 UTF-8, execute 'newlisp -h' for more info.

>
```

This is good for trying out short expressions, testing ideas, and for debugging. You can write multi-line code in this environment by enclosing the lines between `[cmd]` and `[/cmd]`.

```
>[cmd]
(define (fibonacci n)
 (if (< n 2)
  1
  (+ (fibonacci (- n 1))
     (fibonacci (- n 2)))))
[/cmd]
>(fibonacci 10)
89
>
```

The visual interface to newLISP, newLISP-GS, provides a graphical toolkit for newLISP applications, and it also gives you a development environment in which to write and test code: the newLISP editor. On Windows, this is installed as a desktop icon and a folder in the Program Start menu. On MacOS X, an application package and icon is installed in the Applications folder. The newLISP-GS editor provides you with multiple tabbed windows, syntax colouring, and a monitor area for seeing the results of running your code.

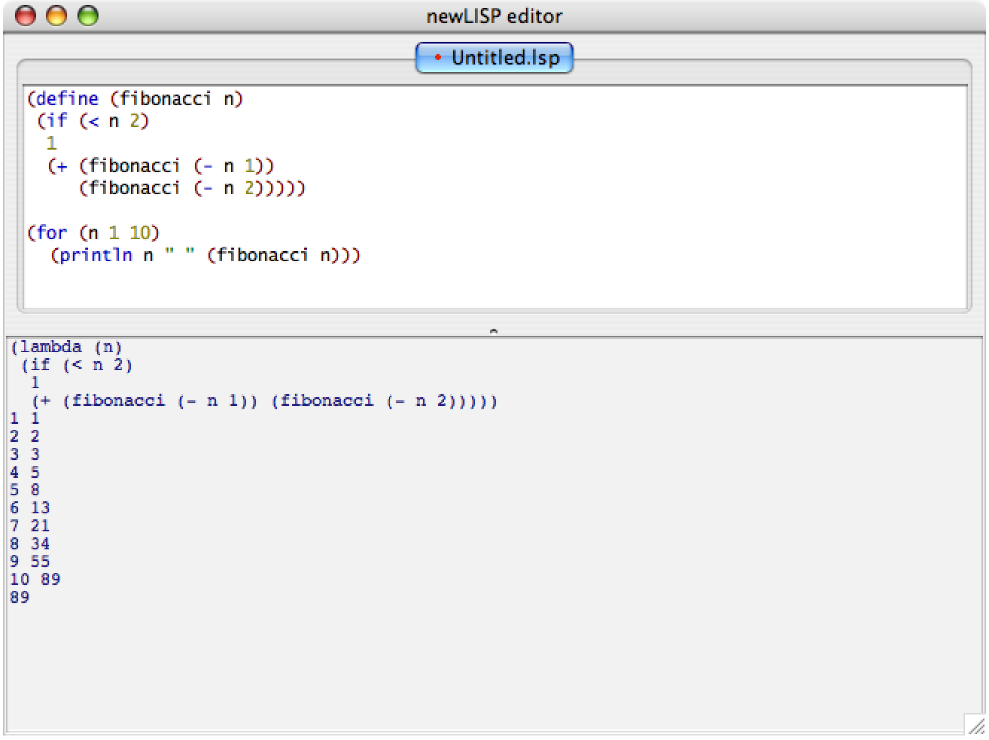
You can also run the newLISP-GS editor from the command line: you can find the file at C:/Program Files/newlisp/newlisp-edit.lsp (Windows), or at /usr/bin/newlisp-edit.lsp (on Unix). (This is a newLISP source file, so you can look at the code too.)

You can edit newLISP scripts in your favourite text editor. On MacOS X you can use BBEdit, TextWrangler, or TextMate to run newLISP scripts, or you can use one of the pre-installed Unix text editors such as vim or emacs. On Windows you can use UltraEdit, EditPlus, or NotePad++, to name just a few. If you use Linux, you probably already know more about text editors than I do, and you probably already have a preference.

The newLISP web site hosts configuration files for a number of popular editors at:

`http://newlisp.org/index.cgi?Code_Contributions`.

On Unix, the first line of a newLISP script should be:

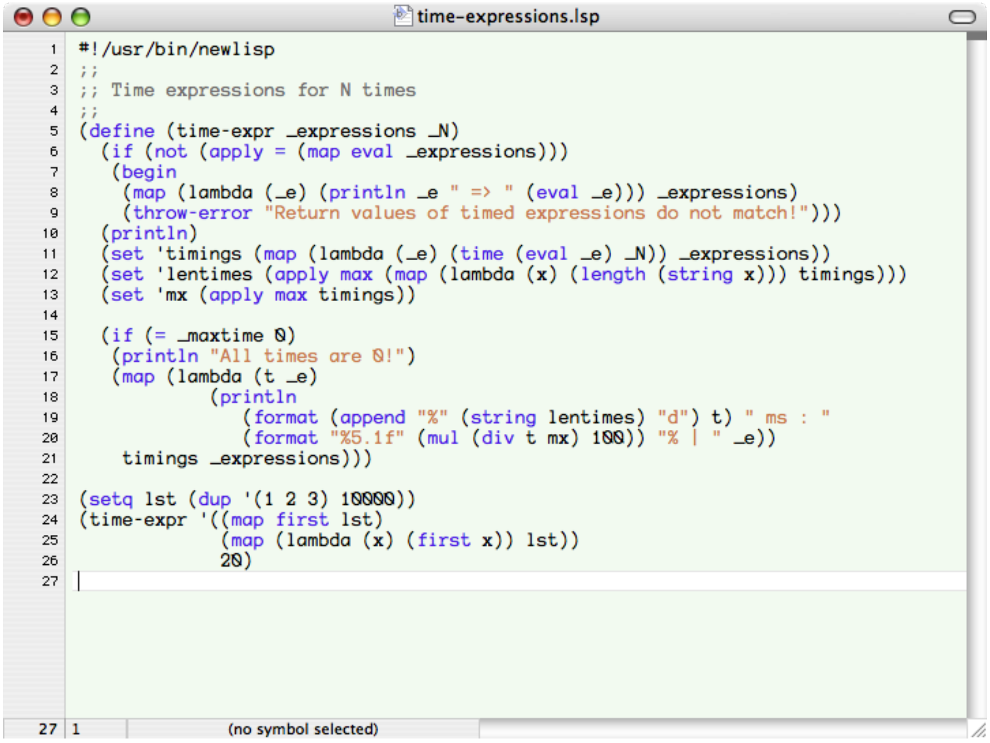


```
(define (fibonacci n)
 (if (< n 2)
  1
  (+ (fibonacci (- n 1))
     (fibonacci (- n 2)))))

(for (n 1 10)
  (println n " " (fibonacci n)))
```

```
(lambda (n)
 (if (< n 2)
  1
  (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
1 1
2 2
3 3
4 5
5 8
6 13
7 21
8 34
9 55
10 89
89
```

Figure 2.1 the newLISP-GS Editor



```
#!/usr/bin/newlisp
;;
;; Time expressions for N times
;;
(define (time-expr _expressions _N)
  (if (not (apply = (map eval _expressions)))
   (begin
    (map (lambda (_e) (println _e " => " (eval _e))) _expressions)
    (throw-error "Return values of timed expressions do not match!")))
  (println)
  (set 'timings (map (lambda (_e) (time (eval _e) _N)) _expressions))
  (set 'lentimes (apply max (map (lambda (x) (length (string x))) timings)))
  (set 'mx (apply max timings))

  (if (= _maxtime 0)
   (println "All times are 0!")
   (map (lambda (t _e)
             (println
                (format (append "%" (string lentimes) "d") t) " ms : "
                (format "%5.1f" (mul (div t mx) 100)) "% | " _e))
    timings _expressions)))

(setq lst (dup '(1 2 3) 10000))
(time-expr '((map first lst)
            (map (lambda (x) (first x)) lst))
            20)
```

Figure 2.2 Editing newLISP in BBEdit

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

or:

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

If you prefer to run newLISP from an external text editor such as TextWrangler, you'll have to use more **println** functions if you want to see the value returned by every function or expression.

Generally, you end a newLISP script or a console session using **exit**:

```
(exit)
```

The three basic rules of newLISP

You have to learn just three basic rules to program in newLISP. Here's the first one:

Rule 1: a list is a sequence of elements

A list is a sequence of elements enclosed in parentheses:

```
(1 2 3 4 5)                ; a list of integers
("the" "cat" "sat")        ; a list of strings
(x y z foo bar)            ; a list of symbol names
(sin cos tan atan)         ; a list of newLISP functions
(1 2 "stitch" x sin)       ; a mixed list
(1 2 (1 2 3) 3 4 )         ; a list with a list inside it
((1 2) (3 4) (5 6))        ; a list of lists
```

The list is the basic data structure in newLISP, and it's also the way you write your program code. But don't type these examples in just yet – there are two more rules to learn!

Rule 2: the first element in a list is special

When newLISP sees a list, it treats the first element as a function, and then tries to use the remaining elements as the information the function needs.

```
(+ 2 2)
```

This is a list of three elements: the function called **+**, followed by two numbers. When newLISP sees this list, it processes it and returns the value 4 (of course). Notice that the first element was treated by newLISP as a function, whereas the rest of the elements were interpreted as arguments to that function – numbers that the function expects.

Here are some more examples that demonstrate these first two rules:

```
(+ 1 2 3 4 5 6 7 8 9)
```

returns 45. The **+** function adds up all the numbers in the list.

```
(max 1 1.2 12.1 12.2 1.3 1.2 12.3)
```

returns 12.3, the biggest number in the list. Again, there's no (reasonable) limit to the length of the list: if a function is happy to accept 137 items (which both **max** and **+** are), than you can pass it 137 items.

```
(print "the sun has put his hat on")
```

```
"the sun has put his hat on"
```

prints the string of characters 'the sun has put his hat on'. (It also returns the string, which is why, when you're working in the console, you sometimes see things repeated twice.) The **print** function can print out a single string of characters, or you can supply a sequence of elements to print:

```
(print 1 2 "buckle" "my" "shoe")
```

```
12bucklemyshoe
```

which prints the two numbers and the three strings (although not very well formatted, because you haven't met the **format** function yet).

The **directory** function:

```
(directory "/")
```

produces a listing of the specified directory, in this case the root directory, /:

```
("." ".." ".DS_Store" ".hotfiles.btree" ".Spotlight-V100"
".Trashes"".vol" ".VolumeIcon.icns" "Applications"
"automount" "bin" "cores" "Desktop DB" "Desktop DF"
"Desktop Folder" "dev""Developer" "etc" "Library"
"mach" "mach.sym" "mach_kernel" "Network" "private"
"sbin" "System" "System Folder" "TheVolumeSettingsFolder"
"tmp" "User Guides And Information" "Users" "usr"
"var" "Volumes")
```

It lists the current directory if you don't specify one:

```
(directory)
```

There's a **read-file** function that reads in the contents of a text file:

```
(read-file "/usr/share/newlisp/modules/stat.lsp")
```

Here the function wants a single argument – the file name – and returns the contents of the file to you, in a string.

These are typical examples of the building blocks of newLISP code – a list containing a function call, followed perhaps by any extra information the function requires. There are over 300 newLISP functions, and you can refer to the excellent newLISP reference manual for details of all of them and how to use them.

You can try these examples. If you're using newLISP at a terminal, just type them in. If you're typing the lines into a text editor and running it as a script, you won't necessarily see the result of a function call unless you enclose the expression in the **println** function. For example, type:

```
(println (read-file "/usr/share/newlisp/modules/stat.lsp"))
```

to print the results of the **read-file** function.

There's one more useful thing to look at before you meet the third rule.

Nested lists

We've already spotted one list nesting inside another. Here's another example:

```
(* (+ 1 2) (+ 3 4))
```

When newLISP sees this, it 'thinks' as follows:

Hmm. Let's start with the first of those inner lists. I can do

```
(+ 1 2)
```

easily. The value of that is 3. I can also do the second list

```
(+ 3 4)
```

easily enough. That evaluates to 7.

So if I replace these two inner lists with these values, I get

```
(* 3 7)
```

which is really easy. I'll return the value 21 for this expression.

```
(* (+ 1 2) (+ 3 4))
(* (+ 1 2) 7)
(* 3 7)
21
```

See those two right parentheses at the end of the first line, after the 4? Both are essential: the first one finishes the `(+ 3 4` list, and the second one finishes the multiplication operation that started with `(*`. When you start writing more complicated code, you'll find that you are putting lists inside lists inside lists inside lists, and you might be ending some of the more complicated definitions with half a dozen right parentheses. A good editor will help you keep track of them.

But the good thing is, you don't have to worry about white space, line terminators, various punctuation marks, or compulsory indentation. And because all your data and your code are stored in the same way, in lists, you can mix them freely. More on that later.

Some people worry about the proliferation of parentheses when they first see LISP code. Others refer to them as 'nail clippings' or say that LISP stands for 'Lots of Irritating Silly Parentheses'. But I prefer to think of the parentheses as small 'handles' that enclose a newLISP 'thought':

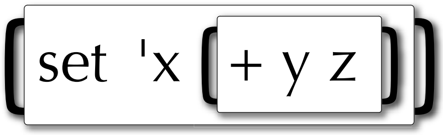


Figure 2.3 Parentheses as 'handles' enclosing a thought

When you're editing newLISP code in a good text editor, you can easily move or edit a thought by 'grabbing its handles', and easily select a thought with a Balance Parentheses command. You'll soon find the parentheses more useful than you first thought!

Quoting prevents evaluation

You can now meet the third rule of programming with newLISP:

Rule 3: Quoting prevents evaluating

To stop newLISP evaluating something, quote it.

Compare these two lines:

```
(+ 2 2)
'(+ 2 2)
```

The first line is a list which contains a function and two numbers. In the second line, the list is quoted – preceded by a single quote or apostrophe ('). You don't need to put another quote at the end, because one is sufficient.

```
> (+ 2 2)
4
> '(+ 2 2)
(+ 2 2)
>
```

For the first expression, newLISP does its job as usual, and enthusiastically evaluates the list, returning the number 4. But for the second expression, as soon as it sees the quotation mark, newLISP doesn't even think about evaluating the list by adding the numbers; it just returns the list, unevaluated.

> This quotation mark does the same job in newLISP that opening and closing quotation marks do in written English – they inform the reader that the word or phrase is not to be interpreted normally, but treated specially in some way: a non-standard or ironic meaning, perhaps, or something spoken by another person.

So why do you want to stop newLISP evaluating things? You'll soon meet some examples where you quote things to prevent newLISP thinking that the first item in a list is a function. For example, when you store information in a list, you don't want newLISP to evaluate them in the usual way:

```
(2006 1 12)                 ; today's year/month/date
("Arthur" "J" "Chopin")     ; someone's full name
```

You don't want newLISP to look for functions called `2006` or `"Arthur"`. Besides, 2006 isn't a valid function name, because it starts with a digit, and function names can't start with a double quotation mark, so in either case your program will stop with an error. Therefore you *quote* the lists to stop their first elements being used as functions rather than data:

```
'(2006 1 12)                 ; evaluates to (2006 1 12)
'("Arthur" "J" "Chopin")     ; evaluates to ("Arthur" "J" "Chopin")
```

newLISP's ability to treat program code as data – and data as program code – is discussed in more detail later.

Use the vertical apostrophe (ASCII code 39) to quote lists and symbols. Unfortunately, the software that generates the PDF version of this document converts vertical apostrophes into right quotation marks. If you copy and paste any newLISP code from the PDF document, remember to change the quotation marks to vertical apostrophes.

Symbols and quotes

A symbol is a newLISP 'thing' with a name. You define something in your code and assign a name to it. Then you can refer to that something later on, using the name rather than the contents. For example, after typing this:

```
(define alphabet "abcdefghijklmnopqrstuvwxyz")
```

there's now a new symbol called `alphabet` whose value is a string consisting of the 26 letters of the alphabet. The **define** function stores the string of characters from a to z in the symbol `alphabet`. Now this symbol can be used elsewhere, and will evaluate to the alphabet whenever it's used. Whenever you want to use the 26 letters of the alphabet, you use this symbol without quoting it. For example, here's the **upper-case** function:

```
(upper-case alphabet)
```

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

I use the symbol without quoting it, because I want newLISP to use the value of the symbol, not its name. I'm really not interested in upper-casing the word 'alphabet', but the alphabet itself. newLISP hasn't permanently changed the value of the symbol, in this example, because **upper-case** always creates and returns a new string, leaving the one stored in the symbol unchanged.

Symbols correspond to variables in other programming languages. In fact, newLISP doesn't use symbols as much as other languages use variables. This is partly because values are continually being returned by expressions and fed directly into other expressions without being stored. For example, in the following code each function hands its result directly to the next, 'enclosing' function:

```
(println (first (upper-case alphabet)))
```

```
"A"
```

upper-case gives its return value directly to **first**, which gives its return value directly to **println**, which both prints it and gives you the string it printed as the return value. So there's less need to store values temporarily. But there are plenty of other places where you do want symbols.

Here are two more examples of symbol-quoting:

```
(define x (+ 2 2 ))
(define y '(+ 2 2))
```

In the first example, I don't quote the `(+ 2 2)` list – newLISP evaluates this to 4 and then assigns 4 to the symbol `x`, which evaluates to 4:

```
x
;-> 4
```

In the second example I quote the list. This means that the symbol `y` is now holding a list rather than a number. Whenever newLISP sees the symbol `y`, it will return the list, rather than 4. (Unless, of course, you quote `y` first as well!)

```
y
;-> (+ 2 2)
'y
;-> y
```

By the way, throughout this document:

```
; the semicolon is the comment character
;-> that ";->" is my way of saying "the value is"
```

Every function expression returns a value. Even a **println** function returns a value. You could say that the printing action is actually just a side effect, and its main task is to return a value. You might notice that when you use **println** interactively, you see the return value twice: once when it's printed, and again when the value is returned to the calling function (or to the top-most level).

Setting and defining symbols

You can create and set the value of symbols using either **define** or **set**. Used in the following way, they're interchangeable:

```
(define x (+ 2 2 ))
;-> 4
(set 'y (+ 2 2 ))
;-> 4
```

In **set**, the first argument is evaluated before it's used, and it should evaluate to a symbol. So you should either quote a symbol to prevent it being evaluated, or supply an expression that evaluates to a symbol.

define is also used to define functions. See "**Make your own functions**" on page **27**.

Destructive functions

Some newLISP functions modify the values of the symbols that they operate on, others create a copy of the value and return that. Technically, the ones that modify the contents of symbols are described as 'destructive' functions – although you'll often be using them to create new data. In this document I'll describe functions such as **push** and **replace** as destructive, although this simply means that they change the value of something rather than return a modified copy.

3 Controlling the flow

There are many different ways to control the flow of your code. If you've used other scripting languages, you'll probably find your favourites here, and many more besides.

All the control flow functions obey the standard rules of newLISP. The general form of each is usually a list in which the first element is a keyword followed by one or more expressions to be evaluated:

```
(keyword expression1 expression2 expression3 ...)
```

Tests: if...

Perhaps the simplest control structure you can write in any language is a simple **if** list, consisting of a test and an action:

```
(if button-pressed? (launch-missile))
```

The second expression, a call to the `launch-missile` function, is evaluated only if the symbol `button-pressed?` evaluates to 'true'. 1 is true. 0 is true – it's a number, after all. -1 is true. Most of the things that newLISP already knows about are true. There are two important things that newLISP knows are false rather than true: **nil** and the empty list (). And anything that newLISP doesn't know the value of is false.

```
(if x 1)
; if x is true, return the value 1

(if 1 (launch-missile))
; missiles are launched, because 1 is true

(if 0 (launch-missile))
; missiles are launched, because 0 is true

(if nil (launch-missile))
;-> nil, there's no launch, because nil is false

(if '() (launch-missile))
;-> (), and the missiles aren't launched
```

You can use anything that evaluates to either true or false as a test:

```
(if (> 4 3) (launch-missile))
;-> it's true that 4 > 3, so the missiles are launched

(if (> 4 3) (println "4 is bigger than 3"))
```

```
"4 is bigger than 3"
```

If a symbol evaluates to **nil** (perhaps because it doesn't exist or hasn't been assigned a value), newLISP considers it false and the test returns **nil**:

```
(if snark (launch-missile))
;-> nil ; that symbol has no value

(if boojum (launch-missile))
;-> nil ; can't find a value for that symbol

(if untrue (launch-missile))
;-> nil ; can't find a value for that symbol either

(if false (launch-missile))
;-> nil
; never heard of it, and it doesn't have a value
```

You can add a third expression, which is the 'else' action. If the test expression evaluates to **nil** or **()**, the third expression is evaluated, rather than the second, which is ignored:

```
(if x 1 2)
; if x is true, return 1, otherwise return 2

(if 1
 (launch-missile)
 (cancel-alert))
; missiles are launched

(if nil
 (launch-missile)
 (cancel-alert))
; alert is cancelled

(if false
 (launch-missile)
 (cancel-alert))
; alert is cancelled
```

Here's a typical real-world three-part **if** function, formatted to show the structure as clearly as possible:

```
(if (and socket (net-confirm-request))     ; test
    (net-flush)                            ; action when true
    (finish "could not connect"))          ; action when false
```

Although there are two expressions after the test – `(net-flush)` and `(finish ...)` – only one of them will be evaluated.

The lack of the familiar 'signpost' words such as 'then' and 'else' that you find in other languages, can catch you out if you're not concentrating! But you can easily put comments in.

You can use **if** with an unlimited number of tests and actions. In this case, the **if** list consists of a series of test-action pairs. newLISP works through the pairs until one of the tests succeeds, then evaluates that test's corresponding action. If you can, format the list in columns to make the structure more apparent:

```
(if
 (< x 0)       (define a "impossible")
 (< x 10)      (define a "small")
 (< x 20)      (define a "medium")
 (>= x 20)     (define a "large")
 )
```

If you've used other LISP dialects, you might recognise that this is a simple alternative to **cond**, the conditional function. newLISP provides the traditional **cond** structure as well. See "**Selection: if, cond, and case**" on page **25**.

You might be wondering how to do two or more actions if a test is successful or not. There are two ways to do this. You can use **when**, which is like an **if** without an 'else' part.

```
(when (> x 0)
  (define a "positive")
  (define b "not zero")
  (define c "not negative"))
```

Another way is to define a block of expressions that form a single expression that you can use in an **if** expression. I'll discuss how to do this shortly, in "**Blocks: groups of expressions**" on page **22**.

Earlier, I said that when newLISP sees a list, it treats the first element as a function. I should also mention that it evaluates the first element first before applying it to the arguments:

```
(define x 1)
((if (< x 5) + *) 3 4)          ; which function to use, + or *?
7                               ; it added
```

Here, the first element of the expression, `(if (< x 5) + *)`, returns an arithmetic operator depending on the results of a test comparing `x` with 5. So the whole expression is either an addition or multiplication, depending on the value of `x`.

```
(define x 10)
;-> 10

((if (< x 5) + *) 3 4)
12                               ; it multiplied
```

This technique can help you write concise code. Instead of this:

```
(if (< x 5) (+ 3 4) (* 3 4))
```

you could write this:

```
((if (< x 5) + *) 3 4)
```

which evaluates like this:

```
;-> ((if (< x 5) + *) 3 4)
;-> ((if true + *) 3 4)
;-> (+ 3 4)
;-> 7
```

Notice how every expression returns a value to the enclosing function. In newLISP *every* expression returns some value, even an **if** expression:

```
(define x (if flag 1 -1))       ; x is either 1 or -1
```

```
(define result
 (if
     (< x 0)      "impossible"
     (< x 10)     "small"
     (< x 20)     "medium"
                  "large"))
```

The value of `x` depends on the value returned by **if** expression. Now the symbol `result` contains a string depending on the value of `x`.

Looping

Sometimes you want to repeat a series of actions more than once, going round in a loop. There are various possibilities. You might want to do the actions:

- on every item in a list
- on every item in a string
- a certain number of times
- until something happens
- while some condition prevails

newLISP has a solution for all of these, and more.

Working through a list

newLISP programmers love lists, so **dolist** is a most useful function that sets a local loop symbol (variable) to each item of a list in turn, and runs a series of actions on each. Put the name for the loop variable and the list to be scanned, in parentheses, after **dolist**, then follow it with the actions.

In the following example, I set another symbol called `counter` as well, before defining a local loop variable `i` that will hold each value of the list of numbers generated by the **sequence** function:

```
(define counter 1)
(dolist (i (sequence -5 5))
 (println "Element " counter ": " i)
 (inc 'counter))                          ; increment counter by 1
```

```
Element 1: -5
Element 2: -4
Element 3: -3
Element 4: -2
Element 5: -1
Element 6: 0
Element 7: 1
Element 8: 2
Element 9: 3
Element 10: 4
Element 11: 5
```

Notice that, unlike **if**, the **dolist** function and many other control words let you write a series of expressions one after the other: here both the **println** and the **inc** (increment) functions are called for each element of the list.

There's a useful shortcut for accessing a system-maintained loop counter. Just now I used a counter symbol, incremented each time through the loop, to keep track of how far we'd got in the list. However, newLISP automatically maintains a loop counter for you, in a system variable called

`$idx`, so I can dispense with the counter symbol, and just retrieve the value of `$idx` each time through the loop:

```
(dolist (i (sequence -5 5))
 (println "Element " $idx ": " i))
```

```
Element 0: -5
Element 1: -4
Element 2: -3
Element 3: -2
Element 4: -1
Element 5: 0
Element 6: 1
Element 7: 2
Element 8: 3
Element 9: 4
Element 10: 5
```

In some situations, you might prefer to use the mapping function **map** for processing a list (described later – see "**Apply and map: applying functions to lists**" on page **81**). **map** can be used to apply a function (either an existing function or a temporary definition) to every element in a list, without going through the list using a local variable. For example, let's use **map** to produce the same output as the above **dolist** function. I define a temporary 'print and increase' function consisting of two expressions, and apply this to each element of the list produced by **sequence**:

```
(define counter 1)
(map  (fn (i)
  (println "Element " counter ": " i)
  (inc 'counter))
  (sequence -5 5))
```

Experienced LISP programmers may be more familiar with **lambda**. **fn** is a synonym for **lambda**: use whichever one you prefer.

You might also find **flat** useful for working through lists, because it flattens out lists containing nested lists for easier processing, by copying:

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

to

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

for example. See "**flat**" on page **40**.

To work through the arguments supplied to a function, you can use the **doargs** function. See "**Arguments: args**" on page **31**.

Working through a string

You can step through every character in a string using the equivalent of **dolist**, **dostring**.

```
(define alphabet "abcdefghijklmnopqrstuvwxyz")
(dostring (letter alphabet)
    (print letter { }))
```

```
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122
```

The numbers are the ASCII/Unicode codes.

A certain number of times

If you want to do something a fixed number of times, use **dotimes** or **for**. **dotimes** carries out a given number of repeats of the actions in the body of the list. You should provide a name for the local variable, just like you did with **dolist**:

```
(dotimes (c 10)
 (println c " times 3 is " (* c 3)))
```

```
0 times 3 is 0
1 times 3 is 3
2 times 3 is 6
3 times 3 is 9
4 times 3 is 12
5 times 3 is 15
6 times 3 is 18
7 times 3 is 21
8 times 3 is 24
9 times 3 is 27
```

You must supply a local variable with these forms. Even if you don't use it, you have to provide one.

Notice that counting starts at 0 and continues to `n - 1`, never actually reaching the specified value. Programmers think this is sensible and logical; non-programmers just have to get used to starting their counting at 0 and specifying 10 to get 0 to 9.

> One way to remember this is to think about birthdays. You celebrate your 1st birthday when you complete your first year, during which time you were 0 years old. You complete your first 10 years of life when you start celebrating your 10th birthday, which starts when you stop being 9. The newLISP function **first** gets the element with index number 0...

Use **dotimes** when you know the number of repetitions, but use **for** when you want newLISP to work out how many repetitions should be made, given start, end, and step values:

```
(for (c 1 -1 .5)
 (println c))
```

```
1
0.5
0
-0.5
-1
```

Here newLISP is smart enough to work out that I wanted to step down from 1 to -1 in steps of 0.5.

Just to remind you of this 'counting from 0' thing again, compare the following two functions:

```
(for (x 1 10) (println x))
```

```
1
...
10
```

```
(dotimes (x 10) (println x))
```

```
0
...
9
```

An escape route is available

for, **dotimes**, and **dolist** like to loop, carrying out a set of actions over and over again. Usually the repetition continues, inexorably, until the limit – the final number, or the last item in the list – is reached. But you can specify an emergency escape route in the form of a test to be carried out before the next loop starts. If this test returns true, the next loop isn't started, and the expression finishes earlier than usual. This provides you with a way of stopping before the official final iteration.

For example, suppose that you want to halve every number in a list, but (for some reason) wanted to stop if one of the numbers was odd. Compare the first and second versions of this **dolist** expression:

```
(define number-list '(100 300 500 701 900 1100 1300 1500))
; first version
(dolist (n number-list)
 (println (/ n 2)))
```

```
50
150
250
350
450
550
650
750
```

```
; second version
(dolist (n number-list (!= (mod n 2) 0)) ; escape if true
 (println (/ n 2)))
```

```
50
150
250
```

The second version stops looping if the test for `n` being odd, `(!= (mod n 2) 0)`, returns true.

> Notice the use of integer-only division here. I've used **/** rather than the floating-point division operator **div** as part of the example. Don't use one if you want the other!

You can supply escape route tests with **for** and **dotimes** too.

For more complex flow control, you can use **catch** and **throw**. **throw** passes an expression to the previous **catch** expression which completes and returns the value of the expression:

```
(catch
 (for (i 0 9)
 (if (= i 5) (throw (string "i was " i)))
 (print i " ")))
```

The output is:

```
0 1 2 3 4
```

and the **catch** expression returns `i was 5`.

You can also devise flows using Boolean functions. See "**Blocks: groups of expressions**" on page **22**.

Until something happens, or while something is true

You might have a test for a situation that returns **nil** or () when something interesting happens, but otherwise returns a true value, which you're not interested in. To repeatedly carry out a series of actions until the test fails, use **until** or **do-until**:

```
(until (disk-full?)
 (println "Adding another file")
 (add-file)
 (inc 'count))

(do-until (disk-full?)
 (println "Adding another file")
 (add-file)
 (inc 'count))
```

The difference between these two is to do with when the test is carried out. In **until**, the test is made first, then the actions in the body are evaluated if the test fails. In **do-until**, the actions in the body are evaluated first, before the test is made, then the test is made to see if another loop is possible.

Which of those two fragments of code is correct? Well, the first one tests the capacity of the disk before adding a file, but the second one, using **do-until**, doesn't check for free disk space until the file is added, which isn't so cautious.

while and **do-while** are the complementary opposites of **until** and **do-until**, repeating a block while a test expression remains true.

```
(while (disk-has-space)
 (println "Adding another file")
 (add-file)
 (inc 'count))

(do-while (disk-has-space)
 (println "Adding another file")
 (add-file)
 (inc 'count))
```

Choose the `do-` variants of each to do the action block before evaluating the test.

Blocks: groups of expressions

A lot of newLISP control functions let you construct a block of actions: a group of expressions that are evaluated one by one, one after the other. Construction is implicit: you don't have to do anything except write them in the right order and in the right place. Look at the **while** and **until** examples above: each has three expressions that will be evaluated one after the other.

However, you can also create blocks of expressions explicitly using the **begin**, **or**, and **and** functions.

begin is useful when you want to explicitly group expressions together in a single list. Each expression is evaluated in turn:

```
(begin
 (switch-on)
 (engage-thrusters)
 (look-in-mirror)
 (press-accelerator-pedal)
 (release-brake)
 ...)
```

and so on. You normally use **begin** only when newLISP is expecting a single expression. You don't need it with **dotimes** or **dolist** constructions, because these already allow more than one expression.

It doesn't matter what the result of each expression in a block is, unless it's bad enough to stop the program altogether. Returning **nil** is OK:

```
(begin
 (println "so far, so good")
 (= 1 3)           ; returns nil but I don't care
 (println "not sure about that last result"))
```

```
so far, so good
not sure about that last result!
```

The values returned by each expression in a block are, in the case of **begin**, thrown away. But for two other 'block' functions, **and** and **or**, the return values are important and useful.

and and or

The **and** function works through a block of expressions but finishes the block immediately if one of them returns **nil** (false). To get to the end of the **and** block, every single expression has to return a true value. If one expression fails, evaluation of the block stops, and newLISP ignores the remaining expressions, returning **nil** so that you know it didn't complete normally.

Here's an example of **and** that tests whether `disk-item` contains a useful directory:

```
(and
 (directory? disk-item)
 (!= disk-item ".")
 (!= disk-item "..")
 ; I get here only if all tests succeeded
 (println "it looks like a directory")
 )
```

The `disk-item` has to 'pass' all three tests: it must be a directory, it mustn't be the '.' directory, and it mustn't be the '..' directory (Unix terminology). When it successfully gets past these three tests, evaluation continues, and the message was printed. If one of the tests failed, the block completes without printing a message.

You can use **and** for numeric expressions too:

```
(and
 (< c 256)
 (> c 32)
 (!= c 48))
```

which tests whether `c` is between 33 and 255 inclusive and not equal to 48. This will always return either true or **nil**, depending on the value of `c`.

In some circumstances **and** can produce neater code than **if**. You can rewrite this example on the previous page:

```
(if (number? x)
 (begin
   (println x " is a number ")
   (inc 'x)))
```

to use **and** instead:

```
(and
 (number? x)
 (println x " is a number ")
 (inc 'x))
```

You could also use **when** here:

```
(when (number? x)
 (println x " is a number ")
 (inc 'x))
```

The **or** function is more easily pleased than its counterpart **and**. The series of expressions are evaluated one at a time until one returns a true value. The rest are then ignored. You could use this to work through a list of important conditions, where any one failure is important enough to abandon the whole enterprise. Or, conversely, use **or** to work through a list where any one success is reason enough to continue. Whatever, remember that as soon as newLISP gets a non-nil result, the **or** function completes.

The following code sets up a series of conditions that each number must avoid fulfilling – just one true answer and it doesn't get printed:

```
(for (x -100 100)
 (or
   (< x 1)               ; x mustn't be less than 1
   (> x 50)              ; or greater than 50
   (> (mod x 3) 0)       ; or leave a remainder when divided by 3
   (> (mod x 2) 0)       ; or when divided by 2
   (> (mod x 7) 0)       ; or when divided by 7
   (println x)))

42                       ; the ultimate and only answer
```

ambiguity: the amb function

You may or may not find a good use for **amb** – the ambiguous function. Given a series of expressions in a list, **amb** will choose and evaluate just one of them, but you don't know in advance which one. If you type **amb** 20 times, you'll get a different result each time.

```
> (amb 1 2 3 4 5 6)
3
> (amb 1 2 3 4 5 6)
2
> (amb 1 2 3 4 5 6)
6
> (amb 1 2 3 4 5 6)
3
> (amb 1 2 3 4 5 6)
5
> (amb 1 2 3 4 5 6)
3
>...
```

Use it to choose alternative actions at random:

```
(dotimes (x 20)
 (amb
   (println "Will it be me?")
   (println "It could be me!")
   (println "Or it might be me...")))
```

```
Will it be me?
It could be me!
It could be me!
Will it be me?
It could be me!
Will it be me?
Or it might be me...
It could be me!
Will it be me?
Will it be me?
It could be me!
It could be me!
Will it be me?
Or it might be me...
It could be me!
...
```

For more about newLISP's random functions, see "**Random numbers**" on page **111**.

Selection: if, cond, and case

To test for a series of alternative values, you can use **if**, **cond**, or **case**. The **case** function lets you execute expressions based on the value of a switching expression. It consists of a series of value/expression pairs:

```
(case n
  (1       (println "un"))
  (2       (println "deux"))
  (3       (println "trois"))
  (4       (println "quatre"))
  (true    (println "je ne sais quoi")))
```

newLISP works through the pairs in turn, seeing if n matches any of the values 1, 2, 3, or 4. As soon as a value matches, the expression is evaluated and the **case** function finishes, returning the value of the expression. It's a good idea to put a final pair with **true** and a catch-all expression to cope with no matches at all. n will always be true, so put this at the end.

The potential match values are not evaluated. This means that you can't write this:

```
(case n
  ((- 2 1)      (println "un"))
  ((+ 2 0)      (println "deux"))
  ((- 6 3)      (println "trois"))
  ((/ 16 4)     (println "quatre"))
  (true         (println "je ne sais quoi")))
```

even though this ought logically to work: if n is 1, you would expect the first expression (- 2 1) to match. But that expression hasn't been evaluated – none of the sums have. In this example, the **true** action (println "je ne sais quoi") is evaluated.

> If you prefer to have a version of case that evaluates its arguments, that's easily done in newLISP. See "**Macros**" on page **99**.

Earlier I mentioned that **cond** is a more traditional version of **if**. A **cond** statement in newLISP has the following structure:

```
(cond
   (test action1 action2 ...)
   (test action1 action2 ...)
   (test action1 action2 ...)
   ...
   )
```

where each list consists of a test followed by one or more expressions or actions that are evaluated if the test returns true. newLISP does the first test, does the actions if the test is true, then ignores the remaining test/action loops. Often the test is a list or list expression, but it can also be a symbol or a value.

A typical example looks like this:

```
(cond
  ((< x 0)       (define a "impossible") )
  ((< x 10)      (define a "small")      )
  ((< x 20)      (define a "medium")     )
  ((>= x 20)     (define a "large")      )
 )
```

which is essentially the same as the **if** version, except that each pair of test-actions is enclosed in parentheses. Here's the **if** version for comparison:

```
(if
  (< x 0)        (define a "impossible")
  (< x 10)       (define a "small")
  (< x 20)       (define a "medium")
  (>= x 20)      (define a "large")
 )
```

For simpler functions, it might be easier to work with **if**. But **cond** can be more readable when writing longer programs. If you want the action for a particular test to evaluate more than one expression, **cond**'s extra set of parentheses can give you shorter code:

```
(cond
 ((< x 0)     (define a -1)      ; if would need a begin before the set
              (println a)      ; but cond doesn't
              (define b -1))
 ((< x 10)    (define a 5))
 ...
```

Variables local to a control structure

The control functions **dolist**, **dotimes**, and **for** involve the definition of temporary local symbols, which survive for the duration of the expression, then disappear.

Similarly, with the **let** and **letn** functions, you can define variables that exist only inside a list. They aren't valid outside the list, and they lose their value once the list has finished being evaluated.

The first item in a **let** list is a sublist containing variables (which don't have to be quoted) and expressions to initialize each variable. The remaining items in the list are expressions that can access those variables. It's a good idea to line up the variable/starting value pairs:

```
(let
   (x (* 2 2)
    y (* 3 3)
    z (* 4 4))
   ; end of initialization
 (println x)
 (println y)
 (println z))
```

```
4
9
16
```

This example creates three local variables, x, y, and z, and assigns values to each. The body contains three **println** expressions. After these finish, the values of x, y, and z are no longer accessible – although the entire expression returns the value 16, the value returned by the final println statement.

The structure of a let list is easily remembered if you imagine it on a single line:

```
(let ((x 1) (y 2))  (+ x y))
```

If you want to refer to a local variable elsewhere in the first, initialization, section, use **letn** rather than **let**:

```
(letn
   (x 2
    y (pow x 3)
    z (pow x 4))
 (println x)
 (println y)
 (println z))
```

In the definition of y, you can refer to the value of x, which we've only just defined to be 2. **letn**, a nested version of **let**, allows you to do this.

Our discussion of local variables leads to functions.

Make your own functions

The **define** function provides a way to store a list of expressions under a name, suitable for running later. The functions you define can be used in the same way as newLISP's built-in functions. The basic structure of a function definition is like this:

```
(define (func1)
 (expression-1)
 (expression-2)
 ...
 (expression-n)
)
```

when you don't want to supply any information to the function, or like this, when you do:

```
(define (func2 v1 v2 v3)
 (expression-1)
 (expression-2)
 ...
 (expression-n)
)
```

You call your newly defined function like any other function, passing values to it inside the list if your definition requires them:

```
(func1)                                ; no values expected
(func2 a b c)                          ; 3 values expected
```

I say expected, but newLISP is flexible. You can supply any number of arguments to func1, and newLISP won't complain. You can also call func2 with any number of arguments – in which case a, b, and c are set to **nil** at the start if there aren't enough arguments to define them.

When the function runs, each expression in the body is evaluated in sequence. The value of the last expression to be evaluated is returned as the function's value. For example, this function returns either **true** or **nil**, depending on the value of `n`:

```
(define (is-3? n)
 (= n 3))
```

```
(println (is-3? 2))
```

```
nil
```

```
(println (is-3? 3))
```

```
true
```

Sometimes you'll want to explicitly specify the value to return, by adding an expression at the end that evaluates to the right value:

```
(define (answerphone)
 (pick-up-phone)
 (say-message)
 (set ’message (record-message))
 (put-down-phone)
message)
```

The `message` at the end evaluates to the message received and returned by `(record-message)` and the `(answerphone)` function returns this value. Without this, the function would return the value returned by `(put-down-phone)`, which might be just a true or false value.

To make a function return more than one value, you can return a list.

Symbols that are defined in the function's argument list are local to the function, and can't be accessed from outside:

```
(define (test v1 v2)
  (println "v1 is " v1)
  (println "v2 is " v2)
  (println "end of function"))
(test 1 2)
```

```
v1 is 1
v2 is 2
end of function
```

```
v1
```

```
nil
```

`v1` has no value outside the function. But the same isn't true of symbols that are defined inside a function body:

```
(define (test v1)
  (set ’x v1)
  (println x))
(test 1)
```

```
1
```

```
x
```

```
1
```

And that's why you'll want to define local variables! See "**Local variables**" on page **29**.

newLISP is smart enough not to worry if you supply more than the required information:

```
(define (test)
  (println "hi there"))
(test 1 2 3 4)                        ; 1 2 3 4 are ignored
```

```
hi there
```

but it won’t fill in gaps for you:

```
(define (test n)
  (println n))
(test)                                ; no n supplied, so print nil
```

```
nil
```

```
(test 1)
```

```
1
```

```
(test 1 2 3)                          ; 2 and 3 ignored
```

```
1
```

Local variables

Sometimes you want functions that change the values of symbols elsewhere in your code, and sometimes you want functions that don’t - or can’t. The following function, when run, changes the value of the x symbol, which may or may not be defined elsewhere in your code:

```
(define (changes-symbol)
 (set ’x 15)
 (println "x is " x))

(set ’x 10)
;-> x is 10

(changes-symbol)
```

```
x is 15
```

If you don’t want this to happen, use **let** or **letn** to define a local x, which doesn’t affect the x symbol outside the function:

```
(define (does-not-change-x)
 (let (x 15)
    (println "my x is " x)))

(set ’x 10)

(does-not-change-x)
```

```
my x is 15
```

```
x
```

```
10
```

x is still 10 outside the function. The x inside the function is not the same as the x outside. When you use **set** to change the value of the local x inside the function, it doesn’t change any x outside:

```
(define (does-not-change-x)
 (let (x 15)            ; this x is inside the 'let' form
  (set 'x 20)))

(set 'x 10)             ; this x is outside the function
;-> 10

x
;-> 10
(does-not-change-x)
x
;-> 10
```

instead of **let** and **letn** you can use the **local** function. This is like **let** and **letn**, but you don't have to supply any values for the local variables when you first mention them. They're just `nil` until you set them:

```
(define (test)
 (local (a b c)
   (println a " " b " " c)
   (set 'a 1 'b 2 'c 3)
   (println a " " b " " c)))
(test)
```

```
nil nil nil
1 2 3
```

There are other ways of declaring local variables. You might find the following technique easier to write when you're defining your own functions. Watch the comma:

```
(define (my-function x y , a b c)
 ; a b and c are local variables inside this function
 ...)
```

In fact, that's just a clever trick: the comma is an ordinary symbol name like `c` or `x`:

```
(set ', "this is a string")
(println ,)
```

```
"this is a string"
```

– it's just less likely that you'd use it as a symbol name, and so it's useful as a visual separator in argument lists.

Default values

In a function definition, the local variables that you define in the function's argument list can have default values assigned to them, which will be used if you don't specify values when you call the function. For example, this is a function with three named arguments, `a`, `b`, and `c`:

```
(define (foo (a 1) b (c 2))
 (println a " " b " " c))
```

The symbols `a` and `c` will take the values 1 and 2 if you don't supply values in the function call, but `b` will be **nil** unless you supply a value for it.

```
(foo)          ; there are defaults for a and c but not b
```

```
1 nil 2
```

```
(foo 2)        ; no values for b or c; c has default
```

```
2 nil 2

(foo 2 3)     ; b has a value, c uses default

2 3 2

(foo 3 2 1)   ; default values not needed

3 2 1
```

Arguments: args

You can see that newLISP is very flexible with its approach to arguments to functions. You can write definitions that accept any number of arguments, giving you (or the caller of your functions) maximum flexibility.

The **args** function returns any unused arguments that were passed to a function:

```
(define (test v1)
 (println "the arguments were " v1 " and " (args)))

(test)

the arguments were nil and ()

(test 1)

the arguments were 1 and ()

(test 1 2 3)

the arguments were 1 and (2 3)

(test 1 2 3 4 5)

the arguments were 1 and (2 3 4 5)
```

Notice that `v1` contains the first argument passed to the function, but that any remaining unused arguments are in the list returned by `(args)`.

With **args** you can write functions that accept different types of input. Notice how the following function can be called without arguments, with a string argument, with numbers, or with a list:

```
(define (flexible)
 (println " arguments are " (args))
 (dolist (a (args))
  (println " -> argument " $idx " is " a)))

(flexible)

arguments are ()

(flexible "OK")

 arguments are ("OK")
 -> argument 0 is OK

(flexible 1 2 3)

 arguments are (1 2 3)
 -> argument 0 is 1
 -> argument 1 is 2
 -> argument 2 is 3
```

```
(flexible '(flexible 1 2 "buckle my shoe"))

 arguments are ((flexible 1 2 "buckle my shoe"))
 -> argument 0 is (flexible 1 2 "buckle my shoe")
```

args allows you to write functions that accept any number of arguments. For example, newLISP is perfectly happy for you to pass a million arguments to a suitably defined function. I tried it:

```
(sum 0 1 2 3 4 5 6 7 8 ...
 999997 999998 999999 1000000)
; all the numbers were there but they've been omitted here
; for obvious reasons...
;-> 500000500000
```

> In practice, newLISP was happy with this but my text editor wasn't. Besides, there are obviously better ways to do the same job. Young Carl Friedrich Gauss could have done it in his head.

The **doargs** function can be used instead of **dolist** to work through the arguments returned by **args**. You could have written the `flexible` function as:

```
(define (flexible)
 (println " arguments are " (args))
 (doargs (a)                            ; instead of dolist
  (println " -> argument " $idx " is " a)))
```

newLISP has yet more ways to control the flow of code execution. As well as **catch** and **throw**, which allow you to handle and trap errors and exceptions, there's **silent**, which operates like a quiet version of **begin**.

If you want more, you can write your own language keywords, using newLISP macros, which can be used in the same way that you use the built-in functions. See "**Macros**" on page **99**.

Scope

Consider this function:

```
(define (show)
  (println "x is " x))
```

Notice that this function refers to some unspecified symbol `x`. This may or may not exist when the function is defined or called, and it may or may not have a value. When this function is evaluated, newLISP looks for the 'nearest' symbol called `x`, and finds its value:

```
(define (show)
  (println "x is " x))

(show)

x is nil

(set 'x "a string")
(show)

x is a string

(for (x 1 5)
  (dolist (x '(sin cos tan))
    (show))
  (show))
```

```
x is sin
x is cos
x is tan
x is 1
x is sin
x is cos
x is tan
x is 2
x is sin
x is cos
x is tan
x is 3
x is sin
x is cos
x is tan
x is 4
x is sin
x is cos
x is tan
x is 5
```

```
(show)
```

```
x is a string
```

```
(define (func x)
  (show))
```

```
(func 3)
```

```
x is 3
```

```
(func "hi there")
```

```
x is hi there
```

```
(show)
```

```
x is a string
```

See how newLISP always gives you the value of the 'current' x by dynamically keeping track of which x is active, even though there might be other x's lurking in the background. As soon as the **for** loop starts, the loop variable x takes over as the 'current' x, but then that x is immediately superseded by the list iteration variable x which takes the value of a few trigonometric functions. In between each set of trig functions, the loop variable version of x pops back again briefly. And after all that iteration, the string value is available again.

In the func function, there's another x which is local to the function. When show is called, it will print this local symbol. A final call to show returns the very first value that x had.

Although newLISP won't get confused with all those different x's, you might! So it's a good idea to use longer and more explanatory symbol names, and use local rather than global variables. If you do, there's less chance of you making mistakes or of misreading your code at a later date. In general it's not a good idea to refer to an undefined symbol in a function unless you know exactly where it came from and how its value is determined.

This dynamic process of keeping track of the current version of a symbol is called dynamic scoping. There'll be more about this topic when you look at contexts ("**Introducing contexts**" on page **87**). These offer an alternative way to organize similarly-named symbols – lexical scoping.

4 Lists

Building lists

Lists are often created for you by other functions, but you can also build your own, using one of these functions:

- **list** makes a new list from expressions
- **set** defines a symbol
- **append** glues lists together to form a new list
- **cons** prepends an element to a list or make a list
- **push** inserts a new member in a list
- **dup** duplicates an expression

You can build a list and assign it to a symbol directly. Quote the list to stop it being evaluated immediately:

```
(set 'vowels '("a" "e" "i" "o" "u"))
;-> ("a" "e" "i" "o" "u") ; an unevaluated list
```

list, append, and cons

Use **list** to build a list from a sequence of expressions:

```
(set 'rhyme (list 1 2 "buckle my shoe"
    '(3 4) "knock" "on" "the" "door"))
; rhyme is now a list with 8 items
;-> (1 2 "buckle my shoe" '(3 4) "knock" "on" "the" "door")
```

Notice that the `(3 4)` element is itself a list, which is nested inside the main list.

cons accepts exactly two expressions, and can do two jobs: insert the first element at the start of an existing list, or build a new two element list. In both cases it returns a new list. newLISP automatically chooses which job to do depending on whether the second element is a list or not.

```
(cons 1 2)                           ; makes a new list
;-> (1 2)

(cons 1 '(2 3))                      ; inserts an element at the start
;-> (1 2 3)
```

To glue two or more lists together, use **append**:

```
(set 'odd '(1 3 5 7) 'even '(2 4 6 8))
(append odd even)
;-> (1 3 5 7 2 4 6 8)
```

Notice the difference between **list** and **append** when you join two lists:

```
(set 'a '(a b c) 'b '(1 2 3))

(list a b)
;-> ((a b c) (1 2 3))                    ; list makes a list of lists

(append a b)
;-> (a b c 1 2 3)                        ; append makes a list
```

list preserves the source lists when making the new list, whereas **append** uses the elements of each source list to make a new list.

> To remember this: List keeps the List-ness of the source lists, but aPPend Picks the elements out and Packs them all together again.

We'll meet **append** again later – it can also turn a bunch of strings into a new string.

push: pushing items onto lists

push is a powerful command that you can use to make a new list or insert an element at any location in an existing list. Pushing an element at the front of the list moves everything one place to the right, whereas pushing an element at the end of the list just attaches it and makes a new last element. You can also insert an element anywhere in the middle of a list.

Despite its constructive nature, it's technically a destructive function, because it changes the target list permanently, so use it carefully. The function returns the value of the element that was inserted, rather than the whole list.

```
(set 'vowels '("e" "i" "o" "u"))
(push (char 97) vowels)
; returns "a"
; but vowels is now ("a" "e" "i" "o" "u")
```

When you refer to the location of an element in a list, you use zero-based numbering, which you would expect if you're an experienced programmer:

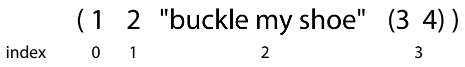


Figure 4.1 Index numbers of list elements

If you don't specify a location or index, **push** pushes the new element at the front. Use a third expression to specify the location or index for the new element. -1 means the last element of the list, 1 means the second element of the list counting from the front from 0, and so on:

```
(set 'vowels '("a" "e" "i" "o"))
(push "u" vowels -1)
;-> "u"
; vowels is now ("a" "e" "i" "o" "u")

(set 'evens '(2 6 10))
(push 8 evens -2)                       ; goes before the 10
(push 4 evens 1)                        ; goes after the 2

; evens is now (2 4 6 8 10)
```

If the symbol you supply as a list doesn't exist, **push** usefully creates it for you, so you don't have to declare it first.

```
(for (c 1 10)
 (push c number-list -1)                ; doesn't fail first time round!
 (println number-list))

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

```
(1 2 3 4 5 6)
(1 2 3 4 5 6 7)
(1 2 3 4 5 6 7 8)
(1 2 3 4 5 6 7 8 9)
(1 2 3 4 5 6 7 8 9 10)
```

By the way, there are plenty of other ways of generating a list of unsorted numbers. You could also do a number of random swaps like this:

```
(set 'l (sequence 1 99))
(dotimes (n 100)
  (swap (rand 100) (rand 100) l)))
```

although it would be even easier to use **randomize**:

```
(randomize (sequence 1 99))
;-> (54 38 91 18 76 71 19 30 ...
```

(That's one of the nice things about newLISP – a more elegant solution is just a re-write away!)

push has an opposite, **pop**, which destructively removes an element from a list, returning the removed element. We'll meet **pop** and other list surgery functions later. See "**List surgery**" on page **53**.

These two functions, like many other newLISP functions, work on strings as well as lists. See "**push and pop work on strings too**" on page **70**.

dup: building lists of duplicate elements

A useful function called **dup** lets you construct lists quickly by repeating elements a given number of times:

```
(dup 1 6)              ; duplicate 1 six times
;-> (1 1 1 1 1 1)

(dup '(1 2 3) 6)
;-> ((1 2 3) (1 2 3) (1 2 3) (1 2 3) (1 2 3) (1 2 3))

(dup x 6)
;-> (x x x x x x)
```

There's a little trick to get **dup** to return a list of strings. Because **dup** can also be used to duplicate strings into a single longer string, you should supply an extra **true** value at the end of the list, and newLISP creates a list of strings rather than a string of strings.

```
(dup "x" 6)            ; a string of strings
;-> "xxxxxx"

(dup "x" 6 true)       ; a list of strings
;-> ("x" "x" "x" "x" "x" "x")
```

Working with whole lists

Once you've got a list, you can start to work on it. First, let's look at the functions that operate on the list as a unit. After that, I'll look at the functions that let you carry out list surgery – operations on individual list elements.

Using and processing lists

dolist works through every item of a list, and **apply** applies a function to a list containing suitable arguments for that function. I examine the **dolist** and **apply** functions elsewhere (see "**Working through a list**" on page **18**, and "**Apply and map: applying functions to lists**" on page **81**), but for now here's a contrived example showing the two cooperating well:

```
(set 'vowels '("a" "e" "i" "o" "u"))
(dolist (v vowels)
  (println (apply upper-case (list v))))
```

```
A
E
I
O
U
```

Using apply and map

In the previous example, **apply** expects a function and a list, and uses the elements of the list as arguments to the function. So it repeatedly applies the **upper-case** function to the loop variable's value in `v`. Since **upper-case** works on strings but **apply** expects a list, I've had to use the **list** function to convert the current value of `v` (a string) in each iteration to a list.

A better way to do this is to use **map**:

```
(map upper-case '("a" "e" "i" "o" "u"))
;-> ("A" "E" "I" "O" "U")
```

map applies the named function, **upper-case** in this example, to each item of the list in turn. The advantage of **map** is that it both traverses the list and applies the function to each item of the list in one pass. The result is a list, too, which might be more useful for subsequent processing.

reverse

reverse does what you would expect, and reverses the list. It's a destructive function, changing the list permanently.

```
(reverse '("A" "E" "I" "O" "U"))
;-> ("U" "O" "I" "E" "A")
```

sort and randomize

In a way, **randomize** and **sort** are complementary, although **sort** changes the original list, whereas **randomize** returns a disordered copy of the original list. **sort** arranges the elements in the list into ascending order, organizing them by type as well as by value.

Here's an example where I create a list of letters and a list of numbers, stick them together, shuffle the result and then sort it again:

```
(for (c (char "a") (char "z"))
 (push (char c) alphabet -1))

(for (i 1 26)
 (push i numbers -1))

(set 'data (append alphabet numbers))

;-> ("a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
"q" "r" "s" "t" "u" "v" "w" "x" "y" "z" 1 2 3 4 5 6 7 8 9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24 25 26)
```

```
(randomize data))

;-> ("l" "r" "f" "k" 17 10 "u" "e" 6 "j" 11 15 "s" 2 22 "d" "q"
"b" "m" 19 3 5 23 "v" "c" "w" 24 13 21 "a" 4 20 "i" "p" "n" "y" 14
"g" 25 1 8 18 12 "o" "x" "t" 7 16 "z" 9 "h" 26)

(sort data)

;-> (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
"p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z")
```

Compare `data` before it was randomized and after it was sorted. The **sort** command sorts the list by data type as well as by value: integers before strings, strings before lists, and so on.

To change the sort method, you can supply one of newLISP's built-in comparison functions (**>** is the obvious choice, since **<** is the default). Adjacent objects are considered to be correctly sorted when the comparison function is true.

```
(for (c (char "a") (char "z"))
  (push (char c) alphabet -1))

alphabet
;-> ("a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
"o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z")

(sort alphabet >)
;-> ("z" "y" "x" "w" "v" "u" "t" "s" "r" "q" "p" "o" "n"
"m" "l" "k" "j" "i" "h" "g" "f" "e" "d" "c" "b" "a")
```

You can also supply a custom sort function. This is a function that takes two arguments and returns true if they're in the right order (first before second), and false if they aren't. For example, suppose you want to sort a list of file names so that the shortest name appears first. First, define a function that returns true if the first argument is shorter than the second, and then use **sort** with this custom sorting function:

```
(define (shorter? a b)        ; two arguments, a and b
 (< (length a) (length b)))

(sort (directory) shorter?)
;->

("." ".." "var" "usr" "tmp" "etc" "dev" "bin" "sbin" "mach" ".vol"
"Users" "cores" "System" "Volumes" "private" "Network" "Library"
"mach.sym" ".Trashes" "Developer" "automount" ".DS_Store"
"Desktop DF" "Desktop DB" "mach_kernel" "Applications" "System Folder"
...)
```

Advanced newLISPers often write a nameless function and supply it directly to the **sort** command:

```
(sort (directory) (fn (a b) (< (length a) (length b))))
```

This does the same job, but saves 25 characters or so.

You can use either **fn** or **lambda** to define an inline or anonymous function.

unique

unique returns a copy of a list with all duplicates removed:

```
(set 'data '( 1 1 2 2 2 2 2 2 2 3 2 4 4 4 4))
(unique d)
;-> (1 2 3 4)
```

There are also some useful functions for comparing lists. See “**Working with two or more lists**” on page **57**.

flat

flat is useful for dealing with nested lists, because it shows what a list looks like without its complicated hierarchical structure:

```
(set 'data '(0 (0 1 2) 1 (0 1) 0 1 (0 1 2) ((0 1) 0)))
(length data)
;-> 8
(length (flat data))
;-> 15
(flat data)
;-> (0 0 1 2 1 0 1 0 1 0 1 2 0 1 0)
```

Fortunately, **flat** is non-destructive, so you can use it without worrying about losing the structure of nested lists:

```
data
;-> (0 (0 1 2) 1 (0 1) 0 1 (0 1 2) ((0 1) 0)) ; still nested
```

transpose

transpose is designed to work with matrices (a special type of list: see “**Matrices**” on page **117**). It also does something useful with ordinary nested lists. If you imagine a list of lists as a table, it flips rows and columns for you:

```
(set 'a-list
 '(("a" 1)
   ("b" 2)
   ("c" 3)))

(transpose a-list)

;->
(("a" "b" "c")
 ( 1 2 3))

(set 'table
'((A1 B1 C1 D1 E1 F1 G1 H1)
  (A2 B2 C2 D2 E2 F2 G2 H2)
  (A3 B3 C3 D3 E3 F3 G3 H3)))

(transpose table)

;->
((A1 A2 A3)
 (B1 B2 B3)
 (C1 C2 C3)
 (D1 D2 D3)
 (E1 E2 E3)
 (F1 F2 F3)
 (G1 G2 G3)
 (H1 H2 H3))
```

And here's a nice bit of newLISP sleight of hand:

```
(set 'table '((A 1) (B 2) (C 3) (D 4) (E 5)))
;-> ((A 1) (B 2) (C 3) (D 4) (E 5))

(set 'table (transpose (swap 1 0 (transpose table))))
;-> ((1 A) (2 B) (3 C) (4 D) (5 E))
```

and each sublist has been reversed. You could, of course, do this:

```
(set 'table (map (fn (i) (rotate i)) table))
```

which is shorter, but a bit slower.

explode

The **explode** function lets you blow up a list:

```
(explode (sequence 1 10))
;-> ((1) (2) (3) (4) (5) (6) (7) (8) (9) (10))
```

You can specify the size of the pieces, too:

```
(explode (sequence 1 10) 2)
;-> ((1 2) (3 4) (5 6) (7 8) (9 10))

(explode (sequence 1 10) 3)
;-> ((1 2 3) (4 5 6) (7 8 9) (10))

(explode (sequence 1 10) 4)
;-> ((1 2 3 4) (5 6 7 8) (9 10))
```

List analysis: testing and searching

Often you don't know what's in a list, and you want some forensic tools to find out more about what's inside it. newLISP has a good selection.

We've already met **length**, which finds the number of elements in a list.

The **starts-with** and **ends-with** functions test the start and ends of lists:

```
(for (c (char "a") (char "z"))
 (push (char c) alphabet -1))

;-> alphabet is ("a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
"m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z")

(starts-with alphabet "a")              ; list starts with item "a"?
;-> true

(starts-with (join alphabet) "abc")     ; convert to string and test
;-> true

(ends-with alphabet "z")                ; testing the list version
;-> true
```

These functions work equally well with strings, too (and they take regular expressions). See "**Testing and comparing strings**" on page **74**.

How about 'contains'? In fact there isn't one single newLISP function that does this job. Instead you have **find**, **match**, **member**, **ref**, **filter**, **index**, and **count**, among others. Which one you use depends on what sort of answer you want to the question ‘Does this list contain this item?’, and whether the list is a nested list or a flat one.

If you want a simple answer with a quick top-level-only search, use **find**. See “**find**” on page **42**.

If you want the item and the rest of the list as well, use **member**. See “**member**” on page **44**.

If you want the index number of the first occurrence, even if the list contains nested lists, you can use **ref**. See “**ref and ref-all**” on page **45**.

If you want a new list containing all the elements that match your search element, use **find-all**. See “**find-all**” on page **45**.

If you want to know whether or not the list contains a certain pattern of elements, use **match**. See “**matching patterns in lists**” on page **44**.

You can find all list items that satisfy a function, either built-in or one of your own, with the **filter**, **clean**, and **index** functions. See “**Filtering lists: filter, clean, and index**” on page **46**.

The **exists** and **for-all** functions check elements in a list to see if they pass a test.

If you want to find elements in a list purely to change them to something else, then don’t bother to look for them first, just use **replace**. See “**Replacing information: replace**” on page **54**. You can also find and replace list elements using the **ref-set** and **set-ref** functions. See “**Find and replace matching elements**” on page **55**.

If you just want to know how many occurrences there are of items in the list, use **count**. See “**Working with two or more lists**” on page **57**.

Let’s look at some examples of each of these.

find

find looks through a list for an expression and returns an integer or **nil**. The integer is the index of the first occurrence of the search term in the list. It’s possible for **find** to return 0 – if the list starts with the item, its index number is 0, but this isn’t a problem – you can use this function in an **if** test, because 0 evaluates to true.

```
(set 'sign-of-four
 (parse (read-file "/Users/me/Sherlock-Holmes/sign-of-four.txt")
 {\W} 0))

(if  (find "Moriarty" sign-of-four)        ; Moriarty anywhere?
  (println "Moriarty is mentioned")
  (println "No mention of Moriarty"))

No mention of Moriarty

(if (find "Lestrade" sign-of-four)
  (println "Lestrade is mentioned")
  (println "No mention of Lestrade"))

Lestrade is mentioned.

(find "Watson" sign-of-four)
;-> 477
```

Here we’ve parsed the text of Sir Arthur Conan Doyle’s ‘The Sign of Four’ (which you can download from Project Gutenberg), and tested whether the resulting list of strings contains various names. The integer returned is the first occurrence of the string element in the list.

find lets you use regular expressions, too, so you can find any string elements in a list that match a string pattern:

```
(set 'loc (find "(tea|cocaine|morphine|tobacco)" sign-of-four 0))
(if loc
  (println "The drug " (sign-of-four loc) " is mentioned.")
  (println "No trace of drugs"))
```

```
The drug cocaine is mentioned.
```

Here we're looking for any traces of chemical indulgence in Sherlock Holmes' bohemian way of life: `"(tea|cocaine|morphine|tobacco)"` means any one of tea, cocaine, morphine, or tobacco.

This form of **find** lets you look for regular expression patterns in the string elements of a list. We'll meet regular expressions again when we explore strings. See "**Regular expressions**" on page **71**.

```
(set 'word-list '("being" "believe" "ceiling" "conceit" "conceive"
"deceive"
"financier" "foreign" "neither" "receive" "science" "sufficient" "their"
"vein" "weird"))

(find {(c)(ie)(?# i before e except after c...)} word-list 0)
;-> 6                                  ; the first one is "financier"
```

Here we're looking for any string elements in the word-list that match our pattern (a `c` followed by `ie`, the old and inaccurate spelling rule 'i before e except after c').

The regular expression pattern here (enclosed in braces, which are string delimiters, doing more or less the same job as quotation marks) is a (c) followed by an (ie). Then there's a comment, starting with `(?#`. Comments in regular expressions are useful when things get cryptic, as they often do.

find can also accept a comparison function. See "**Searching lists**" on page **48**.

find finds only the first occurrence in a list. To find all occurrences, you could keep repeating **find** until it returns **nil**. The word list gets shorter each time through, and the found elements are added at the end of another list:

```
(set 'word-list '("scientist" "being" "believe" "ceiling" "conceit"
"conceive"
"deceive" "financier" "foreign" "neither" "receive" "science"
"sufficient" "their" "vein" "weird"))

(while (set 'temp
 (find {(c)(ie)(?# i before e except after c...)} word-list 0))
   (push (word-list temp) results -1)
   (set 'word-list ((+ temp 1) word-list)))

results
;-> ("scientist" "financier" "science" "sufficient")
```

But in this case it's much easier to use **filter**:

```
(filter (fn (w) (find {(c)(ie)} w 0)) word-list)
;-> ("scientist" "financier" "science" "sufficient")
```

– see "**Filtering lists: filter, clean, and index**" on page **46**.

Alternatively, you can use **ref-all** (see "**ref and ref-all**" on page **45**) to get a list of indices.

If you're not using regular expressions, you can use **count**, which, given two lists, looks through the second list and counts how many times each item in the first list occurs. Let's see how many times the main characters' names are mentioned:

```
(count '("Sherlock" "Holmes" "Watson" "Lestrade" "Moriarty" "Moran")
 sign-of-four)
;-> (34 135 24 1 0 0)
```

The list of results produced by **count** shows how many times each element in the first list occurs in the second list, so there are 34 mentions of Sherlock, 135 mentions of Holmes, 24 of Watson, and only one mention for poor old Inspector Lestrade in this story.

It's worth knowing that **find** examines the list only superficially. For example, if the list contains nested lists, you should use **ref** rather than **find**, because **ref** looks inside sublists:

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

(find 4 maze)
;-> nil                                 ; didn't look inside the lists

(ref 4 maze)
;-> (2 3)                               ; element 3 of element 2
```

member

The **member** function returns the rest of the source list rather than index numbers or counts:

```
(set 's (sequence 1 100 7))             ; 7 times table?
;-> (1 8 15 22 29 36 43 50 57 64 71 78 85 92 99)

(member 78 s))
;-> (78 85 92 99)
```

matching patterns in lists

There's a powerful and complicated function called **match**, which looks for patterns in lists. It accepts the wild card characters *, ?, and +, which you use to define a pattern of elements. + means one or more elements, * means zero or more elements, and ? means one element. For example, suppose you want to examine a list of random digits between 0 and 9 for patterns. First, generate a list of 10000 random numbers as source data:

```
(dotimes (c 10000) (push (rand 10) data))
;-> (7 9 3 8 0 2 4 8 3 ...)
```

Next, you decide that you want to find the following pattern:

```
... 1 2 3 ...
```

somewhere in the list, ie anything followed by 1 followed by a 2 followed by a 3 followed by anything. Call **match** like this:

```
(match '(* 1 2 3 *) data)
```

which looks odd, but it's just a pattern specification in a list, followed by the source data. The list pattern:

```
(* 1 2 3 *)
```

means any sequence of atoms or expressions (or nothing), followed by a 1, then a 2, then a 3, followed by any number of atoms or expressions or nothing. The answer returned by this **match** function is another list, consisting of two sublists, one corresponding to the first *, the other corresponding to the second:

```
((7 9 3 8 ... 0 4 5) (7 2 4 1 ... 3 5 5 5))
```

and the pattern you were looking for first occurred in the gap between these lists (in fact it occurs half a dozen times later on in the list as well). **match** can also handle nested lists.

To find all occurrences of a pattern, not just the first, you can use **match** in a **while** loop. For example, to find and remove every 3 when it's followed by a 4, repeat the **match** on a new version of the list until it stops returning a non-nil value:

```
(set 'number-list '(2 4 3 4 5 4 3 6 2 3 5 2 2 3 5 3 3 4 2 3 4 2))
(while (set 'temp-list (match '(* 3 4 *) number-list))
 (set 'number-list (apply append temp-list)))

;-> (2 4 5 4 3 6 2 3 5 2 2 3 5 3 2 2)
```

You don't have to find elements first before replacing them: just use **replace**, which does the finding and replacing in one operation. And you can use **match** as a comparison function for searching lists. See "**Replacing information: replace**" on page **54**, and "**Searching lists**" on page **48**.

find-all

find-all is a powerful function with a number of different forms, suitable for searching lists, association lists, and strings. For list searches, you supply four arguments: the search key, the list, an action expression, and the functor, which is the comparison function you want to use for matching the search key:

```
(set 'food '("bread" "cheese" "onion" "pickle" "lettuce"))
(find-all "onion" food (print $0 { }) >)
bread cheese lettuce
```

Here, **find-all** is looking for the string "onion" in the list `food`. It's using the **>** function as a comparison function, so it will find anything that "onion" is greater than. For strings, 'greater than' means appearing later in the default ASCII sorting order, so that "cheese" is greater than "bread" but less than "onion". Notice that, unlike other functions that let you provide comparison functions (namely **find**, **ref**, **ref-all**, **ref-set**, **replace** when used with lists, **set-ref**, **set-ref-all**, and **sort**), the comparison function *must* be supplied. With the **<** function, the result is a list of things that "onion" is less than:

```
(find-all "onion" food (print $0 { }) <)
pickle
```

ref and ref-all

The **ref** function returns the index of the first occurrence of an element in a list. It's particularly suited for use with nested lists, because, unlike **find**, it looks inside all the sublists, and returns the 'address' of the first appearance of an element. As an example, suppose you've converted an XML file, such as your iTunes library, into a large nested list, using newLISP's built-in XML parser:

```
(xml-type-tags nil nil nil nil)         ; controls XML parsing
(set 'itunes-data
 (xml-parse
   (read-file "/Users/me/Music/iTunes/iTunes Music Library.xml")
   (+ 1 2 4 8 16)))
```

Now you can look for any expression in the data, which is now in the form of an ordinary newLISP list:

```
(ref "Brian Eno" itunes-data)
```

and the returned list will be the location of the first occurrence of that string in the list:

```
(0 2 14 528 6 1)
```

– this is a set of list index numbers which together define an ‘address’. This example means: in list element 0, look for sublist element 2, then find sublist element 14 of that sublist, and so on, drilling down into the highly-nested XML-based data structure. See “**Working with XML**” on page **141**.

There’s an alternative syntax for **ref**:

```
(ref (itunes-data "Brian Eno"))
```

ref-all does a similar job, but returns a list of every occurrence of the expression:

```
(ref-all "Brian Eno" itunes-data)
;-> ((0 2 14 528 6 1) (0 2 16 3186 6 1) (0 2 16 3226 6 1))
```

These functions can also accept a comparison function. See “**Searching lists**” on page **48**.

Use these functions when you’re searching for something in a nested list. If you want to replace it when you’ve found it, use the **set-ref**, **ref-set**, and **set-ref-all** functions. See “**Find and replace matching elements**” on page **55**.

Filtering lists: filter, clean, and index

Another way of finding things in lists is to filter the list. Like panning for gold, you can create a filter that keeps only the stuff you want, flushing the unwanted stuff away.

The functions **filter** and **index** have the same syntax, but **filter** returns the list elements, whereas **index** returns the index numbers (indices) of the wanted elements rather than the list elements themselves.

The filtering functions **filter**, **clean**, and **index** use another function for testing elements: the element appears in the results list according to whether it passes the test or not. You can either use a built-in function or define your own. Typically, newLISP functions that tests and return true or false (sometimes called predicate functions) have names ending with question marks:

NaN? array? atom? context? directory? empty? file? float? global? integer? lambda? legal? list? macro? nil? null? number? primitive? protected? quote? string? symbol? true? zero?

So, for example, an easy way to find integers in (and remove floating-point numbers from) a list is to use the **integer?** function with **filter**. Only integers pass through this filter:

```
(set 'data '(0 1 2 3 4.01 5 6 7 8 9.1 10))
(filter integer? data)
;-> (0 1 2 3 5 6 7 8 10)
```

filter has a complementary function called **clean** which removes elements that satisfy the test:

```
(set 'data '(0 1 2 3 4.01 5 6 7 8 9.1 10))
(clean integer? data)
;-> (4.01 9.1)
```

Think of **clean** as getting rid of dirt – it gets rid of anything that passes the test. Think of **filter** as panning for gold, keeping what passes the test.

This next filter finds all words in Conan Doyle’s story ‘The Empty House’ that contain the letters ‘pp’. The filter is a lambda expression (a temporary function without a name) that returns **nil** if the element doesn’t contain ‘pp’. The list is a list of string elements generated by **parse**, which breaks up a string into a list of smaller strings according to a pattern.

```
(set ’empty-house-text
 (parse
   (read-file "/Users/me/Sherlock-Holmes/the-empty-house.txt")
   {,\s*|\s+} 0))

(filter (fn (s) (find "pp" s)) empty-house-text)

;->
("suppressed" "supply" "disappearance" "appealed" "appealed"
"supplemented" "appeared" "opposite" "Apparently" "Suppose"
"disappear" "happy" "appears" "gripped" "reappearance."
"gripped" "opposite" "slipped" "disappeared" "slipped"
"slipped" "unhappy" "appealed" "opportunities." "stopped"
"stepped" "opposite" "dropped" "appeared" "tapped"
"approached" "suppressed" "appeared" "snapped" "dropped"
"stepped" "dropped" "supposition" "opportunity" "appear"
"happy" "deal-topped" "slipper" "supplied" "appealing"
"appear")
```

You can also use **filter** or **clean** for tidying up lists before using them – removing empty strings that resulted from a **parse** operation, for example.

When would you use **index** rather than **filter** or **clean**? Well, use **index** when you later want to access the list elements by index number rather than their values: we'll meet functions for selecting list items by index in the next section. For example, whereas **ref** found the index of only the first occurrence, you could use **index** to return the index numbers of every occurrence of an element.

For example, if you have a predicate function that looks for a string in which the letter 'c' is followed by 'ie', you can use that function to search a list of matching strings:

```
(set ’word-list ’("agencies" "being" "believe" "ceiling"
"conceit" "conceive" "deceive" "financier" "foreign"
"neither" "receive" "science" "sufficient" "their" "vein"
"weird"))

(define (i-before-e-after-c? wd)        ; a predicate function
 (find {(c)(ie)(?# i before e after c...)} wd 0))

(index i-before-e-after-c? word-list)
;-> (0 7 11 12)
; agencies, financier, science, sufficient
```

Remember that lists can contain nested lists, and that some functions won't look inside the sublists:

```
(set ’maze
 ’((1 2.1)
   (1 2 3)
   (1 2 3 4)))

(filter integer? maze)
;-> ()                                  ; I was sure it had integers...

(filter list? maze)
;-> ((1 2.1) (1 2 3) (1 2 3 4))         ; ah yes, they’re sublists!

(filter integer? (flat maze))
;-> (1 1 2 3 1 2 3 4)                   ; one way to do it...
```

Testing lists

The **exists** and **for-all** functions check elements in a list to see if they pass a test.

exists returns either the first element in the list that passes the test, or **nil** if none of them do.

```
(exists string? ’(1 2 3 4 5 6 "hello" 7))
;-> "hello"

(exists string? ’(1 2 3 4 5 6 7))
;-> nil
```

for-all returns either true or **nil**. If every list element passes the test, it returns true.

```
(for-all number? ’(1 2 3 4 5 6 7))
;-> true

(for-all number? ’("zero" 2 3 4 5 6 7))
;-> nil
```

Searching lists

As we’ve seen, **find**, **ref**, **ref-all** and **replace** can look for items in lists. Usually, you use these functions to find items that equal what you’re looking for. However, equality is just the default test: all these functions can accept an optional comparison function that’s used instead of a test for equality. This means that you can look for list elements that satisfy any test.

The following example uses the **<** comparison function. **find** looks for the first element that compares favourably with `n`, ie the first element that `n` is less than. With a value of 23, the first element that satisfies the test is 24, the 14th element of the list, and so the returned value is 14.

```
(set ’s (sequence 1000 1020))
;-> (1000 1001 1002 1003 1004 1005 1006 1007 1008 1009
 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020)

(set ’n 1002)
; find the first something that n is less than:
(find n s <)

;-> 3, the index of 1003 in s, the first number that  n is less than
```

You can write your own comparison function:

```
(set ’a-list
   ’("elephant" "antelope" "giraffe" "dog" "cat" "lion" "shark" ))

(define (longer? x y)
  (> (length x) (length y)))

(find "tiger" a-list longer?)
;-> 3 ; "tiger" is longer than "dog"
```

The `longer?` function returns true if the first argument is longer than the second. So **find**, with this function as the comparison, finds the first element in the list that makes the comparison true. Because `tiger` is longer than `dog`, the function returns 3, the index of `dog` in the list.

You could supply an anonymous (or lambda) function as part of the **find** function, rather than write a separate function:

```
(find "tiger" alist (fn (x y) (> (length x) (length y))))
```

If you want your code to be readable, you'll probably move longer or more complex comparators out into their own separate – and documented – functions.

You can also use comparison functions with **ref**, **ref-all**, and **replace**.

A comparison function can be any function that takes two values and returns true or false. For example, here's a function that returns true if y is greater than 6 and less than x. The search is therefore for an element of the `data` list that is both smaller than the searched-for number, which in this case is 15, and yet bigger than 6.

```
(set 'data '(31 23 -63 53 8 -6 -16 71 -124 29))

(define (my-func x y)
  (and (> x y) (> y 6)))

(find 15 data my-func)
;-> 4 ; there's an 8 at index location 4
```

Summary: compare and contrast

To summarize these 'contains' functions, here they are in action:

```
(set 'data
   '("this" "is" "a" "list" "of" "strings" "not" "of" "integers"))

(find "of" data)                       ; equality is default test
;-> 4                                  ; index of first occurrence

(ref "of" data)                        ; where is "of"?
;-> (4)                                ; returns a list of indexes

(ref-all "of" data)
;-> ((4) (7))                          ; list of address lists

(filter (fn (x) (= "of" x)) data)      ; keep every of
;-> ("of" "of")

(index (fn (x) (= "of" x)) data)       ; indexes of the of's
;-> (4 7)

(match (* "of" * "of" *) data)
;-> (("this" "is" "a" "list") ("strings" "not") ("integers"))

(member "of" data)                     ; and the rest
;-> ("of" "strings" "not" "of" "integers")

(count (list "of") data)               ; remember to use a list
;-> (2)                                ; returns list of counts
```

Selecting items from lists

There are various functions for getting at the information stored in a list:

- **first** gets the first element
- **rest** gets all but the first element
- **last** returns the last element

- **nth** gets the nth element
- **select** selects certain elements by index
- **slice** extracts a sublist

The **first** and **rest** functions are more sensible names for the traditional `car` and `cdr` LISP functions, which were based on the names of old computer hardware registers.

Picking elements: nth, select, and slice

nth gets the nth element of a list:

```
(set 'phrase '("the" "quick" "brown" "fox" "jumped" "over" "the" "lazy"
"dog"))

(nth 1 phrase)
;-> "quick"
```

nth can also look inside nested lists, because it accepts more than one index number:

```
(set 'zoo
 '(("ape" 3)
   ("bat" 47)
   ("lion" 4)))

(nth 2 1 zoo)                          ; item 2, then subitem 1
;-> 4
```

If you want to pick a group of elements out of a list, you'll find **select** useful. You can use it in two different forms. The first form lets you supply a sequence of loose index numbers:

```
(set 'phrase '("the" "quick" "brown" "fox" "jumped" "over" "the" "lazy"
"dog"))

(select phrase 0 -2 3 4 -4 6 1 -1))
;-> ("the" "lazy" "fox" "jumped" "over" "the" "quick" "dog")
```

A positive number selects an element by counting forward from the beginning, and a negative number selects by counting backwards from the end:

```
   0      1       2       3       4       5      6      7      8
("the" "quick" "brown" "fox" "jumped" "over" "the" "lazy" "dog")
  -9     -8      -7      -6      -5      -4     -3     -2     -1
```

You can also supply a list of index numbers to **select**. For example, you can use the **rand** function to generate a list of 20 random numbers between 0 and 8, and then use this list to select elements from `phrase` at random:

```
(select phrase (rand 9 20)))
;-> ("jumped" "lazy" "over" "brown" "jumped" "dog" "the" "dog" "dog"
 "quick" "the" "dog" "the" "dog" "the" "brown" "lazy" "lazy" "lazy"
"quick")
```

Notice the duplications. If you had written this instead:

```
(randomize phrase)
```

there would be no duplicates: `(rand 9 20)` produces duplicates, whereas `(randomize phrase)` shuffles elements without duplicating them.

slice lets you extract sections of a list. Supply it with the list, followed by one or two numbers. The first number is the start location. If you miss out the second number, the rest of the list is returned. The second number, if positive, is the number of elements to return.

```
(slice (explode "schwarzwalderkirschtorte") 7)
;-> ("w" "a" "l" "d" "e" "r" "k" "i" "r" "s" "c" "h" "t" "o" "r" "t" "e")

(slice (explode "schwarzwalderkirschtorte") 7 6)
;-> ("w" "a" "l" "d" "e" "r")
```

If negative, the second number specifies an element at the other end of the slice counting backwards from the end of the list, -1 being the final element:

```
(slice (explode "schwarzwalderkirschtorte") 19 -1)
;-> ("t" "o" "r" "t")
```

The cake knife reaches as far as – but doesn't include – the element you specify.

Implicit addressing

newLISP provides a faster and more efficient way of selecting and slicing lists. Instead of using a function, you can use index numbers and lists together. This technique is called implicit addressing.

Select elements using implicit addressing

As an alternative to using **nth**, put the list's symbol and an index number in a list, like this:

```
(set 'r '("the" "cat" "sat" "on" "the" "mat"))
(r 1)                                   ; element index 1 of r
;-> "cat"

(nth 1 r)                               ; the equivalent using nth
;-> "cat"

(r 0)
;-> "the"

(r -1)
;-> "mat"
```

If you have a nested list, you can supply a sequence of index numbers that identify the list in the hierarchy:

```
(set 'zoo
 '(("ape" 3)
   ("bat" 47)
   ("lion" 4)))                         ; three sublists in a list

(zoo 2 1)
;-> 4

(nth 2 1 zoo)                           ; the equivalent using nth
;-> 4
```

where the `2 1` first finds element 2, `("lion" 4)`, then finds element 1 (the second one) of that sublist.

Selecting a slice using implicit addressing

You can also use implicit addressing to get a slice of a list. This time, put one or two numbers to define the slice, before the list's symbol, inside a list:

```
(set 'alphabet '("a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
"l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"))

(13 alphabet)                 ; start at 13, get the rest
;-> ("n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z")

(slice alphabet 13)           ; equivalent using slice
;-> ("n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z")

(3 7 alphabet)                ; start at 3, get 7 elements
;-> ("d" "e" "f" "g" "h" "i" "j")

(slice alphabet 3 7)          ; equivalent using slice
;-> ("d" "e" "f" "g" "h" "i" "j")
```

Earlier, we parsed the iTunes XML library:

```
(xml-type-tags nil nil nil nil)
(silent
 (set 'itunes-data
 (xml-parse
  (read-file
  "/Users/me/Music/iTunes/iTunes Music Library.xml")
  (+ 1 2 4 8 16))))
```

Let's access the inside of the XML structure using the implicit addressing techniques:

```
(set 'eno (ref "Brian Eno" itunes-data))
;-> (0 2 14 528 6 1)                   ; address of Brian Eno

(0 4 eno)                              ; implicit slice
;-> (0 2 14 528)

(itunes-data (0 4 eno))
;->
(dict
 (key "Track ID")
 (int "305")
 (key "Name")
 (string "An Ending (Ascent)")
 (key "Artist")
 (string "Brian Eno") ; this was (0 2 14 528 6 1)
 (key "Album")
 (string "Ambient Journeys")
 (key "Genre")
 (string "ambient, new age, electronica")
 (key "Kind")
 (string "Apple Lossless audio file")
 (key "Size")
 (int "21858166")
 ...
```

How to remember the difference between the two types of implicit addressing? sLice numbers go in the Lead, sElect numbers go at the End. The shorter formulation is also the quicker.

List surgery

Shortening lists

To shorten a list, by removing elements from the front or back, use **chop** or **pop**. **chop** makes a copy and works from the end, **pop** changes the original and works from the front.

chop returns a new list by cutting the end off the list:

```
(set 'vowels '("a" "e" "i" "o" "u"))
(chop vowels)
;-> ("a" "e" "i" "o")

(println vowels)
("a" "e" "i" "o" "u")                   ; original unchanged
```

An optional third argument for **chop** specifies how many elements to remove:

```
(chop vowels 3)
;-> ("a" "e")

(println vowels)
("a" "e" "i" "o" "u") ; original unchanged
```

pop (the opposite of **push**) permanently removes the specified element from the list, and works with list indices rather than lengths:

```
(set 'vowels '("a" "e" "i" "o" "u"))

(pop vowels)                            ; defaults to 0-th element

(println vowels)
("e" "i" "o" "u")

(pop vowels -1)

(println vowels)
("e" "i" "o")
```

You can also use **replace** to remove items from lists.

Changing items in lists

You can easily change elements in lists, using the following functions:

- **replace** changes or removes elements
- **nth-set** changes the nth element
- **set-nth** changes the nth element
- **swap** swaps two elements
- **set-ref** searches for and changes an element

- **ref-set** searches for and changes an element
- **set-ref-all** searches for and changes every element

These are destructive functions, just like **push**, **pop**, **reverse**, and **sort**, and change the original lists, so use them carefully.

Replacing information: replace

You can use **replace** to change or remove elements in lists. Specify the element to change and the list to search in, and also a replacement if there is one.

```
(set 'data (sequence 1 10))
(replace 5 data)                   ; no replacement specified
;-> (1 2 3 4 6 7 8 9 10)           ; the 5 has gone

(set 'data '(("a" 1) ("b" 2)))
(replace ("a" 1) data)             ; data is now (("b" 2))
```

replace returns the changed list:

```
(set 'data (sequence 1 10))
(replace 5 data 0)                   ; replace 10 with 0
;-> (1 2 3 4 0 6 7 8 9 10)
```

The replacement can be a simple value, or any expression that returns a value.

```
(set 'data (sequence 1 10))
(replace 5 data (sequence 0 5))
;->(1 2 3 4 (0 1 2 3 4 5) 6 7 8 9 10)
```

replace updates a set of system variables $0, $1, $2, up to $15, with the matching data. For list replacements, only $0 is used, and holds the value of the found item, suitable for using in the replacement expression.

```
(replace 5 data (list (dup $0 2)))    ; $0 holds 5
;-> (1 2 3 4 ((5 5)) 6 7 8 9 10)
```

For more about system variables and their use with string replacements, see "**System variables: $0, $1 ...**" on page **72**.

The default test for finding things in lists is equality. If you don't supply a test function, **=** is used:

```
(set 'data (sequence 1 10))
(replace 5 data 0 =)
;-> (1 2 3 4 0 6 7 8 9 10)

(set 'data (sequence 1 10))
(replace 5 data 0)              ; = is assumed
;-> (1 2 3 4 0 6 7 8 9 10)
```

You can make **replace** find elements that pass a different test, rather than equality. Supply the test function after the replacement value:

```
(set 'data (randomize (sequence 1 10)))
;-> (5 10 6 1 7 4 8 3 9 2)
(replace 5 data 0 <)  ; replace everything that 5 is less than
;-> (5 0 0 1 0 4 0 3 0 2)
```

The test can be any function that compares two values and returns a true or false value. This can be amazingly powerful. Suppose you have a list of names and their scores:

```
(set 'scores '(
   ("adrian" 234 27 342 23 0)
   ("hermann" 92 0 239 47 134)
   ("neville" 71 2 118 0)
   ("eric" 10 14 58 12 )))
```

How easy is it to add up the numbers for all those people whose scores included a 0? Well, with the help of the **match** function, this easy:

```
(replace '(* 0 *) scores (list (first $0) (apply + (rest $0))) match)

(("adrian" 626)
 ("hermann" 512)
 ("neville" 191)
 ("eric" 10 14 58 12))
```

Here, for each matching element, the replacement expression builds a list from the name and the sum of the scores. **match** is employed as a comparator function - only matching list elements are selected for totalization, so Eric's scores weren't totalled since he didn't manage to score 0.

See "**Changing substrings**" on page **71** for more information about using **replace** on strings.

Changing the nth element

The difference between **nth-set** and **set-nth** is what they return, rather than how they change a list element. **nth-set** returns just the old element, and is therefore faster than **set-nth**, which returns the modified list (which might be quite large).

```
(set 'data (sequence 100 110))
;-> (100 101 102 103 104 105 106 107 108 109 110)
(set-nth 5 data 0)
;-> (100 101 102 103 104 0 106 107 108 109 110)
(nth-set 5 data "")
;-> 0
data
;-> (100 101 102 103 104 "" 106 107 108 109 110)
```

The system variable $0 contains the old element that was replaced.

Both these functions have an alternate form, which allows you to use implicit addressing. Put the list and the index number in a list, inside **nth-set**.

```
(set 'data (sequence 100 110))
;-> (100 101 102 103 104 105 106 107 108 109 110)
(nth-set (data 5) 0)
;-> 105
data
;-> (100 101 102 103 104 0 106 107 108 109 110)
```

Modifying lists

There are even more powerful ways of modifying the elements in lists. Meet **ref-set**, **set-ref**, and **set-ref-all**, introduced with newLISP version 9.3.

You can locate and modify elements using the functions **set-ref**, **ref-set**, and **set-ref-all**. All these functions are designed to work well with nested lists.

Find and replace matching elements

Like **nth-set** and **set-nth**, the **ref-set** and **set-ref** functions do the same job, but differ in the value they return. **set-ref** returns the entire list; **ref-set** returns only the old element that was replaced.

We'll look at the **set-ref** function here, because it returns the whole list, so it's easier to see what's going on. Remember to use its twin when you need to.

Take this list:

```
(set 'l '((aaa 100) (bbb 200)))
;-> ((aaa 100) (bbb 200))
```

To change that 200 to a 300, use **set-ref**, and put the list and desired element together in a sublist:

```
(set-ref (l 200) 300)              ; change the first 200 to 300
;-> ((aaa 100) (bbb 300))
```

Notice that this syntax – `(list element)` – is similar to the implicit indexing form you use with **nth-set** and **set-nth**, but it's doing something different – you're looking for the element itself in the list that's just been specified, not an index number.

ref-set does the same job, but returns only the old element:

```
(set 'l '((aaa 100) (bbb 200)))
(ref-set (l 200) 300)
;-> 200
l
;-> ((aaa 100) (bbb 300))
```

Find and replace all matching elements: set-ref-all

set-ref finds the first matching element in a nested list and changes it, and **set-ref-all** can replace every matching element. Consider the following nested list that contains data on the planets:

```
 (("Mercury"
      (name "Mercury")
      (diameter 0.382)
      (mass 0.06)
      (radius 0.387)
      (period 0.241)
      (incline 7)
      (eccentricity 0.206)
      (rotation 58.6)
      (moons 0))
  ("Venus"
      (name "Venus")
      (diameter 0.949)
      (mass 0.82)
      (radius 0.72)
      (period 0.615)
      (incline 3.39)
      (eccentricity 0.0068)
      (rotation -243)
      (moons 0))
  ("Earth"
      (name "Earth")
      (diameter 1)
      ...
```

How could you change every occurrence of that 'incline' symbol to be 'inclination'? It's easy using **set-ref-all**:

```
(set-ref-all (planets 'incline) 'inclination)
```

This returns the list with every 'incline changed to 'inclination.

As with **replace**, the default test for finding matching elements is equality. But you can supply a different comparison function. This is how you could examine the list of planets and change every entry where the moon's value is greater than 9 to say "lots" instead of the actual number.

```
(set-ref-all '(moons ?) planets (if (> (last $0) 9) "lots" (last $0))
match)
```

The replacement expression compares the number of moons (the last item of the result which is stored in $0) and evaluates to "lots" if it's greater than 9. The search term is formulated using **match**-friendly wildcard syntax.

Swap

The **swap** function can exchange two elements of a list, two characters in a string, or the values of two symbols. This changes the original list:

```
(set 'fib '(1 2 1 3 5 8 13 21))
(swap 1 2 fib)                         ; list swap
;-> (1 1 2 3 5 8 13 21)

fib
;-> (1 1 2 3 5 8 13 21)                ; is 'destructive'

(set 'w "teh")
(swap 1 2 w)                           ; string swap
;-> the

w
;-> the                                ; is 'destructive'
```

Usefully, **swap** can also swap the values of two symbols without you having to use an intermediate temporary variable. This parallel assignment can make life easier sometimes, such as in this slightly unusual iterative version of a function to find Fibonacci numbers:

```
(define (fibonacci n)
 (let (current 1 next 0)
  (dotimes (j n)
   (print current " ")
   (inc 'next current)
   (swap current next))))

(fibonacci 20)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Working with two or more lists

If you have two lists, you might want to ask questions such as 'How many items are in both lists?', 'Which items are in only one of the lists?', 'How often do the items in this list occur in another list?', and so on. Here are some useful functions for answering these questions:

- **count** counts elements of one list in another list
- **difference** finds the set difference of two lists
- **intersect** finds the intersection of two lists
- **count** counts the number of times an element in one list occurs in a second list.

For example, to see how many vowels there are in a sentence, put the known vowels in one list, and the sentence in another (first use **explode** to turn the sentence into a list of characters):

```
(count '("a" "e" "i" "o" "u") (explode "the quick brown fox jumped over
the lazy dog"))
;-> (1 4 1 4 2)
```

The result, `(1 4 1 4 2)`, means that there are 1 a, 4 e's, 1 i, 4 o's, and 2 u's in the sentence.

difference and **intersect** are functions that will remind you of those Venn diagrams you did at school (if you went to a school that taught them). In newLISP lists can represent sets.

difference returns a list of those elements in the first list that are not in the second list. For example, you could compare two directories on your system to find files that are in one but not the other. You can use the **directory** function for this.

```
(set 'd1
 (directory "/Users/me/Library/Application Support/BBEdit"))
(set 'd2
 (directory "/Users/me/Library/Application Support/TextWrangler"))

(difference d1 d2)
;-> ("AutoSaves" "Glossary" "HTML Templates" "Stationery" "Text
Factories")

(difference d2 d1)
;-> ()
```

It's important which list you put first! There are five files or directories in directory `d1` that aren't in directory `d2`, but there are no files or directories in `d2` that aren't also in `d1`.

The **intersect** function finds the elements that are in both lists.

```
(difference d2 d1)
;-> ("." ".." ".DS_Store" "Language Modules" "Menu Scripts" "Plug-Ins"
"Read Me.txt" "Scripts" "Unix Support")
```

Both these functions can take an additional argument, which controls whether to keep or discard any duplicate items.

You could use the **difference** function to compare two revisions of a text file. Use **parse** ("**Parsing strings**" on page **77**) to split the files into lines first:

```
(set 'd1
 (parse (read-file "/Users/me/f1-(2006-05-29)-1.html") "\r" 0))

(set 'd2
 (parse (read-file "/Users/me/f1-(2006-05-29)-6.html") "\r" 0))

(println (difference d1 d2))

(" <p class=\"body\">You could use this function to find" ...)
```

Association lists

There are various techniques available in newLISP for storing information. One very easy and effective technique is to use a list of sublists, where the first element of each sublist is a 'key'. This structure is known as an association list, but you could also think of it as a dictionary, since you look up information in the list by first looking for the key element.

You can also implement a dictionary using newLISP's contexts. See "**Introducing contexts**" on page **87**.

You can make an association list using basic list functions. For example, you can supply a hand-crafted quoted list:

```
(set 'ascii-chart '(("a" 97) ("b" 98) ("c" 99) ...))
```

Or you could use functions like **list** and **push** to build the association list:

```
(for (c (char "a") (char "z"))
 (push (list (char c) c) ascii-chart -1))

ascii-chart
;-> (("a" 97) ("b" 98) ("c" 99) ... ("z" 122))
```

It's a list of sublists, and each sublist has the same format. The first element of a sublist is the key. The key can be a string, a number, or a symbol. You can have any number of data elements after the key.

Here's an association list that contains some data about the planets in the solar system:

```
(set 'sol-sys
 '(("Mercury" 0.382 0.06 0.387 0.241 7.00 0.206 58.6 0)
   ("Venus" 0.949 0.82 0.72 0.615 3.39 0.0068 -243 0)
   ("Earth" 1.00 1.00 1.00 1.00 0.00 0.0167 1.00 1)
   ("Mars" 0.53 0.11 1.52 1.88 1.85 0.0934 1.03 2)
   ("Jupiter" 11.2 318 5.20 11.86 1.31 0.0484 0.414 63)
   ("Saturn" 9.41 95 9.54 29.46 2.48 0.0542 0.426 49)
   ("Uranus" 3.98 14.6 19.22 84.01 0.77 0.0472 -0.718 27)
   ("Neptune" 3.81 17.2 30.06 164.8 1.77 0.0086 0.671 13)
   ("Pluto" 0.18 0.002 39.5 248.5 17.1 0.249 -6.5 3)
   )
   ; 0: Planet name 1: Equator diameter (earth) 2: Mass (earth)
   ; 3: Orbital radius (AU) 4: Orbital period (years)
   ; 5: Orbital Incline Angle 6: Orbital Eccentricity
   ; 7: Rotation (days) 8: Moons
)
```

Each sublist starts with a string, the name of a planet, which is followed by data elements, numbers in this case. The planet name is the 'key'. I've included some comments at the end, because I'm never going to remember that element 2 is the planet's mass, in Earth masses.

You could easily access this information using standard list-processing techniques, but newLISP offers some tailor-made functions that are designed to work specifically with these dictionary or association lists:

- **assoc** finds the first occurrence of the keyword and return the sublist
- **lookup** looks up the value of a keyword inside the sublist
- **set-assoc** replaces the sublist associated with a key

Both **assoc** and **lookup** take the first element of the sublist, the key, and retrieve some data from the appropriate sublist. Here's **assoc** in action, returning the sublist:

```
(assoc "Uranus" sol-sys)
;-> ("Uranus" 3.98 14.6 19.22 84.01 0.77 0.0472 -0.718 27)
```

And here's **lookup**, which goes the extra mile and gets data out of an element of one of the sublists for you, or the final element if you don't specify one:

```
(lookup "Uranus" sol-sys)
;-> 27, moons - value of the final element of the sublist

(lookup "Uranus" sol-sys 2)
;-> 14.6, element 2 of the sublist is the planet's mass
```

This saves you having to use a combination of **assoc** and **nth**.

One problem that you might have when working with association lists with long sublists like this is that you can't remember what the index numbers represent. Here's one solution:

```
(constant 'orbital-radius 3)
(constant 'au 149598000)                 ; 1 au in km
(println "Neptune's orbital radius is "
 (mul au (lookup "Neptune" sol-sys orbital-radius))
 " kilometres")
```

```
Neptune's orbital radius is 4496915880 kilometres
```

Here we've defined `orbital-radius` and `au` (astronomical unit) as constants, and you can use `orbital-radius` to refer to the right column of a sublist. This also makes the code easier to read. The **constant** function is like **set**, but the symbol you supply is protected against accidental change by another use of **set**. You can change the value of the symbol only by using the **constant** function again.

Having defined these constants, here's an expression that lists the different orbits of the planets, in kilometres:

```
(dolist (planet-data sol-sys)             ; go through list
 (set 'planet (first planet-data))        ; get name
 (set 'orb-rad
  (lookup planet sol-sys orbital-radius)) ; get radius
 (println
    planet
    (format "%12.2f %18.0f"
     orb-rad
     (mul au orb-rad)))))
```

```
Mercury 0.39 57894426
Venus 0.72 107710560
Earth 1.00 149598000
Mars 1.52 227388960
Jupiter 5.20 777909600
Saturn 9.54 1427164920
Uranus 19.22 2875273560
Neptune 30.06 4496915880
Pluto 39.50 5909121000
```

When you want to manipulate floating-point numbers, use the floating-point arithmetic operators **add**, **sub**, **mul**, **div** rather than **+**, **-**, *****, and **/**, which work with (and convert values to) integers.

Replacing sublists in association lists

The function **set-assoc** is designed to make it easier for you to change the values stored in an association list. Supply a replacement sublist to the function, consisting of the name of the association list followed by the name of the key for the element you want to modify. Then supply a replacement for that element:

```
(set-assoc (sol-sys "Jupiter") '("Jupiter" 11.2 318 5.20 11.86 1.31
0.0484 0.414 64))
```

The value returned by this function is the whole association list. If your association lists are very large, you might want to use the version of this function that returns only the changed element, **assoc-set**.

There's a set of list-changing functions, and many of them come in pairs. One returns the changed list, the other returns the changed element. If your lists are very large, or you're particularly interested in the changed part of the list, use the 'function-set' version. The 'set-function' version returns the whole changed list.

Adding new items to association lists

Association lists are ordinary lists, too, so you can use all the familiar newLISP techniques with them. Want to add a new 10th planet to our `sol-sys` list? Just use **push**:

```
(push '("Sedna" 0.093 0.00014 .0001 502 11500 0 20 0) sol-sys -1)
```

and check that it was added OK with:

```
(assoc "Sedna" sol-sys)
;-> ("Sedna" 0.093 0.00014 0.0001 502 11500 0 20 0)
```

You can use **sort** to sort the association list. (Remember though that **sort** changes lists permanently.) Here's a list of planets sorted by mass. Since you don't want to sort them by name, you use a custom sort (see "**sort and randomize**" on page **38**) to compare the mass (index 2) values of each pair:

```
(constant 'mass 2)
(sort sol-sys (fn (x y) (> (x mass) (y mass))))

(println sol-sys)

("Jupiter" 11.2 318 5.2 11.86 1.31 0.0484 0.414 63)
("Saturn" 9.41 95 9.54 29.46 2.48 0.0542 0.426 49)
("Neptune" 3.81 17.2 30.06 164.8 1.77 0.0086 0.671 13)
("Uranus" 3.98 14.6 19.22 84.01 0.77 0.0472 -0.718 27)
("Earth" 1 1 1 1 0 0.0167 1 1)
("Venus" 0.949 0.82 0.72 0.615 3.39 0.0068 -243 0)
("Mars" 0.53 0.11 1.52 1.88 1.85 0.0934 1.03 2)
("Mercury" 0.382 0.06 0.387 0.241 7 0.206 58.6 0)
("Pluto" 0.18 0.002 39.5 248.5 17.1 0.249 -6.5 3)
```

You can also easily combine the data in the association list with other lists:

```
; restore to standard order - sort by orbit radius
(sort sol-sys (fn (x y) (< (x 3) (y 3))))

; define Unicode symbols for planets
(set 'unicode-symbols
  '(("Mercury" 0x263F )
    ("Venus" 0x2640 )
    ("Earth" 0x2641 )
    ("Mars" 0x2642 )
    ("Jupiter" 0x2643 )
    ("Saturn" 0x2644 )
    ("Uranus" 0x2645 )
    ("Neptune" 0x2646 )
    ("Pluto" 0x2647)))
(map
 (fn (planet)
 (println (char (lookup (first planet) unicode-symbols))
  "\t"
 (first planet)))
 sol-sys)
```

```
☿ Mercury
♀ Venus
♁ Earth
♂ Mars
♃ Jupiter
♄ Saturn
♅ Uranus
♆ Neptune
♇ Pluto
```

Here we've created a temporary inline function that **map** applies to each planet in `sol-sys` – **lookup** finds the planet name and retrieves the Unicode symbol for that planet from the `unicode-symbols` association list.

> If you're looking at the HTML version of this document, you might see the Unicode symbols and/or the alternative text, 'Unicode symbol for ...'.

You can quickly remove an element from an association list with **pop-assoc**. This works in the same way as **set-assoc**, but you don't supply a replacement.

```
(pop-assoc (sol-sys "Pluto"))
```

This removes the Pluto element from the list.

newLISP offers powerful data storage facilities in the form of contexts, which you can use for building dictionaries, hash tables, objects, and so on. You can use association lists to build dictionaries, and work with the contents of dictionaries using association list functions. See "**Introducing contexts**" on page **87**.

You can also use a database engine – see "**Using a SQLite database**" on page **156**.

find-all and association lists

Another form of **find-all** lets you search an association list for a sublist that matches a pattern. You can specify the pattern with wildcard characters. For example, here's an association list:

```
(set 'symphonies
  '((Beethoven 9)
    (Haydn 104)
    (Mozart 41)
    (Mahler 10)
    (Wagner 1)
    (Schumann 4)
    (Shostakovich 15)
    (Bruckner 9)))
```

To find all the sublists that end with 9, use the match pattern `'(? 9)`, where the question mark matches any single item:

```
(find-all '(? 9) symphonies)
;-> ((Beethoven 9) (Bruckner 9))
```

(For more about match patterns – wild card searches for lists – see "**matching patterns in lists**" on page **44**.)

You can also use this form with an additional action expression after the association list:

```
(find-all '(? 9) symphonies
   (println (first $0) { wrote 9 symphonies.}))
```

```
Beethoven wrote 9 symphonies.
Bruckner wrote 9 symphonies.
```

Here, the action expression uses $0 to refer to each matched element in turn.

5 Strings

String-handling tools are an important part of a programming language. newLISP has many easy to use and powerful string handling tools, and you can easily add more tools to your toolbox if your particular needs aren't met.

Here's a guided tour of newLISP's 'string orchestra'.

Strings in newLISP code

You can write strings in three ways:

- enclosed in double quotes
- embraced by curly braces
- marked-up by markup codes

like this:

```
(set 's "this is a string")
(set 's {this is a string})
(set 's [text]this is a string[/text])
```

Always use the third method for strings longer than 2048 characters. The first and second methods can handle strings with less than 2048 characters.

Use the first method, quotation marks, if you want escaped characters such as \n and \t, or code numbers (\046), to be processed.

```
(set 's "this is a string \n with two lines")
(println s)
```

```
this is a string
with two lines
```

```
(println "\110\101\119\076\073\083\080")    ; decimal ASCII
```

```
newLISP
```

```
(println "\x6e\x65\x77\x4c\x49\x53\x50")    ; hex ASCII
```

```
newLISP
```

Double-quote characters must be escaped with backslashes, as must a backslash.

Use the second method, braces (or 'curly brackets'), for strings shorter than 2048 characters when you don't want any escaped characters to be processed:

```
(set 's {strings can be enclosed in \n"quotation marks" \n })
(println s)
```

```
strings can be enclosed in \n"quotation marks" \n
```

This is a really useful way of writing strings, because you don't have to worry about putting backslashes before every quotation character, or backslashes before other backslashes. You can nest pairs of braces inside a braced string, but you can't have an odd unmatched brace – there's no way to 'escape' it. I like to use braces for strings, because they face the correct way (which plain 'dumb' quotation marks don't) and because your text editor might be able to balance and match them.

The third method, using `[text]` and `[/text]` markup tags, is intended for longer text strings running over many lines, and is used automatically by newLISP when it outputs large amounts of text. Again, you don't have to worry about which characters you can and can't include – you can put anything you like in, with the obvious exception of `[/text]`. Escape characters such as \n or \046 aren't processed either.

```
(set 'novel (read-file {my-latest-novel.txt}))

;->
[text]
It was a dark and "stormy" night...
...
The End.
[/text]
```

If you want to know the length of a string, use **length**:

```
(length novel)
;-> 575196
```

Strings of millions of characters can be handled easily by newLISP.

You can use **utf8len** to get the length of a Unicode string in Unicode characters:

```
(utf8len (char 955))
;-> 1

(length (char 955))
;-> 2
```

Making strings

Many functions, such as the file-reading ones, return strings or lists of strings for you. But if you want to build a string from scratch, one way is to start with the **char** function. This converts the supplied number to the equivalent character string with that code number. It can also reverse the operation, converting the supplied character string to its equivalent code number.)

```
(char 33)
;-> "!"
(char "!")
;-> 33
(char 955)        ; Unicode lambda character, decimal code
;-> "\206\187"
```

```
(char 0x2643)    ; Unicode symbol for Jupiter, hex code
;-> "\226\153\131"
```

These last two examples are available when you're running the Unicode-capable version of newLISP. Since Unicode is hexadecimally inclined, you can give a hex number, starting with `0x`, to **char**. To see the actual characters, use a printing command:

```
(println (char 955))
```

λ

```
;-> "\206\187"
(println (char 0x2643))
```

♃

```
;-> "\226\140\152"
```

The backslashed numbers are the result of the **println** function, presumably the multi-byte values of the Unicode glyph.

You can use **char** to build strings in other ways:

```
(join (map char (sequence (char "a") (char "z"))))
;-> "abcdefghijklmnopqrstuvwxyz"
```

This uses **char** to find out the ASCII code numbers for `a` and `z`, and then uses **sequence** to generate a list of code numbers between the two. Then the **char** function is mapped onto every element of the list, so producing a list of strings. Finally, this list is converted to a single string by **join**.

join can also take a separator when building strings:

```
(join (map char (sequence (char "a") (char "z"))) "-")
;-> "a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
```

Similar to **join** is **append**, which works directly on strings:

```
(append "con" "cat" "e" "nation")
;-> "concatenation"
```

but even more useful is **string**, which turns any collection of numbers, lists, and strings into a single string.

```
(string '(sequence 1 10) { produces } (sequence 1 10) "\n")
;-> (sequence 1 10) produces (1 2 3 4 5 6 7 8 9 10)
```

Notice that the first list wasn't evaluated (because it was quoted) but that the second list was evaluated to produce a list of numbers, and the resulting list – including the parentheses – was converted to a string.

The **string** function, combined with the various string markers such as braces and markup tags, is a good way to include the values of variables inside strings:

```
(define x 42)
(string {the value of } 'x { is } x)
;-> "the value of x is 42"
```

You can also use **format** to combine strings and symbol values. See "**Formatting strings**" on page **78**.

dup makes copies:

```
(dup "spam" 10)
;-> "spamspamspamspamspamspamspamspamspamspam"
```

And **date** makes a date string:

```
(date)
;-> "Wed Jan 25 15:04:49 2006"
```

or you can give it a number of seconds since 1970 to convert:

```
(date 1230000000)
;-> "Tue Dec 23 02:40:00 2008"
```

See "**Date and time functions**" on page **123**.

String surgery

Now you've got your string, there are plenty of functions for operating on them. Some of these are 'destructive' functions – they change the string permanently, possibly losing information for ever. Others are constructive, producing a new string and leaving the old one unharmed. See "**Destructive functions**" on page **13**.

reverse is destructive:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(reverse t)
;-> "elcitrap cimotabus lanoisnemid-eno lacitehtopyh a"
```

Now `t` has changed for ever. However, the case-changing functions aren't destructive, producing new strings without harming the old ones:

```
(set 't "a hypothetical one-dimensional subatomic particle")

(upper-case t)
;-> "A HYPOTHETICAL ONE-DIMENSIONAL SUBATOMIC PARTICLE"

(lower-case t)
;-> "a hypothetical one-dimensional subatomic particle"

(title-case t)
;-> "A hypothetical one-dimensional subatomic particle"
```

Substrings

If you know which part of a string you want to extract, use one of the following constructive functions:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(first t)
;-> "a"

(rest t)
;-> " hypothetical one-dimensional subatomic particle"

(last t)
;-> "e"

(t 2)
;-> "h"
```

You can also use this technique with lists. See "**Selecting items from lists**" on page **49**.

String slices

slice gives you a new slice of an existing string, counting either from the beginning (positive integers) or from the end (negative integers), for a given number of characters:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(slice t 15 13)
;-> "one-dimension"

(slice t -8 8)
;-> "particle"

(slice t 2 -9)
;-> "hypothetical one-dimensional subatomic"

(slice "schwarzwalderkirschtorte" 19 -1)
;-> "tort"
```

There's a shortcut to do this, too. Put the required start and length before the string in a list:

```
(15 13 t)
;-> "one-dimension"

(0 14 t)
;-> "a hypothetical"
```

If you don't want a continuous run of characters, but want to cherry-pick some of them for a new string, use **select** followed by a sequence of character index numbers:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(select t 3 5 24 48 21 10 44 8)
;-> "yosemite"

(select t (sequence 1 49 12)) ; every 12th char starting at 1
;-> " lime"
```

which is good for finding secret coded messages buried in text.

If you just want to swap two characters, use the destructive function **swap**:

```
(set 'typo {teh})
(swap 2 1 typo)
;-> "the"
```

Changing the ends of strings

trim and **chop** are both constructive string-editing functions that work from the ends of the original strings inwards:

```
(chop t)       ; defaults to the last character
;-> "a hypothetical one-dimensional subatomic particl"

(chop t 9)     ; chop 9 characters off
;-> "a hypothetical one-dimensional subatomic"
```

trim removes characters from the ends of a source string:

```
(set 's "  centred  ")
(trim s)       ; defaults to removing spaces
;-> "centred"
```

```
(set 's "------centred------")
(trim s "-")
;-> "centred"

(set 's "------centred********")
(trim s "-" "*")
;-> "centred"
```

push and pop work on strings too

You've seen **push** and **pop** adding and removing items from lists. They work on strings too. Use **push** to add characters to a string, and **pop** to remove one character from a string. Strings are added to or removed from the start of the string, unless you specify an index.

```
(set 't "some ")
(push "this is " t)
(push "text " t -1)
;-> t is now "this is some text"
```

push and **pop** always return what was pushed or popped, not the modified target of the action. This is quicker when you have large lists or strings. It's also useful when you want to break up a string and process the pieces as you go. For example, to print the newLISP version number, which is stored as a 4 or 5 digit integer, use something like this:

```
(set 'version-string (string (sys-info -2)))
; eg: version-string is "9405"
(set 'dev-version (pop version-string -2 2))   ; always two digits
; version-string is now "94"
(set 'point-version (pop version-string -1))   ; always one digit
; version-string is now "9"
(set 'version version-string)                  ; one or two digits
(println version "." point-version "." dev-version " on " ostype)
```

```
9.4.05 on OSX
```

It's easier to work from the right-hand side of the string and use **pop** to extract the information and remove it in one operation.

ostype returns a string that specifies your current operating system.

Modifying strings

There are two approaches to changing characters inside a string. Either use the index numbers of the characters, or specify the substring you want to find or change.

Using index numbers in strings

Use character indexing with the **nth-set** and **set-nth** functions, also used for modifying lists.

nth-set and **set-nth** are twin character assassins – destructive functions for changing strings. They make the same changes, but **nth-set** returns just the part of the string that was destroyed, and **set-nth** returns the modified string. **nth-set** is therefore slightly quicker.

```
(set 't "a b c")
(nth-set (t 0) ">")
;-> "a"
t
;-> "> b c"
```

```
(set 't "a b c")
(set-nth (t 0) ">")
;-> "> b c"
```

To remember which does which, consider that **set-nth** starts with `s` and returns the string, whereas **nth-set** starts with `n` and returns only the nth characters. (If this doesn't work for you, remember them another way!)

Changing substrings

If you don't want to – or can't – deal with index numbers or character positions, use **replace**, a powerful destructive function that does all kinds of useful operations on strings. Use it in the form:

```
(replace old-string source-string replacement)
```

So:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(replace "hypoth" t "theor")
;-> "a theoretical one-dimensional subatomic particle"
```

replace is destructive, but if you want to use **replace** or another destructive function constructively, without affecting the original string, enclose the string in a **string** function call:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(replace "hypoth" (string t) "theor")
;-> "a theoretical one-dimensional subatomic particle"
t
;-> "a hypothetical one-dimensional subatomic particle"
```

This use of **string** creates a new copy that gets operated on by **replace**. The original string `t` is unaffected.

Regular expressions

replace is one of a group of newLISP functions that accept regular expressions for defining patterns in text. For most of them, you add an extra number at the end of the expression which specifies options for the regular expression operation: 0 means basic matching, 1 means case-sensitive matching, and so on.

```
(set 't "a hypothetical one-dimensional subatomic particle")
(replace {h.*?l(?# h followed by l but not too greedy)} t {} 0)

;-> "a  one-dimensional subatomic particle"
```

Sometimes I put comments inside regular expressions, so that I know what I was trying to do when I read the code some days later. Text between (?# and the following closing parenthesis is ignored.

If you're happy working with Perl-compatible Regular Expressions (PCRE), you'll be happy with **replace** and its regex-using cousins (**find**, **regex**, **find-all**, **parse**, **starts-with**, **ends-with**, **directory**, and **search**). Full details are in the newLISP reference manual.

You have to steer your pattern through both the newLISP interpreter and the regular expression processor. Remember the difference between strings enclosed in quotes and strings enclosed in braces? Quotes allow the processing of escaped characters, whereas braces don't. Braces have some advantages: they face each other visually, they don't have smart and dumb versions to confuse you, your text editor might balance them for you, and they let you use the more commonly occurring quotation characters in strings without having to escape them all the time. If you use quotes, you should double the backslashes, so that a single backslash survives intact as far as the regular expression processor:

```
(set 'str "\s")
(replace str "this is a phrase" "|" 0)  ; not searching for \s (white
space) ...
;-> thi| i| a phra|e                    ; but for the letter s

(set 'str "\\s")
(replace str "this is a phrase" "|" 0)
;-> this|is|a|phrase                    ; better!
```

System variables: $0, $1 ...

replace updates a set of system variables $0, $1, $2, up to $15, with the matches. These refer to the parenthesized expressions in the pattern, and are the equivalent of the \1, \2 that you might be familiar with if you've used grep. For example:

```
(set 'quotation {"I cannot explain." She spoke in a low, eager voice,
with a
curious lisp in her utterance. "But for God's sake do what I ask you. Go
back
and never set foot upon the moor again."})

(replace {(.*?),.*?curious\s*(l.*p\W)(.*?)(moor)(.*)}
    quotation
    (println {$1 } $1 { $2 } $2 { $3 } $3 { $4 } $4 { $5 } $5)
    0)
```

```
$1 "I cannot explain." She spoke in a low $2 lisp  $3 in her utterance.
"But
for God's sake do what I ask you. Go back and never set foot upon the  $4
moor
$5  again."
```

Here we've looked for five patterns, separated by any string starting with a comma and ending with the word curious. $0 stores the matched expression, $1 stores the first parenthesized subexpression, and so on.

If you prefer to use quotation marks rather than the braces I used here, remember that certain characters have to be escaped with a backslash.

The replacement expression

The previous example demonstrates that an important feature of **replace** is that the replacement doesn't have to be just a simple string or list, it can be any newLISP expression. Each time the pattern is found, the replacement expression is evaluated. You can use this to provide a replacement value that's calculated dynamically, or you could do anything else you wanted to with the found text. It's even possible to evaluate an expression that's got nothing to do with found text at all.

Here's another example: search for the letter t followed either by the letter h or by any vowel, and print out the combinations that **replace** found:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(replace {t[h]|t[aeiou]} t (println $0) 0)
```

```
th
ti
to
ti
t

;-> "a hypothetical one-dimensional subatomic particle"
```

For every matching piece of text found, the third expression

```
(println $0)
```

was evaluated. This is a good way of seeing what the regular expression engine is up to while the function is running. In this example, the original string appears to be unchanged, but in fact it did change, because `(println $0)` did two things: it printed the string, and it returned the value to **replace**, thus replacing the found text with itself. Invisible mending! If the replacement expression doesn't return a string, no replacement occurs.

You could do other useful things too, such as build a list of matches for later processing, and you can use the newLISP system variables and any other function to use any of the text that was found.

In the next example, we look for the letters a, e, or c, and force each occurrence to upper-case:

```
(replace "a|e|c" "This is a sentence" (upper-case $0) 0)
;-> "This is A sEntEnCE"
```

As another example, here's a simple search and replace operation that keeps count of how many times the letter 'o' has been found in a string, and replaces each occurrence in the original string with the count so far. The replacement is a block of expressions grouped into a single **begin** expression. This block is evaluated every time a match is found:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(set 'counter 0)
(replace "o" t
 (begin
  (inc 'counter)
  (println {replacing "} $0 {" number } counter)
  (string counter))         ; the replacement text should be a string
 0)
```

```
replacing "o" number 1
replacing "o" number 2
replacing "o" number 3
replacing "o" number 4

"a hyp1thetical 2ne-dimensi3nal subat4mic particle"
```

The output from **println** doesn't appear in the string; the final value of the entire **begin** expression is a string version of the counter, so that gets inserted into the string.

Here's yet another example of **replace** in action. Suppose you have a text file, consisting of the following:

```
1 a = 15
2 another_variable = "strings"
4 x2 = "another string"
5 c = 25
3x=9
```

We want to write a newLISP script that re-numbers the lines in multiples of 10, starting at 10, and aligns the text so that the equals signs line up, like this:

```
10 a                     = 15
20 another_variable      = "strings"
30 x2                    = "another string"
40 c                     = 25
50 x                     = 9
```

(I don't know what language this is!)

The following script will do this:

```
(set 'file (open ((main-args) 2) "read"))
(set 'counter 0)
(while (read-line file)
 (set 'temp
   (replace {^(\d*)(\s*)(.*)}            ; the numbering
     (current-line)
     (string (inc 'counter 10) " " \$3)
     0))
 (println
   (replace {(\S*)(\s*)(=)(\s*)(.*)}     ; the spaces around =
    temp
    (string $1 (dup " " (- 20 (length $1))) $3 " " $5)
    0)))
(exit)
```

I've used two **replace** operations inside the **while** loop, to keep things clearer. The first one sets a temporary variable to the result of a replace operation. The search string is a regular expression `({^(\d*)(\s*)(.*)})` that's looking for any number at the start of a line, followed by some space, followed by anything. The replacement string `((string (inc 'counter 10) " " $3) 0))` consists of a incremented counter value, followed by the third match (ie the 'anything' I just looked for).

The result of the second replace operation is printed. I'm searching the temporary variable `temp` for more strings and spaces with an equals sign in the middle:

```
({(\S*)(\s*)(=)(\s*)(.*)})
```

The replacement expression is built up from the important found elements ($1, $3, $5) but it also includes a quick calculation of the amount of space required to bring the equals sign across to character 20, which should be the difference between the first item's width and position 20 (which I've chosen arbitrarily as the location for the equals sign).

Regular expressions aren't very easy for the newcomer, but they're very powerful, particularly with newLISP's **replace** function, so they're worth learning.

Testing and comparing strings

There are various tests that you can run on strings. newLISP's comparison operators work by finding and comparing the code numbers of the characters until a decision can be made:

```
(> {Higgs Boson} {Higgs boson})         ; nil
(> {Higgs Boson} {Higgs})               ; true
(< {dollar} {euro})                     ; true
(> {newLISP} {LISP})                    ; true
(= {fred} {Fred})                       ; nil
(= {fred} {fred})                       ; true
```

and of course newLISP's flexible argument handling lets you test loads of strings at the same time:

```
(< "a" "c" "d" "f" "h")
;-> true
```

These comparison functions also let you use them with a single argument. If you supply only one argument, newLISP helpfully assumes that you mean 0 or "", depending on the type of the first argument:

```
(> 1)                                   ; true - assumes > 0
(> "fred")                              ; true - assumes > ""
```

To check whether two strings share common features, you can either use **starts-with** and **ends-with**, or the more general pattern matching commands **member**, **regex**, **find**, and **find-all**. **starts-with** and **ends-with** are simple enough:

```
(starts-with "newLISP" "new")
;-> true
(ends-with "newLISP" "LISP")
;-> true
```

They can also accept regular expressions, using one of the regex options (0 being the most commonly used):

```
(starts-with {newLISP} {[a-z][aeiou](?\#lc followed by lc vowel)} 0)
;-> true
(ends-with {newLISP} {[aeiou][A-Z](?\# lc vowel followed by UCase)} 0)
;-> false
```

find, **find-all**, **member**, and **regex** look everywhere in a string. **find** returns the index of the matching substring:

```
(set ’t "a hypothetical one-dimensional subatomic particle")
(find "atom" t)
;-> 34

(find "l" t)
;-> 13

(find "L" t)
;-> nil                                 ; search is case-sensitive
```

member looks to see if one string is in another. It returns the rest of the string, including the search string, rather than the index of the first occurrence.

```
(member "rest" "a good restaurant")
;-> "restaurant"
```

Both **find** and **member** also let you use regular expressions:

```
(set ’quotation {"I cannot explain." She spoke in a low,
eager voice, with a curious lisp in her utterance. "But for
Gods sake do what I ask you. Go back and never set foot upon
the moor again."})

(find "lisp" quotation)                    ; without regex
;-> 69                                     ; character 69

(find {i} quotation 0))                    ; with regex
;-> 15                                     ; character 15

(find {s} quotation 1)                     ; case insensitive regex
;-> 20                                     ; character 20

(println "character "
 (find {(l.*?p)} quotation 0) ": " $0)     ; l followed by a p
;->

character 13: lain." She sp
```

find-all works like **find**, but returns a list of all matching strings, rather than the index of just the first match. It always takes regular expressions, so – for once – you don’t have to put regex option numbers at the end.

```
(set 'quotation {"I cannot explain." She spoke in a low,
eager voice, with a curious lisp in her utterance. "But for
Gods sake do what I ask you. Go back and never set foot upon
the moor again."})

(find-all "[aeiou]{2,}" quotation $0)       ; two or more vowels
;-> ("ai" "ea" "oi" "iou" "ou" "oo" "oo" "ai")
```

Or you could use **regex**. This returns **nil** if the string doesn't contain the pattern, but, if it does contain the pattern, it returns a list with the matched strings and substrings, and the start and length of each string. The results can be quite complicated:

```
(set 'quotation
 {She spoke in a low, eager voice, with a curious lisp in her
utterance.})

(println (regex {(.*)(l.*)(l.*p)(.*)} quotation 0))

("She spoke in a low, eager voice, with a curious lisp in
her utterance." 0 70 "She spoke in a " 0 15 "low, eager
voice, with a curious " 15 33 "lisp" 48 4 " in her
utterance." 52 18)
```

This results list can be interpreted as 'the first match was from character 0 continuing for 70 characters, the second from character 0 continuing for 15 characters, another from character 15 for 33 characters', and so on.

The matches are also stored in the system variables ($0, $1, ...) which you can inspect easily with a simple loop:

```
(for (x 1 4)
 (println {$} x ": " ($ x)))

$1: She spoke in a
$2: low, eager voice, with a curious
$3: lisp
$4: in her utterance.
```

Strings to lists

Two functions let you convert strings to lists, ready for manipulation with newLISP's extensive list-processing powers. The well-named **explode** function cracks open a string and returns a list of single characters:

```
(set 't "a hypothetical one-dimensional subatomic particle")
(explode t)

:-> ("a" " " "h" "y" "p" "o" "t" "h" "e" "t" "i" "c" "a" "l"
" " "o" "n" "e" "-" "d" "i" "m" "e" "n" "s" "i" "o" "n" "a"
"l" " " "s" "u" "b" "a" "t" "o" "m" "i" "c" " " "p" "a" "r"
"t" "i" "c" "l" "e")
```

The explosion is easily reversed with **join**. **explode** can also take an integer. This defines the size of the fragments. For example, to divide up a string into cryptographer-style 5 letter groups, remove the spaces and use **explode** like this:

```
(explode (replace " " t "") 5)
;-> ("ahypo" "theti" "calon" "e-dim" "ensio" "nalsu" "batom" "icpar"
"ticle")
```

You can do similar tricks with **find-all** (see the next page). Watch the end, though:

```
(find-all ".{3}" t)                     ; this regex drops leftover
characters
;-> ("a h" "ypo" "the" "tic" "al " "one" "-di" "men" "sio"
"nal" " su" "bat" "omi" "c p" "art" "icl")
```

Parsing strings

parse is a powerful way of breaking strings up and returning the pieces. Used on its own, it breaks strings apart, usually at word boundaries, eats the boundaries, and returns a list of the remaining pieces:

```
(parse t)                               ; defaults to spaces...
;-> ("a" "hypothetical" "one-dimensional" "subatomic" "particle")
```

Or you can supply a delimiting character, and **parse** breaks the string whenever it meets that character:

```
(set 'pathname {/System/Library/Fonts/Courier.dfont})
(parse pathname {/})
;-> ("" "System" "Library" "Fonts" "Courier.dfont")
```

By the way, I could eliminate that first empty string from the list by filtering it out:

```
(filter (fn (s) (not (empty? s))) (parse t {/}))
;-> ("System" "Library" "Fonts" "Courier.dfont")
```

You can also specify a delimiter string rather than a delimiter character:

```
(set 't {spamspamspamspamspamspamspamspam})
;-> "spamspamspamspamspamspamspamspam"

(parse t {am})                          ; break on "am"
;-> ("sp" "sp" "sp" "sp" "sp" "sp" "sp" "sp" "")
```

Best of all, though, you can specify a regular expression delimiter. Make sure you supply the options flag (0 or whatever), as with most of the regex functions in newLISP:

```
(set 't {/System/Library/Fonts/Courier.dfont})
(parse t {[/aeiou]} 0)                  ; split at slashes and vowels
;-> ("" "Syst" "m" "L" "br" "ry" "F" "nts" "C" "" "r" "" "r.df" "nt")
```

Here's that well-known quick and not very reliable HTML-tag stripper:

```
(set 'html (read-file "/Users/Sites/index.html"))
(println (parse html {<.*?>} 4))        ; option 4: dot matches newline
```

For parsing XML strings, newLISP provides the function **xml-parse**. See "**Working with XML**" on page **141**.

Take care when using **parse** on text. Unless you specify exactly what you want, it thinks you're passing it newLISP source code. This can produce surprising results:

```
(set 't {Eats, shoots, and leaves ; a book by Lynn Truss})
(parse t)
;-> ("Eats" "," "shoots" "," "and" "leaves")     ; she's gone!
```

The semicolon is considered a comment character in newLISP so **parse** has ignored it and everything that followed on that line. Tell it what you really want, using delimiters or regular expressions:

```
(set 't {Eats, shoots, and leaves ; a book by Lynn Truss})
(parse t " ")
;-> ("Eats," "shoots," "and" "leaves" ";" "a" "book" "by" "Lynn" "Truss")
```

or

```
(parse t "\\s" 0)
;-> ("Eats," "shoots," "and" "leaves" ";" "a" "book" "by" "Lynn" "Truss")
```

If you want to chop strings up in other ways, consider using **find-all**, which returns a list of strings that match a pattern. If you can specify the chopping operation as a regular expression, you're in luck. For example, if you want to split a number into groups of three digits, use this technique:

```
(set 'a "1212374192387562311")
(println (find-all {\d{3}|\d{2}$|\d$} a))
;-> ("121" "237" "419" "238" "756" "231" "1")

; alternatively
(explode a 3)
;-> ("121" "237" "419" "238" "756" "231" "1")
```

The pattern has to consider cases where there are 2 or 1 digits left over at the end.

parse eats the delimiters once they've done their work – **find-all** finds things and returns what it finds.

```
(find-all {\w+} t )                    ; word characters
;-> ("Eats" "shoots" "and" "leaves" "a" "book" "by" "Lynn" "Truss")

(parse t {\w+} 0 )                     ; eats and leaves delimiters
;-> ("" ", " ", " " " "; " " " " " " " " " "")
```

Other string functions

There are other functions that work with strings. **search** looks for a string inside a file on disk:

```
(set 'f (open {/private/var/log/system.log} {read}))
(search f {kernel})
(seek f (- (seek f) 64))               ; rewind file pointer
(dotimes (n 3)
 (println (read-line f)))
(close f)
```

This example looks in system.log for the string 'kernel'. If it's found, newLISP rewinds the file pointer by 64 characters, then prints out three lines, showing the line in context.

There are also functions for working with base64 encoding files, and for encrypting strings.

Formatting strings

It's also worth mentioning the **format** function, which lets you insert the values of newLISP expressions into a pre-defined template string. Use %s to represent the location of a string expression inside the template, and other % codes to include numbers. For example, suppose you want to display a list of files like this:

```
folder: Library
 file:  mach
```

A suitable template for folders (directories) looks like this:

```
"folder: %s" ; or
"  file: %s"
```

Give the **format** function a template string, followed by the expression (f) that produces a file or folder name:

```
(format "folder: %s" f) ; or
(format "  file: %s" f)
```

When this is evaluated, the contents of f is inserted into the string where the %s is. The code to generate a directory listing in this format, using the **directory** function, looks like this:

```
(dolist (f (directory))
 (if (directory? f)
  (println (format "folder: %s" f))
  (println (format "  file: %s" f))))
```

I'm using the **directory?** function to choose the right template string. A typical listing looks like this:

```
folder: .
folder: ..
  file: .DS_Store
  file: .hotfiles.btree
folder: .Spotlight-V100
folder: .Trashes
folder: .vol
  file: .VolumeIcon.icns
folder: Applications
folder: Applications (Mac OS 9)
folder: automount
folder: bin
folder: Cleanup At Startup
folder: cores
...
```

There are lots of formatting codes that you use to produce the output you want. You can output strings, numbers at different levels of precision, and so on. Numbers control the alignment and precision. Just make sure that the % constructions in the format string match the expressions or symbols that appear after it.

Here's another example. We'll display the first 400 or so Unicode characters in decimal, hexadecimal, and binary. We'll have to quickly rustle up a binary display function first, because **format** doesn't do binary. We feed a list of three values to **format** after the format string, which has three entries:

```
(define (binary x , results)
 (until (<= x 0)
   (push (string (% x 2)) results)
   (set 'x (/ x 2)))
 results)

(for (x 32 0x01a0)
 (println (char x)                ; the character, then
   (format "%4d\t%4x\t%10s"       ; decimal \t hex \t binary-string
    (list x x (join (binary x))))))
```

```
  32 20  100000
! 33 21  100001
" 34 22  100010
# 35 23  100011
$ 36 24  100100
% 37 25  100101
```

```
& 38 26  100110
' 39 27  100111
( 40 28  101000
) 41 29  101001
* 42 2a  101010
+ 43 2b  101011
, 44 2c  101100
- 45 2d  101101
. 46 2e  101110
...
```

Strings that make newLISP think

Lastly, I must mention **eval** and **eval-string**. Both of these let you give newLISP code to newLISP for evaluation. If it's valid newLISP you'll see the result of the evaluation. **eval** wants an expression:

```
(set 'expr (+ 1 2))
(eval expr)
;-> 3
```

eval-string wants a string:

```
(set 'expr "(+ 1 2)")
(eval-string expr)
;-> 3
```

This means that you can build newLISP code, using any of the functions we've met, and then have it evaluated by newLISP. **eval** is particularly useful when you're defining macros – functions that delay evaluation until you choose to do it. See "**Macros**" on page **99**.

You could use **eval** and **eval-string** to write programs that write programs.

The following curious piece of newLISP continually and mindlessly rearranges a few strings and tries to evaluate the result. When it finally becomes valid newLISP it will be evaluated successfully and the result will satisfy the finishing condition and finish the loop. The **nil** in **eval-string** tells it to keep going after an unsuccessful evaluation.

```
(set 'code '(")" "set" "'valid" "true" "("))
(set 'valid nil)
(until valid
 (set 'code (randomize code))
 (println (join code " "))
 (eval-string (join code " ") nil))
```

```
) set ( true 'valid
( set true) 'valid
set true ) ( 'valid
) set ( true 'valid
'valid true set ) (
'valid ( set ) true
'valid set ( ) true
...
( ) 'valid set true
) set true 'valid (
set 'valid true ( )
( set 'valid true )
```

I've used programs that were obviously written using this programming technique... :-)

6 Apply and map: applying functions to lists

Making functions and data work together

Often, you'll find that you've got some data stored in a list and you want to apply a function to it. For example, suppose that a program you're running has acquired some temperature readings from a space probe, and they're stored in a list called `data`:

```
(println data)
```

```
(0.1 3.2 -1.2 1.2 -2.3 0.1 1.4 2.5 0.3)
```

How are you going to add these numbers up (and then divide by the total, to find the average)? Perhaps you think you could use **add**, which totals a list of floating-point numbers, but you're not working interactively, so you can't edit the code to read like this:

```
(add 0.1 3.2 -1.2 1.2 -2.3 0.1 1.4 2.5 0.3)
```

Since we're holding the data in a symbol called `data`, we could try this:

```
(add data)
```

```
value expected in function add : data
```

but no, this doesn't work, because **add** wants numbers to add, not a list. You could of course do it the hard way and write a loop that works through each item in the list and increases a running total each time:

```
(set 'total 0)
(dolist (i data)
 (inc 'total i))
```

```
(println total)
```

```
5.3
```

This works fine. But newLISP has a much more powerful solution, for this and many other problems: you can treat functions as data and data as functions, so you can manipulate functions as easily as you can manipulate your data. You can just 'introduce' **add** and the data list to each other, and then stand back and let them get on with it.

There are two important functions for doing this: **apply** and **map**.

apply

apply takes a function and a list, and makes them work together:

```
(apply add data)
;-> 5.3
```

and this produces the required result. Here we've treated the **add** function like any other newLISP list, string, or number, using it as an argument to another function. You don't need to quote it (although you can), because **apply** is already expecting the name of a function.

map

The other function that can make functions and lists work together is **map**, which applies a function to each item of a list, one by one. For example, if you wanted to apply the **floor** function to each element of the data list (to round them down to the nearest integer) you could combine **map**, **floor**, and the data as follows:

```
(map floor data)
;-> (0 3 -2 1 -3 0 1 2 0)
```

and the **floor** function is applied to each element of the data. The results are combined and returned in a new list.

apply and map in more detail

Both **apply** and **map** let you treat functions as data. They have the same basic form:

```
(apply f l)
(map f l)
```

where `f` is the name of a function and `l` is a list. The idea is that you tell newLISP to process a list using the function you specify.

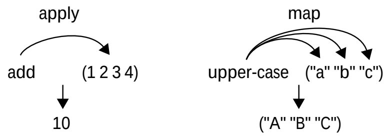


Figure 6.1 apply and map make functions work on lists

The **apply** function uses all the elements in the list as arguments to the function, and evaluates the result.

```
(apply reverse '("this is a string"))
;-> "gnirts a si siht"
```

Here, **apply** looks at the list, which in this case consists of a single string, and feeds the elements to the function as arguments. The string gets reversed. Notice that you don't have to quote the function but you do have to quote the list, because you don't want newLISP to evaluate it before the designated function gets a chance to consume it.

The **map** function, on the other hand, works through the list, element by element, like a sergeant major inspecting a row of soldiers, and applies the function to each element in turn, using the element as the argument. However, **map** remembers the result of each evaluation as it goes, collects them up, and returns them in a new list.

So **map** looks like a control-flow word, a bit like **dolist**, whereas **apply** seems to be way of controlling the newLISP list evaluation process from within your program, calling a function when you want it called, not just as part of the normal evaluation process.

If we adapt the previous example for **map**, it gives a similar result, although the result is a list rather than just a string:

```
(map reverse '("this is a string"))
;-> ("gnirts a si siht")
```

Because we've used a list with only one element, the result is almost identical to the **apply** example, although notice that **map** returns a list whereas, in this example, **apply** doesn't.

```
(apply reverse '("this is a string"))
;-> "gnirts a si siht"
```

The string has been extracted from the list, reversed, and then stored in another list created by **map**.

In this example:

```
(map reverse '("this" "is" "a" "list" "of" "strings"))
;-> ("siht" "si" "a" "tsil" "fo" "sgnirts")
```

you can clearly see that **map** has applied **reverse** to each element of the list in turn, and returned a list of the resulting strings.

Write one in terms of the other?

To illustrate the relationship between these two functions, here is **map** defined in terms of **apply**:

```
(define (my-map f l , r)
 ; declare a local variable r to hold the results
 (dolist (e l)
   (push (apply f (list e)) r -1))
 r)
```

We're pushing the result of applying a function `f` to each list item to the end of a temporary list, and then returning the list at the end, just as **map** would do. This works, at least for simple expressions:

```
(my-map explode '("this is a string"))
;-> (("t" "h" "i" "s" " " "i" "s" " " "a" " " "s" "t" "r" "i" "n" "g"))

(map explode '("this is a string"))
;-> (("t" "h" "i" "s" " " "i" "s" " " "a" " " "s" "t" "r" "i" "n" "g"))
```

Notice that in our function definition we converted the list element to a list (`(list e)`) first, because the **apply** function expects the arguments to be in a list.

This example illustrates why **map** is useful. It's an easy way to transform all the elements of a list without the hassle of working through them element by element using a **dolist** structure.

More tricks

Both **map** and **apply** have more tricks up their sleeves. **map** can traverse more than one list at the same time. If you supply two or more lists, newLISP interleaves the elements of each list together, starting with the first elements of each list, and then passes them in order as arguments to the function:

```
(map append '("cats " "dogs " "birds ") '("miaow" "bark" "tweet"))
;-> ("cats miaow" "dogs bark" "birds tweet")
```

We've taken the first element of each list and passed them to **append**, followed by the second element of each list, and so on. I like this weaving together of strands – like knitting with lists. Or like doing up a zip.

apply has a trick too. A third argument indicates how many of the preceding list's arguments the function should use. So if a function takes two arguments, and you supply three or more, **apply** comes back and makes another attempt, using the result of the first application and another argument. It continues eating its way through the list until all the arguments are used up.

To see this in action, let's first define a function that takes two arguments and compares their lengths:

```
(define (longest s1 s2)
 (println s1 " is longest so far, is " s2 " longer?") ; feedback
 (if (>= (length s1) (length s2))                       ; compare
     s1
     s2))
```

Now you can apply this function to a list of strings, using the third argument to tell **apply** to use up the arguments two strings at a time:

```
(apply longest '("green" "purple" "violet" "yellow" "orange"
"black" "white" "pink" "red" "turquoise" "cerise" "scarlet"
"lilac" "grey" "blue") 2)
```

```
green is longest so far, is purple longer?
purple is longest so far, is violet longer?
purple is longest so far, is yellow longer?
purple is longest so far, is orange longer?
purple is longest so far, is black longer?
purple is longest so far, is white longer?
purple is longest so far, is pink longer?
purple is longest so far, is red longer?
purple is longest so far, is turquoise longer?
turquoise is longest so far, is cerise longer?
turquoise is longest so far, is scarlet longer?
turquoise is longest so far, is lilac longer?
turquoise is longest so far, is grey longer?
turquoise is longest so far, is blue longer?
turquoise
```

It's like walking along the beach and finding a pebble, and holding on to it until an even better one turns up.

apply also gives you a way of working through a list and applying a function to each pair of items:

```
(apply (fn (x y)
    (println {x is } x {, y is } y)) (sequence 0 10) 2)
```

```
x is 0, y is 1
x is 1, y is 2
x is 2, y is 3
x is 3, y is 4
x is 4, y is 5
x is 5, y is 6
x is 6, y is 7
x is 7, y is 8
x is 8, y is 9
x is 9, y is 10
```

What's happening here is that the value returned by the **println** function is the second member of the pair, and this becomes the value of the first element of the next pair.

Lispiness

This thing about passing around the names of functions as if they were bits of data is very characteristic of newLISP, and it's very useful. You will find many uses for it, sometimes using functions that you don't think will be useful with **map**. Here, for example, is **set** working hard under the control of **map**:

```
(map set '(a b) '(1 2))
;-> a is 1, b is 2
```

```
(map set '(a b) (list b a))
;-> a is 2, b is 1
```

This construction gives you another way to assign values to symbols in parallel, rather than sequentially. (You can use **swap** as well.)

Some uses of **map** are simple:

```
(map char (explode "hi there"))
;-> (104 105 32 116 104 101 114 101)

(map (fn (h) (format "%02x" h)) (sequence 0 15))
;-> ("00" "01" "02" "03" "04" "05" "06" "07" "08" "09" "0a" "0b" "0c"
 "0d" "0e" "0f")
```

Others can become quite complex. For example, given a string of data in this form, stored in a symbol `image-data`:

```
("/Users/me/graphics/file1.jpg" "  pixelHeight: 978" "  pixelWidth:
1181")
```

the numbers can be extracted with:

```
(map 'set '(height width) (map int (map last (map parse (rest
image-data)))))
```

currying

Some of the built-in newLISP functions do things with other functions. An example is **curry**, which creates a copy of a two-argument function and creates a single-argument version with a pre-determined first argument. So if a function `f1` was often called like this:

```
(f1 arg1 arg2)
```

you can use **curry** to make a new function `f2` that has a ready-to-use built-in `arg1`:

```
(set 'f2 (curry f1 arg1))
```

now you can forget about that first argument, and just supply the second one to `f2`:

```
(f2 arg2)
```

Why is this useful? Consider the **dup** function which often gets used to insert multiple blank spaces:

```
(dup { } 10)
```

Using **curry**, you can create a new function called, say, `blank`, that's a special version of **dup** that always gets called with a blank space as the string:

```
(set 'blank (curry dup { }))
```

Now you can use `(blank n)`:

```
(blank 10)
;->           ; 10 spaces
```

curry can be useful for creating temporary or anonymous functions with **map**:

```
(map (curry pow 2) (sequence 1 10))
;-> (2 4 8 16 32 64 128 256 512 1024)

(map (fn (x) (pow 2 x)) (sequence 1 10)) ; equivalent
;-> (2 4 8 16 32 64 128 256 512 1024)
```

7 Introducing contexts

We all like to organize our stuff into separate areas or compartments. Chefs keep their fish, meat, and dessert areas separate, electronics engineers keep their power supplies away from their radio frequency and audio stages, and newLISP programmers use contexts to organize their code.

What is a context?

A newLISP context provides a named container for symbols. Symbols in different contexts can have the same name without clashing. So, for example, in one context I can define the symbol called `meaning-of-life` to have the value 42, but, in another context, the identically-named symbol could have the value ‘dna-propagation’, and, in yet another, ‘worship-of-deity’.

Unless you specifically choose to create and/or switch contexts, all your newLISP work is carried out in the default context, called MAIN. So far in this document, when new symbols have been created, they’ve been added to the MAIN context.

Contexts are very versatile – you can use them for dictionaries, or software objects, or super-functions, depending on the task in hand.

Contexts: the basics

The **context** function can be used for a number of different tasks:

- to create a new context
- to switch from one context to another
- to retrieve the value of an existing symbol in a context
- to see what context you’re in
- to create a new symbol in a context and assign a value to it

newLISP can usually read your mind, and knows what you want to do, depending on how you use the **context** function. For example:

```
(context 'Test)
```

creates a new context called Test, as you might expect. If you type this in interactively, you’ll see that newLISP changes the prompt to tell you that you’re now working in another context:

```
> (context 'Test)
Test
Test>
```

And you can switch between contexts freely:

```
> (context MAIN)
MAIN
> (context Test)
Test
Test>
```

Used on its own, it just tells you where you are:

```
> (context)
MAIN
>
```

Once a context exists, you don't have to quote the name (but you can if you like). Notice that I've used an upper-case letter for my context name. This is not compulsory, just a convention.

A context contains symbols and their values. There are various ways to create a symbol and give it a value.

```
> (context 'Doyle "villain" "moriarty")
"moriarty"
>
```

This creates a new context - notice the quote, because newLISP hasn't seen this before - and a new symbol called "villain", with a value of "Moriarty", but stays in the MAIN context. If the context already exists, you can omit the quote:

```
> (context Doyle "hero" "holmes")
"holmes"
>
```

To obtain the value of a symbol, you can do this:

```
> (context Doyle "hero")
"holmes"
>
```

or, if you're using the console, this step by step approach:

```
> (context Doyle)
Doyle
Doyle> hero
"holmes"
Doyle>
```

or, from the MAIN context:

```
> Doyle:hero
;-> "holmes"
```

The full address of a symbol is the context name, followed by a colon (:), followed by the symbol name. Always use the full address if you're in another context.

To see all the symbols inside a context, use **symbols** to produce a list:

```
(symbols Doyle)
;-> (Doyle:hero Doyle:period Doyle:villain)
```

or, if you're already inside the Doyle context:

```
(symbols)
;-> (hero period villain)
```

You can use this list of symbols in the usual way, such as stepping through it with **dolist**.

```
(dolist (s (symbols Doyle))
 (println s))
```

```
Doyle:hero
Doyle:period
Doyle:villain
```

To see the values of each symbol, use **eval** to find its value, and **name** to return just the symbol's name.

```
(dolist (s (symbols Doyle))
 (println (name s) " is " (eval s)))
```

```
hero is Holmes
period is Victorian
villain is Moriarty
```

There's a more efficient (slightly faster) technique for looping through symbols in a context. Use the **dotree** function:

```
(dotree (s Doyle)
 (println (name s) " is " (eval s)))
```

```
hero is Holmes
period is Victorian
villain is Moriarty
```

Creating contexts implicitly

As well as explicitly creating contexts with **context**, you can have newLISP create contexts for you automatically. For example:

```
(define (C:greeting)
  (println "greetings from context " (context)))
(C:greeting)
```

```
greetings from context C
```

Here, newLISP has created a new context C and a function called greeting in that context. You can create symbols this way too:

```
(define D:greeting "this is the greeting string of context D")
(println D:greeting)
```

```
this is the greeting string of context D
```

In both these examples, notice that you stayed in the MAIN context.

The following code creates a new context `L` containing a new list called `ls` that contains strings:

```
(set 'L:ls '("this" "is" "a" "list" "of" "strings"))
;-> ("this" "is" "a" "list" "of" "strings")
```

Functions in context

Contexts can contain functions too. To create a function in a context other than MAIN, either do this:

```
(context Doyle)                          ; switch to existing context
(define (hello-world)                    ; define a local function
 (println "Hello World"))
```

or do this

```
(context MAIN)                           ; stay in MAIN
(define (Doyle:hello-world)              ; define function in context
 (println "Hello World"))
```

This second syntax lets you create both the context and the function inside the context, while remaining safely in the MAIN context all the time.

```
(define (Moriarty:helloworld)
 (println "(evil laugh) Hello World"))
```

You don't have to quote the new context name here because we're using **define**, and **define** (by definition) isn't expecting the name of an existing symbol.

To use functions while you're in another context, remember to call them using this `context:function` syntax.

The default function

If a symbol in a context has the same name as the context, it's known as the default function (it can be either a function or a symbol containing a list or a string). For example, here is a context called Evens, and it contains a symbol called Evens:

```
(define Evens:Evens (sequence 0 30 2))
;-> (0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30)
Evens:Evens
;-> (0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30)
```

And here is a function called Double in a context called Double:

```
(define (Double:Double x)
   (mul x 2))
```

So Evens and Double are the default functions in their contexts. Although they're called the default 'functions', the symbol with the same name as the context can be a function or a symbol containing a list or a string.

There are lots of good things about default functions. If the default function has the same name as the context, it is evaluated whenever you use the name of the context in expressions, unless newLISP is expecting the name of a context. For example, although you can always switch to the Evens context by using the **context** function in the usual way:

```
> (context Evens)
Evens
Evens> (context MAIN)
MAIN
>
```

you can use Evens as a list (because Evens:Evens is a list):

```
(reverse Evens)
;-> (30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0)
```

You can use the default function without supplying its full address. Similarly, you can use the Double function as an ordinary function without supplying the full colon-separated address:

```
> (Double 3)
6
```

You can still switch to the Double context in the usual way:

```
> (context Double)
Double
Double>
```

newLISP is smart enough to be able to work out from your code whether to use the default function or the context itself.

Passing parameters by reference

There is important differences between default functions and their more ordinary siblings. newLISP is using a reference to the data rather than a copy of the data. For larger lists and strings, references are much quicker for newLISP to pass around between functions, so your code will be faster if you can use default functions as parameters to functions.

Also, and as a consequence, functions change the contents of any default functions passed as reference parameters. Ordinary symbols are copied when passed as parameters. Observe the following code:

```
(define Evens:Evens (sequence 0 30 2))   ; a default symbol
;-> (0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30)

(define odds (sequence 1 31 2))          ; ordinary symbol
;-> (1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31)

; this function reverses a list
(define (my-reverse lst)
  (reverse lst))

(my-reverse Evens)                                            ;
default symbol as parameter
;-> (30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0)

(my-reverse odds)                        ; ordinary symbol as parameter
;-> (31 29 27 25 23 21 19 17 15 13 11 9 7 5 3 1)
```

So far, they look as if they're behaving identically. But now inspect the original symbols:

```
> Evens:Evens
(30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0)
> odds
(1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31)
```

The list that was passed as a default function to a function was changed by the function, whereas the ordinary list parameter was copied, as usual.

Functions with a memory

In the following example, we create a context called `Output`, and a default function inside it, also called `Output`. This function prints its arguments, and increments a counter by the number of characters output. Because the default function has the same name as the context, it is executed whenever we use the name of the context in other expressions.

Inside this function, the value of a variable counter (inside the context Output) is incremented if it exists, or created and initialized if it doesn't. Then the function's main task – the printing of the arguments – is done. The counter symbol keeps count of how many characters were output.

```
(define (Output:Output)                 ; define the default function
 (if Output:counter
  (inc 'Output:counter (length (string (args))))
  (set 'Output:counter 0))
 (map print (args))
 (println))

(dotimes (x 90)
 (Output                                ; use context name as a function
 "the square root of " x " is " (sqrt x)))

(Output "you used " Output:counter " characters")
```

```
the square root of 0 is 0
the square root of 1 is 1
the square root of 2 is 1.414213562
the square root of 3 is 1.732050808
the square root of 4 is 2
the square root of 5 is 2.236067977
the square root of 6 is 2.449489743
the square root of 7 is 2.645751311
the square root of 8 is 2.828427125
the square root of 9 is 3
...
the square root of 88 is 9.38083152
the square root of 89 is 9.433981132
you used 3895 characters
```

The Output function effectively remembers how much work it's done since it was first created. It could even append that information to a log file.

> Think of the possibilities. You could log the usage of all your functions, and bill users according to how often they use your functions.

You can override the built-in **println** function so that it uses this code instead when it's called. See "**On your own terms**" on page **155**.

Dictionaries and tables

A common use for a context is a dictionary: a ordered set of key/value pairs, arranged so that you can obtain the current value of a key, or add a new key/value pair. newLISP makes creating dictionaries easy. To illustrate, I'll enlist the help of the great detective, Sherlock Holmes. First, I downloaded Sir Arthur Conan Doyle's 'The Sign of Four' from Project Gutenberg, then I loaded the file as a list of words.

```
(set 'file "/Users/me/Sherlock Holmes/sign-of-four.txt")
(set 'data (clean empty? (parse (read-file file) "\\W" 0)))
```

Next, I define an empty dictionary:

```
(define Doyle:Doyle)
```

This defines the Doyle context and the default function, but leaves the default function uninitialized. If the default function is left empty, you can use the following expressions to build and examine a dictionary:

- (Doyle key value) – set key to value
- (Doyle key) – get value of key
- (Doyle key nil) – delete key

The code is like this:

```
(dolist (word data)
   (set 'lc-word (lower-case word))
   (if (set 'tally (Doyle lc-word))
       (Doyle lc-word (inc 'tally))
       (Doyle lc-word 1)))
```

Each word is added to the dictionary, and the value (the number of occurrences) increased by 1. Inside the context, the names of the keys have been prefixed with an underscore ("_"). This is so that nobody gets confused between the names of keys and reserved words.

There are various ways you can browse the dictionary. To look at individual symbols:

```
(Doyle "baker")
;-> 10
(Doyle "street"))
;-> 26
```

To look at the symbols as they are stored in a context, work through the context evaluating each symbol, using **dotree**:

```
(dotree (wd Doyle)
   (println wd { } (eval wd)))
```

```
Doyle:Doyle nil
Doyle:_1 1
Doyle:_1857 1
Doyle:_1871 1
Doyle:_1878 2
Doyle:_1882 3
Doyle:_221b 1
...
Doyle:_your 107
Doyle:_yours 7
Doyle:_yourself 9
Doyle:_yourselves 2
Doyle:_youth 3
Doyle:_zigzag 1
Doyle:_zum 2
```

To see the dictionary as an association list, use the dictionary name on its own:

```
(Doyle)
```

```
(("1" 1)
 ("1857" 1)
 ("1871" 1)
 ("1878" 2)
 ("1882" 3)
 ("221b" 1)
 ...
 ("you" 543)
 ("young" 19)
 ("your" 107)
```

```
  ("yours" 7)
  ("yourself" 9)
  ("yourselves" 2)
  ("youth" 3)
  ("zigzag" 1)
  ("zum" 2))
```

This is a standard association list, which you can access using the functions described in the Lists chapter (see "**Association lists**" on page **58**). For example, to find all words that occur 20 times, use **find-all**:

```
(find-all '(? 20) (Doyle)  (println $0))
```

```
("friends" 20)
("gone" 20)
("seemed" 20)
("those" 20)
("turned" 20)
("went" 20)
```

Creating dictionaries from association lists

You can also add new entries to dictionaries, or modify existing entries, using data in the form of an association list:

```
(Doyle '(("laser" 0) ("radar" 0)))
```

Saving and loading contexts

If you want to use the dictionary again, you can save the context in a file:

```
(save "/Users/me/Sherlock Holmes/doyle-context.lsp" 'Doyle)
```

This collection of data, wrapped up in a context called Doyle, can be quickly loaded by another script or newLISP session using:

```
(load "/Users/me/Sherlock Holmes/doyle-context.lsp")
```

and newLISP will automatically recreate all the symbols in the Doyle context, switching back to the MAIN (default) context when done.

Using newLISP modules

Contexts are used as containers for software modules because they provide lexically-separated namespaces. The modules supplied with the newLISP installation usually define a context that contains a set of functions handling tasks in a specific area.

Here's an example. The POP3 module lets you check POP3 email accounts. You first load the module:

```
(load "/usr/share/newlisp/modules/pop3.lsp")
```

The module has now been added to the newLISP system. You can switch to the context:

```
(context POP3)
```

and call the functions in the context. For example, to check your email, use the `get-mail-status` function, supplying user name, password, and POP3 server name:

```
(get-mail-status "someone@example.com" "secret" "mail.example.com")
;-> (3 197465 37)
; (totalMessages, totalBytes, lastRead)
```

If you don't switch to the context, you can still call the same function by supplying the full address:

```
(POP3:get-mail-status "someone@example.com" "secret" "mail.example.com")
```

Scoping

You've already seen the way newLISP dynamically finds the 'current' version of a symbol (see "**Scope**" on page **32**). However, when you use contexts, you can employ a different approach, which programmers call lexical scoping. With lexical scoping, you can explicitly control which symbol is used, rather than rely on newLISP to keep track of similarly-named symbols for you automatically.

In the following code, the `width` symbol is defined inside the Right-just context.

```
(context 'Right-just)
(set 'width 30)
(define (Right-just:Right-just str)
  (slice (string (dup " " width) str) (* width -1)))

(context MAIN)
(set 'width 0)                          ; this is a red herring
(dolist (w (symbols))
  (println (Right-just w)))
```

```
                             !
                            !=
                             $
                            $0
                            $1
                           ...
                    write-line
                     xml-error
                     xml-parse
                 xml-type-tags
                         zero?
                             |
                             ~
```

The second `(set 'width ...)` line is a red herring: changing this here makes no difference at all, because the symbol which is actually used by the right-justification function is inside a different context.

You can still reach inside the Right-just context to set the width:

```
(set 'Right-just:width 15)
```

There's been much discussion about the benefits and disadvantages of the two approaches. Whatever you choose, make sure you know where the symbols are going to get their values from when the code runs. For example:

```
(define (f y)
  (+ y x))
```

Here, `y` is the first argument to the function, and is independent of any other `y`. But what about `x`? Is it a global symbol, or has the value been defined in some other function that has just called `f`? Or perhaps it has no value at all!

It's best to avoid using these 'free' symbols, and to use local variables (defined with **let** or **local**) wherever possible. Perhaps you can adopt a convention such as putting asterisks around a global symbol.

Objects

More has been written about object-oriented programming (OOP) than you could possibly read in one lifetime, so this section is just a quick glance at the subject. newLISP is agile enough to enable more than one style of OOP, and you can easily find references to these on the web, together with discussions about the merits of each.

For this introduction, I'll briefly outline just one of these styles: 'FOOP', or Functional Object-Oriented Programming.

FOOP in a nutshell

In FOOP, each object is stored as a list. Class methods and class properties (ie functions and symbols that apply to every object of that class) are stored in a context.

Objects are stored in lists because lists are fundamental to newLISPṪhe first item in an object list is a symbol identifying the class of the object; the remaining items are the values that describe the properties of an object.

All the objects in a class share the same properties but those properties can have different values. The class can also have properties that are shared between all objects in the class; these are the class properties. Functions stored in the class context provide the various methods for managing the objects and processing the data they hold.

To illustrate these ideas, consider the following code that works with times and dates. It builds on top of the basic date and time functions provided by newLISP (see "**Working with dates and times**" on page **123**). A moment in time is represented as a 'time object'. An object holds two values: the number of seconds that have elapsed since the beginning of 1970, and the time zone offset, in minutes west of Greenwich. So the list to represent a typical time object looks like this:

```
(Time 1219568914 0)
```

where Time is a symbol representing the class name, and the two numbers are the values of this particular time (these numbers represent a time around 10 in the morning on Sunday August 24 2008, somewhere in England).

The code required to build this object model starts with a default function to create a new context. This makes a new time object and gives it an initial value. The default function acts as the *constructor*:

```
(define (Time:Time (t (date-value)) (zone 0))
    (list Time t zone))
```

It's a default function for the Time context that builds a list with the class name in the first position, and two more integers to represent the time. If values are not supplied, they default to the current time and zero offset. You use the constructor like this:

```
(set 'time-now (Time))
;-> (Time 1220044684 0)
(set 'my-birthday (Time (date-value 2008 5 26)))
;-> (Time 1211760000 0)
(set 'christmas-day (Time (date-value 2008 12 25)))
;-> (Time 1230163200 0)
```

Next, you can define other functions to inspect and manage time objects. All these functions live together in the context. They can extract the seconds and zone information by picking them from the object list passed as an argument. So `(t 1)` gets the seconds and `(t 2)` gets the time zone offset. Here are a few obvious class functions:

```
(define (Time:show t)
    (date (t 1) (t 2)))

(define (Time:days-between t1 t2)
; return difference in days between two times
    (div (abs (- (t1 1) (t2 1))) (* 24 60 60)))

(define (Time:get-hours t)
; return hours
    (int (date (t 1) (t 2) {%H})))

(define (Time:get-day t)
; return day of week
    (date (t 1) (t 2) {%A}))

(define (Time:leap-year? t)
  (let ((year (int (date (t 1) (t 2) {%Y}))))
   (and (= 0 (% year 4))
        (or (!= 0 (% year 100))  (= 0 (% year 400))))))
```

These functions could be called in the usual way, by providing the context prefix:

```
(Time:show christmas-day)
;-> Thu Dec 25 00:00:00 2008
(Time:show my-birthday)
;-> Mon May 26 01:00:00 2008
```

However, it's also possible to provide just the colon as a prefix to the function, and omit the context specifier altogether:

```
(:show christmas-day)
;-> Thu Dec 25 00:00:00 2008
(:show my-birthday)
;-> Mon May 26 01:00:00 2008
```

This technique allows newLISP to provide an important feature beloved by object-oriented programmers: *polymorphism*.

Polymorphism

Let's add another class which works on durations - the gap between two time objects - measured in days.

```
(define (Duration:Duration (d 0))
   (list Duration d))

(define (Duration:show d)
   (string (d 1) " days "))
```

There's a new class constructor `Duration:Duration` for making new duration objects, and a simple `show` function. They can be used with the time objects like this:

```
; define two times
(set 'time-now (Time) 'christmas-day (Time (date-value 2008 12 25)))
; find days between them
(:show (Duration (:days-between time-now christmas-day)))
;-> "122.1331713 days "
```

Compare that `:show` function call with the `:show` in the previous section:

```
(:show christmas-day)
;-> Thu Dec 25 00:00:00 2008
```

You can see that newLISP is choosing which version of the `show` function to evaluate, according to the class of `:show`'s parameter. Because `christmas-day` is a Time object, newLISP evaluates `Time:show`. But when the argument is a Duration object, it evaluates `Duration:show`.

The idea of this facility is that you can use a function on various types of object: you perhaps don't need to know what class of object you're dealing with. With this polymorphism, you can apply the `show` function to a list of objects of differing types, and newLISP selects the appropriate one each time:

```
(map (curry :show) (list my-birthday (Duration (:days-between time-now
christmas-day))))
;-> ("Mon May 26 01:00:00 2008" "123.1266898 days ")
```

(The syntax is slightly devious because the colon operator has to be applied as well as the function.)

Modifying objects

This particular style of OOP is called FOOP because it's considered to be functional. Here, the term 'functional' refers to the style of programming encouraged by newLISP which emphasizes the evaluation of functions and avoids state and mutable data. As you've seen, many newLISP functions return copies of lists rather than modify the originals. There are, though, a small number of functions that are called 'destructive', and these are considered to be less purely functional. But FOOP doesn't provide for destructive object methods, so can be considered more functional.

A key point to notice about FOOP is that objects are immutable; they can't be modified by class functions. For example, here's a function for the Time class that adds a given number of days to a time object:

```
(define (Time:adjust-days t n)
  (list Time (+ (* 24 60 60 n) (t 1)) (t 2)))
```

When this is called, it returns a modified copy of the object; the original is unchanged:

```
> (set 'christmas-day (Time (date-value 2008 12 25)))
(Time 1230163200 0)
> (:show (:adjust-days christmas-day 3))
Sun Dec 28 00:00:00 2008
"Sun Dec 28 00:00:00 2008"
> (:show christmas-day)
Thu Dec 25 00:00:00 2008
"Thu Dec 25 00:00:00 2008"
>
```

The original date of the `christmas-day` object didn't change, although the `:adjust-days` function returned a modified copy adjusted by 3 days.

In other words, to make changes to objects, use the familiar newLISP approach of using the value returned by a function:

```
(set 'moved-christmas-day (:adjust-days christmas-day 3))
```

`moved-christmas-day` is a new time object containing the modified date, and the original object is left unchanged.

8 Macros

Introducing macros

We've covered the basics of newLISP but there are plenty of powerful features left to discover. Once you've grasped the main rules of the language, you can decide which of the more advanced tools you want to add. One feature that you might want to explore is newLISP's provision of macros.

> Strictly speaking, newLISP's macros are 'fexprs', not macros. But this is probably of interest only if you're an old-school LISP user.

A macro is a special type of function that you can use to change the way your code is processed by newLISP. For example, you can create new types of control functions, such as your own version of **if** or **case**. With macros, you can start to make newLISP work exactly the way you want it to.

When do things get evaluated

To understand macros, let's jump back to the beginning of this introduction, and look again at the way this expression is evaluated:

```
(* (+ 1 2) (+ 3 4))
;-> (* 3 7)
;-> 21
```

Notice that the ***** function doesn't see the **+** expressions at all, only their results. newLISP has enthusiastically evaluated the addition expressions first, before handing just the results to the multiplication function. This is usually what you want, but there are times when you don't want every expression evaluated immediately. Consider the operation of the built-in function **if**:

```
(set 'x 1)
(if (< x 0) (exit))
```

If the test returns nil, the (exit) function isn't evaluated. This is probably what you want.

Now suppose that you want to define your own version of the 'if' function. It should be easy:

```
(define (my-if test true-action false-action)
   (if test true-action false-action))

(my-if (> 3 2) (println "yes it is" ) (exit))
```

But this won't work. If the comparison returns true, newLISP prints a message and then quits. If the comparison returns false, newLISP still quits, although without printing a message. The problem is simply that `(exit)` is evaluated before the `my-if` function is called, even when you don't want it to be. For ordinary functions, expressions in arguments are evaluated first.

Macros are similar to functions, but they let you control when – and if – arguments are evaluated. You use the **define-macro** function to define macros, in the same way that you define functions using **define**. Both these defining functions let you make your own functions that accept arguments. The important difference is that, with **define**, arguments are evaluated before the function runs. When you call a macro function defined with **define-macro**, the arguments are passed to the definition in their raw and unevaluated form.

A macro version of the ‘my-if’ function looks like this:

```
(define-macro (my-if test true-action false-action)
   (if (eval test) (eval true-action) (eval false-action)))


(my-if (> 3 2) (println "yes it is" ) (exit))
```

```
"yes it is"
```

The test and action arguments aren’t evaluated immediately, but only when you want them to be, using **eval**. This ability to postpone evaluation gives you the ability to write your own control structures and add powerful new forms to the language.

Tools for building macros

newLISP provides a number of useful tools for building macros. As well as **define-macro** and **eval**, there’s **letex**, which gives you way of expanding local symbols into an expression before evaluating it, and **args**, which holds all the arguments that are passed to your macro.

Symbol confusion

One problem to be aware of when you’re writing macros is the way that symbols in macros can be confused with symbols in the code that calls the macro. Here’s a simple macro which adds a new looping construct to the language that combines **dolist** and **do-while**. A loop variable steps through a list while a condition is true:

```
(define-macro (dolist-while)
  (letex (var (args 0 0)
    lst (args 0 1)
    cnd (args 0 2)
    body (cons 'begin (1 (args))))
  (let (y)
  (catch (dolist (var lst)
       (if (set 'y cnd) body (throw y)))))))
```

It’s called like this:

```
(dolist-while (x (sequence 20 0) (> x 10))
  (println {x is } (dec 'x 1)))
```

```
x is 19
x is 18
x is 17
x is 16
x is 15
x is 14
x is 13
x is 12
x is 11
x is 10
```

And it appears to work well. But there’s a problem: you can’t use a symbol called `y` as the loop variable, although you can use `x` or anything else. Put a `(println y)` statement in the loop to see why:

```
(dolist-while (x (sequence 20 0) (> x 10))
    (println {x is } (dec 'x 1))
    (println {y is }  y))
```

```
x is 19
y is true
x is 18
y is true
x is 17
y is true
```

If you try to use y, it won't work:

```
(dolist-while (y (sequence 20 0) (> y 10))
  (println {y is } (dec 'y 1)))
```

```
y is
value expected in function dec : y
```

The problem is that y is used by the macro, even though it's in its own let expression. It appears as a true/nil value, so it can't be decremented. To fix this problem, enclose the macro inside a context, and make the macro the default function in that context:

```
(context 'dolist-while)
(define-macro (dolist-while:dolist-while)
      (letex (var (args 0 0)
        lst (args 0 1)
        cnd (args 0 2)
        body (cons 'begin (1 (args))))
      (let (y)
      (catch (dolist (var lst)
           (if (set 'y cnd) body (throw y)))))))
(context MAIN)
```

This can be used in the same way, but without any problems:

```
(dolist-while (y (sequence 20 0) (> y 10))
      (println {y is } (dec 'y 1)))
```

```
y is 19
y is 18
y is 17
```

Other ideas for macros

newLISP users find different reasons to use macros. Here are a couple of macro definitions I've found on the newLISP user forums to start you off. Here's a version of **case**, called ecase (evaluated-case) that really does evaluate the tests:

```
(define-macro (ecase _v)
 (eval (append
   (list 'case _v)
   (map (fn (_i) (set-nth 0 _i (eval (_i 0))))
    (args)))))

(define (test n)
 (ecase n
  ((/ 4 4)     (println "n was 1"))
  ((- 12 10)   (println "n was 2"))))

(set 'n 2)
(test n)
```

```
n was 2
```

You can see that the division `(/ 4 4)` was evaluated. It wouldn't have been with the standard version of **case**.

Here's a macro that creates functions:

```
(define-macro (create-functions group-name)
 (letex
  ((f1 (sym (append (name group-name) "1")))
   (f2 (sym (append (name group-name) "2"))))
 (define (f1 arg) (+ arg 1))
 (define (f2 arg) (+ arg 2))))

(create-functions foo)    ; this creates two functions starting with
'foo'

(foo1 10)
;-> 11
(foo2 10)
;-> 12

(create-functions bar)    ; and this creates two functions starting with
'bar'

(bar1 12)
;-> 13
(bar2 12)
;-> 14
```

A tracer macro

The following code changes the operation of newLISP so that every function defined using **define** will, when evaluated, add the function name and details of its arguments to a log file. When you run a script, the log file will contain a record of the functions and arguments that were evaluated.

```
(context 'tracer)
(define-macro (tracer:tracer farg)
  (set (farg 0)
    (letex (func   (farg 0)
            arg    (rest farg)
            arg-p  (cons 'list (map (fn (x) (if (list? x) (first x) x))
                   (rest farg)))
            body   (cons 'begin (args)))
           (lambda
               arg
               (append-file
                     (string (env "HOME") "/trace.log")
                     (string 'func { } arg-p "\n"))
               body))))

(context MAIN)
(constant (global 'newLISP-define) define)
; redefine the built-in define:
(constant (global 'define) tracer)
```

To run a script with this simple tracer, load the context before you run:

```
(load {tracer.lsp})
```

The log file generated contains a list of every function that was called, and the arguments it received:

```
Time:Time (1211760000 0)
Time:Time (1230163200 0)
Time:Time (1219686599 0)
show ((Time 1211760000 0))
show ((Time 1230163200 0))
get-hours ((Time 1219686599 0))
get-day ((Time 1219686599 0))
days-between ((Time 1219686599 0) (Time 1230163200 0))
leap-year? ((Time 1211760000 0))
adjust-days ((Time 1230163200 0) 3)
show ((Time 1230422400 0))
Time:Time (1219686599 0)
days-between ((Time 1219686599 0) (Time 1230422400 0))
Duration:Duration (124.256956)
period-to-string ((Duration 124.256956))
days-between ((Time 1219686599 0) (Time 1230422400 0))
Duration:Duration (124.256956)
Time:print ((Time 1211760000 0))
Time:string ((Time 1211760000 0))
Duration:print ((Duration 124.256956))
Duration:string ((Duration 124.256956))
```

9 Working with numbers

If you work with numbers, you'll be pleased to know that newLISP includes most of the basic functions that you'd expect to find, plus many more. This section is designed to help you make the best use of them and to avoid some of the minor pitfalls that you might encounter. As always, see the documentation for full details.

Integers and floating-point numbers

newLISP handles two different types of number: the integer and the floating-point number. Integers are precise, but limited in range, whereas floating-point numbers ('floats') are less precise but capable of a much wider range. There are advantages and disadvantages to each.

The largest positive integer you can have is 9 223 372 036 854 775 807, and the largest negative integer is -9 223 372 036 854 775 808. If you try adding 1 to the largest positive integer, you'll find yourself wrapping round to the largest negative integer again:

```
(set 'li 9223372036854775807)
(+ 1 li)
;-> -9223372036854775808
```

The arithmetic operators +, -, *, /, and % always return integer values. A common mistake is to forget this and use / and * without realising that they're carrying out integer arithmetic:

```
(/ 10 3)
;-> 3
```

This might not be what you were expecting!

Floating-point numbers are capable of holding bigger and smaller values than integers, and fractional values too, but they only keep the 15 or 16 most important digits (ie the digits at the left of the number, with the highest place values).

The philosophy of a floating-point number is 'that's close enough', rather than 'that's the exact value'.

Suppose you try to define a symbol PI to store the value of π to 50 decimal places:

```
(constant 'PI 3.14159265358979323846264338327950288419716939937510)
;-> 3.141592654
```

```
(println PI)
```

```
3.141592654
```

It looks like newLISP has cut about 40 digits off the right hand side! In fact about 15 or 16 digits have been stored, and 35 of the less important digits have been discarded.

How does newLISP store this number? Let's look using the **format** function:

```
(format {%1.50f} PI)
;-> "3.14159265358979311599796346854418516159057617187500"
```

But notice how the value is accurate up to 9793, but then drifts away from the more precise string you originally supplied. The numbers after 9793 are typical of the way all computers store floating-point values – it isn't newLISP being creative with your data!

The largest float you can use seems to be – on my machine, at least – about 10^{308}. Only the first 15 or so digits are stored, though, so that's mostly zeroes, and you can't really add 1 to it.

Another example of the motto of a floating-point number: 'that's close enough'!

The above comments are true for most computer languages, by the way, not just newLISP. Floating-point numbers are a compromise between convenience, speed, and accuracy. If you really need to use very large integers, see "**Bigger numbers**" on page **121**.

Integer and floating-point maths

When you're working with floating-point numbers, use the floating-point arithmetic operators **add**, **sub**, **mul**, **div**, and **mod**, rather than **+**, **-**, *****, **/**, and **%**, their integer-only equivalents:

```
(mul PI 2)
;-> 6.283185307
```

and, to see the value that newLISP is storing (because the interpreter's default output resolution is 9 or 10 digits):

```
(format {%1.16f} (mul PI 2))
;-> "6.2831853071795862"
```

If you forget to use **mul** here, and use ***** instead, the numbers after the decimal point are thrown away:

```
(format {%1.16f} (* PI 2))
;-> "6.0000000000000000"
```

Here, π was converted to 3 and then multiplied by 2.

You can re-define the familiar arithmetic operators so that they default to using floating-point routines rather than integer-only arithmetic:

```
; before
(+ 1.1 1.1)
;-> 2
```

```
(constant (global '+) add)

; after
(+ 1.1 1.1)
;-> 2.2
```

You could put these definitions in your init.lsp file to have them available for all newLISP work on your machine. The main problem you'll find is when sharing code with others, or using imported libraries. Their code might produce surprising results, or yours might!

Conversions: explicit and implicit

To convert strings into numbers, or numbers of one type into another, use the **int** and **float** functions.

The main use for these is to convert a string into a number – either an integer or a float. For example, you might be using a regular expression to extract a string of digits from a longer string:

```
(map int (find-all {\d+} {the answer is 42, not 41}))
;-> (42 41)                             ; a list of integers

(map float (find-all {\d+.*} {the value of pi is 3.14, not 1.618}))
;-> (3.14 1.618)                        ; a list of floats
```

A second argument passed to **int** specifies a default value which should be used if the conversion fails:

```
(int "x")
;-> nil
(int "x" 0)
;-> 0
```

int is a clever function that can also convert strings representing numbers in number bases other than 10 into numbers. For example, to convert a hexadecimal number in string form to a decimal number, make sure it is prefixed with `0x`, and don't use letters beyond `f`:

```
(int (string "0x" "1F"))
;-> 31
(int (string "0x" "decaff"))
;-> 14600959
```

And you can convert strings containing octal numbers by prefixing them with just a `0`:

```
(int "035")
;-> 29
```

Even if you never use octal or hexadecimal, it's worth remembering this, because one day you might, either deliberately or accidentally, write this:

```
(int "08")
```

which evaluates to 0 rather than 8 – a failed octal-decimal conversion rather than the decimal 8 that you might have expected! For this reason, it's always a good idea to specify not only a default value but also a number base whenever you use **int** on string input:

```
(int "08" 0 10)                         ; default to 0 and assume base 10
;-> 8
```

Invisible conversion and rounding

Some functions convert floating-point numbers to integers automatically. Two commonly used examples are **inc** and **dec**, which have a default behaviour that involves conversion and rounding:

```
(set 'f 2.5)
;-> 2.5

(inc 'f)
;-> 3     ; f is now 3, and an integer, not 3.5

(set 'f 2.5)
;-> 2.5

(dec 'f)
;-> 1
```

When used like this, the value of `f` is first converted to an integer (and the fractional part is discarded), then increased or decreased by 1. This is **inc**'s and **dec**'s convenient default action (you're often using **inc** with integers). If you're using floats, be sure to specify more precisely what you want by supplying an increment (or decrement) value, even if it's just 1:

```
(set 'f 2.5)
;-> 2.5

(inc 'f 1)
;-> 3.5

(dec 'f 2)
;-> 1.5
```

Many newLISP functions automatically convert integer arguments into floating-point values. This usually isn't a problem. But it's possible to lose some precision if you pass extremely large integers to functions that convert to floating-point:

```
(format {%15.15f} (add 1 9223372036854775800))
;-> "9223372036854775632.000000000000000"
```

Because the **add** function converted the very large integer to a float, a small amount of precision was lost (amounting to about 52, in this case). Close enough? If not, think carefully about how you store and manipulate numbers.

Number testing

Sometimes you will want to test whether a number is an integer or a float:

```
(set 'PI 3.141592653589793)
;-> 3.141592654

(integer? PI)
;-> nil

(float? PI)
;-> true

(number? PI)
;-> true
```

```
(zero? PI)
;-> nil
```

With **integer?** and **float?**, you're testing whether the number is stored as an integer or float, not whether the number is mathematically an integer or a floating-point value. For example, this test returns **nil**, which might surprise you:

```
(integer? (div 30 3))
;-> nil
```

It's not that the answer isn't 10 (it is), but rather that the answer is a floating-point 10, not an integer 10, because the **div** function always returns a floating-point value.

```
(integer? (int (div 30 3)))
;-> true
```

Absolute signs, from floor to ceiling

It's worth knowing that the **floor** and **ceil** functions return floating-point numbers that contain integer values. For example, if you use **floor** to round π down to the nearest integer, the result is 3, but it's stored as a float not as an integer:

```
(integer? (floor PI))
;-> nil

(floor PI)
;-> 3

(float? (ceil PI))
;-> true
```

The **abs** and **sgn** functions can also be used when testing and converting numbers. **abs** always returns a positive version of its argument, and **sgn** returns 1, 0, or -1, depending on whether the argument is positive, negative, or 0.

The **round** function rounds numbers to the nearest whole number, with floats remaining floats. You can also supply an optional additional value to round the number to a specific number of digits. Negative numbers round after the decimal point, positive numbers round before the decimal point.

```
(set 'n 1234.6789)
(for (i -6 6)
 (println (format {%4d %12.5f} i (round n i))))

  -6   1234.67890
  -5   1234.67890
  -4   1234.67890
  -3   1234.67900
  -2   1234.68000
  -1   1234.70000
   0   1235.00000
   1   1230.00000
   2   1200.00000
   3   1000.00000
   4      0.00000
   5      0.00000
   6      0.00000
```

sgn has an alternative syntax that lets you evaluate up to three different expressions depending on whether the first argument is negative, zero, or positive.

```
(for (i -5 5)
 (sgn i (println "below 0") (println "0") (println "above 0")))
```

```
below 0
below 0
below 0
below 0
below 0
0
above 0
above 0
above 0
above 0
above 0
```

Number formatting

To convert numbers into strings, use the **string** and **format** functions:

```
(reverse (string PI))
;-> "456395141.3"
```

Both **string** and **println** use only the first 10 or so digits, even though more (up to 15 or 16) are stored internally.

Use **format** to output numbers with more control:

```
(format {%1.15f} PI)
;-> "3.141592653589793"
```

The **format** specification string uses the widely-adopted `printf`-style formatting. Remember too that you can use the results of the **format** function:

```
(string "the value of pi is " (format {%1.15f} PI))
;-> "the value of pi is 3.141592653589793"
```

The **format** function lets you output numbers as hexadecimal strings as well:

```
(format "%x" 65535)
;-> "ffff"
```

Number utilities

Creating numbers

There are some useful functions that make creating numbers easy.

Series

sequence produces a list of numbers in an arithmetical sequence. Supply start and finish numbers (inclusive), and a step value:

```
(sequence 1 10 1.5)
;-> (1 2.5 4 5.5 7 8.5 10)
```

If you specify a step value, all the numbers are floats, otherwise they're integers.

series multiplies its first argument by its second argument a number of times. The number of repeats is specified by the third argument. This produces geometric sequences:

```
(series 1 2 20)
;-> (1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536
131072 262144 524288)
```

Every number is stored as a float.

The **normal** function returns a list of floating-point numbers with a specified mean and a standard deviation. For example, a list of 6 numbers with a mean of 10 and a standard deviation of 5 can be produced as follows:

```
(normal 10 5 6)
;-> (6.5234375 14.91210938 6.748046875 3.540039062 4.94140625 7.1484375)
```

Random numbers

rand creates a list of randomly chosen integers less than a number you supply:

```
(rand 7 20)
; 20 numbers between 0 and 6 (inclusive) or 7 (exclusive)
;-> (0 0 2 6 6 6 2 1 1 1 6 2 0 6 0 5 2 4 4 3)
```

Obviously `(rand 1)` generates a list of zeroes and isn't useful. `(rand 0)` doesn't do anything useful either, but it's been assigned the job of initializing the random number generator.

If you leave out the second number, it just generates a single random number in the range.

random generates a list of floating-point numbers multiplied by a scale factor, starting at the first argument:

```
(random 0 2 10)
; 10 numbers starting at 0 and scaled by 2
;-> (1.565273852e-05 0.2630755763 1.511210644 0.9173002638
1.065534475 0.4379183727 0.09408923243 1.357729434
1.358592812 1.869385792))
```

Randomness

Use **seed** to control the randomness of **rand** (integers), **random** (floats), **randomize** (shuffled lists), and **amb** (list elements chosen at random).

If you don't use **seed**, the same set of random numbers appears each time. This provides you with a predictable randomness – useful for debugging. When you want to simulate the randomness of the real world, seed the random number generator with a different value each time you run the script:

Without **seed**:

```
; today
(for (i 10 20)
 (print (rand i) { }))

7 1 5 10 6 2 8 0 17 18 0

; tomorrow
(for (i 10 20)
 (print (rand i) { }))

7 1 5 10 6 2 8 0 17 18 0                ; same as yesterday
```

With **seed**:

```
; today
(seed (date-value))
(for (i 10 20)
 (print (rand i) { }))
```

```
2 10 3 10 1 11 8 13 6 4 0
```

```
; tomorrow
(seed (date-value))
(for (i 10 20)
 (print (rand i) { }))
```

```
0 7 10 5 5 8 10 16 3 1 9
```

General number tools

min and **max** work as you would expect. Like many of the arithmetic operators, you can supply more than one value:

```
(max 1 2 13.2 4 2 1 4 3 2 1 0.2)
;-> 13.2
(min -1 2 17 4 2 1 43 -20 1.1 0.2)
;-> -20
```

The comparison functions allow you to supply just a single argument. If you use them with numbers, newLISP helpfully assumes that you're comparing with 0. Remember that you're using postfix notation:

```
(set 'n 3)
(> n)
;-> true, assumes test for greater than 0
(< n)
;-> nil, assumes test for less than 0

(set 'n 0)
(>= n)
;-> true
```

The **factor** function finds the factors for an integer and returns them in a list. It's a useful way of testing a number to see if it's prime:

```
(factor 5)
;-> (5)

(factor 42)
;-> (2 3 7)

(define (prime? n)
 (and
   (set 'lst (factor n))
   (= (length lst) 1)))

(for (i 0 30)
 (if (prime? i) (println i)))
```

```
2
3
5
7
11
13
17
19
23
29
```

Or you could use it to test if a number is even:

```
(find 2 (factor n))
;-> true if n is even
```

gcd finds the largest integer that exactly divides into two or more numbers:

```
(gcd 8 12 16)
;-> 4
```

Floating-point utilities

If omitted, the second argument to the **pow** function defaults to 2.

```
(pow 2)                                ; default is squared
;-> 4

(pow 2 2 2 2)                          ; (((2 squared) squared) squared)
;-> 256

(pow 2 8)                              ; 2 to the 8
;-> 256

(pow 2 3)
;-> 8

(pow 2 0.5)                            ; square root
;-> 1.414213562
```

You can also use **sqrt** to find square roots. To find cube and other roots, use **pow**:

```
(pow 8 (div 1 3))                      ; 8 to the 1/3
;-> 2
```

The **exp** function calculates e^x, where e is the mathematical constant 2.718281828, and x is the argument:

```
(exp 1)
;-> 2.71828128
```

The **log** function has two forms. If you omit the base, natural logarithms are used:

```
(log 3)                                ; natural (base e) logarithms
;-> 1.098612289
```

Or you can specify another base, such as 2 or 10:

```
(log 3 2)
;-> 1.584962501

(log 3 10)                             ; logarithm base 10
;-> 0.4771212547
```

Other mathematical functions available by default in newLISP are **fft** (fast Fourier transform), and **ifft** (inverse fast Fourier transform).

Trigonometry

All newLISP's trigonometry functions, **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, **atan2**, and the hyperbolic functions **sinh**, **cosh**, and **tanh**, work in radians. If you prefer to work in degrees, you can define alternative versions as functions:

```
(constant 'PI 3.141592653589793)

(define (rad->deg r)
 (mul r (div 180 PI)))

(define (deg->rad d)
 (mul d (div PI 180)))

(define (sind _e)
 (sin (deg->rad (eval _e))))

(define (cosd _e)
 (cos (deg->rad (eval _e))))

(define (tand _e)
 (tan (deg->rad (eval _e))))

(define (asind _e)
 (rad->deg (asin (eval _e))))

(define (atan2d _e _f)
 (rad->deg (atan2 (deg->rad (eval _e)) (deg->rad (eval _f)))))
```

and so on.

When writing equations, one approach is to build them up from the end first. For example, to convert these equations:

$$\alpha = \arctan\left\{\frac{\sin\lambda\cos\epsilon - \tan\beta\sin\epsilon}{\cos\lambda}\right\}$$

build it up in stages, like this:

```
1 (tand beta)
2 (tand beta) (sind epsilon)
3 (mul (tand beta) (sind epsilon))
4 (sind lamda)           (mul (tand beta) (sind epsilon))
5 (sind lamda) (cosd epsilon)          (mul (tand beta) (sind epsilon))
6 (sub   (mul (sind lamda) (cosd epsilon))
   (mul (tand beta) (sind epsilon)))
7 (atan2d ((sub      (mul (sind lamda) (cosd epsilon))
      (mul (tand beta)(sind epsilon)))
    (cosd lamda))
8 (set 'alpha
```

and so on...

It's often useful to line up the various expressions in your text editor:

```
(set 'right-ascension
 (atan2d
   (sub
      (mul
        (sind lamda)
        (cosd epsilon))
      (mul
        (tand beta)
        (sind epsilon)))
   (cosd lamda)))
```

If you have to convert a lot of mathematical expressions from infix to postfix notation, you might want to investigate the infix.lsp module (available from the newLISP website):

```
(load "/usr/share/newlisp/modules/infix.lsp")
(INFIX:xlate
 "(sin(lamda) * cos(epsilon)) - (cos(beta) * sin(epsilon))")
;->
(sub (mul (sin lamda) (cos epsilon)) (mul (tan beta) (sin epsilon)))
```

Arrays

newLISP provides multidimensional arrays. Arrays are very similar to lists, and you can use most of the functions that operate on lists on arrays too.

A large array can be faster than a list of similar size. The following code uses the **time** function to compare how fast arrays and lists work.

```
(for (size 200 1000)
 ; create an array
 (set 'arry (array size (randomize (sequence 0 size))))
 ; create a list
 (set 'lst (randomize (sequence 0 size)))

 (set 'array-time
 (time (dotimes (x (/ size 2))
  (nth x arry)) 50))                    ; repeat at least 50 times
                                        ; to get non-zero time!
 (set 'list-time
 (time (dotimes (x (/ size 2))
  (nth x lst)) 50))

 (println "with " size " elements: array access: "
   array-time
   "; list access: "
   list-time
   " "
   (div list-time array-time )))
```

```
with 200 elements: array access: 1; list access: 1 1
with 201 elements: array access: 1; list access: 1 1
with 202 elements: array access: 1; list access: 1 1
with 203 elements: array access: 1; list access: 1 1
...
with 992 elements: array access: 6; list access: 24 4
with 993 elements: array access: 6; list access: 23 3.833333333
with 994 elements: array access: 6; list access: 23 3.833333333
with 995 elements: array access: 6; list access: 23 3.833333333
with 996 elements: array access: 6; list access: 23 3.833333333
with 997 elements: array access: 6; list access: 24 4
with 998 elements: array access: 6; list access: 23 3.833333333
with 999 elements: array access: 6; list access: 23 3.833333333
with 1000 elements: array access: 6; list access: 25 4.166666667
```

The exact times will vary from machine to machine, but typically, with 200 elements, arrays and lists are comparable in speed. As the sizes of the list and array increase, the execution time of the **nth** accessor function increases. By the time the list and array contain 1000 elements each, the array is 3 to 4 times faster to access than the list.

To create an array, use the **array** function. You can make a new empty array, make a new one and fill it with default values, or make a new array that's an exact copy of an existing list.

```
(set 'table (array 10))                 ; new empty array
(set 'lst (randomize (sequence 0 20)))  ; new full list
(set 'arry (array (length lst) lst))    ; new array copy of a list
```

To make a new list that's a copy of an existing array, use the **array-list** function:

```
(set 'lst2 (array-list arry))           ; makes new list
```

To tell the difference between lists and arrays, you can use the **list?** and **array?** tests:

```
(array? arry)
;-> true
(list? lst)
;-> true
```

Functions available for arrays

The following general-purpose functions work equally well on arrays and lists: **first**, **last**, **rest**, **nth**, **set-nth**, **nth-set**, **sort**, **append**, and **slice**.

There are also some special functions for arrays and lists that provide matrix operations: **invert**, **det**, **multiply**, **transpose**. See "**Matrices**" on page **117**.

Arrays can be 'multi-dimensional'. For example, to create a 2 by 2 table, filled with 0s, use this:

```
(set 'arry (array 2 2 '(0)))
;-> ((0 0) (0 0))
```

The third argument to **array** supplies some initial values that newLISP will use to fill the array. newLISP uses the value as effectively as it can. So, for example, you can supply a more than sufficient initializing expression:

```
(set 'arry (array 2 2 (sequence 0 10)))
arry
;-> ((0 1) (2 3))                        ; don't need all of them
```

or just provide a hint or two:

```
(set 'arry (array 2 2 (list 1 2)))
arry
;-> ((1 2) (1 2))

(set 'arry (array 2 2 '(42)))
arry
;-> ((42 42) (42 42))
```

Getting and setting values

To get values from an array, use the **nth** function, which expects indices for each dimension of the array, followed by the name of the array:

```
(set 'size 10)
(set 'table (array size size (sequence 0 (pow size))))

(dotimes (row size)
   (dotimes (column size)
     (print (format {%3d} (nth row column table))))
   ; end of row
   (println))
```

```
 0 1 2 3 4 5 6 7 8 9
 10 11 12 13 14 15 16 17 18 19
 20 21 22 23 24 25 26 27 28 29
 30 31 32 33 34 35 36 37 38 39
 40 41 42 43 44 45 46 47 48 49
 50 51 52 53 54 55 56 57 58 59
 60 61 62 63 64 65 66 67 68 69
 70 71 72 73 74 75 76 77 78 79
 80 81 82 83 84 85 86 87 88 89
 90 91 92 93 94 95 96 97 98 99
```

(**nth** also works with lists.)

As with lists, you can use implicit addressing to get values:

```
(set ’size 10)
(set ’table (array size size (sequence 0 (pow size))))

(table 3)
;-> (30 31 32 33 34 35 36 37 38 39)     ; row 3 (0-based!)

(table 3 3)                             ; row 3 column 3
;-> 33
```

To set values, use **nth-set** and **set-nth**. The following code replaces every number that isn’t prime with 0.

```
(set ’size 10)
(set ’table (array size size (sequence 0 (pow size))))
(dotimes (row size)
  (dotimes (column size)
    (if (not (= 1 (length (factor (nth row column table)))))
        (nth-set row column table 0))))
table

((0 0 2 3 0 5 0 7 0 0)
 (0 11 0 13 0 0 0 17 0 19)
 (0 0 0 23 0 0 00 0 29)
 (0 31 0 0 0 0 0 37 0 0)
 (0 41 0 43 0 0 0 47 0 0)
 (0 0 0 53 0 0 0 0 0 59)
 (0 61 0 0 0 0 0 67 0 0)
 (0 71 0 73 0 0 0 0 0 79)
 (0 0 0 83 0 0 0 0 0 89)
 (0 0 0 0 0 0 0 97 0 0))
```

To use implicit addressing – which is slightly quicker than the **nth-set** and **set-nth** functions – use `(table row column)` instead of `(nth row column table)`.

Matrices

There are functions that treat an array or a list (with the correct structure) as a matrix.

- **invert** returns the inversion of a matrix
- **det** calculates the determinant
- **multiply** multiplies two matrices

- **mat** applies a function to two matrices or to a matrix and a number
- **transpose** returns the transposition of a matrix

transpose is also useful when used on nested lists (see "**Association lists**" on page **58**).

Statistics, financial, and modelling functions

newLISP has an extensive set of functions for financial and statistical analysis, and for simulation modelling.

- **beta** calculate the beta function
- **betai** calculate the incomplete beta function
- **binomial** calculate the binomial function
- **crit-chi2** calculate the Chi square for a given probability
- **crit-z** calculate the normal distributed Z for a given probability
- **erf** calculate the error function of a number
- **gammai** calculate the incomplete gamma function
- **gammaln** calculate the log gamma function
- **normal** produce a list of normal distributed floating point numbers
- **prob-chi2** calculate the cumulated probability of a Chi square
- **prob-z** calculate the cumulated probability of a Z value

Bayesian analysis

Statistical methods developed initially by Reverend Thomas Bayes in the 18th century have proved versatile and popular enough to have made it into the programming languages of today. In newLISP, two functions, **bayes-train** and **bayes-query**, work together to provide an easy way to calculate Bayesian probabilities for datasets.

Here's how to use the two functions to predict the likelihood that a short piece of text is written by one of two authors.

First, choose texts from the two authors, and generate datasets for each. I've chosen Oscar Wilde and Conan Doyle.

```
(set 'doyle-data
 (parse (lower-case
   (read-file "/Users/me/Documents/sign-of-four.txt")) {\W} 0))
(set 'wilde-data
 (parse (lower-case
   (read-file "/Users/me/Documents/dorian-grey.txt")) {\W} 0))
```

The **bayes-train** function can now scan these two data sets and store the word frequencies in a new context, which I'm calling Lexicon:

```
(bayes-train doyle-data wilde-data 'Lexicon)
```

This context now contains a list of words that occur in the lists, and the frequencies of each. For example:

```
Lexicon:_always
;-> (21 110)
```

ie the word `always` appeared 21 times in Conan Doyle's text, and 110 times in Wilde's. Next, the Lexicon context can be saved in a file:

```
(save "/Users/me/Documents/lex.lsp" 'Lexicon)
```

and reloaded whenever necessary with:

```
(load "/Users/me/Documents/lex.lsp")
```

With training completed, you can use the **bayes-query** function to look up a list of words in a context, and return two numbers, the probabilities of the words belonging to the first or second set of words. Here are three queries. Remember that the first set was Doyle, the second was Wilde:

```
(set 'quote1
 (bayes-query
   (parse (lower-case
    "the latest vegetable alkaloid" ) {\W} 0)
   'Lexicon))
;-> (0.973352412 0.02664758802)

(set 'quote2
 (bayes-query
   (parse
    (lower-case
    "observations of threadbare morality to listen to" ) {\W} 0)
    'Lexicon))
;-> (0.5 0.5)

(set 'quote3
 (bayes-query
   (parse
    (lower-case
    "after breakfast he flung himself down on a divan
     and lit a cigarette" ){\W} 0)
   'Lexicon))
;-> (0.01961482169 0.9803851783)
```

These numbers suggest that `quote1` is probably (97% certain) from Conan Doyle, that `quote2` is neither Doylean nor Wildean, and that `quote3` is likely to be from Oscar Wilde.

Perhaps we were lucky, but that seems to be a good result. The first quote is from Doyle's 'A Study in Scarlet', and the third is from Wilde's 'Lord Arthur Savile's Crime', both texts that were not included in the training process but – apparently – typical of the author's vocabulary. The second quote is from Jane Austen, and the methods developed by the Reverend are unable to assign it to either of the authors.

Financial functions

newLISP offers the following financial functions:

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

Logic programming

The programming language Prolog promoted a type of logic programming called unification. newLISP provides a **unify** function that can carry out unification, which involves matching expressions.

To use **unify**, you can build up a small database of rules and statements, and then carry out queries on the database by trying to match expressions.

Bit operators

The bit operators treat numbers as if they consist of 1’s and 0’s. You can see the way they operate if we define a utility function that prints out numbers in binary format:

```
(define (binary n (width 24) , n1 temp)
 (set ’n1 n)
 (dotimes (i width)
   (push (& n 0x1) temp)
   (set ’n (>> n)))
   (println (format "%6d %s" n1 (join (map string temp)))))
```

This function prints out both the original number and a binary representation of it:

```
(binary 6)

   6 000000000000000000000110
```

The shift **«** and **»** functions move the bits to the right or left:

```
(binary 6)

   6 000000000000000000000110

(binary (<< 6))             ; shift left

  12 000000000000000000001100

(binary (>> 6))             ; shift right

   3 000000000000000000000011
```

The following operators compare the bits of two or more numbers. Using 4 and 5 as examples:

```
(map binary (5 4))

  5 000000000000000000000101
  4 000000000000000000000100

(binary (^ 4 5))      ; exclusive or: 1 if only 1 of the two bits is 1

  1 000000000000000000000001

(binary (| 4 5))      ; or: 1 if either or both bits are 1

  5 000000000000000000000101

(binary (& 4 5))      ; and: 1 only if both are 1

  4 000000000000000000000100
```

The negate or not function **~** reverses all the bits in a number, exchanging 1’s and 0’s:

```
(binary (~ 5))        ; not: 1 <-> 0
```

```
-6 11111111111111111111010
```

The binary function that prints out these strings uses the & function to test the last bit of the number to see if it's a 1, and the » function to shift the number 1 bit to the right, ready for the next iteration.

One use for the or operator (|) is when you want to combine regular expression options with the **regex** function.

crc32 calculates a 32 bit CRC (Cyclic Redundancy Check) for a string.

Bigger numbers

If you really want to use numbers larger than 9223372036854775807, and/or numbers with more precision than the default floating-point, you can either:

- send the calculations to a high-precision calculator process
- import a precision arithmetic library such as the GNU Multiple Precision Arithmetic Library

To send a calculation to a high-precision calculator process (such as bc on Unix), wrap the calculation in a string and send it to the shell:

```
(replace "\\\\" (join (exec "echo 'scale=300; 4 * a(1)' | bc -ql")) "")
```

```
3.1415926535897932384626433832795028841971693993751058209749445
5923078164062862089986280348253421170679821480865132823066470
0938446095505822317253594081284811174502841027019385211055596
4462294895493038196442881097566593344612847564823378678316527
7120190914564856692346034861045432664821339360726024914127
```

This calculation uses the value of `4 * arctangent(1)` to generate π. The **replace** and **join** functions seem to be required to correctly format the string returned by `bc`.

If you have installed the GMP (GNU Multiple Precision Arithmetic Library) on your computer, and downloaded the GMP interface module from the newLISP site, you can import those superhuman numeric functions into your code:

```
(load "gmp.lsp")
(define (factorial-gmp num)
 (if (= num 0)
  "1"
  (GMP:* (string num) (factorial-gmp (- num 1)))))
```

10 Working with dates and times

Date and time functions

To work with dates and times, use the following functions:

- **date** convert a seconds count to a date/time, or return date/time for now
- **date-value** return the time in seconds since 1970-1-1 for a date and time, or for now
- **now** return the current date/time information in a list
- **time-of-day** return milliseconds since the start of today till now

date-value and **now** work in UT, not your local time. **date** can take account of the time difference between your local time and UT.

The current time and date

All four functions can be used to return information about the current time. **date-value** returns the number of seconds between 1970 and the current time (in UT):

```
1142798985
```

and **now** returns a list of integers containing information about the current date and time (in UT):

```
(now)
;-> (2006 3 19 20 5 2 125475 78 1 0 0)
```

This provides the following information:

- year, month, day (2006, 3, 19)
- hour, minute, second, microsecond (20, 5, 2, 125475)
- day of current year (78)
- day of current week (1)
- local time zone offset (in minutes west of GMT) (0)
- daylight savings time flag (0)

To extract the information you want, use a slice or pick out the elements:

```
(slice (now) 0 3)             ; year month day using explicit slice
(0 3 (now))                   ; year month day using implicit slice
(select (now) (0 1 2))        ; year month day using selection
(3 3 (now))                   ; hour minute second
(nth 8 (now))                 ; day of the week, starting from Sunday
```

date used on its own will give you the current date and time for the local time zone (**now** and **date-value** return values in UCT/UTC, not relative to your local time zone):

```
(date)
;-> "Mon Mar 19 20:05:02 2006"
```

It can also tell you the date of an integer number of seconds since 1970 (the start of the Unix epoch), adjusted for your local time zone:

```
(date 0)                                ; a US newLISP user sees this
;-> "Wed Dec 31 16:00:00 1969"
(date 0)                                ; a European newLISP user sees
this
;-> "Thu Jan 1 01:00:00 1970"
```

date-value can calculate the number of seconds for a specific date or date/time (in UT):

```
(date-value 2006 5 11)                  ; just the date
;-> 1147305600

(date-value 2006 5 11 23 59 59)         ; date and time (in UT)
;-> 1147391999
```

Because **date-value** can accept the year, month, day, hours, minutes, and seconds as input, it can be applied to the output of **now**:

```
(apply date-value (now))
;-> 1164723787
```

By converting different times to these date values, you can do calculations. For example, to subtract November 13 2003 from January 3 2005:

```
(- (date-value 2005 1 3) (date-value 2003 11 13))
;-> 36028800
; seconds, which is

(/ 36028800 (* 24 60 60))
;-> 417
; this is a duration in days - don't convert this to a date!
```

You can find out the date that is 12 days after Christmas Day 2005 by adding 12 days-worth of seconds to the date:

```
(+ (date-value 2005 12 25) (* 12 24 60 60))
; seconds in 12 days
;-> 1136505600
; this is an instant in time, so it can be converted!
```

This seconds value can be converted to a human-friendly date by **date** in its longer form, when it takes a seconds-since-1970 value and converts it to a local time zone representation of this UT-based value:

```
(date 1136505600)
;-> "Fri Jan 6 00:00:00 2006"           ; for this European user...
```

Of course `(date (date-value))` is the same as `(date)`, but you'll have to use the longer form if you want to change the date format. **date** accepts an additional formatting string (preceded by a time-zone offset in minutes). If you're familiar with C-style `strftime` formatting, you'll know what to do:

```
(date (date-value) 0 "%Y-%m-%d %H:%M:%S") ; ISO 8601
;-> 2006-06-08 11:55:08

(date 1136505600 0 "%Y-%m-%d %H:%M:%S")
;-> "2006-01-06 00:00:00"
```

```
(date (date-value) 0 "%Y%m%d-%H%M%S")   ; in London
;-> "20061207-144445"

(date (date-value) (* -8 60) "%Y%m%d-%H%M%S") ; in Los Angeles
;-> "20061207-064445"                          ; 8 hours offset
```

Reading dates and times: parse-date

The **parse-date** function can convert date and time strings to seconds-since-1970 values. You supply a date-time format string after the string:

```
(parse-date "2006-12-13" "%Y-%m-%d")
;-> 1165968000

(date (parse-date "2007-02-08 20:12" "%Y-%m-%d %H:%M"))
;-> "Thu Feb  8 20:12:00 2007"
```

Timing and timers

For timing purposes, you can use these functions:

- **time** return the time taken to evaluate an expression, in milliseconds
- **timer** set a timer to wait for a certain number of seconds and then evaluate expression
- **sleep** stop working for a certain number of milliseconds

time is useful for finding out how much time expressions take to evaluate:

```
(time (read-file "/Users/me/Music/iTunes/iTunes Music Library.xml"))
;-> 27                                ; milliseconds
```

You can supply a repetition count as well, which probably gives a more accurate picture:

```
(time (for (x 1 1000) (factor x)) 100)  ; 100 repetitions
;-> 426
```

If you can't or don't want to enclose expressions, more simple timing can be done using **time-of-day**:

```
(set 'start-time (time-of-day))
(for (i 1 1000000)
  (set 'temp (sqrt i)))

(string {that took } (div (- (time-of-day) start-time) 1000) { seconds})
;-> "that took 0.238 seconds"
```

timer is basically an alarm clock. Set it and then forget about it until the time comes. You supply a symbol specifying the alarm action, followed by the number of seconds to wait:

```
(define (teas-brewed)
 (println (date) " Your tea has brewed, sir!"))

(timer teas-brewed (* 3 60))
```

and three minutes later you'll see this:

```
Sun Mar 19 23:36:33 2006 Your tea has brewed, sir!
```

Without any arguments, this function returns the name of the current symbol that's been assigned as the alarm action:

```
(timer)
;-> teas-brewed
```

If you're waiting for the alarm to go off, and you're impatient to see how much time has elapsed so far, use the function with the name of the assigned symbol but without a seconds value:

```
(timer teas-brewed)
;-> 89.135747
; waited only a minute and a bit so far
```

For another example of the use of these functions, see "**Simple countdown timer**" on page **159**.

11 Working with files

Functions for working with files can be grouped into two main categories: interacting with the operating system, and reading and writing data to and from files.

Interacting with the file system

The concept of a current working directory is maintained in newLISP. When you start newLISP by typing newLISP in a terminal, your current working directory becomes newLISP's current directory.

```
$ pwd
/Users/me/projects/programming/lisp
$ newlisp
newLISP v.9.3 on OSX UTF-8, execute ’newlisp -h’ for more info.

> (env "PWD")
"/Users/me/projects/programming/lisp"
> (exit)

$ pwd
/Users/me/projects/programming/lisp
$
```

You can also check your current working directory using the **real-path** function without arguments:

```
(real-path)
;-> "/Users/me/projects/programming/lisp"
```

But when you run a newLISP script from elsewhere, such as from inside a text editor, the current working directory and other settings may be different. It's a good idea, therefore, to use **change-dir** to establish the current working directory, if you're not sure:

```
(change-dir "/Users/me/Documents")
;-> true
```

Other environment variables can be accessed with **env**:

```
(env "HOME")
;-> "/Users/me"
(env "USER")
;-> "cormullion"
```

Again, running newLISP from a text editor rather than in an interactive terminal session will affect which environment variables are available and their values.

Once you have the correct working directory, use **directory** to list its contents:

```
(directory)
("." ".." ".bash_history" ".bash_profile" ".inputrc" ".lpoptions"
".sqlite_history" ".ssh" ".subversion" "bin" "Desktop" "Desktop Folder"
"Documents" "Library" ...
```

and so on. Notice that it gives you the relative filenames, not the absolute pathnames. **directory** can list the contents of directories other than the current working directory too, if you supply a path. And in this extended form, you can use regular expressions to filter the contents:

```
(directory "./")                         ; just a pathname
;-> ("." ".." ".bash_history" ".bash_profile" ".inputrc" ".lpoptions"
".sqlite_history" ".ssh" ".subversion" "bin" "Desktop" "Desktop Folder"
"Documents" "Library" ...

(directory "./" {^[^.]})                 ; exclude files starting "."
;-> ("bin" "Desktop" "Desktop Folder" "Documents" "Library" ... )
```

Again, notice that the results are relative to the current working directory. It's often useful to store the path of the directory that you're listing, in case you have to use it later to build full pathnames. **real-path** returns the full pathname of a file or directory, either in the current working directory:

```
(real-path ".subversion")
;-> "/Users/me/.subversion"
```

or specified by another relative pathname:

```
(real-path "projects/programming/lisp/lex.lsp")
;-> "/Users/me/projects/programming/lisp/lex.lsp"
```

To find the containing directory of an item on disk, you can just remove the file name from the full pathname:

```
(set 'f "lex.lsp")
(replace f (real-path f) "")
;-> "/Users/me/projects/programming/lisp/"
```

This won't always work, by the way, if the file name appears as a directory name as well earlier in the path. A simple solution is to do a regex search for `f` occurring only at the very end of the pathname, using the $ option:

```
(replace (string f "\$") (real-path f) "" 0).
```

To scan a section of your file system recursively, use a function that calls itself recursively. Here, just the full path name is printed:

```
(define (search-tree dir  )
 (dolist (item (directory dir {^[^.]}))
   (if (directory? (append dir item))
    ; search the directory
    (search-tree (append dir item "/"))
    ; or process the file
    (println (append dir item)))))

(search-tree {/usr/share/newlisp/})
```

```
/usr/share/newlisp/guiserver/allfonts-demo.lsp
/usr/share/newlisp/guiserver/animation-demo.lsp
...
/usr/share/newlisp/util/newlisp.vim
/usr/share/newlisp/util/syntax.cgi
```

See also "**Editing text files in folders and hierarchies**" on page **161**.

You'll find some testing functions useful:

- **file?** does this file exist?
- **directory?** is this pathname a directory or a file?

Remember the difference between relative and absolute pathnames:

```
(file? "System")
;-> nil
(file? "/System")
;-> true
```

File information

You can get information about a file with **file-info**. This function asks the operating system about the file and returns the information in a series of numbers:

- 0 size
- 1 mode
- 2 device mode
- 3 user id
- 4 group id
- 5 access time
- 6 modification time
- 7 status change time

To get the size of files, for example, get the first number returned by **file-info**. The following code lists the files in a directory and includes their sizes too.

```
(set 'dir {/usr/share/newlisp})
(dolist (i (directory dir {^[^.]}))
 (set 'item (string dir "/" i))
 (if (not (directory? item))
  (println (format {%7d %-30s} (nth 0 (file-info item)) item))))
```

```
 281615 /usr/share/newlisp/guiserver.jar
 131381 /usr/share/newlisp/guiserver.lsp
      0 /usr/share/newlisp/init.lsp
   1730 /usr/share/newlisp/init.lsp.example
...
```

Notice that we stored the name of the directory in `dir`. The **directory** function returns relative file names, but you must pass absolute pathname strings to **file-info** unless the string refers to a file that's in the current working directory.

You can use implicit addressing to select the item you want. So instead of `(nth 0 (file-info item))`, you can write: `(file-info item 0)`.

MacOS X: resource forks

If you tried the previous script on MacOS X, you might notice that some files are 0 bytes in size. This might indicate the presence of the dual-fork system inherited from the days of the classic (ie old) Macintosh. Use the following version to access the resource forks of files. A good hunting ground for the increasingly rare resource fork is the fonts folder:

```
(set 'dir "/Users/me/Library/Fonts") ; fonts folder
(dolist (i (directory dir "^[^.]"))
 (set 'item (string dir "/" i))
  (and
    (not (directory? item))                     ; don't do folders
    (println
      (format "%9d DF %-30s" (nth 0 (file-info item)) item))
    (file? (format "%s/..namedfork/rsrc" item)) ; there’s a resource fork
too
    (println (format "%9d RF"
      (first (file-info (format "%s/..namedfork/rsrc" item)))))
  )
)

...
        0 DF /Users/me/Library/Fonts/AvantGarBoo
    26917 RF
        0 DF /Users/me/Library/Fonts/AvantGarBooObl
    34982 RF
        0 DF /Users/me/Library/Fonts/AvantGarDem
    27735 RF
        0 DF /Users/me/Library/Fonts/AvantGarDemObl
    35859 RF
        0 DF /Users/me/Library/Fonts/ITC Avant Garde Gothic 1
   116262 RF
...
```

File management

To manage files, you can use the following functions:

- **rename-file** renames a file or directory
- **copy-file** copies a file
- **delete-file** deletes a file
- **make-dir** makes a new directory
- **remove-dir** removes an empty directory

For example, to renumber all files in the current working directory so that the files sort by modification date, you could write something like this:

```
(set 'dir {/Users/me/temp/})
(dolist (i (directory dir {^[^.]}))
 (set 'item (string dir "/" i))
 (set 'mod-date (date (file-info item 6) 0 "%Y%m%d-%H%M%S"))
 (rename-file item (string dir "/" mod-date i)))

;-> before
image-001.png
image-002.png
image-003.png
image-004.png

;-> after
20061116-120534image-001.png
20061116-155127image-002.png
20061117-210447image-003.png
20061118-143510image-004.png
```

The `(file-info item 6)` is a handy way of extracting the modification time (item 6) of the result returned by **file-info**.

Always test scripts like this before you use them for real work! A misplaced punctuation character can wreak havoc.

Reading and writing data

newLISP has a good selection of input and output functions.

An easy way of writing stuff to file is **append-file**, which adds a string to the end of a file. The file is created if it doesn't exist. It's very useful for creating log files and files that you write to periodically:

```
(dotimes (x 10)
 (append-file "/Users/me/Desktop/log.log"
 (string (date) " logging " x "\n")))
```

To load the contents of a file into a symbol in one gulp, use **read-file**.

```
(set 'contents (read-file "/usr/share/newlisp/init.lsp.example"))
;->
";; init.lsp - newLISP initialization file\n;; gets loaded automatically
on
...
(load (append $HOME \"/.init.lsp\")) 'error))\n\n;;;; end of file
;;;;\n\n\n          "
```

open returns a value which acts as a file handle to a file. You'll probably want to use the file handle later, so store it in a symbol:

```
(set 'data-file (open "newfile.data" "read"))  ; in current directory

; and later

(close data-file)
```

Use **read-line** to read a file one line at a time from a file handle. Each time you use **read-line**, the next line is stored in a buffer which you can access with the **current-line** function. The basic approach for reading a file is like this:

```
(set 'file (open ((main-args) 2) "read"))      ; the argument to the
script
(while (read-line file)
 (println (current-line)))                     ; just output the line
```

read-line discards the line feed at the end of each line. **println** adds one at the end of the text you supply. For more about argument handling and **main-args**, see "**Standard input and output**" on page **132**.

For small to medium-size files, Working through a source file a line at a time is much slower than loading the whole file into memory. For example, the source document for this introduction is about 6000 lines of text, or about 350KBytes. It's about 10 times faster to process the file using **read-file** and **parse**, like this:

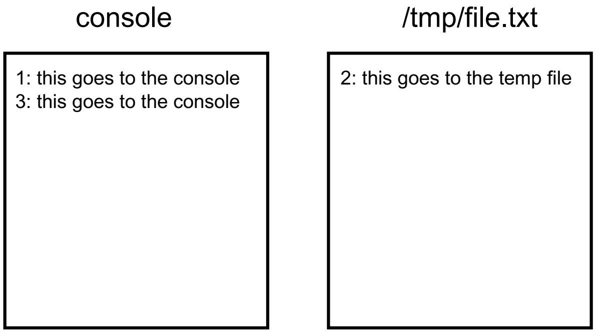
```
(set 'source-text (read-file "/Users/me/introduction.txt"))
(dolist (ln (parse source-text "\n" 0))
     (process-line ln))
```

than using **read-line**, like this:

```
(set 'source-file (open "/Users/me/introduction.txt" "read"))
(while (read-line source-file)
     (process-line (current-line)))
```

The **device** function is a handy way of switching output between the console and a file:

```
(set 'output-file (open "/tmp/file.txt" "write"))
(println "1: this goes to the console")
(device output-file)
(println "2: this goes to the temp file")
(device 0)
(println "3: this goes to the console")
(close output-file)
```



```
1: this goes to the console
3: this goes to the console
```

```
2: this goes to the temp file
```

Figure 11.1 device: file or console

Suppose that your script accepts a single argument, and you want to write the output to a file with the same name but with a `.out` suffix. Try this:

```
(device (open (string ((main-args) 2) ".out") "write"))
...
(set 'file-contents (read-file ((main-args) 2)))
```

Now you can process the file's contents and output any information using `println` statements.

The **load** and **save** functions are used to load newLISP source code, and to save it into a file.

The **read-line** and **write-line** functions can be used to read and write lines to threads as well as files. See "**Reading and writing to threads**" on page **137**.

Standard input and output

To read from STDIO (standard input) and write to STDOUT (standard output), use **read-line** and **println**. For example, here's a simple filter that converts standard input to lower-case and outputs it:

```
#!/usr/bin/newlisp

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

(exit)
```

And it can be run in the shell like this:

```
$ ./lower-case.lsp
HI
hi
HI THERE
hi there
...
```

Command-line arguments

To use newLISP programs on the command line, you can access the arguments with the **main-args** function. For example, if you create this file:

```
#!/usr/bin/newlisp
(println (main-args))
(exit)
```

make it executable, and then run it in the shell, you'll see a list of the arguments that were supplied to the script when you run it:

```
$ test.lsp 1 2 3
("/usr/bin/newlisp" "/Users/me/bin/test.lsp" "1" "2" "3")
$
```

main-args returns a list of the arguments that were passed to your program. The first two arguments, which you probably don't want to process, are the path of the newLISP program, and the pathname of the script being executed:

```
(main-args)
;-> ("/usr/bin/newlisp" "/path/script.lsp" "1" "2" "3")
```

So you probably want to process the arguments starting with the one at index 2:

```
((main-args) 2)
;-> 1.txt
```

which is returned as a string. Or, to process all the arguments, starting at index 2, use a slice:

```
(2 (main-args))
;-> ("1" "2" "3")
```

and the arguments are returned in a list of strings.

Often, you'll want to work through all the arguments in your script: a convenient phrase for this is:

```
(dolist (a (2 (main-args)))
 (println a)
  ...)
```

A slightly more readable equivalent is this, which works through the rest of the rest of the arguments:

```
(dolist (a (rest (rest (main-args))))
 (println a)
 ...)
```

Here's a short script that filters out unwanted Unicode characters from a text file, except for a few special ones that are allowed through:

```
(set 'file (open ((main-args) 2) "read")) ; one file

(define (in-range? n low high)
 ; is n between low and high inclusive?
 (and (<= n high) (>= n low)))

(while (read-line file)
 (dostring (c (current-line))
   (if
      (or
          (in-range? c 32 127)       ; ascii
          (in-range? c 9 10)         ; tab newline
          (in-range? c 12 13)        ; \f \r
          (= c (int "\0xbb"))        ; right double angle
          (= c (int "\0x25ca"))      ; diamond
          (= c (int "\0x2022"))      ; bullet
          (= c (int "\0x201c"))      ; open double quote
          (= c (int "\0x201d"))      ; close double quote
          )
       (print (char c))))            ; nothing to do
      (println) ; because read-line swallows line endings
)
```

For some more examples of argument processing, see “**Simple countdown timer**” on page **159**.

12 Working with pipes, threads, and processes

Processes, pipes, threads, and system functions

The following functions allow you to interact with the operating system:

- **!** run a command in the operating system
- **abort** stop all spawned processes
- **exec** run a process and read from or write to it
- **fork** launch a newLISP child process thread (Unix)
- **pipe** create a pipe for interprocess communication
- **process** launch a child process, remap standard I/O and standard error
- **semaphore** create and control semaphores
- **share** share memory with other processes and threads
- **spawn** create a new newLISP child process
- **sync** monitor and synchronize spawned processes
- **wait-pid** wait for a child process to end

Because these commands interact with your operating system, you should refer to the documentation for platform-specific issues and restrictions.

! runs a system command and shows you the results in the console. The **exec** function does a similar job, but it waits for the operating system to finish, then returns the standard output as a list of strings, one for each line:

```
(exec "ls -Rat /usr | grep newlisp")
;->
("newlisp" "newlisp-edit" "newlispdoc" "newlisp" "newlisp.1"
"newlispdoc.1" "/usr/share/newlisp:"
"/usr/share/newlisp/guiserver:"
"/usr/share/newlisp/modules:" "/usr/share/newlisp/util:"
"newlisp.vim" "newlisp" "/usr/share/doc/newlisp:"
"newlisp_index.html" "newlisp_manual.html"
"/usr/share/doc/newlisp/guiserver:")
```

As usual, you'll have all the 'fun' of quoting and double-quoting to get your commands through the shell.

With **exec**, your script will wait until the command finishes before your script continues.

```
(exec (string "du -s " (env "HOME") "/Desktop"))
```

You'll see the results only when the command finishes.

To interact with another process running alongside newLISP, rather than wait until a process finishes, use **process**. See "**Processes, pipes, threads, and system functions**" on page **135**.

Multitasking

All the newLISP expressions we've evaluated so far have run serially, one after the other, so one expression has to finish evaluating before newLISP can start on the next one. This is usually OK. But sometimes you want to start an expression and then move on to another while the first is still being evaluated. Or you might want to divide a large job into a number of smaller ones, perhaps taking advantage of any extra processors your computer has. newLISP does this type of multitasking using three functions: **spawn** for creating new processes to run in parallel, **sync** for monitoring and finishing them, and **abort** for stopping them before they've finished.

For the examples that follow, I'll use this short pulsing function:

```
(define (pulsar ch interval)
 (for (i 1 20)
   (println ch)
   (sleep interval))
 (println "I've finished"))
```

When you run this normally, you'll see 20 characters printed, one every `interval` milliseconds. The execution of this function blocks everything else, and you'll have to wait for all 20, or stop execution using Control-C.

To run this function in a parallel process alongside the current one, use the **spawn** function. Supply a symbol to hold the result of the expression, followed by the expression to be evaluated:

```
> (spawn 'r1 (pulsar "." 3000))
2882
> .
```

The number returned by the function is the process ID. You can now carry on using newLISP in the terminal while the pulsar continues to interrupt you (yes, it's irritating!). Start a few more:

```
> (spawn 'r2 (pulsar "-" 5000))
2883
> (spawn 'r3 (pulsar "!" 7000))
2884
> (spawn 'r4 (pulsar "@" 9000))
2885
```

To see how many processes are active (and haven't yet finished), use the **sync** function without arguments:

```
> (sync)
(2885 2884 2883 2882)
```

If you want to stop all the pulsar processes, use **abort**:

```
> (abort)
true
```

sync also lets you keep an eye on currently running processes. Supply a value in milliseconds; newLISP will wait for that time, and then check to see if the spawned processes have finished:

```
> (spawn 'r1 (pulsar "." 3000))
2888
> .
> (sync 1000)
nil
```

If the result is nil, then the processes haven't finished. If the result is true, then they've all finished. Now – and only now, after the **sync** function runs – the value of the return symbol `r1` is set to be the value returned by the process. In the case of the pulsar, this will be the string "I've finished".

```
I've finished
> r1
nil
> (sync 1000)
true
> r1
"I've finished"
>
```

Notice that the process finished – at least, it printed its concluding message – but the symbol `r1` wasn't set until the **sync** function was executed and returned true. This is because **sync** returns true, and the return symbols have values, only if *all* the spawned processes have finished.

You could execute a loop if you wanted to wait for all the processes to complete:

```
(until (sync 1000))
```

which checks every second to see if the processes have completed.

As a bonus, many modern computers have more than one processor, and your scripts might be able to run faster if each processor can devote its energies to one task. newLISP leaves it to the operating system to schedule tasks and processors according to the hardware at its disposal.

forked processes

There are some lower-level functions for manipulating processes. These aren't as convenient or as easy to use as the spawned process technique described in the previous section, but they provide a few additional features that you might find a use for some day.

You can use **fork** to evaluate an expression in another process. Once the process is launched, it won't return a value to the parent process, so you'll have to think about how to obtain its results. Here's a way of calculating prime numbers in a separate process and saving the output in a file:

```
(define (isprime? n)
 (if (= 1 (length (factor n)))
   true))

(define (find-primes l h)
 (for (x l h)
   (if (isprime? x)
       (push x results -1)))
 results)

(fork (append-file "/Users/me/primes.txt"
  (string "the result is: " (find-primes 500000 600000))))
```

Here, the new child process launched by **fork** knows how to find prime numbers, but, unlike the spawned processes, can't return information to its parent process to report how many it's found.

It is possible for processes to share information. The **share** function sets up a common area of memory, like a notice board, that all processes can read and write to. There's a simple example pf **share** in a later chapter: see "**A simple IRC client**" on page **152**.)

To control how processes access shared memory, newLISP provides a **semaphore** function.

Reading and writing to threads

If you want forked threads to communicate with each other, you'll have to do a bit of plumbing first. Use **pipe** to set up communications channels, then arrange for one thread to listen to another. **pipe** returns a list containing read and write handles to a new inter-process communications pipe, and you can then use these as channels for the **read-line** and **write-line** functions to read and write to.

```
(define (isprime? n)
 (if (= 1 (length (factor n)))
 true))

(define (find-primes l h)
 (for (x l h)
   (if (isprime? x)
       (push x results -1)))
 results)

(define (generate-primes channel)
 (dolist (x (find-primes 100 300))
  (write-line (string x) channel)))         ; write a prime

(define (report-results channel)
 (do-until (> (int i) 290)
  (println (setq i (read-line channel)))))  ; get next prime

(define (start)
 (map set '(in out) (pipe))                 ; do some plumbing
 (set 'generator (fork (report-results in)))
 (set 'reporter (fork (generate-primes out)))
 (println "they've started"))

(start)
```

```
they've started
101
103
107
109
...
```

```
(wait-pid generator)
(wait-pid reporter)
```

Notice how the "they've started" string appears before any primes are printed, even though the **println** expression occurs after the threads start.

The **wait-pid** function waits for the thread launched by **fork** to finish – of course you don't have to do this immediately.

Communicating with other processes

To launch a new operating system process that runs alongside newLISP, use **process**. As with **fork**, you first want to set up some suitable plumbing, so that newLISP can both talk and listen to the process, which in the following example is the Unix calculator `bc`. The **write-buffer** function writes to the `myout` pipe, which is read by bc via `bcin`. bc's output is directed through `bcout` and read by newLISP using **read-line**.

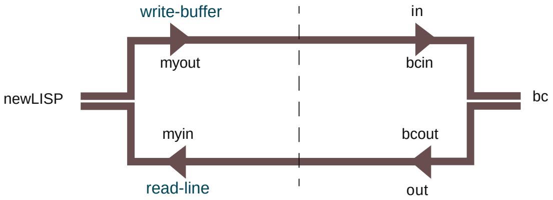


Figure 12.1 plumbing: organizing pipes for communication with threads

```
(map set '(bcin myout) (pipe))                  ; set up the plumbing
(map set '(myin bcout) (pipe))
(process "/usr/bin/bc" bcin bcout)              ; start the bc process
(set 'sum "123456789012345 * 123456789012345")
(write-buffer myout (string sum "\n"))
(set 'answer (read-line myin))
(println (string sum " = " answer))
```

```
123456789012345 * 123456789012345 = 15241578753238669120562399025
```

```
(write-buffer myout "quit\n")                   ; don't forget to quit!
```

13 Working with XML

Converting XML into lists

XML files are widely used these days, and you might have noticed that the highly organized tree structure of an XML file is similar to the nested list structures that we've met in newLISP. So wouldn't it be good if you could work with XML files as easily as you can work with lists?

You've met the two main XML-handling functions already. (See "**ref and ref-all**" on page **45**.) The **xml-parse** and **xml-type-tags** functions are all you need to convert XML into a newLISP list. (**xml-error** is used for diagnosing errors.) **xml-type-tags** determines how the XML tags are processed by **xml-parse**, which does the actual processing of the XML file, turning it into a list.

To illustrate the use of these functions, we'll use the RSS newsfeed from the newLISP forum:

```
(set 'xml (get-url "http://www.alh.net/newlisp/phpbb/rss.php"))
```

and store the retrieved XML in a file, to save hitting the server repeatedly:

```
(save {/Users/me/Desktop/newlisp.xml} 'xml)  ; save symbol in file
(load {/Users/me/Desktop/newlisp.xml})       ; load symbol from file
```

The XML starts like this:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<rss version="0.92">
<channel>
 <docs>http://backend.userland.com/rss092</docs>
 <title>newLISP Fan Club</title>
 <link>http://www.alh.net/newlisp/phpbb/</link>
 <description>Friends and Fans of newLISP</description>
 <managingEditor>newlispfanclub-at-excite.com</managingEditor>
 <webMaster>newlispfanclub-at-excite.com</webMaster>
 <lastBuildDate>Thu, 30 Nov 2006 14:22:25 GMT</lastBuildDate>
<item>
 <title>newLISP newS :: RE: mystery quoting</title>
...
```

If you use **xml-parse** to parse the XML, but don't use **xml-type-tags** first, the output looks like this:

```
(xml-parse xml)
(("ELEMENT" "rss" (("version" "0.92")) (("TEXT" "\n")
 ("ELEMENT" "channel"
 ()
 (("TEXT" "\n\t") ("ELEMENT" "docs" ()
 (("TEXT" "http://backend.userland.com/rss092")))
 ("TEXT" "\n\t")
 ("ELEMENT" "title" () (("TEXT" "newLISP Fan Club")))
 ("TEXT" "\n\t")
 ("ELEMENT" "link" ()
 (("TEXT" "http://www.alh.net/newlisp/phpbb/")))
 ("TEXT" "\n\t")
...
```

Although it looks a bit LISP-y already, you can see that the elements have been labelled as "ELEMENT", "TEXT". For now, we could do without all these labels, and that's basically the job that **xml-type-tags** does. It lets you determine the labels for four types of XML tag: TEXT, CDATA, COMMENTS, and ELEMENTS. We'll hide them altogether with four **nil**s. We'll also use some options for **xml-parse** to tidy up the output even more.

```
(xml-type-tags nil nil nil nil)
(set 'sxml (xml-parse xml 15))        ; options: 15 (see below)
;->
((rss ((version "0.92"))
 (channel (docs "http://backend.userland.com/rss092")
 (title "newLISP Fan Club")
 (link "http://www.alh.net/newlisp/phpbb/")
 (description "Friends and Fans of newLISP")
 (managingEditor "newlispfanclub-at-excite.com")
 (webMaster "newlispfanclub-at-excite.com")
 (lastBuildDate "Thu, 30 Nov 2006 14:22:25 GMT")
 (item (title "newLISP newS :: RE: mystery quoting")
  (link "http://www.alh.net/newlisp/phpbb
  viewtopic.php?p=7899\#7899")
 (description [text]Author: &lt;a
...
```

This is now a useful newLISP list, albeit quite a complicated one, stored in a symbol called `sxml`. (This way of representing XML is called S-XML.)

If you're wondering what that 15 was doing in the **xml-parse** expression, it's just a way of controlling how much of the auxiliary XML information is translated: the options are as follows:

- 1 – suppress whitespace text tags
- 2 – suppress empty attribute lists
- 4 – suppress comment tags
- 8 – translate string tags into symbols
- 16 – add SXML (S-expression XML) attribute tags

You add them up to get the options code number – so 15 (+ 1 2 4 8) uses the first four of these options: suppress unwanted stuff, and translate strings tags to symbols. As a result of this, new symbols have been added to newLISP's symbol table:

```
(channel description docs item lastBuildDate link
managingEditor rss sxml title version webMaster xml)
```

These correspond to the string tags in the XML file, and they'll be useful almost immediately.

Now what?

The story so far is basically this:

```
(set 'xml (get-url "http://www.alh.net/newlisp/phpbb/rss.php"))
; we stored this in a temporary file while exploring
(xml-type-tags nil nil nil nil)
(set 'sxml (xml-parse xml 15))
```

which has given us a list version of the news feed stored in the `sxml` symbol.

Because this list has a complicated nested structure, it's best to use **ref** and **ref-all** rather than **find** to look for things. **ref** finds the first occurrence of an expression in a list and returns the address:

```
(ref 'item sxml)
;-> (0 2 8 0)
```

These numbers are the address in the list of the first occurrence of the symbol `item`: `(0 2 8 0)` means 'start at item 0 of the whole list, then go to item 2 of that item, then go to item 8 of that one, then item 0 of that one'. (0-based indexing, of course!)

To find the higher-level or enclosing item, use **chop** to remove the last level of the address:

```
(chop (ref 'item sxml))
;-> (0 2 8)
```

This now points to the level that contains the first item. It's like chopping the house number off an address, leaving the street name.

Now you can use this address with other expressions that accept a list of indices. The most convenient and compact form is probably the implicit address, which is just the name of the source list followed by the set of indices in a list:

```
(sxml (chop (ref 'item sxml)))            ; a (0 2 8) slice of sxml
;->
(item
 (title "newLISP newS :: RE: mystery quoting")
 (link "http://www.alh.net/newlisp/phpbb/
  viewtopic.php?p=7899\#7899")
 (description [text]Author: &lt;a href=&qu
...
```

That found the first occurrence of `item`, and returned the enclosing portion of the SXML.

Another technique available to you is to treat sections of the list as association lists:

```
(lookup 'title (sxml (chop (ref 'item sxml))))
;->
newLISP newS :: RE: mystery quoting
```

Here we've found the first item, as before, and then looked up the first occurrence of title using **lookup**.

Use **ref-all** to find all occurrences of a symbol in a list. It returns a list of addresses:

```
(ref-all 'title sxml)
;->
((0 2 2 0)
 (0 2 8 1 0)
 (0 2 9 1 0)
 (0 2 10 1 0)
 (0 2 11 1 0)
 (0 2 12 1 0)
 (0 2 13 1 0)
 (0 2 14 1 0)
 (0 2 15 1 0)
 (0 2 16 1 0)
 (0 2 17 1 0)
 (0 2 18 1 0)
 (0 2 19 1 0)
 (0 2 20 1 0)
 (0 2 21 1 0)
 (0 2 22 1 0))
```

With a simple list traversal, you can quickly show all the titles in the file, at whatever level they may be:

```
(dolist (el (ref-all 'title sxml))
 (println (rest (sxml (chop el)))))
```

```
("newLISP Fan Club")
("newLISP newS :: RE: mystery quoting")
("newLISP newS :: RE: mystery quoting")
("newLISP newS :: RE: mystery quoting")
("newLISP newS :: RE: development release version 9.0.4")
("newLISP newS :: RE: development release version 9.0.4")
...
```

Without the **rest** in there, you would have seen this:

```
(title "newLISP Fan Club")
(title "newLISP newS :: RE: mystery quoting")
...
```

As you can see, there are many different ways to access the information in the SXML data. To produce a concise summary of the news in the XML file, one approach is to go through all the items, and extract the title and description entries. Because the description elements are a mass of escaped entities, we'll write a quick and dirty tidying routine as well:

```
(define (cleanup str)
 (let (replacements
   (({&amp;}  {&})
    ({&gt;}   {>})
    ({&lt;}   {<})
    ({&nbsp;} { })
    ({&apos;} {'})
    ({&quot;} {"})
    ({&#40;}  {(})
    ({&#41;}  {)})
    ({&#58;}  {:})
    ("\n"      "")))
 (and
  (!= str "")
  (map
   (fn (f) (replace (first f) str (last f)))
   replacements)
  (join (parse str {<.*?>} 4) " "))))

(set 'items (sxml (chop (chop (ref 'title sxml)))))

(dolist (e (ref-all 'item items))
 (set 'entry (items (chop e)))
 (set 'title (first (rest (entry (chop (ref 'title entry))))))
 (set 'body
  (0 18 (cleanup (first (rest (entry (chop
   (ref 'description entry))))))))
 (println title "\n\t" body))
```

```
newLISP newS :: RE: mystery quoting
 Author: rickyboy
newLISP newS :: RE: mystery quoting
 Author: Lutz Pos
newLISP newS :: RE: mystery quoting
 Author: nikus80
newLISP newS :: RE: development release version 9.0.4
...
```

That description section is a bit messy, and would require a bit more code to tidy it up and extract the right amount of detail. But that's another story.

Changing SXML

You can use similar techniques to modify data in XML format. For example, suppose you keep the periodic table of elements in an XML file, and you wanted to change the data about elements' melting points, currently stored in degrees Kelvin, to degrees Celsius. The XML data looks like this:

```
<?xml version="1.0"?>
<PERIODIC_TABLE>
  <ATOM>
  ...
  </ATOM>
  <ATOM>
    <NAME>Mercury</NAME>
    <ATOMIC_WEIGHT>200.59</ATOMIC_WEIGHT>
    <ATOMIC_NUMBER>80</ATOMIC_NUMBER>
    <OXIDATION_STATES>2, 1</OXIDATION_STATES>
    <BOILING_POINT UNITS="Kelvin">629.88</BOILING_POINT>
    <MELTING_POINT UNITS="Kelvin">234.31</MELTING_POINT>
    <SYMBOL>Hg</SYMBOL>
...
```

When the table has been loaded into the symbol `sxml`, using `(set 'sxml (xml-parse xml 15))` (where xml contains the XML source), we want to change each sublist that has the following form:

```
(MELTING_POINT ((UNITS "Kelvin")) "629.88")
```

You can use the **set-ref-all** function to find and replace elements in one expression. First, here's a function to convert a temperature from Kelvin to Celsius:

```
(define (convert-K-to-C n)
 (sub n 273.15))
```

Now the **set-ref-all** function can be called just once to find all references and modify them in place, so that every melting-point is converted to Celsius:

```
(set-ref-all
  '(MELTING_POINT ((UNITS "Kelvin")) *)
  sxml
  (list
    (first $0)
    '((UNITS "Celsius"))
    (string (convert-K-to-C (float (last $0)))))
  match)
```

Here the **match** function searches the SXML list using a wild-card construction (`'(MELTING_POINT ((UNITS "Kelvin")) *)`) to find every occurrence. The replacement function builds a replacement sublist from the matched expression stored in $0. The SXML changes from this:

```
...
(ATOM
    (NAME "Mercury")
    (ATOMIC_WEIGHT "200.59")
    (ATOMIC_NUMBER "80")
    (OXIDATION_STATES "2, 1")
    (BOILING_POINT
        ((UNITS "Kelvin")) "629.88")
    (MELTING_POINT
        ((UNITS "Kelvin")) "234.31")
   ...
```

to this:

```
...
(ATOM
    (NAME "Mercury")
    (ATOMIC_WEIGHT "200.59")
    (ATOMIC_NUMBER "80")
    (OXIDATION_STATES "2, 1")
    (BOILING_POINT
        ((UNITS "Kelvin")) "629.88")
    (MELTING_POINT
        ((UNITS "Celsius")) "-38.84")
   ...
```

Another way of doing this – which you should adopt if you're using version 9.2 of newLISP because **set-ref-all** was introduced in version 9.3 – is to iterate over the list of references obtained with **ref-all**:

```
; get all melting point elements
(set 'melt-pts (ref-all 'MELTING_POINT sxml))
(dolist (e melt-pts)
  ; get last part and convert to float
  (set 'ktemp (float (last (sxml (chop e)))))
  ; convert from K to C
  (set 'ctemp (sub ktemp 273.15))
  ; replace element with new version
  (nth-set (sxml (chop e))
     (list 'MELTING_POINT '((UNITS "Celsius")) (string ctemp))))
  ; remember to convert the temperature value back to a string
```

This `(set 'ktemp (float (last (sxml (chop e)))))` formulation is concise: the **chop** 'goes up' a level in the SXML relative to the address in `e`; the `(sxml ...)` uses implicit addressing to return the data, of which we want the last item, the temperature. Then it's converted to a float, and `ktemp` is now the value of the element's melting point.

XML isn't always as easy to manipulate as this - there are attributes, CDATA sections, and so on.

Outputting SXML to XML

If you want to go the other way and convert a newLISP list to XML, the following function suggests one possible approach. It recursively works through the list:

```
(define (expr2xml expr (level 0))
 (cond
   ((or (atom? expr) (quote? expr))
      (print (dup " " level))
      (println expr))
   ((list? (first expr))
      (expr2xml (first expr) (+ level 1))
      (dolist (s (rest expr)) (expr2xml s (+ level 1))))
   ((symbol? (first expr))
      (print (dup " " level))
      (println "<" (first expr) ">")
      (dolist (s (rest expr)) (expr2xml s (+ level 1)))
      (print (dup " " level))
      (println "</" (first expr) ">"))
   (true
    (print (dup " " level))
    (println "<error>" (string expr) "<error>"))))

(expr2xml sxml)
```

```
 <rss>
   <version>
   0.92
   </version>
  <channel>
   <docs>
   http://backend.userland.com/rss092
   </docs>
   <title>
   newLISP Fan Club
   </title>
   <link>
   http://www.alh.net/newlisp/phpbb/
   </link>
   <description>
     Friends and Fans of newLISP
   </description>
   <managingEditor>
   newlispfanclub-at-excite.com
   </managingEditor>
...
```

Which is – almost – where we started from!

A simple practical example

The following example was originally set in the shipping department of a small business. I've changed the items to be pieces of fruit. The XML data file contains entries for all items sold, and the charge for each. We want to produce a report that lists how many at each price were sold, and the total value.

Here's an extract from the XML data:

```
<FRUIT>
  <NAME>orange</NAME>
  <charge>0</charge>
  <COLOR>orange</COLOR>
</FRUIT>
<FRUIT>
  <NAME>banana</NAME>
  <COLOR>yellow</COLOR>
  <charge>12.99</charge>
</FRUIT>
<FRUIT>
  <NAME>banana</NAME>
  <COLOR>yellow</COLOR>
  <charge>0</charge>
</FRUIT>
```

```
<FRUIT>
  <NAME>banana</NAME>
  <COLOR>yellow</COLOR>
  <charge>No Charge</charge>
</FRUIT>
```

This is the main function that defines and organizes the tasks:

```
(define (work-through-files file-list)
 (dolist (fl file-list)
   (set 'table '())
   (scan-file fl)
   (write-report fl)))
```

Two functions are called: `scan-file`, which scans an XML file and stores the required information in a table, which is going to be some sort of newLISP list, and `write-report`, which scans this table and outputs a report.

The `scan-file` function receives a pathname, converts the file to SXML, finds all the `charge` items (using **ref-all**), and keeps a count of each value. We allow for the fact that some of the free items are marked variously as No Charge or no charge or nocharge:

```
(define (scan-file f)
 (xml-type-tags nil nil nil nil)
 (set 'sxml (xml-parse (read-file f) 15))
 (dolist (r (ref-all charge sxml))                ; all charge items
   (set 'charge-text (last (sxml (chop r))))
   (if (= (lower-case (replace " " charge-text "")) "nocharge")
       (set 'charge charge-text)
       (set 'charge (float charge-text 0 10)))
   (if (set 'result (lookup charge table 1))
      ; if this price already exists in table, add it
       (assoc-set (table charge)  (list charge (+ result 1)))
      ; or create it
       (push (list charge 1) table -1))))
```

The `write-report` function sorts and analyses the table, keeping running totals as it goes:

```
(define (write-report fl)
   (set 'total-items 0 'running-total 0 'total-priced-items 0)
   (sort table (fn (x y) (< (float (x 0)) (float (y 0)))))
   ; heading
   (println " Price    Quantity Subtotal\n")
   (dolist (c table)
     (set 'price (float (first c)))
     (set 'quantity (int (last c)))
     (inc 'total-items quantity)
     (cond
       ; do the No Charge items:
       ((= price nil)
         (println (format " No charge  %8d\n" quantity)))
       ; do 0.00 items
       ((= price 0)
         (println (format " 0.00  %8d\n" quantity)))
       ; do priced items:
       (true
         (begin
          (set 'subtotal (mul price quantity))
          (inc 'running-total subtotal)
          (if (> price 0) (inc 'total-priced-items quantity))
          (println
          (format "%8.2f  %8d %8.2f" price quantity subtotal))))))
   ; totals
   (println
    (format " Total priced: %8d %8.2f" total-priced-items running-total)
"\n")
   (println
    (format "Grand Total  %8d %8.2f" total-items running-total)))
```

The report requires a bit more fiddling about than the `scan-file` function, particularly as the user wanted – for some reason – the 0 and no charge items to be kept separate.

```
 Price        Quantity Subtotal
   No charge      42
   0.00           79
   0.11            1     0.11
   0.29            1     0.29
   1.89           39    73.71
   1.99           17    33.83
   2.99           18    53.82
   12.99          22   285.78
   17.99           1    17.99
   Total priced:  99   465.53
   Grand Total   220   465.53
```

14 The internet

HTTP and networking

Most networking tasks are possible with newLISP's networking functions:

- **base64-dec** decode a string from BASE64 format
- **base64-enc** encode a string to BASE64 format
- **delete-url** delete a URL
- **get-url** read a file or page from the web
- **net-accept** accept a new incoming connection
- **net-close** close a socket connection
- **net-connect** connect to a remote host
- **net-error** return the last error
- **net-eval** evaluate expressions on multiple remote newLISP servers
- **net-listen** listen for connections to a local socket
- **net-local** local IP and port number for a connection
- **net-lookup** the name for an IP number
- **net-peek** number of characters ready to read
- **net-peer** remote IP and port for a net-connect
- **net-ping** send a ping packet (ICMP echo request) to one or more addresses
- **net-receive** read data on a socket connection
- **net-receive-from** read a UDP datagram on an open connection
- **net-receive-udp** read a UDP datagram on and closes connection
- **net-select** check a socket or list of sockets for status
- **net-send** send data on a socket connection
- **net-send-to** send a UDP datagram on an open connection
- **net-send-udp** send a UDP datagram and closes connection
- **net-service** translate a service name to a port number
- **net-sessions** return a list of currently open connections
- **post-url** post info to a URL address
- **put-url** upload a page to a URL address.
- **xml-error** return last XML parse error

- **xml-parse** parse an XML document
- **xml-type-tags** show or modify XML type tags

With these networking functions you can build all kinds of network-capable applications. With functions like **net-eval** you can start newLISP as a daemon on a remote computer and then use on a local computer to send newLISP code across the network for evaluation.

Accessing web pages

Here's a very simple example using **get-url**. Given the URL of a web page, obtain the source and then use **replace** and its list-building ability to generate a list of all the JPEG images on that page:

```
(set 'the-source (get-url "http://www.apple.com"))
(replace {src="(http\S*?jpg)"} the-source (push $1 images-list -1) 0)
(println images-list)
```

```
("http://images.apple.com/home/2006/images/ipodhifititle20060228.jpg"
"http://images.apple.com/home/2006/images/ipodhifitag20060228.jpg"
"http://images.apple.com/home/2006/images/macminiwings20060228.jpg"
"http://images.apple.com/home/2006/images/macminicallouts20060228.jpg"
"http://images.apple.com/home/2006/images/ipodhifititle20060228.jpg"
"http://images.apple.com/home/2006/images/ipodhifitag20060228.jpg")
```

A simple IRC client

The following code implements a simple IRC (Internet Relay Chat) client, and it shows how the basic network functions can be used. The script logs in to the server using the given username, and joins the #newlisp channel. Then the script divides into two threads: the first thread displays any channel activity in a continuous loop, while the second thread waits for input at the console. The only communication between the two threads is through the shared `connected` flag.

```
(set 'server (net-connect "irc.freenode.net" 6667))
(net-send server "USER newlispnewb 0 * :XXXXXXX\r\n")
(net-send server "NICK newlispnewb \r\n")
(net-send server "JOIN #newlisp\r\n")

(until (find "366" buffer)
  (net-receive server 'buffer 8192 "\n")
  (print buffer))

(set 'connected (share))
(share connected true)

(fork
    (while (share connected)
      (cond
        ((net-select server "read" 1000) ; read the latest
            (net-receive server 'buffer 8192 "\n")
            ; ANSI colouring: output in yellow then switch back
            (print "\n\027[0;33m" buffer "\027[0;0m"))
        ((regex {^PING :(.*)\r\n} buffer) ; play ping-pong
            (net-send server (append "PONG :" (string $1 ) "\r\n"))
            (sleep 5000))
        ((net-error) ; error
            (println "\n\027[0;34m" "UH-OH: " (net-error) "\027[0;0m")
            (share connected nil)))
     (sleep 1000)))
```

```
(while (share connected)
   (sleep 1000)
   (set 'message (read-line))
   (cond
     ((starts-with message "/")  ; a command?
          (net-send server (append (rest message) "\r\n"))
          (if
            (net-select server "read" 1000)
            (begin
                (net-receive server 'buffer 8192 "\n")
                (print "\n\027[0;35m" buffer "\027[0;0m"))))
     ((starts-with message "quit") ; quit
            (share connected nil))
     (true  ; send input as message
          (net-send server (append "PRIVMSG #newlisp :" message
"\r\n")))))

(println "finished; closing server")
(close server)
(exit)
```

15 More examples

This section contains some simple examples of newLISP in action. You can find plenty of really good newLISP code on the web and in the standard newLISP distribution.

On your own terms

You might find that you don't like the names of some of the newLISP functions. You can use **constant** and **global** to assign another symbol to the function:

```
(constant (global 'nth-set!) nth-set)
```

You can now use `nth-set!` instead of **nth-set**. There's no speed penalty for doing this.

It's also possible to define your own alternatives to built-in functions. For example, earlier we defined a context and a default function that did the same job as **println**, but kept a count of how many characters were output. For this code to be evaluated rather than the built-in code, do the following.

First, define the function:

```
(define (Output:Output)
 (if Output:counter
   (inc 'Output:counter (length (string (args))))
   (set 'Output:counter 0))
 (map print (args))
 (print "\n"))
```

Keep the original newLISP version of **println** available by defining an alias for it:

```
(constant (global 'newLISP-println) println)
```

Assign the **println** symbol to your Output function:

```
(constant (global 'println) Output)
```

Now you can use **println** as usual:

```
(for (i 1 10)
 (println (+ 1 i)))
```

```
1
4
9
16
25
36
49
64
```

```
(map println '(1 2 3 4 5))
```

```
1
2
3
4
5
```

And it appears to do the same job as the original function. But now you can also make use of the new features of the alternative **println** that you defined:

```
Output:counter
;-> 36
```

In case you've been counting carefully – the counter has been counting the length of the arguments supplied to the Output function. These include the parentheses, of course...

Using a SQLite database

Sometimes it's easier to use existing software rather than write all the routines yourself (even though it can be fun designing something from the beginning). For example, you can save much time and effort by using an existing database engine such as SQLite instead of building custom data structures and database access functions. Here's how you might use the SQLite database engine with newLISP.

Suppose you have a set of data that you want to analyse. For example, I've found a list of information about the elements in the periodic table. We'll start off by storing this in a symbol:

```
(set 'elements
  [text]1 1.0079 Hydrogen H -259 -253 0.09 0.14 1776 1 13.5984
  2 4.0026 Helium He -272 -269 0 0 1895 18 24.5874
  3 6.941 Lithium Li 180 1347 0.53 0 1817 1 5.3917
  ...
  108 277 Hassium Hs 0 0 0 0 1984 8 0
  109 268 Meitnerium Mt 0 0 0 0 1982 9 0[/text])
```

The columns here are Atomic Weight, Melting Point, Boiling Point, Density, Percentage in the earth's crust, Year of discovery, Group, and Ionization energy. I've used 0 to mean Not Applicable (not a very good choice, as it turns out).

To make use of newLISP's SQLite module, use the following line:

```
(load "/usr/share/newlisp/modules/sqlite3.lsp")
```

This loads in the newLISP source file which contains the SQLite interface. It also creates a new context called `sql3`, containing functions and symbols for working with SQLite databases.

Next, we want to create a new database or open an existing one:

```
(if (sql3:open "periodic_table")
  (println "database opened/created")
  (println "problem: " (sql3:error)))
```

This creates a new SQLite database file called `periodic_table`, and opens it. If the file already exists, it will be opened ready for use. We don't have to refer to this database again, because newLISP's SQLite library routines maintain a current database in the `sql3` context. If the **open** function fails, the most recent error stored in `sql3:error` is printed instead.

We've just created this database, so the next step is to create a table. First, though, we'll define a symbol containing a string of column names and the SQLite data types each one should use. We don't have to do this, but you have to write it down somewhere, so instead of the back of an envelope, write it in a newLISP symbol:

```
(set 'column-def "number INTEGER, atomic_weight FLOAT,
element TEXT, symbol TEXT, mp FLOAT, bp FLOAT, density
FLOAT, earth_crust FLOAT, discovered INTEGER, egroup
INTEGER, ionization FLOAT")
```

Now I can make a function that creates the table:

```
(define (create-table)
 (if (sql3:sql (string "create table t1 (" column-def ")"))
    (println "created table ... OK")
    (println "problem " (sql3:error))))
```

It's easy because we've just created the `column-def` symbol in exactly the right format! This function uses the `sql3:sql` function to create a table called `t1`.

We want one more function: one that fills the SQLite table with the data stored in the list elements. It's not a pretty function, but it does the job, and it only has to be called once.

```
(define (init-table)
 (dolist (e (parse elements "\n" 0))
 (set 'line (parse e))
 (if (sql3:sql
Âă (format "insert into t1 values
(%d,%f,'%s','%s',%f,%f,%f,%f,%d,%d,%f);"
    (int (line 0))
    (float (line 1))
    (line 2)
    (line 3)
    (float (line 4))
    (float (line 5))
    (float (line 6))
    (float (line 7))
    (int (line 8))
    (int (line 9))
    (float (line 10))))
  ; success
  (println "inserted element " e)
  ; failure
  (println (sql3:error) ":" "problem inserting " e))))
```

This function calls **parse** twice. The first **parse** breaks the data into lines. The second **parse** breaks up each of these lines into a list of fields. Then we can use **format** to enclose the value of each field in single quotes, remembering to change the strings to integers or floating-point numbers (using **int** and **float**) according to our column definitions.

It's now time to build the database:

```
(if (not (find "t1" (sql3:tables)))
 (and
   (create-table)
   (init-table)))
```

– if the `t1` table doesn't exist in a list of tables, the functions that create and fill it are called.

Querying the data

The database is now ready for use. But first let's write a simple utility function to make queries easier:

```
(define (query sql-text)
 (set 'sqlarray (sql3:sql sql-text))    ; results of query
 (if sqlarray
   (map println sqlarray)
   (println (sql3:error) " query problem ")))
```

This function submits the supplied text, and either prints out the results (by mapping **println** over the results list) or displays the error message.

Here are a few sample queries.

Find all elements discovered before 1900 that make up more than 2% of the earth's crust, and display the results sorted by their discovery date.

```
(query
 "select element,earth_crust,discovered
 from t1
 where discovered < 1900 and earth_crust > 2
 order by discovered")
```

```
("Iron" 5.05 0)
("Magnesium" 2.08 1755)
("Oxygen" 46.71 1774)
("Potassium" 2.58 1807)
("Sodium" 2.75 1807)
("Calcium" 3.65 1808)
("Silicon" 27.69 1824)
("Aluminium" 8.07 1825)
```

When were the noble gases discovered (they are in group 18)?

```
(query
 "select symbol, element, discovered
 from t1
 where egroup = 18")
```

```
("He" "Helium" 1895)
("Ne" "Neon" 1898)
("Ar" "Argon" 1894)
("Kr" "Krypton" 1898)
("Xe" "Xenon" 1898)
("Rn" "Radon" 1900)
```

What are the atomic weights of all elements whose symbols start with A?

```
(query
 "select element,symbol,atomic_weight
 from t1
 where symbol like A%
 order by element")
```

```
("Actinium" "Ac" 227)
("Aluminium" "Al" 26.9815)
("Americium" "Am" 243)
("Argon" "Ar" 39.948)
("Arsenic" "As" 74.9216)
("Astatine" "At" 210)
("Gold" "Au" 196.9665)
("Silver" "Ag" 107.8682)
```

It's elementary, my dear Watson! Perhaps the scientists out there can supply some examples of more scientifically interesting queries?

You can find MySQL newLISP modules on the net.

Simple countdown timer

Next is a simple countdown timer that runs as a command-line utility. This example shows some techniques for accessing the command-line arguments in a script.

To start counting down, you type the command (the name of the newLISP script) followed by a duration. The duration can be in seconds; minutes and seconds; hours, minutes, and seconds; or even days, hours, minutes, and seconds, separated by colons. It can also be any newLISP expression.

```
> countdown 30

Started countdown of 00d 00h 00m 30s at 2006-09-05 15:44:17
Finish time:       2006-09-05 15:44:47
Elapsed: 00d 00h 00m 11s Remaining: 00d 00h 00m 19s
```

or:

```
> countdown 1:30

Started countdown of 00d 00h 01m 30s at 2006-09-05 15:44:47
Finish time:       2006-09-05 15:46:17
Elapsed: 00d 00h 00m 02s Remaining: 00d 00h 01m 28s
```

or:

```
> countdown 1:00:00

Started countdown of 00d 01h 00m 00s at 2006-09-05 15:45:15
Finish time:       2006-09-05 16:45:15
Elapsed: 00d 00h 00m 02s Remaining: 00d 00h 59m 58s
```

or:

```
> countdown 5:04:00:00

Started countdown of 05d 04h 00m 00s at 2006-09-05 15:45:47
Finish time:       2006-09-10 19:45:47
Elapsed: 00d 00h 00m 05s Remaining: 05d 03h 59m 55s
```

Alternatively, you can supply a newLISP expression instead of a numerical duration. This might be a simple calculation, such as the number of seconds in π minutes:

```
> countdown "(mul 60 (mul 2 (acos 0)))"

Started countdown of 00d 00h 03m 08s at 2006-09-05 15:52:49
Finish time:       2006-09-05 15:55:57
Elapsed: 00d 00h 00m 08s Remaining: 00d 00h 03m 00s
```

or, more usefully, a countdown to a specific moment in time, which you supply by subtracting the time now from the time of the target:

```
> countdown "(- (date-value 2006 12 25) (date-value))"

Started countdown of 110d 08h 50m 50s at 2006-09-05 16:09:10
Finish time:        2006-12-25 00:00:00
Elapsed: 00d 00h 00m 07s Remaining: 110d 08h 50m 43s
```

– in this example we’ve specified Christmas Day using **date-value**, which returns the number of seconds since 1970 for specified dates and times.

The evaluation of expressions is done by **eval-string**, and here it’s applied to the input text if it starts with "(" – generally a clue that there’s a newLISP expression around! Otherwise the input is assumed to be colon-delimited, and is split up by **parse** and converted into seconds.

The information is taken from the arguments given on the command line, and extracted using **main-args**, which is a list of the arguments that were used when the program was run:

```
(main-args 2)
```

This fetches argument 2; argument 0 is the name of the newLISP program, argument 1 is the name of the script, so argument 2 is the first string following the `countdown` command.

Save the file as `countdown`, and make it executable.

```
#!/usr/bin/newlisp
(if (not (main-args 2))
 (begin
   (println "usage: countdown duration [message]\n
    specify duration in seconds or d:h:m:s")
   (exit)))

(define (set-duration)
; convert input to seconds
  (if (starts-with duration-input "(")
      (set ’duration-input (string (eval-string duration-input))))
  (set ’duration
   (dolist (e (reverse (parse duration-input ":")))
    (if (!= e)
     (inc ’duration (mul (int e) (’(1 60 3600 86400) $idx)))))))

(define (seconds->dhms s)
; convert seconds to day hour min sec display
  (letn
    ((secs (mod s 60))
     (mins (mod (div s 60) 60))
     (hours (mod (div s 3600) 24))
     (days (mod (div s 86400) 86400)))
   (format "%02dd %02dh %02dm %02ds" days hours mins secs)))

(define (clear-screen-normans-way)
; clear screen using codes - thanks to norman on newlisp forum :-)
 (println "\027[H\027[2J"))

(define (notify announcement)
; MacOS X-only code. Change for other platforms.
  (and
   (= ostype "OSX")
   ; beep thrice
   (exec (string {osascript -e ’tell application "Finder" to beep 3’}))

   ; speak announcment:
   (if (!= announcement nil)
     (exec (string {osascript -e ’say "} announcement {"’})))

   ; notify using Growl:
   (exec (format "/usr/local/bin/growlnotify %s -m \"Finished count down
\""
      (date (date-value) 0 "%Y-%m-%d %H:%M:%S")))))
```

```
(set 'duration-input (main-args 2) 'duration 0)

(set-duration)

(set 'start-time (date-value))
(set 'target-time (add (date-value) duration))

(set 'banner
  (string  "Started countdown of "
    (seconds->dhms duration)
    " at "
    (date start-time 0 "%Y-%m-%d %H:%M:%S")
    "\nFinish time:                              "
    (date target-time 0 "%Y-%m-%d %H:%M:%S")))

(while (<= (date-value) target-time)
  (clear-screen-normans-way)
  (println
     banner
     "\n\n"
    "Elapsed: "
    (seconds->dhms (- (date-value) start-time ))
    " Remaining: "
    (seconds->dhms (abs (- (date-value) target-time))))
  (sleep 1000))

(println
  "Countdown completed at "
  (date (date-value) 0
  "%Y-%m-%d %H:%M:%S") "\n")

; do any notifications here
(notify (main-args 3))

(exit)
```

Editing text files in folders and hierarchies

Here's a simple function that updates some text date stamps in every file in a folder, by looking for enclosing tags and changing the text between them. For example, you might have a pair of tags holding the date the file was last edited, such as <last-edited> and </last-edited>.

```
(define (replace-string-in-files start-str end-str repl-str folder)
  (set 'path (real-path folder))
  (set 'file-list (directory folder {^[^.]}))
  (dolist (f file-list)
    (println "processing file " f)
    (set 'the-file (string path "/" f))
    (set 'page (read-file the-file))
    (replace
      (append start-str "(.*?)" end-str)  ; pattern
      page                                ; text
      (append start-str repl-str end-str) ; replacement
      0)                                  ; regex option number
    (write-file the-file page)
   ))
```

which can be called like this:

```
(replace-string-in-files
 {<last-edited>} {</last-edited>}
 (date (date-value) 0 "%Y-%m-%d %H:%M:%S")
 "/Users/me/Desktop/temp/")
```

The `replace-string-in-files` function accepts a folder name. The first task is to extract a list of suitable files – we're using `directory` with the regular expression `{^[^.]}` to exclude all files that start with a dot. Then, for each file, the contents are loaded into a symbol, the **replace** function replaces text enclosed in the specified strings, and finally the modified text is saved back to disk. To call the function, specify the start and end tags, followed by the text and the folder name. In this example we're just using a simple ISO date stamp provided by **date** and **date-value**.

Recursive version

Suppose we now wanted to make this work for folders within folders within folders, ie to traverse a hierarchy of files, changing every file on the way down. To do this, re-factor the `replace-string` function so that it works on a passed pathname. Then write a recursive function to look for folders within folders, and generate all the required pathnames, passing each one to the `replace-string` function. This re-factoring might be a good thing to do anyway: it makes the first function simpler, for one thing.

```
(define (replace-string-in-file start-str end-str repl-str pn)
 (println "processing file " pn)
 (set 'page (read-file pn))
 (replace
  (append start-str "(.*?)" end-str)   ; pattern
  page                                 ; text
  (append start-str repl-str end-str)  ; replacement
  0)                                   ; regex option number
 (write-file pn page))
```

Next for that recursive tree-walking function. This looks at each normal entry in a folder/directory, and tests to see if it's a directory (using **directory?**). If it is, the `replace-in-tree` function calls itself and starts again at the new location. If it isn't, the pathname of the file is passed to the `replace-string-in-file` function.

```
(define (replace-in-tree dir s e r)
 (dolist (nde (directory dir {^[^.]}))
   (if (directory? (append dir nde))
    (replace-in-tree (append dir nde "/") s e r)
    (replace-string-in-file (append dir nde) s e r))))
```

To change a tree-full of files, call the function like this:

```
(replace-in-tree
  {/Users/me/Desktop/temp/}
  {<last-edited>}
  {</last-edited>}
  (date (date-value) 0 "%Y-%m-%d %H:%M:%S"))
```

It's important to test these things in a scratch area first; a small mistake in your code could make a big impact on your data. Caveat newLISPer!

Talking to other applications (MacOS X example)

newLISP provides a good environment for gluing together features found in application programs with their own scripting languages. It's fast and small enough to 'keep out of the way' of the other components of a scripted solution, and it's easy to process information as it passes through your workflow.

Here is an example of how you can use a newLISP script to send non-newLISP scripting commands to applications. The task is to construct a circle in Adobe Illustrator, given three points on the circumference.

The solution is in three parts. First, we obtain the coordinates of the selection from the application. Next we calculate the radius and centre point of the circle that passes through these points. Finally, we can draw the circle. The first and final parts use AppleScript, which is run using the `osascript` command, because Adobe Illustrator doesn't understand any other scripting language (on Windows, you use Visual Basic rather than AppleScript).

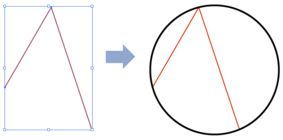


Figure 15.1 Using a newLISP script in Adobe Illustrator

The calculations and general interfacing are carried out using newLISP. This can often be a better solution than using native AppleScript, because newLISP offers many powerful string and mathematical functions that can't be found in the default AppleScript system. For example, if I want to use trigonometry I would have to install an extra component – AppleScript doesn't provide any trig functions at all.

The newLISP script can sit in the Scripts menu on the menu bar (by putting it in the Library>Scripts>Applications>Adobe Illustrator folder, even if it's just a text file, rather than an AppleScript), ready for selection while you're working in Illustrator. To use it, just select a path with at least three points, then choose the script. The first three points define the location for the new circle.

```
#!/usr/bin/newlisp

; http://cgafaq.info/wiki/Circle_Through_Three_Points
; given three points, draw a circle through them

(set 'pointslist
  (exec
    (format [text]osascript  -e 'tell application "Adobe Illustrator 10"
  tell front document
    set s to selection
    repeat with p in s
    set firstItem to p
    set pathinfo to entire path of firstItem
    set pointslist to ""
    repeat with p1 in pathinfo
    set a to anchor of p1
    set pointslist to pointslist & " " & item 1 of a
    set pointslist to pointslist & " " & item 2 of a
    end repeat
    end repeat
  end tell
end tell
pointslist'
[/text])))

; cleanup
(set 'points
  (filter float?
    (map float (parse (first pointslist) { } 0))))

(set  'ax (points 0)
      'ay (points 1)
      'bx (points 2)
      'by (points 3)
      'cx (points 4)
      'cy (points 5))

(set  'A (sub bx ax)
      'B (sub by ay)
      'C (sub cx ax)
      'D (sub cy ay)
      'E (add
          (mul A (add ax bx))
          (mul B (add ay by)))
      'F (add
          (mul C (add ax cx))
          (mul D (add ay cy)))
      'G (mul 2
            (sub
              (mul A (sub cy by))
              (mul B (sub cx bx)))))

(if (= G 0) ; collinear, forget it
  (exit))

(set  'centre-x (div (sub (mul D E) (mul B F)) G)
      'centre-y (div (sub (mul A F) (mul C E)) G)
      'r
        (sqrt
          (add
            (pow (sub ax centre-x))
            (pow (sub ay centre-y)))))

; we have coords of centre and the radius
; in centre-x, centre-y, and r
; Illustrator bounds are left-x, top-y, right-x, bottom-y
; ie centre-x - r, centre-y + r, centre-x + r, centre-y -r

(set 'bounds-string
  (string "{" (sub centre-x r) ", "
   (add centre-y r) ", "
   (add centre-x r) ", "
   (sub centre-y r) "}"))

(set 'draw-circle
  (exec (format [text]osascript  -e 'tell application "Adobe Illustrator
10"
  tell front document
    set e to make new ellipse at beginning with properties {bounds:%s}
  end tell
end tell
'
[/text] bounds-string)))
(exit)
```

There's hardly any error handling in this script! More should definitely be added to the first stage (because the selection might not be suitable for subsequent processing).

16 The debugger

This section takes a brief look at the built-in debugger.

The key function is **trace**. To start and stop the debugger, use true or nil:

```
(trace true)  ; start debugging
(trace nil)   ; stop debugging
```

Used on its own, it returns true if the debugger is active.

The **trace-highlight** command lets you control the display of the expression that's currently being evaluated, and some of the prompts. I'm using a VT100-compatible terminal, so I can use weird escape sequences to set the colours. Type this in at the newLISP prompt:

```
(trace-highlight "\027[0;37m" "\027[0;0m")
```

But since I can never remember this, it's in my .init.lsp file, which runs if /usr/share/newlisp/init.lsp does when you start newLISPİf you can't use these sequences, you can use plain strings instead.

The other debugging function is **debug**. This is really just a short-cut for switching tracing on, running a function in the debugger, and then switching tracing off again. So, suppose we want to run a file called old-file-scanner.lsp, which contains the following code:

```
(define (walk-tree folder)
  (dolist (item (directory folder))
   (set 'item-name (string folder "/" item))
   (if (and (not (starts-with item ".")) (directory? item-name))
       ; folder
       (walk-tree item-name)
       ; file
       (and
        (not (starts-with item "."))
        (set 'f-name (real-path item-name))
        (set 'mtime (file-info f-name 6))
        (if
          (> (- (date-value) mtime) (* 5 365 24 60 60)) ; non-leap years
:)
            (push (list mtime item-name) results))))))
(set 'results '())
(walk-tree {/usr/share})
(map (fn (i) (println (date (first i)) { } (last i))) (sort results))
```

This scans a directory and subdirectories for files previously modfied 5 or more years ago. (Notice that the file ends with expressions that will be evaluated immediately the file is loaded.) First, switch tracing on:

```
(trace true)
```

Then load and start debugging:

```
(load {old-file-scanner.lsp})
```

Or, instead of those two lines, type this one:

```
(debug (load {old-file-scanner.lsp}))
```

Either way, you'll see the first expression in the `walk-tree` function highlighted, awaiting evaluation.

Now you can press the s, n, or c keys ('step', 'next', and 'continue') to proceed through your functions: 'step' evaluates every expression, and steps into other functions as they are called;

’next’ evaluates everything until it reaches the next expression at the same level; and ’continue’ runs through without stopping again.

If you’re clever, you can put a `(trace true)` expression just before the place where you want to start debugging. If possible, newLISP will stop just before that expression and show you the function it’s just about to evaluate. In this case, you can start executing the script with a simple (load...) expression - don’t use **debug** if you want to skip over the preliminary parts of the script. I think newLISP usually prefers to enter the debugger at the start of a function or function call - you probably won’t be able to drop in to a function part way through. But you might be able to organize things so that you can do something similar.

Here, for example, is a script that drops into the debugger halfway through a loop:

```
(set ’i 0)
(define (f1)
  (inc ’i))

(define (f2)
  (dotimes (x 100)
    (f1)
    (if (= i 50) (trace true))))

(f2)
```

```
> (load {simpleloop.lsp})

-----
(define (f1 )
  (inc ’i))

[-> 5 ] s|tep n|ext c|ont q|uit > i

50

[-> 5 ] s|tep n|ext c|ont q|uit >
```

Notice how the function `f1` appeared - you don’t get a chance to see anything in `f2`. At the debugger prompt, you can type in any newLISP expression, and evaluate any functions. newLISP seems to be happy to let you change the values of some symbols, too. You should be able to change the loop variable, for example. But don’t redefine any functions ... if you try to pull the rug from underneath newLISP’s feet you’ll probably succeed in making it fall over.

The source code displayed by the debugger doesn’t include comments, so if you want to leave yourself helpful remarks - or inspiration - when looking at your code, use text strings rather than comments:

```
(define (f1)
  [text]This sentence will appear in the debugger.[/text]
  ; But this sentence won’t.
  (inc ’i))
```

17 Graphical interfaces

Introduction

With newLISPyou can easily build graphical interfaces for your applications. This introductory document is long enough already, so I won't describe the newLISP-GS feature set in any detail. But there's room for a short example to give you a taste of how it works.

The basic ideas of newLISP-GS are *containers*, *widgets*, *events*, and *tags*. Your application consists of containers, which hold widgets and other containers. By giving everything a tag (a symbol), you can control them easily. And when the user of the application clicks, presses, and slides things, events are sent back to newLISP-GS, and you write code to handle each event.

A simple application

To introduce the basic ideas, this chapter shows how easy it is to build a simple application, a colour mixer:

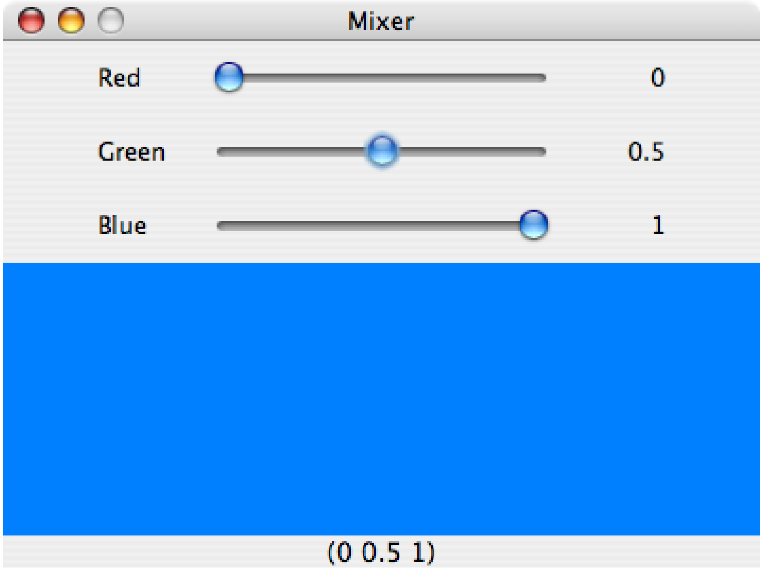


Figure 17.1 a simple colour mixer

You can move the sliders around to change the colour of the central area of the window. The colour components (numbers between 0 and 1 for red, green, and blue), are shown in a text string at the bottom of the window.

In newLISP-GS the contents of a container are arranged depending on the type of layout manager you choose – at present you can have *flow*, *grid*, or *border* layouts.

The following diagram shows the structure of the application's interface. The primary container, which in this case is a frame called 'Mixer', is filled with other containers and widgets. The top area of the window, containing the sliders, consists of a panel called 'SliderPanel', which in turn is filled with three panels, one for each slider, called 'RedPanel', 'GreenPanel', and 'BluePanel'. Below, the middle area holds a canvas called 'Swatch' to show the colour, and at the bottom area there's a text label, called 'Value', displaying the RGB values as text. Each area is laid out using a different layout manager.

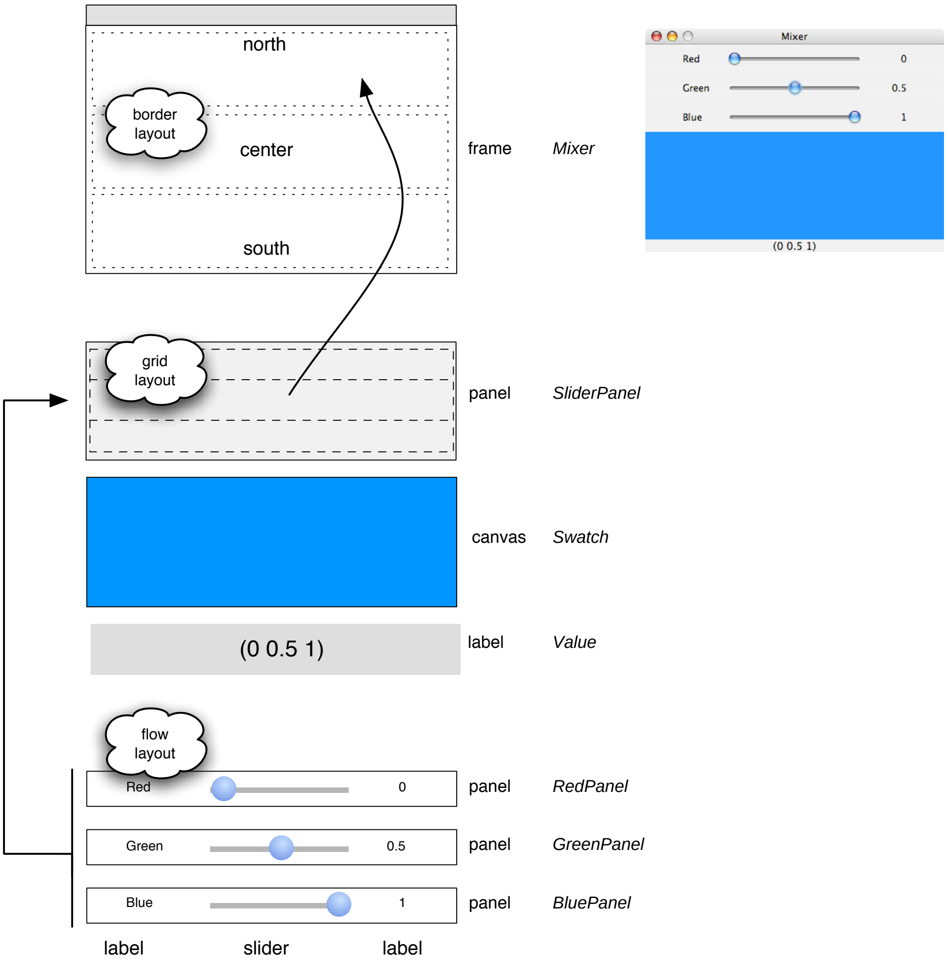


Figure 17.2 interface structure for the mixer application

There's just a single handler required. This is assigned to the sliders, and it's triggered whenever a slider is moved.

The first step is to load the newLISP-GS module:

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

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

This provides all the objects and functions required, in a context called `gs`.

The graphics system is initialized with a single function:

```
(gs:init)
```

The various parts of the interface can be added one by one. First I define the main window, and choose the border layout. Border layouts let you place each component into one of five zones, labelled "north", "west", "center", "east" and "south".

```
(gs:frame 'Mixer 200 200 400 300 "Mixer")
(gs:set-resizable 'Mixer nil)
(gs:set-border-layout 'Mixer)
```

The top panel to hold the sliders can now be added. I want the sliders to be stacked vertically, so I'll use a grid layout of 3 rows and 1 column:

```
(gs:panel 'SliderPanel)
(gs:set-grid-layout 'SliderPanel 3 1)
```

Each of the three colour panels, with its companion labels and slider, is defined. The sliders are assigned the `slider-handler` function. I can write that when I've finished defining the interface.

```
(gs:panel  'RedPanel)
(gs:panel  'GreenPanel)
(gs:panel  'BluePanel)

(gs:label  'Red   "Red"   "left" 50 10 )
(gs:label  'Green "Green" "left" 50 10 )
(gs:label  'Blue  "Blue"  "left" 50 10 )

(gs:slider 'RedSlider   'slider-handler "horizontal" 0 100 0)
(gs:slider 'GreenSlider 'slider-handler "horizontal" 0 100 0)
(gs:slider 'BlueSlider  'slider-handler "horizontal" 0 100 0)

(gs:label  'RedSliderStatus   "0"  "right" 50 10)
(gs:label  'GreenSliderStatus "0"  "right" 50 10)
(gs:label  'BlueSliderStatus  "0"  "right" 50 10)
```

The `gs:add-to` function adds components to a container, using the layout that's been assigned to it. If no layout has been assigned, the flow layout, a simple sequential layout, is used. Specify the target container first, then give the components to be added. So the objects tagged with 'Red, 'RedSlider, and 'RedSliderStatus are added one by one to the 'RedPanel container. When the three panels have been done, they can be added to the SliderPanel:

```
(gs:add-to 'RedPanel 'Red 'RedSlider 'RedSliderStatus)
(gs:add-to 'GreenPanel 'Green 'GreenSlider 'GreenSliderStatus)
(gs:add-to 'BluePanel 'Blue 'BlueSlider 'BlueSliderStatus)

(gs:add-to 'SliderPanel 'RedPanel 'GreenPanel 'BluePanel)
```

You can draw all kinds of graphics on a canvas, although for this application I'm just going to use a canvas as a swatch, a single area of colour:

```
(gs:canvas 'Swatch)

(gs:label 'Value "")
(gs:set-font 'Value "Sans Serif" 16)
```

Now the three main components – the slider panel, the colour swatch, and the label for the value – can be added to the main frame. Because I assigned the border layout to the frame, I can place each component using directions:

```
(gs:add-to 'Mixer 'SliderPanel "north" 'Swatch "center" 'Value "south")
```

We haven't used the east and west areas.

By default, frames and windows start life as invisible, so now is a good time to make our main frame visible:

```
(gs:set-visible 'Mixer true)
```

That completes the application's structure. Now some initialization is required, so that something sensible appears when the application launches:

```
(set 'red 0 'green 0 'blue 0)
(gs:set-color 'Swatch (list red green blue))
(gs:set-text  'Value (string (list red green blue)))
```

Finally, I mustn’t forget to write the handler code for the sliders. The handler is passed the id of the object which generated the event, and the slider’s value. The code converts the value, an integer less than 100, to a number between 0 and 1. Then the **set-color** function can be used to set the colour of the canvas to show the new mixture.

```
(define (slider-handler id value)
  (cond
     ((= id "MAIN:RedSlider")
        (set ’red (div value 100))
        (gs:set-text ’RedSliderStatus (string red)))
     ((= id "MAIN:GreenSlider")
       (set ’green (div value 100))
        (gs:set-text ’GreenSliderStatus (string green)))
     ((= id "MAIN:BlueSlider")
       (set ’blue (div value 100))
       (gs:set-text ’BlueSliderStatus (string blue)))
     )
  (gs:set-color ’Swatch (list red green blue))
  (gs:set-text  ’Value (string (list red green blue))))
```

Only one more line is necessary and we’ve finished. The `gs:listen` function listens for events and dispatches them to the handlers. It runs continuously, so you don’t have to do anything else.

```
(gs:listen)
```

This tiny little application has barely scratched the surface of newLISP-GS, so take a look at the documentation and have a go!

Index