EVIKA

Lua

Встроенный язык программирования контроллеров *LogicMachine*

ВРЕМЕННОЕ

1.1.1.3 2015.08.23.

Назначение руководства.

В данном документе описываются основные положения языка программирования Lua, встроенного в универсальные контроллеры EVIKA LogicMachine.
Документ содержит сведения о синтаксисе, базовых конструкциях языка, а также особенности реализации Lua на LogicMachine: редактор, средства отладки и описание ресурсов.

Руководство предназначено для Инженеров:

- Программистов;
- Проектировщиков;
- Инсталляторов.

СОДЕРЖАНИЕ

▲ ... ►

Авторские права

Уведомление

EVIKA сохраняет за собой право вносить изменения в данный документ без оповещений.
EVIKA не несет ответственности за любые ошибки, которые могут быть допущены в данном документе.

Товарные знаки

Товарный знак ***EVIKA*** принадлежит компании ооо "Эвика Системс". Настоящим подтверждается, что все прочие наименования и товарные знаки являются собственностью их владельцев.

Техническая поддержка

Ремонт устройств реализованных на территории РФ и СНГ осуществляется ***EVIKA***.

Ремонт устройств реализованных на территории стран ЕвроСоюза осуществляется ***Embedded Systems*** SIA.

Служба технической поддержки:

Время работы:	по рабочим дням Понедельник, .., Пятница 08:30 .. 18:30 (Москва).
Телефон:	8-800-775-06-34 (звонки из любых регионов России - бесплатны).
E-Mail:	Support@***Evika***.Ru
Site:	www.***Evika***.Ru

Обзор Lua для программистов

Переменные

Переменные используются для хранения значений в процессе выполнения программы. В Lua есть три вида переменных:

- глобальные,
- локальные,
- поля таблиц.

Переменная может быть глобальной или локальной. Для определения локальной переменной нужно ставить ключевое слово **local**. Область видимости локальной переменной: внутри пары **do .. end** или файле скрипта.

В Lua (как, например и в javascript), имеется специальное значение **nil** (аналог null), означающее отсутствие 'содержательных' данных. До первого явного присвоения значением переменной является **nil**. При инициализации переменных им не нужно указывать тип - Lua сам разберется, и объявление переменных фактически не требуется, так как объявлением является присвоение им значения.

Преобразование из одного типа в другой часто происходит неявно, что упрощает читаемость программы. При этом для любой переменной можно в процессе выполнения узнать ее тип с помощью специального оператора **type()**. Более того, тип одной и той же переменной может быть изменен в процессе ее существования при присвоении нового значения (другого типа). Это явление называется динамической типизацией. Как и C/C++, Lua является регистро-чувствительным. Переменные **AAA** и **aaa** - это разные переменные!

Переменные в языке Lua могут быть объявлены в произвольном месте.

В Lua восемь основных типов данных:

nil	неопределенный
boolean	логический
number	числовой, аналог типа данных "double" C++
string	строковый
function	функция
table	таблица
userdata	пользовательские данные, используется для связи с программами написанными на языке C
thread	поток

Типы userdata и thread в скриптах LogicMachine обычно не используются.

Таблицы, функции, потоки и userdata являются объектами, т.е. переменная содержит не непосредственное значение, а ссылку на объект, со всеми вытекающими последствиями.

Присваивание, передача параметров и возврат результата из функции оперируют только ссылками на значения, и эти операции никогда не ведут к созданию копий переменных.

Примеры объявления переменных:

```
    a = 5                              -- number, числовой
    b = true                           -- boolean, логический (true/false)
  str = 'строковая переменная'
str_1 = "тоже строковая переменная"
str_2 = [[ много строчек
                 много строчек
                     и еще немного ]]
```

▲ ... ◄

Комментарии

Для написания комментария к коду или отключения куска кода можно использовать комментарии. Комментарии бывают двух типов - строчные и блочные. Блочные комментарии используются когда требуется закомментарить несколько строк.

Примеры комментариев:

```
a = 5         -- это строчный комментарий, всё что располагается
              --   после символов сдвоенных тире и до конца строки
b = 6         --[[ блочный комментарий начинается с сдвоенных символов
                   тире и квадратных скобок "[[".

                   Выше расположена пустая строка, она тоже часть блока
                   комментария.
                   Блочный Комментарий заканчивается сдвоенными
                   обратными скобками
                ]]

-- метки комментариев могут располагаться в любом месте строки,
--[[ но не должны разрывать операторы ]]
```

▲ ... ◄

Таблицы

Таблица в языке Lua - это ассоциативный массив, представляющий собой множество пар

```
Table::
{(key1, value1), (key2, value2), .., (keyn, valuen)  }
```

, для которого определена операция [] (получение элемента *valueN* по ключу *keyN*).

В отличие от низкоуровневых языков программирования, элемент пары **value** может быть различных типов. Он может содержать как числовые, строковые, логические данные, так и вложенные таблицы и функции (функции в языке Lua такие же переменные как и числа).

Индексом ***keyN*** может быть число, тогда мы получаем обычный массив (вектор) нумерованных переменных. В этом случае индексы начинаются с 1. В качестве элемента пары *keyN* в таблицах можно использовать не только строки и числа, а вообще всё что угодно, в том числе и другие таблицы. Обращаться к элементам можно по этим самым индексам. Само собой, при попытке взять элемент, которого не существует, вам вернется значение **nil**.

Вот несколько примеров объявления таблиц:

```
 t = {}                              -- пустая таблица
t1 = {3,2,1}                         -- массив из 3 чисел,
                                     -- t[1] <-- 3
                                     -- t[2] <-- 2
                                     -- t[3] <-- 1

t2 = {key1='значение1', key2=3.566}
t3 = {{1,2}, {3,4}}                  -- двумерный массив 2x2,
                                     -- t3[2][1] <-- 3

phones = {['Иван Иванов']  = '+7(495) 333-33-33',
          ['Таня Иванова'] = '+7(495) 444-44-44'
         }
```

Для таблиц со строковыми ключами, которые записаны латинскими буквами и не содержат в себе пробелов (то есть по сути являются идентификаторами языка), можно использовать упрощенный синтаксис при обращении к элементам таблицы, делая ее похожей на структуру (или класс). А именно:

```
t = {}
t.field_x = 10   -- соответствует t['field_x'] = 10
t.field_y = 20   -- соответствует t['field_y'] = 20
```

Для получения размера таблицы можно использовать оператор #, но будет посчитано только то количество элементов, которые имеют числовые индексы. То есть:

```
t1 = {['a'] = 'text1', ['b'] = 'text2'}    -- #t1 = 0

t2 = {[1]   = 'text1', ['b'] = 'text2'}    -- #t2 = 1
```

▲ ... ◄

Операторы

Порции

Единица исполнения Lua называется *chunk* **(порция)**.

Порция – это любая последовательность операторов Lua. Операторы в порции могут разделяться запятыми:

```
chunk ::=
{stat [';']}
```

Пустого оператора в языке нет, поэтому выражение: **';;'** не допустимо.

Lua воспринимает порцию как неименованную функцию с произвольным набором параметров. Порция может определять локальные переменные и возвращать значения.

Порция может храниться в файле или в строке базовой программы.

В момент запуска порции на выполнение осуществляется компиляция ее в промежуточный байт-код (инструкции для виртуальной машины). Затем полученный код исполняется виртуальной машиной.

▲ ... ◄

Блоки (block)

Блок это список операторов; синтаксически блок тождественно равен порции *chunk*:

```
block ::=
chunk
```

Блок операторов *stat* может быть явно ограничен, таким образом представляется составной оператор:

```
stat::=
do block end
```

С помощью составных операторов можно ограничивать области видимости локальных переменных. Также составные операторы используются в циклах и условном операторе.

▲ ... ◄

Присваивание

Lua поддерживает списочное присваивание. В общем случае, оператор присваивания выглядит как:

- список переменных,
- символ `'='`,
- список выражений.

Элементы списков указываются через запятую:

```
stat ::=
varlist1 '=' explist1
```

где:

```
varlist1   = var {',' var}
explist1   = exp {',' exp}
```

Перед выполнением присваивания список переменных *varlist1* согласовывается по длине со списком *explist1* выражений. Если список справа длиннее, то его последние элементы просто отбрасываются. Если короче, то недостающие позиции *varlist1* дополняются значениями nil. Если список операторов оканчивается вызовом функции, то перед согласованием все возвращаемые оттуда значения вставляются в список *explist1* (за исключением случаев, когда вызов взят в скобки).

Перед выполнением присваивания вычисляется значение всех выражений.

Примеры:

```
i = 3
i, a[i] = i+1, 20      -- На момент вычисления
```

Присваивание означает, что переменной **a[3]** присваивается значение **20**, потому что i в выражении **a[i]** имеет то же самое значение, что и в момент вычисления выражения **i+1.** Аналогично, строка

```
x, y = y, x
```

является простым способом обмена значениями двух переменных (при «традиционном» способе требуется дополнительная переменная).

▲ … ◄

Управляющие конструкции

Операторы **if**, **while**, и **repeat** имеют обычное значение и знакомый синтаксис:

```
stat ::=
while exp do block end

stat ::=
repeat block until exp

stat ::=
if exp then block {elseif exp then block} [else block] end
```

В Lua также имеется выражение **for** в двух вариантах (см. ниже).

Логическое выражение в управляющих конструкциях может возвращать любое значение. Значения **false** и **nil** считаются ложными. Все остальные значения считаются истинными (в том числе значение 0 и пустая строка !).

Условие проверки в цикле **repeat**–**until** входит в блок, поэтому, поэтому в условии можно ссылаться на локальные переменные, описанные внутри цикла.

Выражение **return** используется для того, чтобы возвратить значения из функции или порции. Синтаксис оператора **return** позволяет функции или порции вернуть несколько значений:

```
stat ::=
return [explist1]
```

Оператор **break** используется для досрочного выхода из циклов **while**, **repeat** и **for**:

```
stat ::=
break
```

Break прерывает цикл, в котором он расположен, внешние циклы продолжают выполнение.

▲ … ◄

Оператор For

Оператор **for** допускает простую и расширенную формы записи.

В простой форме **for** выполняет блок кода до тех пор, пока переменная цикла, изменяющаяся в арифметической прогрессии, не достигнет установленного порога.

```
stat ::=
for Name ' = ' Vstart ',' Vend [', ' Vstep] do block end
```

block повторяется для переменной цикла `name` начиная со значения первого выражения `Vstart`, до тех пор, пока `name` будет менее или равна значению второго выражения *Vend* с шагом третьего выражения *Vstep*.

Таким образом, запись:

```
for v = vstart, vend, vstep do block end
```

эквивалентна коду:

```
do

  -- присваивание начальных значений
  local var, limit, step = tonumber(vstart), tonumber(vend), tonumber(vstep)
  -- проверка на достаточность
  if not (var and limit and step) then error() end

  -- присваивание начальных значений
  while (step > 0 and var <= limit) or (step <= 0 and var >= limit) do
    local v = var

    block              -- операторы цикла

    var = var + step   -- новое значение переменной цикла
  end
end
```

Обратите внимание, что:

- Все три выражения **vstart**, **vend**, **vstep** вычисляются только один раз перед началом цикла, причем полученные значения приводятся к числам.
- **var**, **limit**, и **step** - локальные переменные, невидимые вне цикла.
- Если выражение **vstep** (шаг) отсутствует, то по умолчанию используется 1
- Для выхода из цикла **for** используется **break**.
- Переменная **v** является локальной для цикла; вы не сможете использовать ее значение после выхода из цикла **for**. Если Вам необходимо значение этой переменной, присвойте его другой нелокальной переменной перед выходом из цикла.

Расширенная форма оператора *for* реализована с использованием функций итераторов.

На каждом цикле для получения нового значения переменной цикла вызывается итератор. Цикл заканчивается, когда итератор вернет nil. Синтаксис расширенного оператора **for**:

```
stat ::=
for namelist in explist1 do block end

namelist ::=
Name {', '~ Name}
```

Запись

```
for var_1, ···, var_n in explist do block end
```

можно представить как :

```
do

  local f, s, var = explist
  while true do

    local var_1, ···, var_n = f(s, var)
    var = var_1

    if var == nil then break end

    block

  end
end
```

Заметим, что

- *explist* вычисляется только однажды. Его результатом является функция-итератор, таблица состояний и начальное значение индекса
- *f*, *s*, и *var* неявные переменные, именованные здесь для примера

- Выйти из цикла можно с помощью оператора break.
- Переменная *var_1* является локальной; Вы не сможете использовать ее значение после выхода из for. Если вам необходимо ее значение, заранее сохраните его в другой нелокальной переменной.

Например, "заготовки" LM :

```
-- arraytable = { 'a', 'b', 'c' }
for index, value in ipairs(arraytable) do
  dosomething()
end

-- hashtable = { a = 1, b = 2, c = 3 }
for key, value in pairs(hashtable) do
  dosomething()
end
```

▲ … ◄

Арифметические операции

Lua поддерживает обычные арифметические операции:

- Бинарные,
- + (сложение),
- - (вычитание),
- * (умножение),
- / (деление),
- % (остаток от деления),
- ^ (возведение в степень),
- унарный минус - (изменение знака числа).

Если операнды являются числами или строками (которые могут быть преобразованы в числа), то операции выполняются обычным образом.

Возведение в степень работает для любого показателя степени. Например, x^(-0.5) подсчитывает величину, обратную квадратному корню из x.

Остаток от деления определен как

```
a % b == a - math.floor(a/b)*b
```

▲ … ◄

Операции сравнения

Операции сравнения в Lua:

- == (по типу и значению)
- ~= (неравенство)
- <
- >
- <=
- >=

Эти операции всегда возвращают false или true.

Сравнение на равенство (==) сначала сравнивает типы операндов. Если типы различны, то результатом будет false. Иначе сравниваются значения операндов. Числа и строки сравниваются обычным способом. Объекты (таблицы, пользовательские данные, потоки и функции) сравниваются по ссылке: два объекта считаются равными, только если они являются одним и тем же объектом. Создаваемый объект (таблица, пользовательские данные, поток или функция) не может быть равен ни одному из уже существующих.

Оператор "~=" прямо противоположен оператору равенства "==".

Операторы сравнения на больше-меньше работают следующим образом: Если оба параметра - числа, то они сравниваются как обычно. Если оба параметра строки, то их значения сравниваются в соответствии с лексикографическим порядком. Во всех остальных ситуациях будет вызван метаметод "lt" или "le".

▲ … ◄

Логические операции

В Lua это операции:

- not (не),
- and (и),
- or (или).

Так же, как и в управляющих конструкциях, все логические операции рассматривают false и nil как ложь, а все остальное как истину.

Операция not (отрицание) всегда возвращает false или true.

Операция and (конъюнкция) возвращает свой первый параметр, если его значение false или nil; в противном случае and возвращает второй параметр.

Оператор or (дизъюнкция) возвращает первый параметр, если его значение отлично от nil и false; в противном случае возвращает второй параметр.

Оба оператора вычисляют второй операнд только в случае необходимости.

Примеры:

```
10 or 20            --> 10
10 or error()       --> 10
nil or "a"          --> "a"
nil and 10          --> nil
false and error()   --> false
false and nil       --> false
false or nil        --> nil
10 and 20           --> 20
```

▲ … ◄

Конкатенация (соединение строк)

Оператор конкатенации (соединения) строк в Lua обозначается двумя точками '..'. Если оба операнда являются строками или числами, то они будут преобразованы в строки согласно правилам. Иначе будет вызван метаметод "concat".

▲ … ◄

Получение длины

Операция получения длины переменой обозначается унарным оператором #.

В результате применения операции к строке, возвращается количество байт в ней (в обычном понимании это длина строки, в которой каждый символ занимает 1 байт).

Длиной таблицы t считается количество значений целочисленного индекса в последовательности элементов начинающейся с первого элемента таблицы. Последним элементом этой последовательности будет последний элемент таблицы или последний перед пропущенным элементом.

Для регулярных массивов от 1 до n, не содержащих значений nil, длиной является n, то есть индекс последнего значения. Если в массиве присутствуют "дыры" (т.е., значения nil между ненулевыми значениями), то значением #t является индекс элемента, непосредственно предшествующего элементу nil (поэтому любое значение nil по сути означает конец массива).

▲ … ◄

Приоритет операций

Приоритет операций Lua показан на таблице ниже. Самым высоким приоритетом обладает операция возведения в степень, далее по убыванию:

- or
- and
- <
- >
- <=
- >=
- ~=
- ==
- +
- -
- %
- not
- #
- -(unary)
- ^

Как обычно, для изменения порядка вычисления выражений Вы можете использовать скобки.

Конкатенация '..' и возведение в степень '^' - право ассоциативные операторы. Все остальные бинарные операторы лево ассоциативные.

▲ … ◄

Конструкторы таблиц

Конструкторы таблиц тоже относятся к выражениям. Обработка любого встречающегося в коде конструктора ведет к созданию новой таблицы. С помощью конструкторов можно создать как пустые, так и частично либо полностью заполненные таблицы. Полное описание синтаксиса конструкторов:

```
tableconstructor::=
'{' [fieldlist] '}'

fieldlist::=
field {fieldsep field} [fieldsep]

field::=
'[' exp ']' '=' exp | Name '=' exp | exp

fieldsep::=
',' | ';'
```

Каждое поле вида [exp1] = exp2 добавляет в новую таблицу значение exp2 с ключом exp1. Поле вида name = exp эквивалентно ["name"] = exp. Поле вида exp эквивалентно [i] = exp, где i – целочисленный автоинкрементный счетчик, начинающийся с 1. Поля в других форматах не оказывают влияния на этот счетчик. Например:

```
a = { [f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45 }
```

эквивалентно

```
do
  local t = {}
  t[f(1)] = g
  t[1]    = "x"
  t[2]    = "y"
```

```
  t.x     = 1
  t[3]    = f(x)
  t[30]   = 23
  t[4]    = 45

  a       = t
end
```

Если последнее поле в списке задано в форме exp, и exp – это вызов функции или неопределенный список параметров, то все значения, возвращаемые этим выражением, последовательно включаются в этот список. Чтобы этого избежать, необходимо заключить вызов функции (или список неопределенных параметров) в скобки.

▲ ... ◄

Вызовы функций

Вызовы функций в Lua имеют следующий синтаксис:

```
functioncall ::=
prefixexp args
```

В вызове функции сначала вычисляются префиксное выражение и аргументы. Если значение префиксного выражения имеет тип *function*, то эта функция будет вызвана с указанными аргументами. В противном случае вызывается метаметод “call”, параметрами которого будет значение префиксного выражения, за которым следуют первоначальные аргументы.

Форма записи:

```
functioncall ::=
prefixexp ':' Name args
```

может использоваться для вызова "методов". Запись v:name(args) синтаксически аналогична записи v.name(v,args), только v вычисляется один раз.

Аргументы описываются следующим образом:

```
args ::=
'(' [explist1] ')'

args ::=
tableconstructor

args ::=
String
```

Все выражения вычисляются перед вызовом. Вызов в форме f{fields} синтаксически аналогичен f({fields}); то есть список аргументов является по сути новой таблицей. Вызов в форме f'string' (или f"string" или f[[string]]) синтаксически равен f('string'); в данном случае список аргументов - единственная символьная строка.

Исключением в довольно свободном синтаксисе Lua является правило, по которому нельзя переходить на новую строку непосредственно перед символом '(' в вызове функции. Это ограничение позволяет избежать некоторой двусмысленности в языке. Если вы напишите

```
a = f
(g).x(a)
```

Lua трактует эту запись как выражение a = f(g).x(a). Поэтому, если вам нужно 2 выражения, вы должны добавить ";" между ними. Если вы действительно хотите вызвать f, вы необходимо убрать переход на новую строку перед (g).

Вызов в форме return *function call* называется *концевым вызовом*. Lua также поддерживает *концевой вызов «себя»* (или рекурсивный *концевой вызов*): в этом случае вызванная функция использует стек вызывающей функции. Поэтому количество вложенных концовых вызовов может быть любым. Заметим только, что концевой вызов стирает отладочную информацию о вызывающей функции. Синтаксис концевого вызова допускает только единичный вызов функции после оператора return. Таким образом, return вернет в точности тот результат, что вернет вызов функции. Ни один из представленных ниже примеров не является допустимым концевым вызовом:

```
return (f(x))        -- список-результат обрезается
```

```
return 2 * f(x)      -- удвоение результата функции
return x, f(x)       -- возвращается несколько значений
f(x); return         -- результат вызова отбрасывается
return x or f(x)     -- список-результат обрезается
```

▲ ... ◄

Объявление функций

Синтаксис объявления функций:

```
function ::=
function funcbody

funcbody ::=
'(' [parlist1] ')' block end
```

Или в упрощенном виде

```
stat ::=
function funcname funcbody

stat ::=
local function Name funcbody

funcname ::=
Name {'.' Name} [':' Name]
```

Выражение

```
function f () body end
```

транслируется в

```
f = function () body end
```

Выражение:

```
function t.a.b.c.f () body end
```

транслируется в:

```
t.a.b.c.f = function () body end
```

Выражение:

```
local function f () body end
```

транслируется в:

```
local f; f = function () body end
```

а не в:

```
local f = function () body end
```

(Разница проявится в том случае, если в теле функции используется имя этой функции, например при рекурсивном вызове)

Объявление функции является выполняемым выражением, его результатом будет значение типа function. Когда Lua перекомпилирует порцию, тела всех упоминающихся в ней функций также перекомпилируются. Таким образом, всякий раз, когда Lua обрабатывает объявление функции, функция уже конкретизирована (или замкнута). Этот конкретный экземпляр функции (или замыкание) и является конечным значением выражения «объявление функции». Различные экземпляры одной и той же функции могу ссылаться на различные внешние локальные переменные и иметь различные таблицы окружения.

Параметры функции фактически являются локальными переменными, которые инициализированы входными значениями:

```
parlist1 ::=
```

```
namelist [',' '...'] | '...'
```

В момент вызова функции длина списка передаваемых параметров приводится в соответствие спецификации, если это не *функция* с неопределенным количеством параметров. Для функций с неопределенным количеством параметров такая коррекция не проводится; все входные параметры попадают в функцию в виде неопределенного выражения, которое также обозначается с тремя точками. Значением этого выражения является список всех полученных входных параметров, как в случае множественного результата функции. Если неопределенное выражение используется внутри другого выражения или в середине списка выражений, то его значение-список урезается до одного элемента. Если это выражение стоит в конце списка выражений, урезания не происходит (если конечно вызов не заключен в круглые скобки).

Например:

```
function f(a, b) end
function g(a, b, c, d,...) end
function r() return 1,2,3 end

-- передача параметров для function f
f(3)             -- a=3, b=nil
f(3, 4)          -- a=3, b=4
f(3, 4, 5)       -- a=3, b=4
f(r(), 10)       -- a=1, b=10
f(r())           -- a=1, b=2

-- передача параметров для function g
g(3)             -- a=3, b=nil, ...
g(3, 4)          -- a=3, b=4, c=nil, ...
g(3, 4, 5, 8)    -- a=3, b=4, c=nil, d=nil, ...
g(5, r())        -- a=5, b=1, c=2, d=3, ...
```

Результаты возвращаются из функции оператором return. Если управление достигает конца функции, а оператор return не встретился, то функция завершается и ничего не возвращает.

Объявление методов.

Синтаксис с двоеточием ':' используется для определения методов. Эти функции неявно получают параметр self в качестве первого аргумента. Таким образом, выражение:

```
function t.a.b.c:f (params) body end
```

аналогично

```
t.a.b.c.f = function (self, params) body end
```

▲ ... ◄

Библиотека табличных функций

Большинство функций в библиотеке таблиц предполагают, что таблица является массивом или списком. Для этих функций, когда мы говорим о параметре "длина" таблицы, мы имеем в виду результат оператора length.

Над таблицами можно проводить следующие операции:

▲ … ◄

table.concat (table [, sep [, i [, j]]])

Задан массив в котором все элементы – строки или числа, возвращает

table[i]..sep..table[i+1] ··· sep..table[j].

Значение по умолчанию для sep – пустая строка, значение по умолчанию для i – 1, а для j – длина таблицы.

Если i больше j, функция возвращает пустую строку.

▲ … ◄

table.insert (table, [pos,] value)

Вставляет элемент value в позицию pos в table, сдвигая вверх остальные элементы. Значение по умолчанию для pos равно n+1, где n это длина таблицы.

Вызов table.insert(t,x) добавляет x в конец таблицы t.

table.maxn (table)

Возвращает наибольший положительный числовой индекс заданной таблицы, или zero если таблица не имеет положительных числовых индексов. Для выполнения запроса эта функция перебирает все индексы таблицы.

▲ … ◄

table.remove (table [, pos])

Удаляет из table элемент в позиции pos, сдвигая вниз остальные элементы, если это необходимо. Возвращает значение удаленного элемента. Значение по умолчанию для – n, где n – длина таблицы.

Вызов table.remove(t) удаляет последний элементы таблицы t.

Примечание: использование insert-remove со значениями по умолчанию позволяет работать с таблицей как со стандартным LIFO – стеком.

▲ … ◄

table.sort (table [, comp])

Сортирует все элементы внутри таблицы в заданном порядке.

Если параметр comp задан, то он должен быть функцией, принимающей 2 параметра. В процессе сортировки в эти параметры будут подставляться сравниваемые элементы таблицы. Функция должна возвращать true, если первый из них должен идти впереди второго и наоборот. Если comp не задан, то вместо него будет использован стандартный оператор Lua "<".

Алгоритм сортировки не стабилен; в том смысле, что равные элементы могут быть переставлены в процессе сортировки.

простая сортировка:

```
t = {2,4,1,3,6}
table.sort(t)      -- 12346
```

Сортировка, использующая функцию:

Предположим у Вас есть структура описывающая помещения, состоящих из поля prior - приоритет обогрева, и текущей температуры:

```
rooms = {{prior = 1, temp = 25 },
         {prior = 2, temp = 23 },
         {prior = 2, temp = 18 }
        }
-- функция сравнения для сортировки
function sorter(a,b)
-- если приоритеты равны, то смотрим на температуру и определяем в какой
-- комнате большая температура
  if(a.prior == b.prior) then

    return a.temp > b.temp
  else
    -- иначе определяем по приоритету
    return a.prior > b.prior
  end
end


table.sort(rooms,sorter)
-- {temp:25,prior:1}, {temp:23,prior:2}, {temp:18,prior:2} до сортировки
-- {temp:23,prior:2}, {temp:18,prior:2}, {temp:25,prior:1} после
```

▲ ... ◄

Работа со строками

Библиотека предоставляет основные функции для работы со строками, такие как поиск и выделение подстрок, а также поиск по шаблону. Строки в Lua индексируются с 1 (а не 0, как в C). Индексы могут быть отрицательными и интерпретируются как индекс с конца строки. Т.е. последний символ имеет позицию -1, и т.д.

Библиотека работы со строками предоставляет все функции в таблице string. Она также устанавливает поле __index метатаблицы строк на таблицу string. Также, Вы можете использовать строковые функции в объектно-ориентированном стиле. Например, string.byte(s, i) может быть записано как s:byte(i).

string.byte (s [, i [, j]])

Возвращает числовые коды символов s[i], s[i+1], ···, s[j]. По умолчанию значение для i равно 1; по умолчанию значение для j равно i.

Напоминаем, что числовые коды символов не одинаковы на разных платформах.

string.char (···)

Принимает 0 или более целых чисел. Возвращает строку, длина которой равна количеству параметров, и каждый элемент строки установлен равным соответствующему параметру.

Напоминаем, что числовые коды символов не одинаковы на разных платформах.

string.dump (function)

Возвращает строку, содержащую двоичное представление данной функции function, так, что если после этого вызвать функциюloadstring на эту строку, мы получим копию функции. function должна быть Lua функцией без внешних локальных переменных (upvalues).

string.find (s, pattern [, init [, plain]])

Ищет первое вхождение шаблона pattern в строку s. Если поиск успешен, то find возвращает индексы s где найдено совпадение с шаблоном, т.е. где начинается и кончается; иначе возвращает nil. Третий, необязательный числовой параметр init указывает откуда начинать поиск; по умолчанию он равен 1, также он может быть отрицательным. Значение true в четвертом, необязательном параметре plain выключает возможность поиска по шаблону, в этом случае производится поиск подстроки как есть, т.е. считается, что она не содержит «шаблонных» ("magic") симолов. Помните, что если указан параметр plain, то параметр init должен быть указан тоже.

Если шаблон содержит захваты (captures) и поиск успешен, то захваченные значения также возвращаются, сразу после двух индексов.

string.format (formatstring, ···)

Возвращает параметры, передаваемые в функцию, отформатированные в соответствии первым параметром (который должен быть строкой). Строка формата должна строиться по тем же правилам, что и строка формата для семейства C функций printf. Отличие только в том, что опции/модификаторы *, l, L, n, p и h не поддерживаются, а также не поддерживается опция q. Опция q позволяет вернуть строку в формате, безопасно воспринимаемом Lua интерпретатором: строка выводится в двойных кавычках, а все двойные кавычки, перевод строки, символы с кодом 0 и обратный слеш внутри строки перекодируются (escaped). Например, вызов

```
string.format('%q', 'a string with "quotes" and \n new line')
```

будет возвращать строку:

```
"a string with \"quotes\" and \
 new line"
```

Опции c, d, E, e, f, g, G, i, o, u, X и x должны использоваться только для числовых параметров, а q и s - строковых.

Эта функция не принимает строковые параметры, содержащие символы с кодом 0, кроме параметров для формата, имеющего опцию q.

string.gmatch (s, pattern)

Возвращает итератор, который, при каждом вызове, возвращает следующее захваченное значение. Если шаблон pattern не содержит захватов (captures), то простое сравнение будет выполнено при каждом вызове.

Например,следующий цикл

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

будет проходить по всем словам в строке s, печатая их по одному в строке. Следующий пример собирает все парные ключи (pairs key)=value из указанной строки в таблицу:

```
t = {}
s = "from=world, to=Lua"
   for k, v in string.gmatch(s, "(%w+)=(%w+)") do
   t[k] = v
end
```

Для данной функции, символ '^' в начале шаблона не работает как признак поиска с начала строки, поскольку это помешает итерации.

string.gsub (s, pattern, repl [, n])

Возвращает копию s в которой все вхождения pattern заменяются на repl, который может быть строкой, таблицей или функцией. gsub также возвращает как второе значение – общеее количество проведенных подстановок.

Если repl строка, то используется ее значение для замены. Символ % работает как символ со специальным назначением: любая последовательность в repl в виде %n, где n от 1 до 9, заменяется на n-ную захваченную подстроку (см. ниже). Последовательность %0 заменяется на найденную подстроку. Последовательность %% считается как одиночный символ %.

Если repl является таблицей, то она запрашивается для каждого сравнения, с использованием первого захваченного значения как ключ; если шаблон не содержит захватов, то используется результат простого сравнения как ключ.

Если repl является функцией, то эта функция вызывается каждый раз, когда обнаруживается совпадение. В качестве параметров ей передаются все захваченные подстроки; если шаблон не содержит захватов (captures), то передается результат сравнения как один параметр.

Если значение, возвращаемое таблицей или функцией является строкой или числом, то это значение используется для замены; в противном случае, если значение равно false или nil, то замена не производится (т.е. найденное значение остается без замены).

Необязательный последний параметр n ограничивает максимальное количество замен. Например, если n равно 1, то будет выполнено не более одной замены.

Несколько примеров

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"
x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"
x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"
x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"
x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
    return loadstring(s)()
            end)
--> x="4+5 = 9"
local t = {name="lua", version="5.1"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.1.tar.gz"
```

string.len (s)

Возвращает длину строки, переданной в качестве параметра. Пустая строка "" имеет длину 0. Вложенные символы с кодом 0 также считаются как символ, т.е. "a\000bc\000" имеет длину 5.

string.lower (s)

Возвращает копию строки s, где все большие буквы заменены на маленькие. Все остальные символы остаются неизменными. Понятие «большие буквы» зависит от текущей кодовой страницы.

string.match (s, pattern [, init])

Поиск первого вхождения шаблона pattern в строку s. В случае обнаружения, match возвращает захваченные значения (captures); в противном случае возвращает nil. Если pattern не содержит захватов, то производится простое сравнение. Третий, необязательный числовой параметр init указывает с какого символа строки необходимо начинать поиск; по умолчанию этот параметр равен 1. Также он может быть отрицательным.

string.rep (s, n)

Возвращает строку, в которой содержится n копий строки s.

string.reverse (s)

Возвращает строку, в которой символы строки s расположены в обратном порядке.

string.sub (s, i [, j])

Возвращает подстроку строки s, которая начинается с символа с индексом i и продолжается до символа с индексом j; i и j могут быть отрицательными. Если j не указан, то считается, что он равен -1 (то же самое, что длина строки). В частности, вызов string.sub(s,1,j) возвращает начальную часть строки с длиной j, а string.sub(s, -i) возвращает конец строки s длиной i.

string.upper (s)

Принимает строку и возвращает копию этой строки, где все маленькие буквы меняются на большие. Все остальные остаются неизменными. Понятие «маленькие буквы» зависит от текущей кодовой страницы.

Математические функции

Эта библиотека – интерфейс к стандартной библиотеке math языка C. Она предоставляет все функции внутри (инкапсулированные???) в таблицу math.

math.abs (x)

Возвращает модуль x.

math.acos (x)

math.acos (x)

math.asin (x)

Возвращает аркчинус x (в радианах).

math.atan (x)

Возвращает арктангенс x (в радианах).

math.atan2 (x, y)

Возвращает арктангенс x/y (в радианах), но использует знаки обоих параметров для вычисления «четверти» на плоскости. (Также корректно обрабатывает случай когда y равен нулю.)

math.ceil (x)

Возвращает наименьшее целое число, большее или равное x. (Округление «вверх»).

math.cos (x)

Возвращает косинус x (Угол – в радианах).

math.cosh (x)

Возвращает кошинус (гиперболический косинус) x.

math.deg (x)

Переводит угол, заданный в радианах (x)в градусы.

math.exp (x)

Возвращает ex.

math.floor (x)

Возвращает наибольшее целое число, меньшее или равное x. (Округление «вниз»)

math.fmod (x, y)

Возвращает остаток от деления x на y.

math.frexp (x)

Возвращает m и e такие, что x = m2e, e – целое, а модуль m находится в интервале [0.5, 1) (либо ноль, если x равен нулю). (Разложение числа с фиксированной запятой).

math.huge

Значение HUGE_VAL, значение большее, либо равное любому числовому значению.

math.ldexp (m, e)

Возвращает m2e (e должно быть целым). (Восстановление значения по мантиссе и показателю).

math.log (x)

Возвращает натуральный логарифм x.

math.log10 (x)

Возвращает логарифм x по основанию 10.

math.max (x,···)

Возвращает максимальный из аргументов.

math.min (x, ···)

Возвращает минимальный из аргументов.

math.modf (x)

Возвращает два числа, Целую чать x и дробную часть x.

math.pi

Значение pi.

math.pow (x, y)

Возвращает xy. (Вы также можете использовать запись x^y для вычисления значения этой функции.)

math.rad (x)

Конвертирует угол x, заданный в градусах, в радианы.

math.random ([m [, n]])

Эта функция является интерфейсом к простейшему генератору псевдослучайных чисел rand, предоставляемому ANSI C. (Нет никаких гарантий по поводу его статистических свойств.)

При вызове без аргументов, возвращает псевдо-случайное действительной число в интервале [0,1). При вызове с аргументом m, math.random возвращает псевдослучайное целое число из отрезка [1, m]. При вызове с двумя аргументами – m и n, math.random возвращает псевдослучайное целое число из отрезка [m, n].

math.randomseed (x)

Инициализирует генератор псевдослучайных чисел параметром x ("seed"): каждый параметр порождает соответствующую (но одну и ту же) последовательность псевдослучайных чисел.

math.sin (x)

Возвращает синус x (аргумент – в радианах).

math.sinh (x)

Возвращает шинус (гиперболический синус) x.

math.sqrt (x)

Возвращает квадратный корень x. (Вы также можете использовать выражение x^0.5 для вычисления этого значения.)

math.tan (x)

Возвращает тангенс угла x (аргумент – в радианах)

math.tanh (x)

Возвращает гиперболический тангенс x.

Ввода-вывод

Библиотека ввода-вывода предоставляет два различных возможных «стиля» для работы с файлами. Первый использует неявные дескрипторы файлов; т.е., существуют операции по установке «умолчательного» файла ввода или вывода, и все операции ввода-вывода работают с этими файлами. Второй «стиль» использует явные дескрипторы файлов.

При использовании неявных файловых дескрипторов, все операции предоставляются таблицей io. При использовании явных дескрипторов, операция io.open возвращает дескриптор, а после этого все операции являются методами данного дескриптора.

Таблица io также предоставляет три предопределенных (predefined) файловых дескриптора со стандартными (в C) значениями: io.stdin, io.stdout, и io.stderr.

Если не указано иное, все функции ввода-вывода возвращают nil при ошибочном завершении (а также сообщение об ошибке как второй результат и системно-независимый код ошибке в третьем результате) и какое-либо значение, отличное от nil при успешном завершении.

io.close ([file])

Эквивалентна file:close(). Без параметра file, закрывает стандартный поток вывода.

io.flush ()

Эквивалентна file:flush для стандартного потока вывода./p>

io.input ([file])

При вызове с указанием имени файла, открывает данный файл (в текстовом режиме), и направляет его поток на стандартынй поток ввода. При вызове с хендлером файла, делает хендлер файла стандартным хендлером ввода (перенаправляет поток, соответствующий хендлеру файла на стандартный поток ввода). При вызове функции без параметров, возвращает текущий файл ввода по умолчанию.

В случае лшибок, данная функция возбуждает ошибку вместо того, чтобы возвратить код ошибки.

io.lines ([filename])

Открывает файл с данным именем в режиме чтения и возвращает функцию-итератор (iterator function) которая при каждом последующем вызове возвращает новую строчку из файла. Т.о., конструкция

```
for line in io.lines(filename) do body end
```

обработает все строки файла. При обнаружении функцией-итератором конца файла, она фозвращает nil (для окончания цикла) и автоматически закрывает файл.

Вызов io.lines() (без имени файла) эквивалентен io.input():lines(); т.о., он обрабатывает строки стандартного файла ввода. В этом случае, файл по окончанию итераций не закрывается автоматически.

io.open (filename [, mode])

Эта функция открывает файл в режиме, указанном в строке mode. Возвращает хендлер файла, или, в случае ошибок, nil и сообщение об ошибке.

Стрка mode может содержать следующие значения:

"r":	режим чтения (используется по умолчанию);
"w":	режим записи;
"a":	режим дозаписи в конец файла;
"r":	режим изменения, все ранее хранившиеся данные сохраняются;
"w":	режим изменения, все ранее хранившиеся данные сохраняются;
"a":	режим изменения с дозаписью в конец, все ранее хранившиеся данные защищены, запись разрешена только в конец файла.

Строка mode может также содержать 'b' в конце; этот символ нужен для некоторых систем для открытия файла в двоичном режиме. Эта строка полностью повторяет синтаксис функции C fopen.

io.output ([file])

Аналогична io.input,но работает с стандартным файлом вывода.

io.popen (prog [, mode])

Запускает программу prog в отдельном процессе и возвращает хендлер файла, который вы можете использовать для чтения данных из этой программы (если mode равен "r", значение по умолчанию) или для записи данных в эту программу ??? (если mode равен "w").

Эта функция системно зависима и доступна не на всех платформах.

io.read (···)

Аналогична io.input():read.

io.tmpfile ()

Возвращаер хендлер для временного файла. Этот файл открывается в режиме изменения и автоматически удаляется при завершении программы.

io.type (obj)

Проверяет, является ли obj валидным хендлером файла. Возвращает строку "file" если obj – открытый хендлер файла, "closed file" если obj закрытый хендлер файла, или nil если obj не является хендлером файла.

io.write (···)

Эквивалентна io.output():write.

file:close ()

Закрывает file. Обратите внимание, что файлы автоматически закрываются когда их хендлеры уничтожаются сборщиком мусора, но это может случиться в любой момент и не предсказуемо.

file:flush ()

Сохраняет все данные, записанные в файл file.

file:lines ()

Возвращает функцию-итератор, которая при каждом вызове возвращает новую строку из файла. Т.о. код

```
for line in file:lines() do body end
```

обработает все строки файла. (В отличие от io.lines , эта функция не закрывает файл по окончании цикла (т.е. достижении конца файла).)

file:read (···)

Читает данные из файла file, в соответствии с заданными форматами, которые определяют, что читать. Для каждого формата, функция возвращает строку (или число) с прочитанными символами, или nil если не может прочитать данные в указанном формате. При вызове без указания формата, использует стандартный формат, соответствующий чтению всей следующей строки (см. ниже).

Возможные форматы:

"*n":	читает число; это – единственный формат, возвращающий число вместо строки.
"*a":	читает весь файл, начиная с текущей позиции. Если позиция совпадает с концом файла, возвращает пустую строку.
"*l":	читает следующую строку (пропуская конец строки), возвращает nil в конце файла. Это – формат по умолчанию.
число:	читает строку, но не более заданного количества символов, возвращает nil по достижении конца файла. Если число равно нулю, функция ничего не читает, возвращая постую строку, или nil по достижении конца файла.

file:seek ([whence] [, offset])

Получает и ыыставляет текущую позицию в файле, отсчитываемую от начала файла, в позицию, заданную параметром offset плюс значение (исходная позиция), заданное строкой whence, следующим образом:

"set": исходная позиция равна 0 (начало файла);

"cur": исходная позиция – текущая;

"end": исходная позиция – конец файла.

В случае успешного выполнения, функция seek возвращает выставленную позицию в файле, отсчитываемую от начала файла. Если функция завершается неудачно, она возвращает nil и строку – описание ошибки.

Значение по умолчанию для параметра whence равно "cur", а offset – 0. Т.о. вызов file:seek() возвращает текущую позицию в файле, не изменяя ее; вызов file:seek("set") перемещает указатель текущей позиции в начало файла (и возвращает 0); а вызов file:seek("end") перемещает указатель текущей позиции в конец файла, и возвращает его длину.

file:setvbuf (mode [, size])

Изменяет режим файла записи (output file) на буферизованный режим. Существует 3 возможных режима:

"no": отключить буферизацию; результат записи в файл немедленно «сбрасывается» на диск.

"full": полная буферизация; операции записи на диск выполняются только при переполнении буфера (или при вызове функции очистки буфера (см. io.flush)).

"line": построчная буферизация; запись буферизуется до достижения конца строки или происходит ввод из особенных (специальных) файлов (таких как, например, терминал).

В двух последних случаях, size указывает размер буфера в байтах. По умолчанию используется «необходимый и достаточный размер».

file:write (···)

Записывает значение каждого из аргументов в файл file. Аргументами могут быть строки или числа. Для записи других значений, используйте функции tostring или string.format перед вызовом функции write.

Функции операционной системы

Эта библиотека включена в таблицу os.

os.clock ()

Возвращает примерное количество времени в секундах, которое программа выполнялась на CPU.

os.date ([format [, time]])

Возвращает строку или таблицу, содержащую дату и время, отформатированные в соответствии с заданным параметром format..

Если аргумент time передается функции, то должно быть отформатировано «время» (см. функцию os.time). В противном случае, параметр date используется для форматирования текущего времени.

Если параметр format начинается с '!', то время форматируется в соответствии с универсальным глобальным временем (по Гринвичу). После этого опционального символа, если format равен "*t", то date возвращает таблицу со следующими полями: year (год, четыре цифры), month (месяц, 1 – 12), day (день, 1 – 31), hour (час, 0 – 23), min (минуты, 0 – 59), sec (секунды, 0 – 61), wday (день недели, воскресенью соответствует 1), yday (день года), и isdst (флаг дневного времени суток, тип boolean).

Если format не равен "*t", то функция date возвращает дату в виде строки, отформатированной в соответствии с правилами функции C strftime.

При вызове без аргументов, date возвращает данные «в обычном формате», который зависит от системы и текущих настроек операционной системы (т.е., os.date() эквивалентна os.date("%c")).

os.difftime (t2, t1)

Возвращает число секунд, прошедшее от времени t1 до времени t2. В POSIX, Windows, и некоторых других системах, это значение равно в точности t2-t1.

os.execute ([command])

Эта функция эквивалентна функции C system. Она передает параметр command на выполнение шеллу операционной системы. Она возвращает системно-зависимый статус. Если параметр command не передается, то возвращает не ноль, если шелл доступен и ноль в противном случае.

os.exit ([code])

Вызывает функцию C exit, с опциональным параметром code, для останова выполнения программы-хозяина. Значение по умолчанию для параметра code – код успешного завершения. Примечание: в ISO C описаны коды 0, EXIT_SUCCESS (обычно равен 0, но не стандартизирован, может обозначать «успешное завершение, отличное от состояния 0»), EXIT_FAILURE (обычно какое-либо ненулевое значение, но не стандартизирован).

os.getenv (varname)

Возвращает значение переменной окружения varname, или nil если переменная не определена.

os.remove (filename)

Удалаяет файл или директорию с заданным именем. Директории должны быть пусты. Если функция не может провести удаления, она возвращает nil, плюс строку, содержащую описание ошибки.

os.rename (oldname, newname)

Переименовывает файл или директорию oldname в newname. Если функция не модет провести переименования, она возвращает nil, плюс строку, содержащую описание ошибки.

os.setlocale (locale [, category])

Изменяет текущие региональные настройки (locale) программы. Параметр locale является строкой, содержащей региональные настройки (locale); category – опциональный строчный параметр, содержащий категорию, для которой производится изменение: "all", "collate", "ctype", "monetary", "numeric", или "time"; значением по умолчанию является категория "all". Функция возвращает значение новых настроек (locale), или nil, если вызов не может быть обработан.

Если locale – пустая строка, региональные настройки изменяются на региональные настройки, определяемые реализацией (implementation-defined native locale). Если locale – строка "C", текущие реиональные настройки изменяются на стандартные C-настройки.

При вызове с nil вместо первого аргумента, эта функция возвращает текущие региональные настройки для заданной категории.

os.time ([table])

Возвращает текущее время при вызове без аргументов, или время и дату, указанные в передаваемой таблице. Эта таблица должна иметь поля year, month, и day, и может иметь поля hour, min, sec, и isdst (описание этих полей см. в описании функции os.date).

Возвращаемое значание – это число, значение которого зависит от системы. В POSIX, Windows, и некоторых других системах, это число соответствует количеству секунд, отсчитываемому от некоторого заданного момента времени ("эпоха"). В других системах, значение не специфицировано, и число, возвращаемое функцией time, может быть использовано только как аргумент функций date и difftime.

os.tmpname ()

Возвращает строку с именем файла, который может быть использован в качестве временного файла. Файл должен быть явно открыт до использования и явно удален, если больше не будет нужен.

Перечень использованных рессурсов

[01] **Справочное руководство по языку Lua 5.1.**
http://www.lua.ru/doc/

[02] **Lua 5.3 Reference Manual.**
Ierusalimschy Roberto, Henrique de Figueiredo Luiz, Celes Waldemar.
http://www.lua.org/manual/5.3/

[03] **Logic Machine Lua reference manual.**
http://openrb.com/docs/lua.htm

[04] **Programming in Lua.**
Ierusalimschy Roberto.
ISBN 859037985X
January 2013
http://www.lua.org/pil/

▲ … ◄