LUA Programming Language

RESERVED KEYWORDS

and break do else elseif end false for function if in local nil not or repeat return then true until while

OPERATORS

+	addition
-	subtraction/negation
*	multiplication
/	division
^	exponent
==	equal to
~=	not equal to
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to
..	string concencation (eg: "Hello ".."World" => "Hello World")

LOGICAL OPERATORS

and	arguments on both sides must be true
or	argument on 1 side must be true
not	reverse argument (eg: true => false)

OTHER OPERATORS

=	assignment
()	function arguments
{}	table constructors
[]	table index
;	end statement
:	table methods (eg: v:name(...) == v.name(v,...))
,	function argument separation/multiple variable (declaration/returning)/table declaration separation
.	table index (eg: a.name == a["name"])
...	variadic (eg: foo(...) will collect all arguments into a table called "arg")

OPERATOR PRECEDENCE

If operators are next to each other it indicates that they are married and go in order of first, to last.
NOTE: you can change the precedence with parentheses

```
or
and
< > <= >= ~= ==
..
+ -
* /
not
^
```

VARIABLE TYPES

nil	
boolean	(eg: TRUE or FALSE)
number	(eg: 1,2, or 13.021)
string	(eg: "Hello World!")
function	(eg: foo(bar))
table	(eg: foo[5] = "bar")
thread	(eg: co-routines)

CONTROL STRUCTURES

```
while EXP do
  CODE_BLOCK
end;

repeat
  CODE_BLOCK
until EXP;

if EXP1 then
  CODE_BLOCK1
else
  CODE_BLOCK2
end;

for VAR = START_VALUE, END_VALUE do
  CODE_BLOCK
end;

for VAR {,VAR} in TABLE do
  CODE_BLOCK
end;
```

CONTROL FLOW

return	returns a value from a function or chunk
break	stops the current control structure (only in while, repeat and for)
continue	switches to next iteration (only in while, repeat and for)

DECLARATION VARIABLES

```
Global scope: VAR_NAME1{, VAR_NAME2,...} = EXP1{,EXP2,...};
Local scope: local VAR_NAME1{, VAR_NAME2,...} = EXP1{,EXP2,...};
Table: TABLE_NAME = {EXP,...};
```

DECLARATION FUNCTIONS

```
function FUNCTION_NAME(ARG)
  BLOCK
end;
```

COMMENTS

```
-- This is a single line comment

--[[
  This is a multiline...
  comment!
--]]
```

ESCAPE-SEQUENCES

\n	newline
\\	backslash
\"	quotation mark
\'	apostrophe
\[	left square bracket
\]	right square bracket

STANDARD LIBRARIES

error(message{,level})
stops execution, and displays the message. The level argument specifies where the error message points the error. With level 1, the error position is where the error function was called. Level 2 points the error to where the function that called error was called, and so on. if level is nil the level = 1.

tonumber(VAR{,base})
attempts to convert VAR to a number, if it can't the function will return nil. An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter `A´ (in either upper or lower case) represents 10, `B´ represents 11, and so forth, with `Z´representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part. In other bases, only unsigned integers are accepted.

tostring(VAR)
accepts any type of variable and converts it into a string type(VAR) returns a string that Identifies what type of value VAR is possible returned strings are: "nil", "number", "string", "boolean", "table", "function", "thread"

string.find(VAR,PATTERN{,INIT{,PLAIN}})
Looks for the first match of PATTERN in VAR. If it dosn't find it, the function returns nil else it returns the indices's of VAR where this occurrence starts and ends. INIT is an optional argument that states where to start searching from, the default is 1. Lastly the argument PLAIN, is a boolean that if TRUE it should do a search on VAR without considering "magic" characters, (NOTE: INIT must be included if PLAIN is included)

string.len(VAR)
returns the number of characters in VAR

string.lower(VAR)
returns an identical string of VAR, except all uppercase letters are now lowercase.

string.sub(VAR,START{,FINISH})
returns a substring of VAR in between position START and position FINISH, if FINISH is absent, then it assumes until the end of the string

string.upper(VAR)
returns an identical string of VAR, except all lowercase letters are now uppercase.

string.gsub(VAR,PATTERN,REPLACE{,MAX})
returns an identical string to VAR, except it replaces all occurrences of PATTERN with REPLACE. If MAX is present it will limit the number of replacements to MAX, so if MAX == 1 then it will only replace the first occurrence of PATTERN with REPLACE. If replace is a function then that function is called every time it finds an occurrence of PATTERN with all captured substrings passed as arguments in order. if the pattern specifies no captures, then the whole match is passed as an argument. If the value returned by this function is a string, then it is used as the replacement string, else the replacement string is an empty string.

table.getn(VAR)
returns the length of the table VAR

table.insert(VAR{,POSITION},VALUE)
inserts VALUE into VAR at index POSITION, while moving other index's up one index, so to make room for the new one. If POSITION is not given then it will insert VALUE at the last index of VAR

table.remove(VAR{,POSITION})
removes the index POSITION from VAR and moves all other index's down, so it fills the gap and then returns the value of the removed index. If POSITION is absent, it will remove the last index from VAR

math.abs(VAL)
absolute value of VAL

math.acos(VAL)
inverse cosine of VAL

math.asin(VAL)
inverse sine of VAL

math.atan(VAL)
inverse tangent of VAL

math.sin(VAL)
sine of VAL

math.deg(VAL)
convert VAL into degree's

math.floor(VAL)
floor of VAL (eg: math.floor(5.9) == 5)

math.log(VAL)
returns the inverse of math.exp on VAL

math.log10(VAL)
returns the base 10 logarithm of VAL

math.max(VAL1,...)
returns the maximum value of its list of arguments

math.min(VAL1,...)
returns the minimum value of its list of arguments

math.mod(X,Y)
returns the remainder of X divided by Y

math.pow(X,Y)
returns X to the Y power

math.sqrt(VAL)
square root of VAL

math.random(VAL1,VAL2)
if their are no arguments, it will randomly return a real number between 0 and 1. If VAL1 is present but not VAL2, it will return a random number between 1 and VAL1. If both VAL1 and VAL2 are present it will return a random number between VAL1 and VAL2

math.randomseed(VAL)
sets the (pseudo-)random number generator's seed to VAL, if you use a different seed each time it will produce unsimular patterns, therefore creating actual random numbers. A good way of making sure the seed is different MOST of the time just call it like this math.randomseed(os.time()) os.time() will return the time and then the time will be used as the seed, which can create almost real random numbers. math.pi a variable that contains the value of PI