Expressions Tutorial

Expressions are evaluated in order to perform calculations which may assign values to variables or pass arguments to functions. Expressions are covered pretty well in section 2.5 of the Reference Manual. Expressions are covered here for completeness and to offer more examples.

We'll use the = *expression* shorthand notation for this page. The values can easily be assigned to a variable, e.g.,

```
> x = 7
> print(x)
7
> = 7
7
```

Arithmetic expressions

Lua has the usual binary arithmetic operators.

```
> = 2+3, 5-12, 2*7, 7/8
5       -7      14      0.875
> = 5*(2-8.3)/77.7+99.1
98.694594594595
```

Unary negation:

```
> = -(-10), -(10)
10      -10
```

Power of:

```
> = 7^2, 107^0, 2^8
49      1       256
```

Relational expressions

Relational operators are supplied which return the boolean values `true` or `false`.

- == equal to
- ~= not equal to
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to

Examples:

```
> = 1 == 1, 1 == 0
true    false
> = 1 ~= 1, 1 ~= 0
false   true
> = 2 < 7, 2 > 7
true    false
> = 3 <= 7, 7 <= 7, 8 <= 7
true    true    false
> = 3 >= 7, 7 >= 7, 8 >= 7
false   true    true
```

These also work on strings and other types.

```
> = "abc" < "def"
true
> = "abc" > "def"
false
> = "abc" == "abc"
true
> = "abc" == "a".."bc"
true
```

Objects will not be equal if the types are different or refer to different objects.

```
> = {} == "table"
false
> = {} == {}  -- two different tables are created here
false
> t = {}
> t2 = t
> = t == t2   -- we're referencing the same table here
true
```

Coercion does not work here, the types must be converted explicitly. See NumbersTutorial and StringsTutorial for explanation of coercion.

```
> = "10" == 10
false
> = tonumber("10") == 10
true
```

Logical operators

Lua provides the logical operators `and`, `or` and `not`. In Lua both `nil` and the boolean value `false` represent false in a logical expression. Anything that is not false (either `nil` or `false`) is `true`. There are more notes on the implications of this at the end of this page.

```
> = false==nil   -- although they represent the same thing they are not equivalent
false
> = true==false, true~=false
false   true
> = 1==0
false
> = does_this_exist  -- test to see if variable "does_this_exist" exists. no, false.
nil
```

not

The keyword `not` inverts a logical expression value:

```
> = true, false, not true, not false
true    false   false   true
> = not nil       -- nil represents false
true
> = not not true  -- true is not not true!
true
> = not "foo"     -- anything not false or nil is true
false
```

and

The binary operator `and` does *not necessarily* return a boolean value `true` or `false` to the logical expression *x and y*. In some languages the `and` operator returns a boolean dependent on the two inputs. Rather in Lua, it returns the first argument if its value is `false` or `nil`, and the second argument if the first argument was not false. So, a boolean is only returned if the value passed in was a boolean.

```
> = false and true  -- false is returned because it is the first argument
false
> = nil and true    -- as above
nil
> = nil and false
nil
> = nil and "hello", false and "hello"
nil     false
```

All of the above expressions return the first argument. All of the following expressions return the second argument, as the first is true.

```
> = true and false
false
> = true and true
true
> = 1 and "hello", "hello" and "there"
hello   there
> = true and nil
nil
```

As you can see the logical expressions are still evaluated correctly but we have some interesting behaviour because of the values returned.

or

The `or` binary operator also does *not necessarily* return a boolean value (see notes for `and` above). If the first argument is not false it is returned, otherwise the second argument is returned.

```
> = true or false
true
> = true or nil
true
> = "hello" or "there", 1 or 0
hello   1
```

All of the above expressions return the first argument. All of the following expressions return the second argument, as the first is false.

```
> = false or true
true
> = nil or true
true
> = nil or "hello"
hello
```

This can be a very useful property. For example, setting default values in a function:

```
> function foo(x)
>>  local value = x or "default"  -- if argument x is false or nil, value becomes "default"
>>  print(value, x)
>> end
>
> foo()       -- no arguments, so x is nil
default nil
> foo(1)
1       1
> foo(true)
true    true
> foo("hello")
hello   hello
```

Ternary operators

Ternary operators [1] are a useful feature in C. e.g.

```
int value = x>3 ? 1 : 0;
```

This behaviour can be partially emulated in Lua using the logical operators `and` and `or`. The C form:

```
value = test ? x : y;
```

translates to the following Lua:

```
value = test and x or y
```

E.g.

```
> print( 3>1 and 1 or 0 )
1
> print( 3<1 and 1 or 0 )
0
> print( 3<1 and "True" or "False" )
False
> print( 3>1 and true or "false" )
true
```

However, there is a caveat, this only works when the first return value is not `nil` or `false`.

```
> print( 3>1 and 1 or "False" )        -- works
1
> print( 3>1 and false or "oops" )     -- failed, should return false
oops
> print( 3>1 and nil or "oops" )       -- failed, should return nil
oops
```

Note on test expressions and nil

An important point to note is that the value `0` is not a false test condition in Lua. In some languages, for example C, a test of:

```
if (0)
  printf("true");
else
  printf("false");
```

would display "false". In Lua,

```
> if 0 then
>>  print("true")
>> else
>>  print("false")
>> end
true
```

prints "**true**"! You should use `false`, or `nil` in place of 0:

```
> if false then print("true") else print("false") end
false
> if nil then print("true") else print("false") end
false
```

Why?

The reason for this is historical. Lua did not support boolean types (i.e. `true` and `false`) before version 5.0. Previous to version 5.0 a value of `nil` represented false. Now, both `nil` and `false` will act as a false condition in a test expression. E.g.,

```
> if nil then print("true") else print("false") end
```

```
false
> if 1 then print("true") else print("false") end
true
> if 0 then print("true") else print("false") end
true
> if 1==2 then print("true") else print("false") end
false
```

Another point to note is that `true` and `false` are not numerical values, e.g., 1 and 0 as they are in some languages.

```
> = true, false
true    false
> = 1 + true
stdin:1: attempt to perform arithmetic on a boolean value
stack traceback:
        stdin:1: in main chunk
        [C]: ?
```

Also, `nil` is coerced into a boolean value when used with a logical operator:

```
> = not nil
true
> = not 1
false
> = not 0
false
```
