Lua is a programming language designed primarily for embedded systems. It is popular in the video game industry as a language that can be embedded in a larger game engine.

Pico-8 implements a subset of Lua for writing game cartridges. Because it is a subset, not all features of Lua are supported. Most notably, Pico-8 does not include the Lua standard library, and instead provides a proprietary collection of global functions.

- Official Lua documentation
- Lua (Wikipedia)

Contents

Features of Pico-8 Lua

The following are the major language features of Lua as implemented by Pico-8.

Pico-8 code comments are preceded by two hyphens (`--`) and go to the end of the line. They also support a multi-line syntax using double-brackets (`--[[ ... ]]`).

```
-- one-line comment
--[[ multi-line
comment ]]
```

Values

Pico-8 supports these fundamental types of values:

- Numbers, in the range supported by 16:16 bit fixed point (-32768.0 to 32767.99)
  Number literals can use decimal (`17.25`) or hexadecimal (`0x11.4000`).
- Booleans (`true` and `false`)
- Strings
  String literals can use single quotes (`'hello'`) or double quotes (`"hello"`), and the quote character can appear in the string escaped with a leading backslash (`"you said \"hello\", yes?"`).
- Nil (`nil`)
- Functions (see below)
- Coroutines (see below)
- Tables
  - Sequences, indexed from 1: `x = {2, 3, 5, 7, 11}` ; `x[1] == 2`
  - Mappings: `x = {a=1, b=2, c=3}` ; `x.a == x['a'] == 1`
  - Unset indexes evaluate to `nil` : `x[0] == nil`

As in Lua, all composite and custom data types are based on tables.

Arithmetic operators

Pico-8 supports these arithmetic operators:

- plus: `a + b`
- minus: `a - b`
- times: `a * b`
- divide: `a / b`
- modulo: `a % b`
- exponent: `a^b`

Arithmetic operators take numbers as arguments and return a number.

Relational operators

Pico-8 supports these relational operators:

- less than: `a < b`
- greater than: `a > b`
- less than or equal: `a <= b`
- greater than or equal: `a >= b`
- equal: `a == b`
- not equal: `a ~= b`; Pico-8 synonym: `a != b`

Relational operators take numbers as arguments and return either `true` or `false`.

Logical operators

Pico-8 supports the following logical operators:

- and: `a and b`
- or: `a or b`
- not: `not a`

In logical expressions, `false` and `nil` are false, all other values are true.

Logical expressions "short circuit," and stop evaluating expressions left to right as soon as the value of the expression is known. For example, `foo() and bar()` calls `foo()`, and only calls `bar()` if `foo()` returns true.

A useful equivalent of the "ternary operator" from the C language is `a and b or c`, which evaluates to `b` if `a` is true, or `c` if `a` is false.

String operators

Pico-8 supports the string concatenation operator: `a..b`

`a` must be a string. `b` can be a string or a number (which is coerced into a string).

No other type of value can be concatenated with a string. You can use the tostr() function to convert other values to strings:

```
print("a > b: "..tostr(a > b))
```

You can determine the length of a string using the sequence length operator (#):

```
x = "hello there"
print(#x)            -- 11
```

See also sub().

Assignment operators

As in Lua, you can assign a value to a variable or table slot

`myvar = value`

Pico-8 adds shortcut assignment operators for the arithmetic operators, often found in other languages like C:

```
myvar += value       -- myvar = myvar + value
myvar -= value       -- myvar = myvar - value
myvar *= value       -- myvar = myvar * value
myvar /= value       -- myvar = myvar / value
myvar %= value       -- myvar = myvar % value
```

(There is no `^=` assignment operator.)

Variables

Pico-8 supports global variables accessible to the entire program, and local variables accessible only within the function where they are declared. If a variable is not declared as local, then it is global.

```
-- a global variable
player_pos = {20, 60}

function move_player(newx, newy)
  player_pos = {newx, newy}
end

function circumference(r)
  -- a local variable
  local pi = 3.14
  return 2 * pi * r
end
```

Caution: A mistyped variable name is often interpreted as a global variable with no value assigned, which evaluates to `nil`. Most uses of unexpectedly `nil` values result in a runtime error, but these are not always easy to find.

Functions

A function is a collection of statements that can be executed by calling it. The behavior of a function can be parameterized, and a function may return a value as a result.

```
function distance(x1, y1, x2, y2)
  return sqrt((x2 - x1)^2 + (y2 - y1)^2)
end
```

Pico-8 includes many built-in global functions, such as `sqrt()` in this example. See APIReference.

If control reaches a `return` statement, then the function exits. If `return` includes a value, then the function call evaluates to that value. If control reaches the end of the function's statement block without seeing a `return` statement, then the function returns `nil`.

A function in Lua is a "first class" value, just like other values. A named function in the outermost block is equivalent to a global variable whose value is a function. A named function can also appear inside another function, which is equivalent to a local variable.

A function can omit the name if it is called right away or otherwise used as a value (an "anonymous function").

```
x = {2, 3, 5, 7, 11}
foreach(x, function(v) print(x^2) end)
```

Parentheses can be omitted when there is only one argument and it is a table or a literal string:

```
add_particle{type=snow, x=rnd(128), y=0, c=7}
print"hello, winter!"
```

Lua functions are lexically scoped.

Conditional statements

The Lua `if` statement takes a condition and a block of statements, and executes the statements only if the condition is true:

```
if score >= 1000 then
  print("you win!")
  score = 0
end
```

An `if` statement can include any number of `elseif` sections to test multiple conditions until one is found true, and an optional final `else` section to evaluate if none of the conditions were true. The general form is:

```
if cond1 then
  ...
elseif cond2 then
  ...
else
  ...
end
```

Pico-8 extends standard Lua with an abbreviated, single-line `if` statement, differentiated by having parentheses around the condition and no `then` or `end` keywords. You may use `else`, but it must be on the same line. There is no support for `elseif`. Some examples:

```
if (cond1) print("cond1")

if (cond2) print("cond2") else print("not cond2")
```

A common misconception is that you must compare a boolean variable to `true` or `false` in an `if` statement, but a condition *is* a boolean, so you can drop any boolean variable straight into an `if` statement:

```
-- these two blocks are functionally identical

if cond == true then
  print("cond")
elseif cond == false then
  print("not cond")
end

if cond then
  print("cond")
else
  print("not cond")
end
```

The only time you might need to compare a boolean variable to a value is when it may not be initialized yet. When variables are uninitialized, their value will be `nil`. When a `nil` value is used as a conditional expression, it is treated the same as `false`. Thus, if you have a boolean variable that may not be set yet, you should check for `nil` and initialize it before testing it as a boolean.

while and repeat loops

The `while` statement executes a block of statements repeatedly as long as a given conditional expression is true:

```
x = 0
while x < 5 do
  print(x)
  x += 1
end
```

The `repeat` statement executes a block of statements repeatedly until a given conditional expression is true:

```
x = 0
repeat
  print(x)
  x += 1
until x > 4
```

`while` does not execute its block if the condition is already false. `repeat` always executes its block at least once, then tests the condition.

The `break` statement anywhere in a loop terminates the loop immediately without testing the condition. If the loop is nested inside another loop, only the innermost loop is terminated.

for loops

There are two kinds of for loops in Lua. The numeric for loop traverses a numeric sequence from start to end, using an optional "stride" (with a default stride of 1).

```
-- draws 16 color bars
for c=0,15 do
  rectfill(c*8, 0, c*8+7, 127, c)
end

-- prints 1, 3, 5, 7, 9
for i=1,10,2 do
  print(i)
end
```

The other kind of for loop is the "generic for." In Pico-8, this is most commonly used with the all() and pairs() built-ins for traversing tables:

```
tbl = {2, 3, 5, 7, 11}
for v in all(tbl) do
  print(v)
end

tbl = {a=1, b=2, c=3}
for k,v in pairs(tbl) do
  print(k.."="..v)
end
```

The generic for statement expects the "in" expression to return a special kind of value called an *iterator*. The built-ins all() and pairs() return iterators. For information on how to create your own iterators, see Iterators and the Generic for in the Lua manual.

As with `while` and `repeat`, you can use the `break` statement to terminate a for loop immediately.

Tables

Tables are the primary composite data structure in Lua. They are used as containers, especially sequences (like lists or arrays) and mappings (also known as dictionaries or hashtables). Tables can be used as general purpose objects, and when combined with metatables (see below) can implement object-oriented concepts such as inheritance.

See Tables for a complete introduction to using tables in Pico-8.

Methods

A method is a function that is stored as a value in a table. Lua has special syntax for defining and calling methods that cause the table itself to be passed to the method as a parameter named `self`.

```
ball = {
  xpos = 60,
  ypos = 60
}

function ball:move(newx, newy)
  self.xpos = newx
  self.ypos = newy
end

print(ball.xpos)           -- 60

ball:move(100, 120)
print(ball.xpos)           -- 100
```

In both the definition and the method call, using the colon (`:`) instead of the dot (`.`) implies the `self` behavior. If you define the method as a simple property of the table, you must remember to explicitly mention the `self` argument. This is equivalent:

```
ball = {
  xpos = 60,
  ypos = 60,

  -- without the colon syntax, must mention self argument explicitly
  move = function(self, newx, newy)
    self.xpos = newx
    self.ypos = newy
  end
}

-- using the colon, ball is passed as self automatically
ball:move(100, 120)

-- using the dot, must pass self explicitly
ball.move(ball, 100, 120)
```

Metatables

The metatable for a table defines the behavior of using the table as a value with Lua operators. You can customize a table's metatable to specify custom operator behaviors.

The most common use of metatables is to implement object-oriented inheritance by redefining the `__index` operator. The new definition tells Lua to check for a given property on a parent prototype object if the property is not set on the current object.

```
function myclass:new(o)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  return o
end
```

Pico-8 supports the setmetatable() and getmetatable() built-ins. It does not support other related Lua functions.

See setmetatable() for more information, links to references, and a more complete example.

Coroutines

A coroutine is a special kind of function that can yield control back to the caller without completely exiting. The caller can then resume the coroutine as many times as needed until the function exits.

Pico-8 supports coroutines using built-in global functions instead of Lua's `coroutine` library. See cocreate(), coresume(), costatus(), and yield.

See cocreate() for more information, links to references, and a complete example.

Differences from Lua

Pico-8 does not include the Lua standard library. See Math for a description of the Pico-8 mathematical functions. See APIReference for a complete list of Pico-8 built-in functions.

josefnpat wrote a list of technical differences between Pico-8's Lua and the official Lua 5.2.

- List of differences: https://gist.github.com/josefnpat/bfe4aaa5bbb44f572cd0
- Discussion: http://www.lexaloffle.com/bbs/?tid=2611