Function Call Tutorial

This tutorial follows on from the FunctionsTutorial.

Passing arguments

Since Lua is dynamically typed and has no need for function prototypes, checking the number and type of arguments required for a function seems somewhat redundant. Lua deals gracefully with misaligned numbers of calling and received function arguments. Function arguments receive a default value of `nil` if they are not filled. Where too many arguments are passed they are simple ignored. No type is specified for each argument as they are dynamically typed. i.e. we only need to know the type of an object when we use it, not when we reference it. We'll use the following function as an example:

```
> function foo(a,b,c) print(a,b,c) end
```

Here's what happens when we call it with no arguments:

```
> foo()
nil     nil     nil
```

Notice that each of the arguments defaulted to the value `nil`, or no value, and we don't get an error. Here's what happens when we pass too many arguments:

```
> foo(1,2,3,4)
1       2       3
```

Again, no error and the last argument is ignored. Because Lua is dynamically typed we can pass any argument type. e.g. we can pass strings in as well as numbers.

```
> foo("hello")
hello   nil     nil
> foo("pi", 3.1415, { comment="this is a table" })
pi      3.1415  table: 002FDBE8
```

Variable arguments

It is often useful to have variable numbers of arguments to a function. e.g. the `printf(format,...)` function in C. Lua does this by placing the variable

argument list into a array table called `arg`, useable by the function. e.g.,

```
> function foo(...) print(arg) end
> foo("abc",3,77)
table: 002FD3B8
```

In this example we can only see that this is a table. We can use the `table.foreach(table,function)` to print out the values of the variable argument table as thus:

```
> function foo(...) table.foreach(arg,print) end
> foo()
n       0
```

From looking at an empty variable argument list it is easy to see that an extra table pair is added to provide the number of elements in the table, `arg.n`. In this case the number of arguments is zero. Let's try passing variable numbers of arguments:

```
> foo(1,2,3)
1       1
2       2
3       3
n       3
> foo("apple",2,"banana",99,3.1415927,foo)
1       apple
2       2
3       banana
4       99
5       3.1415927
6       function: 002FB5C8
n       6
```

unpack

A useful function for variable arguments is `unpack()`. This takes a table and returns a list of variables, e.g.:

```
> = unpack({1,2,3})
1       2       3
```

This can be used with variable argument lists as follows:

```
> function listargs(...)
>>  return unpack(arg)
```

```
>> end
> = listargs(1,2,3)
1       2       3
> = listargs("hello", {1,2,3}, function (x) return x*x end)
hello   table: 0035F0B8 function: 00357860
```

Multiple return values

Lua can return more than one value from a function. This is done by returning a comma seperated list of values:

```
> function foo(angle)
>>  return math.cos(angle), math.sin(angle)
>> end
>
> print( foo(1) )     -- returns 2 values...
0.54030230586814        0.8414709848079
>
> c,s = foo(3.142/3)  -- assign the values to variables
> = math.atan(s/c)
1.0473333333333
> = c,s
0.49988240461137        0.86609328686923
>
> function many(x)
>>  return x, x*x, x*x*x, x*x*x*x, x*x*x*x*x
>> end
> = many(5)
5       25      125     625     3125
> = many(0.9)
0.9     0.81    0.729   0.6561  0.59049
```

The above function could have a variable number of return values if we construct a table, containing the values, and use `unpack`. E.g.,

```
> function many2(x,times)
>>  local t = { [0]=1 }
>>  for i=1,times do t[i] = x*t[i-1] end
>>  return unpack(t)
>> end
> = many2(5,10)
5       25      125     625     3125    15625   78125   390625  1953125 9765625
> = many2(0.5,7)
0.5     0.25    0.125   0.0625  0.03125 0.015625        0.0078125
```

Values as a table

We can also return the values in a table. To to this we add curly brackets around the function call, which will construct a table, i.e.:

{ *function_name* (*args*) }

Here is an example using the previous function example:

```
> = { foo(1.5) }
table: 0035E088
> t = { foo(1.5) }
> table.foreach(t,print)
1       0.070737201667703
2       0.99749498660405
```

Single Value

If a function returns multiple values, but we only want the first value, we put parenthesis around the function call, i.e.

(*function_name* (*args*))

Here is an example:

```
> = (foo(1.5))
0.070737201667703
> = foo(1.5)
0.070737201667703       0.99749498660405
```

The same could be achieved by returning a table and taking the first element but the above syntax is more efficient. E.g.,

```
> = ({foo(1.5)}) [1]
0.070737201667703
```
