newLISP An Interactive Tutorial

This document was reformatted for HTML and a few corrections and
updates made by Rick Hanson in May 2006, cryptorick@gmail.com.
updates for v.10.0 by L.M. January 2008, December 2011.

You can download and install newLISP from www.newLISP.org.

Please send any comments or questions regarding this tutorial to jsmall@atlaol.net.

Hello World!

With newLISP installed on your system, at the shell command line prompt enter `newlisp` to start the REPL (Read, Eval, Print Loop).

On Linux, your console would look something like this:

```
$ newlisp
> _
```

And on Windows platforms, it would look something like this:

```
c:\> newlisp
> _
```

After starting up, newLISP responds with a prompt.

```
> _
```

Enter the expression below in order to print "Hello World!" on the console.

```
> (println "Hello World!")
```

newLISP prints the value of the expression entered at the REPL prompt before looping and prompting for more input.

```
> (println "Hello World!")
Hello World!
"Hello World!"
> _
```

Why did it print twice?

The `println` function prints the first line, i.e.

```
Hello World!
```

to the console as a side effect of calling the function.

The `println` function then returns the "Hello World!" string, i.e. its last argument, to the REPL which in turn prints the second line, i.e.

```
"Hello World!"
```

The REPL will evaluate any expression and not just function calls.

```
> "Hello World!"
"Hello World!"
> _
```

If you enter the string expression `"Hello World!"`, as shown above, it simply returns itself as would any literal expression such as literal numbers.

```
> 1
1
> _
```

By now you may be turned off by the parentheses. If you are coming to newLISP from a mainstream computer language, it would seem more natural to write a function call like this:

```
println("Hello World!")
```

You'll just have to take my word for it in time you will actually prefer

```
(println "Hello World!")
```

to

```
println("Hello World!")
```

for reasons that cannot be adequately explained until you have seen a lot more examples of symbolic list processing.

Source Code and Data are Interchangeable

Lisp stands for List Processor. Since lists are used to represent both code and data in Lisp they are essentially interchangable.

The previous `println` expression is really a list with two elements.

```
(println "Hello World!")
```

The first element is

`println`

and the last element is

`"Hello World!"`

Lisp always interprets a list as a function call unless you quote it, thereby indicating that it should be taken literally as a symbolic expression, i.e. data.

```
> '(println "Hello World!")
(println "Hello World!")
> _
```

A symbolic expression can be evaluated as source code, however.

```
> (eval '(println "Hello World!"))
Hello World!
"Hello World!"
> _
```

A Lisp program can literally extend itself at run time by building lists of data and then executing them as source code!

```
> (eval '(eval '(println "Hello World!")))
Hello World!
"Hello World!"
> _
```

Actually the `'` is syntactical sugar for `quote`.

```
> (quote (println "Hello World!"))
(println "Hello World!")
> _
```

Think of `quote` as taking its argument literally, i.e. symbolically.

```
> 'x
x
> (quote x)
' x
> '(1 2 three "four")
(1 2 three "four")
> _
```

Symbols, such as `x` and `three` above, and symbolic lists play a vitally important role in AI, i.e. Artifical Intelligence. This tutorial will not cover AI topics. However, once you know how to program in Lisp, you'll be able to readily follow the ubiquitous Lisp examples found in most textbooks on AI.

Consider the following example.

```
> 'Hello
Hello
> "Hello"
"Hello"
> _
```

The symbol `'Hello` is not the same as the literal string `"Hello"`. Now you can appreciate why the REPL prints the double quotes to indicate a literal string, thereby distinguishing it from a symbol having the same letters.

Function arguments

The `println` function can also take a variable number of arguments.

```
> (println "Hello" " World!")
Hello World!
" World!"
> _
```

The arguments are merely concatenated on the output stream while the last argument is returned as the value of the function call.

Typically, arguments passed to a function are evaluated from left to right and the resulting values are then passed on to the function. The actual arguments to a function are said to be strictly evaluated. This is known as applicative-order evaluation.

But notice that, for the `quote` function, this is not the case.

```
> (quote (println "Hello World!"))
(println "Hello World!")
> _
```

If its argument, namely

```
(println "Hello World!")
```

had been strictly evaluated, we would have seen

```
Hello World!
```

displayed on the console. The `quote` function is an atypical function sometimes referred to as a "special form".

You can write your own special form functions in newLISP as well. These are called macros and their arguments are said to be called by name, i.e. literally. This is known as normal-order evaluation and we say the evaluation strategy is lazy. In other words, a macro's arguments aren't evaluated until, and only if, you direct them to be evaluated (as we shall see how later).

Thus, the argument to `quote` is taken literally and returned. In a sense, `quote` is an identity function with a lazy evaluation strategy. It never bothers to evaluate its argument, but instead, simply returns it literally in its symbolic source code form.

Without special forms, the flow control constructs found in other languages could not be implemented in a language having only list expressions as syntax to work with. For example, consider the `if` list below.

```
> (if true (println "Hello") (println "Goodbye"))
Hello
"Hello"
> _
```

The `if` special form takes three arguments.

```
syntax: (if condition consequence alternative)

condition        =>   true
consequence      =>   (println "Hello")
alternative      =>   (println "Goodbye")
```

The `condition` argument is always evaluated, i.e. strict, but the `consequence` and `alternative` expressions are lazy. Furthermore the `alternative` expression is optional.

Notice that `if` is an expression. It returns the value of its consequence or alternative expression depending on whether the condition is true or false respectively. In the example above, we know that the alternative expression wasn't evaluated, because its side effect of printing "Goodbye" on the console never occurred.

The value of an `if` expression with a false condition having no alternative is simply `nil`. The `nil` value indicates either void or false depending upon the interpretation required.

Note: In most mainstream computer languages `if` is a statement, and thus doesn't have a return value.

If Lisp lacked a lazy evaluation strategy, it could not be used to implement special forms or macros. Without a lazy evaluation strategy, additional keywords and/or syntax would have had to have been added to the language. What syntax have you seen thus far, other than parenthesis and quotes? Answer: not much!

The flip side of lazy evaluation is you can now add your own flow control to the language thereby extending the "syntax" of Lisp allowing you to embed application specific mini-languages. Writing functions and macros will be covered in a later section of this tutorial.

Side Effects and Contexts

Without side effects, having an REPL would be pointless. To see why, consider the following sequence of expressions.

```
> (set 'hello "Hello")
"Hello"
> (set 'world " World!")
" World!"
> (println hello world)
Hello World!
" World!"
> _
```

The `set` function above has a side effect, as demonstrated below.

```
> hello
"Hello"
> world
" World!"
> _
```

The symbols `'hello` and `'world` are bound in the current context to `"Hello"` and `" World!"` respectively.

All the built-in functions are bound to symbols in the `MAIN` context.

```
> println
println <409040>
> set
set <4080D0>
> _
```

This tells us that `println` is bound to a function named `println` with an entry point of 409040. (Different builds of newLISP will obviously have different entry points for `println`.)

The default context is `MAIN`. A context is essentially a stateful namespace. We'll learn about user defined contexts later.

Notice that the literal symbol `'hello` evaluates to itself.

```
> 'hello
hello
> _
```

Evaluating the `'hello` symbol returns the value it is bound to in the current context.

```
> (eval 'hello)
"Hello"
```

```
> _
```

If the symbol is unbound when evaluated, it simply returns `nil`.

```
> (eval 'z)
nil
> _
```

Actually we don't need the `eval`, since the symbol without the quote is automatically evaluated in the current context.

```
> hello
"Hello"
> z
nil
> _
```

Thus the value of `hello` and `world` below are `"Hello"` and `" World!"` respectively.

```
> (println hello world)
Hello World!
" World!"
> _
```

What would be displayed, if we enter the following?

```
> (println 'hello 'world)
?
```

Think about it for a moment.

The `println` function displays the symbols one immediately after other on the first line.

```
> (println 'hello 'world)
helloworld
world
> _
```

Expression Sequences

A sequence of expressions can be combined into a compound expression with the `begin` function.

```
> (begin "Hello" " World!")
" World!"
> _
```

What happened to `"Hello"`? Since a compound expression returns a single value, it returns the value of its last expression. But the expressions are indeed evaluated in sequence. It's just that the `"Hello"` expression has no side effect, so its return value is discarded and you never see any evidence of its evaluation.

```
> (begin (print "Hello") (println " World!"))
Hello World!
" World!"
> _
```

This time, the side effects of `print` and `println` are evidenced in the console window and the last value returned is displayed by the REPL.

The `begin` function is useful in combining expressions into a single expression. Reconsider the `if` special form.

```
>
(if true
  (begin
     (print "Hello")
     (println " newLISP!"))
  (println "So long Java/Python/Ruby!"))

Hello newLISP!
" newLISP!"
> _
```

Multi-line statements and functions are entered hitting the [enter] key at the prompt. To exit multi-line mode hit the [enter] key again at the prompt.

Recall that the `if` form only takes three arguments.

```
syntax: (if condition consequence alternative)
```

The `(begin ...)` expression is used to combine two expressions into a single expression, which is then passed as the `consequence` argument.

Let's rap up this section by seeing how to turn our example into a program.

Please note that you can exit the REPL at any time by calling exit.

```
> (exit)
$
```

Or, on windows

```
> (exit)
c:\>
```

You can also exit with an optional integer argument.

```
> (exit 3)
```

This would be useful in shell or batch file processing that branched on reported error codes.

We can now put our hello world expression sequence into a source file.

```
;  This is a comment

;  hw.lsp

(println "Hello World!")
(exit)
```

And we can execute it from the command line, like this:

```
$ newlisp hw.lsp
Hello World!
```

Or in windows:

```
c:\> newlisp hw.lsp
Hello World!
```

Executables and Dynamic Linking

Making platform native executables and linking to dynamic link libraries with newLISP is simple.

Depending on your distribution, you should find the `link.lsp` file in the `examples` subdirectory or you may have to download the examples and modules separately from `www.newlisp.org`.

First load `link.lsp` into newLISP. At the command prompt you would type in:

```
$ newlisp link.lsp
```

or in windows:

```
c:\> newlisp link.lsp
```

Alternatively you can load `link.lsp` from the newLISP REPL prompt:

```
> (load "link.lsp")
> _
```

To make `hw.lsp` an executable, you will type either of the following expressions.

```
;; Linux/BSD
```

```
> (link "newlisp" "hw" "hw.lsp")

;; Windows

> (link "newlisp.exe" "hw.exe" "hw.lsp")
```

You can now execute `hw` or `hw.exe` from the the Linux/BSD or Windows command line respectively.

```
> hw
Hello World!
```

Linking to a dynamic link library is likewise straight forward.

*(** Examples for Linux will be added here later.)*

On Windows platforms, the following will render an alert dialog box.

```
(import "user32.dll" "MessageBoxA")

(MessageBoxA 0 "Hello World!"
"newLISP Scripting Demo" 0)
```

Please note that `MessageBoxA` is C function interface in the user library of the win32 system.

The example below demonstrates calling an external `echo` function written in C and compiled with Visual C++.

```
// echo.c

#include <STDIO.H>
#define DLLEXPORT _declspec(dllexport)

DLLEXPORT void echo(const char * msg)
{
printf(msg);
}
```

After compiling the `echo.c` file above into a DLL, it can be imported with the following code.

```
(import "echo.dll" "echo")

(echo "Hello newLISP scripting World!")
```

The ease with which newLISP can link to DLLs is one feature that makes newLISP an ideal component scripting language.

Be sure to check out the other examples and modules that demonstrate sockets programming, database connectivity, etc.

Binding

As shown previously, the `set` function is used to bind a value to a symbol.

```
(set 'y 'x)
```

In this case the value `'x`, a symbol, has been bound to the variable named `y`.

Now consider the following binding.

```
(set y 1)
```

Since there is no quote, `y` evaluates to `'x` and consequently `1` is bound to the variable named `x`.

```
> y
x
> x
1
> _
```

And of course `y` remains bound to `'x` as shown above.

The `setq` function saves you from having to write a quote each time.

```
(setq y 1)
```

Now the variable named `y` has been rebound to the value `1`.

```
> y
1
> _
```

The `define` function achieves the same thing.

```
> (define y 2)
2
> y
2
> _
```

Please note that both `set` and `setq` can bind multiple associations at a time.

```
> (set 'x 1 'y 2)
2
> (setq x 3 y 4)
```

```
4
> x
3
> y
4
> _
```

(You should verify these examples as we go along so they stick in your

memory.)

Unlike `setq` the `define` function can bind only one association at a time. However there are other uses for `define` which will be discussed shortly.

Obviously the `set`, `setq`, and `define` functions have side effects besides returning a value. The side effect is that an association binding the varible to a value is established in the current, implicit, symbol table.

We can visualize this implicit symbol table as an association list.

```
> '((x 1) (y 2))
((x 1) (y 2))
> _
```

The association list above is a list of lists. The nested lists have two elements each, i.e. a key-value pair. The first element represents the name of the association while the last element represents its value.

```
> (first '(x 1))
x
> (last '(x 1))
1
> _
```

The first element of an association list is naturally an association.

```
> (first '((x 1) (y 2)))
(x 1)
> _
```

The built-in functions `assoc` and `lookup` are provided to facilitate working with association lists.

```
> (assoc 'x '((x 1) (y 2) (x 3)))
(x 1)
> (lookup 'x '((x 1) (y 2) (x 3)))
1
> _
```

(The lookup function has other uses as well which you can find in the

newLISP documentation.)

Please be sure to notice that both `assoc` and `lookup` returned the association and value respectively of the first association having the key `'x`. This point will be important later on as the discussion on symbol tables and scope unfolds.

List as a recursive structure

Any list including an association list can be thought of as a recursive, perhaps nested, data structure. A list by definition has a first element, a tail list and a last element.

```
> (first '(1 2 3))
1
> (rest '(1 2 3))
(2 3)
> (last '(1 2 3))
3
```

But consider the following.

```
> (rest '(1))
()
> (rest '())
()
> (first '())
nil
> (last '())
nil
```

The `rest` of a empty list or a list with only one element is again the empty list. The first and last elements of an empty list are always `nil`. Notice that `nil` never represents an empty list! Only non-existent elements are represented with `nil`!

(Please note that the newLISP definition of a list differs from that

found in Lisp and Scheme.)

A list can be processed with a recursive algorithm.

For example, a recursive algorithm for calculating the length of a generic list could be defined as follows.

```
(define (list-length a-list)
(if (first a-list)
  (+ 1 (list-length (rest a-list)))
  0))
```

First of all, notice that `define` can be used not only to define variables but also functions. The name of our function is `list-length` and it takes one argument

namely `a-list`. All remaining arguments to `define` constitute the body of the function being defined.

The names of symbols can use all sorts of characters thereby facilitating rich naming styles not found in typical mainstream langauges. Be sure to consult the newLISP documentation for the complete rules for naming symbols!

The `if` condition interprets any value that is not `nil` or the empty list, i.e. `'()`, as being true. Thus we could have simply tested a-list instead with the same effect.

```
(if a-list
    ...
```

So long as a list has a remaining first or head element, the counting continues by adding 1 to the result of calling `list-length` on the rest, or tail, of the list. Since the first element of an empty list is `nil`, the alternative value of zero is returned and this serves as the recursive exit of the algorithm resulting in popping the stack frame of nested recursive calls to `list-length`.

We say that a list is a recursive data structure because its definition is recursive and not simply because it is amenable to recursive processing algorithms.

A recursive definition of a list would read something like this.

```
type list ::=  empty-list | first * list
```

A list is either the empty list or a list having a first element and a tail which is itself a list.

Since computing the length of a list is quite common there is a built-in library functional called `length` that does the job for us.

```
> (list-length '(1 2 5))
3
> (length '(1 2 5))
3
> _
```

We'll return to our discussion of user defined functions later.

The concept of an implicit symbol table can seen as succeeding expressions are evaluated.

```
> (set 'x 1)
1
> (+ x 1)
2
> _
```

Thus side effects typically effect either the output stream or this implicit context. An association list is but one way to conceptually visualize this implicit symbol table.

Suppose we wish now to momentarily change the binding of a variable without over-writing it permanently.

```
> (set 'x 1 'y 2)
2
>
(let  ((x 3) (y 4))
  (println x)
  (list x y))

3
(3 4)
> x
1
> y
2
> _
```

Notice that `x` and `y` are bound respectively to `1` and `2` in the implicit symbol table. The `let` expression momentarily (dynamically) rebinds `x` and `y` to `3` and `4` for the duration of the `let` expression. In other words, the first argument of `let` is an association list and the remaining arguments are executed in sequence.

The `list` function takes a variable number of arguments that are strictly evaluated returning each resulting value in a list.

The `let` form is similar to the `begin` form shown earlier except it dynamically extends the implicit symbol table momentarily for the life the "`let` block" which includes all the arguments of the `let` expression. This is possible because these arguments are lazily evaluated within the extended context of the "`let` block". If we visualized the implicit symbol table inside the `let` block it would look like the following extended association list.

```
'((y 4) (x 3) (y 2) (x 1))
```

Since the lookup starts from the left the rebound values of `x` and `y` are returned thereby shadowing their original values outside of the `let` expression. When the `let` expression returns the symbol table returns to looking like the following.

```
'((y 2) (x 1))
```

And consequently `x` and `y` evaluate to their original values after the `let` expression returns.

To keep things straight, compare the following.

```
> (begin (+ 1 1) (+ 1 2) (+ 1 3))
4
> (list (+ 1 1) (+ 1 2) (+ 1 3))
(2 3 4)
> (quote (+ 1 1) (+ 1 2) (+ 1 3))
(+ 1 1)
> (quote (2 3 4))
(2 3 4)
> (let () (+ 1 1) (+ 1 2) (+ 1 3))
4
```

Notice `quote` only takes one argument. (We'll learn later how it is able to ignore additional extraneous arguments.) The `let` expression with no dynamic bindings defaults to behaving just like `begin`.

Now determine what the following expressions will return (answers follow).

```
> (setq x 3 y 4)
> (let ((x 1) (y 2)) x y)
?
> x
?
> y
?

> (setq x 3 y 4)
> (begin (set 'x 1 'y 2) x y)
?
> x
?
> y
?
```

Answers:

```
> (setq x 3 y 4)
> (let ((x 1) (y 2)) x y)
2
> x
3
> y
4

> (setq x 3 y 4)
> (begin (set 'x 1 'y 2) x y)
2
> x
1
> y
2
```

Let's try something a little harder this time.

```
> (setq x 3 y 4)
> (let ((y 2)) (setq x 5 y 6) x y)
?
> x
?
> y
?
```

Answer:

```
> (setq x 3 y 4)
> (let ((y 2)) (setq x 5 y 6) x y)
6
> x
5
> y
4
```

To visualize why the above answer is correct consider the following.

```
'((y 2) (y 4) (x 3))
```

The association list above represents the symbol table upon entering the `body` of the `let` expression immediately after the extended (i.e. dynamic) binding of `y`.

After

```
(setq x 5 y 6)
```

the extended symbol table would look like this:

```
'((y 6) (y 4) (x 5))
```

And upon returning from the `let` expression it would look like this:

```
'((y 4) (x 5))
```

Thus `set`, `setq`, and `define` rebind the symbol if it found in the symbol table or else it prefixes the new binding association onto the front of the association list. We'll return to scoping after exploring functions further.

Functions

User defined functions can be defined (as discussed earlier). The following function `f` returns the sum of its two arguments.

```
(define (f x y) (+ x y))
```

This is actually short hand for any of the following.

```
(define f (lambda (x y) (+ x y)))

(setq f (lambda (x y) (+ x y)))

(set 'f (lambda (x y) (+ x y)))
```

The lambda expression defines an anonymous function, i.e. a nameless function. The first argument of the lambda expression is its formal argument list and the remaining expressions make up a delayed sequence of expressions constituting the body of the function.

```
> (f 1 2)
3
> ((lambda (x y) (+ x y)) 1 2)
3
> _
```

Recall that an unquoted list is interpreted as a function call where all arguments are strictly evaluated. The first element of the list above is a lambda expression so it is evaluated returning an anonymous function which is then applied to the arguments `1` and `2`.

Notice that the following two expressions as essentially the same.

```
> (let ((x 1) (y 2)) (+ x y))
3
> ((lambda (x y) (+ x y)) 1 2)
3
> _
```

The only real difference is the sequence of expressions in the lambda expression are delayed until it is applied to arguments. Applying the lambda expression to arguments in effect associates a binding of the formal arguments to the actual arguments to which the function is applied.

What are the values in the following expressions?

```
> (setq x 3 y 4)
> ((lambda (y) (setq x 5 y 6) (+ x y)) 1 2)
?
> x
?
> y
?
```

Remember the lambda and let expressions are essentially the same.

```
> (setq x 3 y 4)
> ((lambda (y) (setq x 5 y 6) (+ x y)) 1 2)
11
> x
```

```
5
> y
4
```

The arguments `1` and `2` are superfluous. The formal argument `y` shadows the `y` defined outside of the lambda expression so that setting `x` to `5` is the only one with lasting effect after the lambda returns.

Higher Order Functions

Functions in Lisp are first class values. Like data they can be created at run time and passed around like any data value to effect higher order functional programming. Please note that function pointers found in C (and event listeners in Java/C#) for example are not first class functions While they may be passed around like data they never the less cannot be created like data at run time.

Perhaps the most commonly used higher order function is `map` (sometimes called `collect` in object oriented (constraint) languages which got the idea originally from Lisp via Smalltalk).

```
> (map eval '((+ 1) (+ 1 2 3) 11))
(1 6 11)
> _
```

The `map` function applies the `eval` function to each element of the given list. Notice that the + function takes a variable number of arguments.

In this case we could have simply written the following.

```
> (list (+ 1) (+ 1 2 3) 11)
(1 6 11)
> _
```

But `map` can do perform more interesting operations as well.

```
> (map string? '(1 "Hello" 2 " World!"))
(nil true nil true)
> _
```

The `map` function can also take more than one list argument.

```
> (map + '(1 2 3 4) '(4 5 6 7) '(8 9 10 11))
(13 16 19 22)
> _
```

On the first iteration + is applied to the first element of each list.

```
> (+ 1 4 8)
13
```

```
> _
```

Let's say we wish to detect which elements are even.

```
> (map (fn (x) (= 0 (% x 2))) '(1 2 3 4))
(nil true nil true)
> _
```

`fn` is shorthand for lambda.

```
> (fn (x) (= 0 (% x 2)))
(lambda (x) (= 0 (% x 2)))
> _
```

The remainder operator `%` is used to determine whether a number is evenly divisible by 2.

The `filter` function is another commonly used higher order function (sometimes called `select` in OO language libraries).

```
> (filter (fn (x) (= 0 (% x 2))) '(1 2 3 4))
(2 4)
> _
```

The `index` function can be used instead to identify the indices of the elements in the original list.

```
> (index (fn (x) (= 0 (% x 2))) '(1 2 3 4))
(1 3)
> _
```

The `apply` function is another higher order function.

```
> (apply + '(1 2 3))
6
> _
```

Why not simply write `(+ 1 2 3)` instead?

Sometimes you may not know ahead of time which function will be applied.

```
> (setq op +)
+ <40727D>
> (apply op '(1 2 3))
6
> _
```

This approach could be used to implement a dynamic method dispatcher, for example.

lambda lists

Consider the following function.

```
> (define (f x y) (+ x y z))
(lambda (x y) (+ x y z))
> f
(lambda (x y) (+ x y z))
> _
```

The function is a special kind of list known as a lambda list.

```
> (first f)
(x y)
> (last f)
(+ x y z)
> _
```

Thus a "compiled" function can be introspected at run time. In fact it can even be changed at run time!

```
> (setf (nth 1 f) '(+ x y z 1))
(lambda (x y) (+ x y z 1))
> _
```

(Be sure to check up the `nth-set` function in the newLISP

documentation also.)

The `expand` function is useful for modifying lists in general including lambda lists.

```
> (let ((z 2)) (expand f 'z))
(lambda (x y) (+ x y 2 1))
> _
```

The `expand` function takes a list argument and replaces the symbols within by the values of all of its remaining symbolic arguments.

Dynamic Scope

Consider the following function definition.

```
>
(define f
(let ((x 1) (y 2))
  (lambda (z) (list x y z))))

(lambda (z) (list x y z))
> _
```

Notice that the value of `f` is the `lambda` only.

```
> f
(lambda (z) (list x y z))
> (setq x 3 y 4 z 5)
5
> (f 1)
(3 4 1)
> (let ((x 5)(y 6)(z 7)) (f 1))
(5 6 1)
```

Even though the lambda expression is defined within the lexical scope of the `let` expression binding `x` to `1` an `y` to `2`, at the time of its invocation it's the dynamic scope that matters. We say that the binding of newLISP lambda expressions is dynamic (versus lexical binding in Common Lisp and Scheme).

Any free variables of a lambda expression are bound dynamically at the time its body of expressions are evaluated. Variables not specified (bound) in the formal argument list are said to be free.

We can use the `expand` function shown previously to close a lambda expression, i.e. bind over all free variables.

```
>
(define f
(let ((x 1) (y 2))
  (expand (lambda (z) (list x y z)) 'x 'y)))

(lambda (z) (list 1 2 z))
> _
```

Notice that the lambda expression does not have any free variables now.

"Closing" the lambda expression with the expand function is not quite

the same thing as the lexical lambda closure found in CL (Common Lisp) and Scheme, however. Lexical closures exist in newLISP and will be discussed in a later section on contexts.

Function Argument Lists

A newLISP function can have any number of arguments (within reason).

```
>
(define (f z , x y)
(setq x 1 y 2)
(list x y z))

(lambda (z , x y) (setq x 1 y 2) (list x y z))
> _
```

The four formal arguments of `f` are

```
z , x y
```

Notice that the comma is the name of an argument (see symbol naming rules). It is used here as a visual gimmick.

The only intentional argument is `z`.

If the number of formal arguments exceeds the number of actual arguments to which the function is applied then the remaining formal arguments are simply initialized to `nil`.

```
> (f 3)
(1 2 3)
> _
```

In this case

```
, x y
```

are all three initialized to `nil`. Since `x` and `y` appear as formal arguments they act like local variables so that

```
(setq x 1 y 2)
```

doesn't overwrite the binding of `x` and `y` outside the scope of the lambda expression.

We could have written the following instead to achieve the same effect of local variables.

```
>
(define (f z)
(let ((x 1)(y 2))
  (list x y z)))

(lambda (z) (let ((x 1)(y 2)) (list x y z)))
> _
```

The comma and unused formal arguments is an idiom used often in newLISP to provide local variables.

A function can even be called with more actual arguments than it has specified in its formal argument list. In this case the excess arguments are simply ignored.

Trailing formal arguments can thus be treated as optional arguments.

```
(define (f z x y)
(if (not x) (setq x 1))
```

```
(if (not y) (setq y 2))
(list x y z))
```

Now if `f` is called with only one argument the `x` and `y` default to `1` and `2` respectively.

lambda-macro

The actual arguments of a `lambda-macro` function are not strictly evaluated like they are for a `lambda` function.

```
(define-macro (my-setq _key _value)
(set _key (eval _value)))
```

Since `_key` is not evaluated, it is in symbolic, i.e. quote equivalent, form already. Its `_value` is also symbolic and must be evaluated therefore.

```
> (my-setq key 1)
1
> key
1
> _
```

The underscores are used to prevent variable capture. Consider the following.

```
> (my-setq _key 1)
1
> _key
nil
> _
```

What happened?

The statement

```
(set _key 1)
```

merely set the `_key` local variable. We say the `_key` variable was captured by the macro's "expansion." Scheme has "hygenic" macros that are "clean" in that they are guaranteed to prevent variable capture. Usually the leading underscore character used in the macro's formal arguments is sufficient to prevent variable capture however.

The `define-macro` function is shorthand for binding a `lambda-macro` expression in one step.

```
(define my-setq
(lambda-macro (_key _value)
(set _key (eval _value))))
```

The definition above is equivalent to the previous definition of `my-setq`.

Besides lazy evaluation, `lambda-macro`'s provide for a variable number of arguments.

```
(define-macro (my-setq )
(eval (cons 'setq (args))))
```

The `cons` function prefixes a list with a new first element.

```
> (cons 1 '(2 3))
(1 2 3)
> _
```

The definition of `my-setq` is now a more complete implementation allowing for multiple bindings.

```
> (my-setq x 10 y 11)
11
> x
10
> y
11
> _
```

The `(args)` function call returns the list all the arguments to the `lambda-macro`, i.e. unevaluated.

Thus the `my-setq` macro first constructs the symbolic expression shown below.

```
'(setq x 10 y 11)
```

This expression is then evaluated.

The main purpose of macros is to extend the syntax of the language however.

Suppose we want to introduce *repeat until* flow control as a syntax extension to the language.

```
(repeat-until condition body ...)
```

The following macro achieves this.

```
(define-macro (repeat-until _condition )
(let ((body (cons 'begin (rest (args)))))
(eval (expand (cons 'begin
  (list body
    '(while (not _condition) body)))
  'body '_condition))))
```

Using `repeat-until`:

```
(setq i 0)
(repeat-until (> i 5)
(println i)
(inc i))
; =>
0
1
2
3
4
5
```

Macros can become quite complex quite quickly. One trick for validating them is to replace `eval` with `list` or `println` to inspect what the expansion looks like just before it is evaluated.

```
(define-macro (repeat-until _condition )
(let ((body (cons 'begin (rest (args)))))
(list (expand (cons 'begin
  (list body
    '(while _condition body)))
  'body '_condition))))
```

Now we can check what the expansion looks like.

```
> (repeat-until (> i 5) (println i) (inc i))
((begin
(begin
(println i)
(inc i))
(while (> i 5)
(begin
(println i)
(inc i)))))
> _
```

Contexts

Upon startup the default context is `MAIN`.

```
> (context)
MAIN
```

A context is a name space.

```
> (setq x 1)
1
> x
1
> MAIN:x
```

```
1
> _
```

A context variable can be used to fully qualify a variable name. `MAIN:x` refers to the variable `x` in context `MAIN`.

To create a new name space use the `context` function.

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

The statement above creates the namespace `FOO`, if it doesn't already exist, and switches to it. The prompt tells you the current namespace if it is other than `MAIN`.

Use the `context?` predicate to determine whether a variable is bound to a context.

```
FOO> (context? FOO)
true
FOO> (context? MAIN)
true
FOO> (context? z)
nil
FOO> _
```

The `set`, `setq`, and `define` functions bind associations in the current context, i.e. namespace.

```
FOO> (setq x 2)
2
FOO> x
2
FOO> FOO:x
2
FOO> MAIN:x
1
FOO> _
```

A fully qualified name, e.g. `FOO:x`, is not required to specify a variable when it is bound in the current context.

To switch back to the `MAIN` (or any other) context use either the variable `MAIN` or the symbol `'MAIN`.

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

or

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

The only time you must use the quote is for creating new contexts.

Contexts cannot be nested they all reside at the top level.

Notice in the example below that the name `y`, which is defined in `MAIN`, is not known in context `FOO`.

```
> (setq y 3)
3
> (context FOO)
FOO
FOO> y
nil
FOO> MAIN:y
3
FOO> _
```

The next example shows that `MAIN` is not special in any way other than being the default context. `MAIN` does not know about `z` for example.

```
FOO> (setq z 4)
4
FOO> (context MAIN)
MAIN
> z
nil
> FOO:z
4
```

All the built-in names reside in a special global section of the `MAIN` context.

```
> println
println <409040>

> (context FOO)
FOO
FOO> println
println <409040>
```

The `println` built-in function is known in both the `MAIN` and `FOO` contexts. The `println` function has been "exported" to global status.

The sequence of expressions below shows that `MAIN:t` is not known initially within contexts `FOO` or `BAR` until it has been elevated to global status.

```
FOO> (context MAIN)
MAIN
> (setq t 5)
5
> (context 'BAR)
BAR
BAR> t
```

```
nil
BAR> (context FOO)
FOO
FOO> t
nil
FOO> (context MAIN)
MAIN
> (global 't)
t
> (context FOO)
FOO
FOO> t
5
FOO> (context BAR)
BAR
BAR> t
5
```

Only names in `MAIN` can be elevated to global status.

Lexical Scope

The `set`, `setq`, and `define` functions bind names in the current context.

```
> (context 'F)
F
F> (setq x 1 y 2)
2
F> (symbols)
(x y)
F> _
```

Notice that the `symbols` function returns the names of symbols bound in the current context.

```
F> (define (id z) z )
(lambda (z) z)
F> (symbols)
(id x y z)
F> _
```

The lexical scope of the current context continues until the next context switch. Since you can later switch back to a particular context its lexical scope can be augmented and may appear fragmented in the source file(s).

```
F> (context 'B)
B
B> (setq a 1 b 2)
2
B>
```

By lexical scope, we mean the scope as defined by the source code. The names `x` and `y` occur in the lexical scope of context `F` while the names `a` and `b` occur within the lexical scope of context `B`.

All lambda expressions are lexically scoped in the context in which they are defined. Consequently lambda expressions are ultimately "closed" by the context.

The lambda expression below occurs in the lexical scope of `MAIN` and also in the lexical scope of the `(let ((x 3)) ...)` expression.

```
> (setq x 1 y 2)
2
>
(define foo
 (let ((x 3))
  (lambda () (list x y))))

(lambda () (list x y))
> (foo)
(1 2)
> _
```

Recall that lambda invocations in general are dynamically scoped. While this is true notice that this lambda invocation is ultimately "closed" by the lexical scope of the `MAIN` context and not the `let` expression.

Continuing with the last example we can see this hybrid lexical / dynamic scoping at work.

```
> (let ((x 4)) (foo))
(4 2)
> _
```

The lexical scoped context is dynamically extended during the execution of the `let` expression this time so that `(foo)` is invoked in the dynamic scope of the `let` expression.

What will happen if we invoke `(foo)` in an alien context?

```
> (context 'FOO)
FOO
FOO> (let ((x 5)) (MAIN:foo))
?
```

Think about this for a moment. The `let` expression above dynamically extends the lexical scope of `FOO` instead of `MAIN`.

```
FOO> (let ((x 5)) (MAIN:foo))
(1 2)
FOO> _
```

What happened? `MAIN:foo`'s dynamic scope includes only the scope of the `MAIN` context possibly dynamically extended. Since the `let` expression extends the dynamic scope of `FOO` the invocation of `MAIN:foo` doesn't see the `FOO:x => 5` binding.

The following expression is revealing.

```
FOO> MAIN:foo
(lambda () (list MAIN:x MAIN:y))
FOO> _
```

When we introspected `foo` in the `MAIN` context we didn't see the default `MAIN` qualifier.

```
> foo
(lambda () (list x y))
> _
```

So even though the `FOO` context was dynamically extended with the `FOO:x => 5` binding, we can now see that when `MAIN:foo` is executing it confines its lookup to the `MAIN` (possibly dynamically extended) context only.

What would be the value of the following expression?

```
FOO> (let ((MAIN:x 5)) (MAIN:foo))
?
```

If you answered the following, you are correct.

```
FOO> (let ((MAIN:x 5)) (MAIN:foo))
(5 2)
FOO> _
```

We say that the context, or namespace, is the lexical closure of all functions defined within that context.

Understanding how newLISP translates and evaluates source code is vital to your understanding of contexts.

Each top level expression is first translated and then evaluated in order by newLISP before stepping forward to the next top level expression. During the translation phase all (unqualified) symbols are understood to be bound in the current context. Thus the context expression is merely a directive to switch (or create and switch) to the indicated context. This has important ramifications as we'll see momentarily.

```
(context 'FOO)
(setq r 1 s 2)
```

Each of the expressions above are top level expressions, despite the suggestive indentation. The first expression is translated in the current context. Thus `FOO` becomes a symbol bound in the current context (i.e. `MAIN` if it isn't already) before the translated expression is actually evaluated. Once the translated expression is evalauted, the context switch takes place which can be clearly seen when operating in the interpreter mode.

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

So by the time newLISP gets around to interpreting

```
FOO> (setq r 1 s 2)
```

the current context is now `FOO`.

Now consider the following code snippet

```
> (begin (context 'FOO) (setq z 5))
FOO> z
nil
FOO> MAIN:z
5
FOO> _
```

What happened?

First the single top level expression

```
(begin (context 'FOO) (setq z 5))
```

was translated in the `MAIN` context. Thus `z` is taken be:

```
(setq MAIN:z 5)
```

As the translated `begin` compound expression begins to be evaluated first the context is switched, but then the `MAIN:z` variable is set to `5`. Upon returning from the compound expression, the context remains switched to `FOO`.

It helps to keep things straight by thinking about the source code in its two phases, i.e. translation and execution especially when dealing with contexts.

Contexts can be used to organize data and/or functions as records or structures, classes, and modules.

```
(context 'POINT)
(setq x 0 y 0)
(context MAIN)
```

The POINT context shown above can be thought of a structure having two fields (or slots).

```
> POINT:x
0
> _
```

But contexts can also be cloned, thus serving as a simple class or prototype. The new function as shown below creates a new context called p if it doesn't already exist and then merges a clone of the bindings found in POINT.

```
> (new POINT 'p)
p
> p:x
0
> (setq p:x 1)
1
> p:x
1
> POINT:x
0
```

The sequence of expressions above shows that context p is a distinct and separate copy of POINT.

The following example shows how contexts could be used to provide a simple structure inheritance mechanism.

```
(context 'POINT)
(setq x 0 y 0)
(context MAIN)

(context 'CIRCLE)
(new POINT CIRCLE)
(setq radius 1)
(context MAIN)

(context 'RECTANGLE)
(new POINT RECTANGLE)
(setq width 1 height 1)
(context MAIN)
```

Notice how the new merges the x and y fields of POINT into CIRCLE which adds an additional field called radius. RECTANGLE "inherits" from POINT in a similar fashion.

The def-make macro below allows us to define named instances of a context and optionally specify initializers.

```
(define-macro (def-make _name _ctx )
(let ((ctx (new (eval _ctx) _name))
```

```
    (kw-args (rest (rest (args)))))
(while kw-args
  (let ((slot (pop kw-args))
        (val (eval (pop kw-args))))
    (set (symbol (name slot) ctx) val)))
ctx))
```

For example we could instantiate a `RECTANGLE` named `r` and override the default values for `x` and `height` with the following expression.

```
(def-make r RECTANGLE x 2 height 2)
```

The following function will convert a context "instance" to a string.

```
(define (context->string _ctx)
(let ((str (list (format "#S(%s" (string _ctx)))))
(dotree (slot _ctx)
  (push (format " %s:%s" (name slot)
    (string (eval (symbol (name slot) _ctx))))
    str -1))
  (push ")" str -1)
(join str)))
```

Now we can verify the contents of `r`.

```
> (context->string r)
"#S(r height:2 width:1 x:2 y:0)"
> _
```

Notice how various characters such as "`->`" can be used in identifier names.

You know enough about newLISP now to decypher the `def-make` and `context->string` definitions above. Be sure to look up in the regular newLISP documentation any primitive operations such as `dotree`, `push`, `join`, etc. that are not familiar to you.

Both Common Lisp and Scheme have lexically scoped functions meaning that a closure is exclusive to a particular function. Functions in newLISP can share a lexical closure, i.e. context, which is similar to an object whose methods share a common state. The context examples shown thus far could have included functions as well. The newLISP documentation gives several examples of using contexts as simple objects.