Краткая справка по языку Lua

Оглавление

Введение

В справке кратко и, при этом, практически *полно* описана архитектура языка **Lua**, его синтаксис и семантика, а также интеграция с **C/C++**. В ней же приведены коды модификации исходников **Lua** для реализации особого режима запуска и выполнения *отдельных потоков* на стеках *сопрограмм* **Lua** (обеспечивающего параллелизм исполнения **C**-функций, вызываемых в **Lua**). Но следует учитывать, что первоисточником описания языка является его официальная документация. Детали конкретных версий **Lua** рекомендуется уточнять, используя документацию по конкретной версии. На сайте https://antirek.github.io/luabook/ выложено подробное описание **Lua** 5.3 (на русском языке).

В тексте справки использованы некоторые примеры из книги «Программирование на языке Lua» Р. Иерузалимски и фрагмент описания **C API** из статьи Борескова А.В «Язык Lua и использование скриптов на нем в программах на C++».

Краткая характеристика Lua

Lua - интерпретатор *динамический*: переменным можно присваивать данные любого типа ***Lua***. Данные языка имеют свойство *Тип*. Контроль типов данных выполняется только в *момент* исполнения скрипта **Lua**. По сути, **Lua** двухуровневый язык программирования: Lua/C. Эта двухуровневость была заложена в проект языка и реализована *изначально* (Lua тесно интегрирован с **C/C++**). С учетом этого, функциональность, разрабатываемых на **Lua** программ и их эффективность, фактически *определяется* возможностями **C/C++**.

Исходный код скрипта на **Lua** транслируется в *байт-код* либо предварительно, либо непосредственно в начале выполнения скрипта. *Байт-код* выполняется виртуальной машиной **Lua**. Управление памятью автоматическое, с использованием потокоопасного мусорщика (из-за этого, в стандартном **Lua** параллелизм выполнения его функций не допускается).

Функционально все (*исключая* параллелизм выполнения его функций) можно написать на «чистом» **Lua**, но при необходимости, из **Lua** можно легко *перейти* в **C/C++** (используя все его возможности) и вернуться обратно.

Lua очень *компактный* как по коду его реализации, так и по его описанию, достаточного для его *начального* использования. Особенно это заметно на фоне многочисленных монструозных средств разработки программ. Пожалуй, отношение <функциональности Lua> к <его компактности> одно из самых высоких среди известных языков программирования.

Для хорошего понимания **Lua** необходимо четкое представление об устройстве его ассоциативных *таблиц* (тип **table**), элементами которых являются *неупорядоченные* пары: <Ключ таблицы (данные любого типа Lua кроме **nil**)> <Поле таблицы (данные любого типа Lua кроме **nil,** при присвоении которого полю, соответствующая запись таблицы удаляется>. Доступ к полям таблицы выполняется по ее ключам. *Таблица* используется, в том числе, и для хранения всех *переменных окружения **Lua*** (переменных, без спецификации локализации: ***local***). Имена этих переменных всего лишь ключи, а данные переменных - поля этой таблицы окружения (видимой, при отсутствии экранирования *локальными* переменными, описанными далее, из любого места скрипта).

В **Lua** используются переменные двух видов: *локальные переменные* и *переменные окружения* (общие для каждой единицы компиляции **Lua** - *куска*). *Локальные переменные со* спецификацией **local** имеют блочную лексическую область видимости (в тексте кода видится только то, что создано/объявлено ранее во внешних или текущем блоке, а повторно объявленное всегда экранирует предыдущее с учетом блочной структуры скрипта, а также переменные окружения). Видимость *переменных окружения* (в таблице окружения) из некоторого места *куска* определяется выполнением обращения к переменным/ключам таблицы окружения (доступной в любом месте *куска*) неэкранированных из этого места локальными переменными.

Окружение любой единицы трансляции ***Lua*** (*куска*), а это и *основной* скрипт, определяется (начиная с версии 5.2) служебной переменной *куска* ***_ENV*** (по *умолчанию* ссылающейся на глобальную служебную таблицу **_G**), которой можно присвоить в нем (*куске*) любую таблицу **Lua**. После такого присвоения, окружением *куска* станет присвоенная таблица.

Рекомендуется там, где это возможно, использовать локальные переменные, объявляемые со спецификацией: ***local*** <Список локальных переменных>. Эти переменные хранятся в *стеках* блоков ***Lua*** и обращение к ним эффективнее, чем к *переменным окружения*. При выходе из любого блока, его локальные переменные перестают существовать.

Типы данных полей таблиц ***Lua*** это:

- значения: **nil, boolean , number, string**;
- ссылочные данные: **function, thread, table, userdata**.

При присвоении значения копируются, а на ссылочные данные создаются ссылки. На одно и то же ссылочное данное может быть несколько одинаковых ссылок из разных переменных.

Функции (**function**) в ***Lua*** *анонимные* (данные *первого класса*, при присвоении и других базовых операций не отличающиеся от остальных ссылочных данных ***Lua***), не имеющие встроенных имен для обращения к ним. Существует конструктор создания функции с присвоением переменной ссылки на нее. Вариантом конструктора функции является строка с текстом скрипта, из которой может быть создана функция скрипта с использованием служебной функции **Lua**. Любую функцию **Lua** можно вызвать на исполнение с использованием тех переменных, в которых есть ссылка на нее. Как и остальные ссылочные данные, функции, если в процессе выполнения скрипта на них нет ни одной ссылки, *подпадают* под уборку мусорщиком ***Lua***. Функцией является сам скрипт **Lua**. Функция может быть создана/определена внутри любой функции в любом месте и может быть выдана как результат функции. Результатов у функции может быть больше одного. В функциях обеспечивается *рекурсия*, а также они представляют собой *замыкания* (в них допускается использование локальных переменных тех блоков, внутри которых создаются функции и которые (внешние локальные переменные) являются составной частью функций – ее *внешним локальным окружением*). При создании функции, в том числе и в качестве результата некоторой функции, она создается с состоянием своего *внешнего локального окружения,* имеющегося на момент ее создания.

Строки (**string**) в ***Lua*** битовые (в байтах строки могут быть любые комбинации 1 и 0), семантически являются значениями (внутренне неизменяемыми после их создания, копируемыми при присвоении) и у них есть свойство *длина строки*. Символ '\0' не является признаком конца строки. Более детально все типы ***Lua*** описаны в основном тексте справки.

Таблицы в ***Lua*** можно использовать как метатаблицы. Это таблицы **Lua**, которые:

1) подключаются служебным оператором подключения *метатаблицы* к таблице (в том числе можно и к любой метатаблице или к самой себе);
2) при определенных операциях над таблицами (у которых есть метатаблицы) или ситуациях, возникающих при их использовании, **Lua** обрабатывает служебные ключи (с преопределенными именами) метатаблиц специальным образом (например, запускаются функции, ссылки на которые присвоены полям таких ключей и т.д.).

В **Lua** можно использовать сопрограммы: тип **thread.** Они запускаются в отдельных стеках сопрограмм (**thread**) и в *стандартной* конфигурации **Lua** реализуют *псевдопотоки* (нити, обслуживаемые *одним* потоком). В сопрограммах обеспечивается режим вызова функции с возможностью ее продолжения после программного прерывания, вызываемого с помощью служебной функции ***yield***.

--

Отдельные стеки сопрограмм *реентерабельны* и могли бы выполняться в различных потоках, но так как сборка мусора *потокоопасная*, то в нескольких потоках **Lua** можно использовать только как *разделяемый* ресурс под синхронизацией (в *специальной* конфигурации-сборки **Lua**). Это относится и к его *функциям **C-API**,* являющиеся частью ***Lua***. При компиляции интерпретатора **Lua** (после внесения сравнительно простой, но обязательной модификации его кода в части реализации синхронизации в нем) есть возможность указания режима использования стеков сопрограмм (**thread),** под *синхронизацией*, в *отдельных* потоках. При этом, **C**-функции, запускаемые в различных стеках, могут выполняться (в разных потоках) *параллельно*, но код **Lua** может выполняться в потоках только в разделяемом режиме (под синхронизацией).

В переводе на другие известные языки **ООП**: таблицы **Lua** это динамический список свойств и ее методов (заданных в том числе и в их метатаблицах), связанных с этими свойствами.

Профессионал, наверное, сможет изучить этот язык, почти в полном объеме, дня за два-три. Для начинающих программировать это язык в своей основе (без использования средств **ООП**) тоже, наверное, один из самых простых.

Если делать в программах на ***Lua*** проверку типов параметров (а может быть и областей значений) более-менее содержательных функций, то можно обеспечить и надежность разрабатываемых на **Lua** программ.

Код реализации **Lua** всего: *~400kb.*

Доступный исходный код **Lua** может служить одним из примеров того, как можно/нужно программировать на языке **C**.

Несколько общих замечаний относительно **Lua**:

1) *Нотация* записи программ *не последнее дело* и авторы языка *зря* не использовали *проверенную* временем нотацию **C/C++**, как в части записи кода, так и в написании комментариев. Конструктор таблиц можно было при этом представить, например, в виде: <….>. Как представляется *издалека*, сишная нотация **Lua**, обеспечила бы ему гораздо *большую* популярность среди программистов, чем это есть сейчас.

   Показательным подтверждением высказанного выше, является язык **python.** В нем была выбрана нотация записи программ, которая «*зашла*» многим программистам (психология восприятия текстов человеком) и этот язык, использующий мно-

гие архитектурные решения **Lua** (как и Java-sript, появившийся на 2 года позже **Lua**), стал популярным.

2) *Множественное* присвоение в **Lua**, а также, возможные, *множественные* значения результата его функций обеспечивают дополнительную гибкость (вариантность) написания скриптов **Lua**, однако, являются существенными *потенциальными* источниками ошибок в них, так как семантика реализация этой множественности слабо отражается в синтаксисе ее записи и это надо контролировать на семантическом уровне (например, если в параметрах *функции1* есть вызов *функции2*, то в качестве параметров могут вставиться несколько значений *функции2*, но сколько их будет, из имени *функции2* формально не следует).
3) Влючительно до версии **5.4.1**, в кодах стандартных библиотек **Lua** *не был учтен* существующий в нем режим конфигурирования (представленный в исходниках **Lua**) использования стеков сопрограмм (**thread)** в *отдельных* потоках. Функции стандартных библиотек *определены* в **Lua** как *сишные* (потокобезопасные, допускающие параллельное выполнение), но в их коде нередко используется внутренняя *среда* интерпретатора **Lua** (не являющаяся потокобезопасной из-за однопоточного управления автоматической памятью)**,** и *нет гарантии* на уровне *архитектуры* реализации языка, того, что стандартные функции **Lua** *потокобезопасны* для упомянутого режима.
4) Синтаксис **Lua** *простой*: основная его сложность перенесена в функциональность его таблиц, реализованных на **C**. Поэтому трансляция текста кода **Lua** в его исполняемый *байт-код*, реализована *эффективно*, и это надо учитывать и использовать при написании скриптов на **Lua** (понимая возможность динамического создания текстов-скриптов внутри выполняемого скрипта).
5) Выбор в **Lua** фактически единственного, но универсального типа **table**, обеспечивающего его полную функциональность, позволяет кратко записывать семантику выполнения скриптов. Кроме того, короткий технологический цикл, от написания кода, до его запуска, обеспечивает оперативное получение конечного результата. Интеграция **Lua** с **C/C++** обеспечивает его расширяемость, как в части использования существующих библиотек, так и в части эффективности выполнения его скриптов. При этом надо учитывать то, что *динамизм* **Lua** *потенциально* отрицательно влияет на надежность кодов, написанных *непосредственно* на нем.

Бэкусо-Науровская Форма описания языков программирования (БНФ).

Для *обеспечения краткости* описания синтаксиса языка **Lua** (*допустимых* в нем текстов) в данной справке часто используется **БНФ**.

Этот метаязык (язык описания *языков*) в своей простой основе обеспечивает краткое и достаточно точное описание синтаксиса почти любых языков программирования.

*Основные понятия **БНФ**, используемые при описании **Lua***

Символы: **<**, **>**, **[**, **]**, **{**, **}**, **|**, **<->**, **~<->**, **::=**, **:=** , выделенные жирным шрифтом, в описании синтаксиса Lua используются как металингвистические (служащие для описания языка). В тех случаях, когда эти символы являются элементами языка программирования, они представлены обычным шрифтом.

Определяемые понятия языка заключаются в скобки **<>**.

Метаформула **<**Определение 1**> ::= <**Определение 2**>** означает, что **<**Определение 1**>** *определяется* (раскрывается/заменяется) **<**Определением 2**>**.

Пусть **A** некоторое понятие (определение).

Как обычно в расширенной **БНФ**, **{A}** означает 1 или более **A**, **[A]** означает опциональное **A** (**A** может отсутствовать), а выражение **A | B** это либо **A**, либо **B**.

Символ **<->** означает эквивалентность (идентичность) того что им разделяетя.
Символ **~<->** означает неэквивалентность (неидентичность) того что им разделяется.
Символ **::=** означает, что правая часть раскрывает (определяет) левое понятие.
Символ **:=** означает, что правая часть показыает, что является конкретным (детальным) значением левой части и далее не раскрывается.
Терминальные (первичные, не определяемые далее) слова языка программирования в описании языка упоминаются *без угловых скобок*.

Основные понятия Lua

Ключевые слова Lua (его словарь) и комментарии

Ключевые слова Lua (в новых версиях могут возникать изменения в этом списке):

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

Lua является языком, чувствительным к регистру символов, тоесть один и тот же символ в нижнем и верхнем регистре это разные символы. Использование точки с запятой в конце операторов Lua не является обязательным.

Типы данных Lua (определяют их представление и допустимые операции с ними)

*В **Lua** 8 типов данных:*

Типы значения (хранятся в переменных-полях таблиц):

1) **nil** (*неопределенный/несуществующий*) -- одно значение: nil
2) **boolean** (*логический*) -- два значения: *true, false*
3) **number** (*числовой*) -- Примеры задания: 0.005 (значение 0,005), 1000
4) **string** (*строковый*) -- Примеры задания: 'Это строка', "И это строка", [[\0 А также это строка]]

Строки внутри неизменяемы и битные, при их присвоении переменным всегда создаются их копии. Значение символа строки '\0' не является признаком конца строки (как это определено в языке C). При этом внутри строки можно задавать специальные (*escape*) символы, аналогично языку C - '\b', '\n', '\r', '\t', '\\', '\"', '\'', '\0' и '\ddd' (в последнем случае в через *ddd* обозначен десятичный код символа от нуля до 255). Также для задания строк можно использовать так называемые длинные скобки (*long brackets*). Открывающая длинная скобка уровня *n* состоит из обычной открывающей квадратной скобки ('['), *n* знаков равенства ('=') и еще одной открывающей квадратной скобки ('['). Так открывающая скобка нулевого уровня это просто *[[*, открывающая скобка первого уровня это *[=[* и т.д. Полностью аналогично вводится закрывающая скобка уровня *n*. Например, закрывающая длинная скобка уровня 4 записывается как]====].

Ссылочные типы (в переменных находятся ссылки на такие данные):

5) **function** (*функция*) - объекты функции, запускаемые на исполнение

6) **userdata** (пользовательские данные) – объекты, полностью реализуемые разработчиком и используемые в скрипте **Lua.** Значения этого типа не могут быть непосредственно созданы в программах на ***Lua***, а должны создаваться только через **C API** (т.е. в вызывающей скрипт программе).

7) **thread** (*поток-нить*) -- сопрограммы (особый вид функций, описанных подробнее далее)

---

8) **table** (*таблица*)

*Таблица **Lua*** это *неупорядоченный* список элементов вида (две состовляющие):

<Ключ (данное *любого* типа, кроме **nil**, уникальное в пределах таблицы)**>** **<**Поле (значение) таблицы (данное **Lua** *любого* типа, кроме **nil**)**>**

Пример задания (создания) таблицы (здесь символы конструирования таблиц { } и символы индексации [ ] *не металингвистические*, а являются элементами синтаксиса **Lua**):

{ [7] =4, [6] = 7, ['вася'] = "916-335 78 67", ['Таблица'] = { [true] = 4, [5] = false} } -- в этой таблице полем элемента с ключем 'Таблица' является таблица { [true] = 4, [5] = false}

Доступ к *любому* полю таблицы: **<** Ссылка на таблицу**> [<**Ключ**>]**. Если в таблице нет **<**Ключа**>**, то результат доступа равен **nil**.

Операция индексация **[<**Значение индекса**>]** допустима *только* для таблиц (иначе будет ошибка).

Если в присвоении **<**Таблица**> [<**Ключ**>]** = **<**Данное **Lua>** задан существующий **<**Ключ**>**, то поле существующего элемента таблицы (с **<**Ключом**>**) заменяется на **<**Данное **Lua>**. Если же такого элемента нет, то *создается* новый элемент таблицы с **<**Ключом**>**. **nil** в качестве значения **<**Ключа**>** не допускается.

Для удаления элемента таблицы с существующим **<**Ключом**>** следует выполнить присвоение **<**Таблица**> [<**Ключ**>]** = **nil**.

--

Существует дополнительный синтаксис доступа к полям таблиц:

<Таблица (переменная, ссылающаяся на таблицу)**>.<** *Переменная-ключ* **>**, где **<** *Переменная-ключ***>** должен быть синтаксически переменной (смотри п.4).

Примеры индексации таблицы **T**.

1) **T**.yes = 1 **<->** **T**['yes'] = 1 --- строка **<->** - метасимвол эквивалентности (полного совпадения)
2) **T**.2 = 1 -- ошибка так как 2 не может быть именем переменной
3) **T**.a1 = 1 **<->** T['a1'] = 1
4) **T**.a1 **~<->** **T**[a1] --- так как во втором случае a1 это переменная (например a1 **:=** 1), а не строка
5) **T**.a1.kl._r **<->** **T**['a1'] ['kl'] ['_r'] -- многократная индексация таблицы **T** (может быть ошибка, если на каком то этапе индексируется не таблица)

Использование приведенного выше синтаксиса доступа к полям таблиц ограничено тем, что **<** *Переменная-ключ***>** должен быть *синтаксически* переменной **Lua**, но в тоже время существуют ситуации в программировании, когда такой синтаксис является более удобным и наглядным чем универсальная индексация с использованием квадратных скобок.

К любой таблице **T** можно *подключить* (для управления поведением в операциях с ней) таблицу **MT** (одну) с помощью служебной функции **setmetatable** следующим образом: **setmetatable**(**T**, **MT**). После такого подключения **MT** становится метатаблицей для таблицы **T**. Допустимой являктся и следующая операция **setmetatable**(**T**, **T**).

Для отключения метатаблицы таблицы **T** надо выполнить **setmetatable**(**T**, **nil**).

Метатаблицы позволяют изменить поведение таблицы **T** при выполнении операций с нею. Например, при помощи метатаблиц мы можем определить то, как **Lua** вычисляет выражение **a + b**, где **a** и **b** — таблицы. Когда **Lua** пытается сложить две таблицы, он проверяет, есть ли хотя бы в одной из них метатаблица и содержит ли эта метатаблица поле со служебным ключем **__add**. Если Lua находит это поле, он вызывает (как функцию) соответствующее значение поля - так называемый метаметод, который должен быть функцией для вычисления суммы таблиц.

Существует список *служебных* ключей, используемых в метатаблицах, а также описание их использования при выполнении операций с таблицами.

--

Для отключения срабатывания метаметодов, при присвоении в таблице полю значения, используется стандартная функция:

rawset (**<**Таблица**>**, **<**Поле таблицы**>**, **<**Значение поля**>**).

Для отключения срабатывания метаметодов, при чтении в таблице поля, используется стандартная функция:

rawget (**<**Таблица**>**, **<**Поле таблицы**>**).
Операция определения длины таблицы: #**<**Таблица**>** (*только* для таблиц-массивов с ключами-индексами последовательностями: от 1 до n).

Более детально работа с таблицами/метатаблицами будет рассмотрена далее в разделе «Таблицы, метатаблицы, слабые таблицы и финализаторы».
--
Функция ***type***(**<**Данное Lua **>**) возвращает строку-тип (nil", "number", "string", "boolean", "table", "function", "thread" и "userdata").

! Таблицы, функции, нити и *пользовательские данные* являются объектами. Это значит, что переменная содержит не непосредственное значение, а ссылку на соответствующий объект. Остальные данные (в том числе и строки) являются непосредственными *значениями*.

Переменные Lua, блоки, области видимости, выражения, операции, скрипты, автоматическое управление памятью

Переменные **Lua** создаются и используются для доступа к его данным. **Lua** язык с динамической типизацией, поэтому в любой его переменной можно хранить данные любого типа **Lua**.
<Переменная**>** это символьная строка (в тексте скрипта без выделения кавычками) из одного или нескольких символов (в начале может бвть (но не рекомендуется) несколько _ символов а далее только латиница, цыфры и символ_).
Переменные могут быть двух видов:
1) Переменные *окружения* (без спецификации *local*) Примеры задания: TYU = 1 , r = 'туци'.
Переменные окружения создаются, при первом присвоении переменной данного Lua, отличного от **nil**, в **<**Таблице контекста выполнения скрипта**>** имеющей тип **table**. Ссылка на эту таблицу хранится (для **Lua** версий > 5.1) для каждой единицы трансляции **Lua** (*куска*) в служебной переменной *куска* с именем **_ENV** (первоначально это глобальная таблица с именем **_G,** хранящая, в том числе стандартные модули **Lua**).
2) *Локальные* блока объявляются (специфицируются) ключевым словом **local**. Примеры задания: **local** uu = 4, L (можно задать список, но следует учитывать, что в **Lua** присвоение *множественное*).
Такие переменные создаются в стеке-блоке выполняемого скрипта, при первом их объявлении.

Блок **Lua** - это тело *управляющей* структуры, тело *функции* или *кусок* кода (файл или строка, где переменная была объявлена). Кроме того группа операторов **Lua** может быть *объединена* в блок (составной оператор) при помощи конструкции **do <**Текст кода Lua**> end**.

Блоки не *пересекаются* друг с другом, но могут быть *вложенны* друг в друга.
Области *видимости* переменной - это все те фрагменты кода ("места" программы), из которых к данной переменной можно обратиться.
1. Области видимости локальных переменных (создаваемых динамически в стеках Lua) определяются *текстуально*, областями, ассоциированными с *блоками* **Lua**, с учетом вложенности и перекрытия/экранирования идентичными именами локальных переменных, внешних областей (в том числе и переменных *окружения*). Обращение к локальным переменным более *быстрое*, чем к глобальным. Спецификатор **local** без присвоений создает локальные переменные со значением **nil**. Повторное (с ранее объявленным именем) объявление локальной переменной в том же блоке создает *новый* экземпляр переменной в блоке, но последующее обращение к переменной (после повторного создания) будет к вновь созданному экземпляру. Причем, локальные переменные внутренних (вложенных) блоков *не доступны* из внешних блоков.

В процессе исполнения скрипта, выход из любого *блока*, *удаляет* все его локальные переменные, создаваемые всякий раз при входе в него. Есть ограничение на количество локальных переменных *блока* (*дефолтное* 256).

2. Область видимости переменных *окружения* определяется динамически (в процессе создания таких переменных присвоением) элементами **<**Таблицы контекста выполнения скрипта**>**.

<Таблица контекста выполнения скрипта**>** доступна из *любого* его блока. Но переменные в этой таблице возникают только после их создания-присвоения значений, *отличных* от **nil**. Если в скрипте идет обращение к несуществующей в нем переменной, то выдается значение **nil**. Видимость *переменных окружения* в **<**Таблица контекста выполнения скрипта**>** из некоторого места скрипта определяется выполнением обращения к переременным/ключам этой таблицы, не экранированных из этого места локальными переменными.

--

В любом месте кода **Lua** можно создать (динамически) пространство имен в виде таблицы: [**local]** **<**Название пространства имен**>** = { }

Использование пространства имен в его области видимости: **<**Название пространства имен**>.<**Имя переменной Lua**>**.

--

Под все данные, создаваемые непосредственно в **Lua**, память выделяется *автоматически*. Память, занимаемая неиспользуемыми (недостижимыми из скрипта) данными **Lua**, освобождается также *автоматически* (при сборке мусора). Но следует учитывать, что при использовании **C**-функций в **Lua**, в некоторых случаях (описанных в спецификациях таких функций) могут требоваться финализирующие действия по освобождению ресурсов, использованных в них.

Хотя сборка мусора автоматическая, в **Lua** существует возможность влиять на то, как это происходит, используя служебную функцию сборщика мусора: **collectgarbage**. Существует и аналог сборщика мусора в **C API**.

Сборщик мусора (GC)

Lua реализует пошаговый отмечающий-и-очищающий сборщик. При сборке мусора, выполнение кода Lua-скрипта останавливается.

Параметры сборщика мусора будут описаны далее, но для управления циклами очистки мусора, в основном, используются два параметра: *пауза сборщика мусора* (*"setpause"*) и *множитель шагов сборщика мусора* (*"setstepmul"*). Оба параметра используют процентные пункты как единицы (т.е., значение 100 означает внутреннее значение 1).

Пауза сборщика мусора контролирует, как долго сборщик ждет перед началом нового цикла (полной уборки мусора). Чем больше значение, тем менее агрессивен сборщик. Значения меньше 100 означают, что сборщик не останавливается перед началом нового цикла. Значение 200 (по умолчанию) означает, что сборщик, перед тем как начать новый цикл, ждет повышения использования общей памяти в два раза.

Множитель шагов сборщика мусора контролирует относительную скорость сборщика (частоту инкрементной сборки мусора) по отношению к скорости выделения памяти. Большие значения делают сборщик более агрессивным. Не рекомендуется использовать значения меньше 100, т.к. они сделают сборщик настолько медленным, что он никогда не завершит цикл. По умолчанию, используется значение 200, которое означает, что сборщик выполняется в два раза быстрее скорости выделения памяти.

Служебная функция управления сборкой мусора в **Lua**:

collectgarbage(what [, data])

Аргумент **what** (перечислимое значение в **API C**, строка в **Lua**) определяет, что требуется

делать. Возможные варианты:
LUA_CGSTOP ("*stop*"): останавливает сборщик мусора до другого вызова **collectgarbage** (или до lua_gc) с опцией "*restart*".
LUA_GSRESTART ("*restart*"): перезапускает сборщик мусора.
LUA_GCCOLLECT ("*collect*"): осуществляет полный цикл сборки мусора, при этом все недоступные объекты собираются и финализируются. Это значение по умолчанию для **collectgarbage**.
LUA_GCSTEP ("*step*"): выполняет некоторую работу по сборке мусора. Объем этой работы эквивалентен тому, что сборщик мусора сделает после выделения *data* байт.
LUA_GCCOUNT ("*count*"): возвращает количество килобайт памяти, используемой Lua в данный момент. Это количество включает в себя «мертвые», но еще не собранные объекты.
LUA_GCCOUNTB (*без аргументов*): возвращает дробную часть числа килобайт памяти, используемой Lua в данный момент.
LUA_GCSETPAUSE ("*setpause*"): задает параметр ***pause*** сборщика мусора. Это значение задается параметром *data* в процентах: когда *data* равен 100, параметр устанавливается на единицу (100%).
LUA_GCSETSTEPMUL ("*setstepmul*"): задает параметр пошагового множителя (***stepmul***) для сборщика мусора. Эта значение также задано *data* в процентах.

Любой сборщик мусора расплачивается за экономию памяти процессорным временем. В одном крайнем случае сборщик может вообще не запуститься. Он совсем не будет расходовать процессорное время за счет огромного потребления памяти. В другом крайнем случае сборщик может производить полный цикл сборки мусора при каждом изменении графа доступности данных. Такая программа будет использовать самый минимум необходимой ей памяти ценой большого потребления процессорного времени. Хорошие сборщики мусора пытаются найти баланс между этими двумя крайностями.

Сборщик мусора **Lua** достаточно хорош для большинства приложений. Тем не менее, в некоторых случаях сборщик мусора стоит попробовать оптимизировать. Два параметра, ***pause*** и ***stepmul***, предоставляют некоторое управление поведением сборщика.Параметр ***pause*** управляет тем, как долго сборщик ждет между окончанием последней сборки мусора и началом новой. Значение *pause* равное нулю заставит **Lua** запустить новую сборку мусора сразу по завершении предыдущей. Параметр ***pause*** в 200% ожидает удвоения использования памяти перед тем, как начать сборку мусора. Вы можете понизить значение *pause*, если хотите пожертвовать процессорным временем ради меньшего потребления памяти. Обычно вы должны держать это значение между 0 и 200%. Параметр ***stepmul*** управляет тем, как много работы сборщик мусора выполняет для каждого килобайта выделенной памяти. Чем выше это значение, тем менее *инкрементальным* становится сборщик мусора.

Огромное значение вроде 100 000 000% заставит сборщик мусора вести себя как *неинкрементальный* сборщик. Значением по умолчанию является 200%. Значения, меньшие 100%, сделают сборщик настолько медленным, что он может так никогда и не завершить сборку мусора.

Другие опции **GC** дают вам контроль над тем, когда запускается сборщик мусора. Игры являются типичными клиентами для данного вида контроля. Например, если вы не хотите, чтобы сборка мусора выполнялась во время определенных периодов времени, вы можете остановить его при помощи **collectgarbage**("*stop*") и затем запустить снова при помощи **collectgarbage**("*restart*"). В системах, где возникают постоянные периоды ожидания, вы можете держать сборщик мусора остановленным и вызывать **collectgarbage**("*step*",**n**) лишь в эти периоды. Чтобы определить, как много работы нужно выполнить во время такого периода, либо экспериментально подберите подходящее значение для **n**, либо в цикле вызывайте **collectgarbage** с **n**, равным нулю, до самого истечения этого периода ожидания.

Операции Lua (выражения)

Выражения **Lua** определяют действия над его данными:
<Выражение (формула) Lua**> ::= <**Переменная**>**

<Выражение (формула) Lua**> ::= <**Унарная операция**>** <Выражение (формула) Lua**>** **|**
<Выражение (формула) Lua**> <**Бинарная операция**> <**Выражение (формула) Lua**>**

1) Математические операции
Сложение/Вычитание/Унарный минус: x + y; x - y; -x
Умножение/Деление: x * y; x / y
Деление по модулю: x % y
Инкремент/декремент: x++ ; ++x ; x-- ; --x ;
Возведение в степень: x ^ y
Конкатенация (сцеление) строк: a = b .. c (a := 'bc')
Получение размера таблицы -массива или строки: v = #a
---
2) Логические операции.
not (унарный) - отрицание: ***not a***. Результат: **<**истина**>** если a: **<**ложь**>** иначе **<**ложь**>** (если a: **<**истина**>**)
and - и: *a and b*. Результат: **<**истина**>** если a: **<**истина**>** и b: **<**истина**>** иначе **<**ложь**>**
or - или: *a or b*. Результат: **<**ложь**>** если a: **<**ложь**>** и b: **<**ложь**>** иначе **<**истина**>**

! Значения, участвующие в логических операциях, могут быть *любыми* допустимыми в Lua типами и *интерпретируются* как **<**истина**>** или **<**ложь**>**.
Только значение ***false*** и ***nil*** в таких операциях считаются **<**ложь**>**. Любые другие значения, в том числе **true** в логических операциях допустимы и считаются **<**истиной**>** (далее истинные значения).
Особенности результатов операций **and** и **or**:
1) Если результат выполнения **and <**истина**>**, то результатом будет значение *второго* операнда. Иначе результат: **false**
2) Если результат выполнения **or <**ложь**>**, то результатом будет второе *значение*. В противном случае, если первый операнд **<**истина**>**, то его *значение*, иначе, значение *второго* операнда.
!! Описанные особенности позволяют использовать в **Lua** условные выражения, результаты которых зависят от логических операций.
Например:
- эквивалент тернарного оператора в C++ (c = a ? b: d): c = a **and** b **or** d
- установка значения по умолчанию: x = b **or <**Умолчание**>** (если b равно **nil**, то x будет присвоено значение <Умолчания>)
3) Операции сравнения (результат сравнения true или false) ----
<
>
<=
>=
== --- тождество – *полная* идентичность (могут использоваться для любых типов данных Lua)
~= --- не тождество (могут использоваться для любых типов данных Lua)

!! Операции (приведены не все) имеют строгий приоритет (в соответствии с тем, как они перечисленны ниже):
^ - возведение в степень (самый высокий)
not - (унарный) (может использоваться для любых типов данных Lua) и **#** (получение длины строки и таблицы)
* / %
+ -
< > <= >= ~= ==
and (может использоваться для любых типов данных Lua)
or (самый низкий приоритет) (может использоваться для любых типов данных Lua)

- оператор выдачи длины (для *строк* и только для *таблиц-массивов* с индексами 1-n). Для таблиц, которые не являются массивами (в момент выполнения оператора #), он

выдает неправильное значение количеста элементов. Длину произвольной таблицы (не массива) можно определить, только *перебором* в цикле ее элементов.
Быстрая проверка *любой* таблицы **T** на отсутствие элементов: если *next*(**T**) равно **nil**, то **T** пустая.

Приоритеты для групп операций в ***Lua*** определены следующим образом (от низкого приоритета к высокому):

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unary)
```

---
5. Срипт **Lua** (программа) - это **<**Список операторов Lua (несколько операторов)**>**, разделенных пробелами (переходами на новую строку) или символом ;
В тексте скрипта можно использовать комментарии, которые не влияют на его выполнение , но позволяют комментировать программу, добавляя в нее произвольные тексты.

Комментарии бывают однострочные и многострочные.
В скрипте однострочный комментарий начинаются с двух черточек: **--** **<**Однострочный комментарий (произвольный текст до конца строки) **>**
Многострочный комментарий оформляется следующим образом: --[[**<** Мнгострочный комментарий (произвольный текст) **>**]]

Перед выполнением текст скрипта транслируется **Lua** в специальный формат, называемый его *байт-кодом* и этот код сразу запускается на исполнение.
При исполнении скрипта создаются и обрабатываются данные **Lua**.
Для хранения всех данных выполняемого скрипта (он является функцией) могут быть использованы следующие *виды памяти:*
1) **<***стек скрипта***>** хранит локальные переменные его блоков.
2) **<***таблица контекста выполнения скрипта***>** хранит его глобальные переменные в виде полей (созданных при начальном запуске скрипта, а также создаваемых в самом скрипте оператором присвоения переменным (без спецификации *local*) данных **Lua** отличных от **nil**.
3) **<***"куча" скрипта***>** хранит данные всех ссылочных типов, а также строки **Lua**.
Кроме того, из "кучи" запрашивается память для начального создания или увеличения **<**стека**>**, а также при создании и увеличении <таблицы контекста выполнения скрипта>.
Данные скрипта, на которые, при его выполнении, нет ни одной ссылки (из существующих в текущий момент переменных скрипта), как было отмечено ранее, автоматически убираются "мусорщиком" **Lua**, с возвратом их памяти операционной системе ПК. Поэтому разработчику скрипта **Lua** не надо заниматься в нем управлением его памятью.

Операторы Lua

Операторы **Lua** включают в себя присваивание, управляющие структуры и вызовы функций. **Lua** также поддерживает такие операторы, как множественное присваивание и объявления локальных переменных.

Присвоение

1) Обычное присвоение (одинарное): **<**Переменная **>** = **<**Данное Lua любого типа**>**
Например: u = {4, 7, 'вася', { true, 5} } -- присвоить таблицу---
2) Множественное присвоение: **<**Список переменных (через запятую)**>** = **<**Список данных (через запятую) **>**. Соответствие присвоения определяется порядком списков.

<Список переменных (через запятую)**>** может быть длиннее чем **<**Список данных (через запятую) **>** и тогда "лишние" переменные не присваиваются.

Например: local a, b, c = 1, 2 Результат a := 1, b := 2, c := nil (у существующих в скрипте переменных, которым ничего не присвоено есть начальное значение nil)

Управляющие конструкции Lua

Lua предоставляет небольшой, традиционный набор управляющих структур: **if** для условного выполнения, а **while**, **repeat** и **for** для итерации. Все управляющие структуры обладают явным завершающим элементом: **end** завершает if, **for** и **while**, а **until** завершает **repeat**.

Условное выражение управляющей структуры может быть любым значением. **Lua** считает все значения, отличные от **false** и **nil**, истинными (в частности, **Lua** считает истинными 0 и пустую строку).

!! Символы: **<**, **>**, **[**, **]**, **{**, **}**, **|** в *описании* синтаксиса управляющих конструкций Lua используются как металингвистические.

---

1) *Оператор группировки вычислений* (создает блок, может быть использована в любом месте, в котором можно использовать операторы Lua)

```
do
   -- блок --
   [ <Список операторов Lua> ]
end
```

---

2) *Оператор Если* **<***Условие***>**

```
if  <Условие (любое выражение)>  then
   -- блок --
   [<Список операторов Lua>]
[ { elseif <Условие (любое выражение)>  then
   -- блок --
   <Список операторов Lua> } ]
[  else
    -- блок --
   [<Список операторов Lua>]  ]
 end
```

---

3) *Оператор Пока* **<***Условие***>**

```
while  <Условие (любое выражение)>  do
   -- блок --
   [ <Список операторов Lua> ]
end
```

Цикл while повторяется пока Условие **<**истинно**>**.

---

4) *Оператор цикла repeat–until* повторяет свое тело до тех пор, пока условие не станет истинным.

```
repeat
      -- блок --
      [ <Список операторов Lua> ]
until <Условие (любое выражение)>
```

Цикл *repeat* повторяется пока Условие **<***ложно***>**.

---

5) *Числовой оператор цикла for*.

```
for <Переменная цикла> = <Начало цикла>, <Конец цикла>[ , <Шаг цикла>] do
      -- блок --
```

[<Список операторов Lua>]

end

Если **<**Шаг цикла**>** отсутствует, то по умолчанию он равен 1.

У цикла **for** есть особенности:

-- во-первых, все три выражения вычисляются только *один* раз, перед *началом* цикла

-- во-вторых, управляющая <Переменная цикла> является *локальной* переменной, автоматически объявляемой оператором **for**, и она видна лишь внутри цикла.

Типичная ошибка полагать, что эта переменная все еще существует после окончания цикла.

6) *Общий оператор цикла for* обходит все значения, возвращаемые итерирующей функцией.

for k, v in ***<****Функция итерации****>*** *do*

-- блок --

[<Список операторов Lua>]

end

k, **v** объявляются *локальными* в блоке цикла со всеми последствиями такого объявления (доступны только в теле цикла). **k**, **v** -- переменные итерации (два результата выдаваемые <Функцией итерации> при очередном ее вызове). При **k** := **nil** цикл завершается.

Стандартные библиотеки предоставляют несколько итераторов (<Функций итерации>), позволяющих нам перебирать строки файла (**io.lines**), пары произвольной таблицы (**pairs**), элементы таблицы-массива с последовательными ключами от 1до n (**ipars**), слова внутри строки (**string.gmatch**).

При необходимости можно запрограммировать и свои итераторы.

Функции итерации особые (будут детально рассмотрены далее). Они должны хранить между своими вызовами состояния с тем, чтобы выдавать значения итерации. Причем, для завершения цикла, необходимо, чтобы при каком-то вызове итератора первым его результатом выдалось значение **nil**.

Пример использования итератора **pairs**.

```
-- Цикл печатает все значения таблицы T ---
for k, v in pairs(T) do message ('Ключ = ' .. tostring (k) .. '  Поле = ' .. tostring (v) ) end
```

7) *Операторы* ***break*** *и* ***return*** позволяют "выпрыгнуть" (осуществить безусловный переход) из блока.

Оператор ***goto*** позволяет нам «прыгнуть» практически в любую точку функции. **Lua** накладывает некоторые ограничения на то, куда вы можете прыгнуть при помощи **goto**. Во-первых, метки следуют обычным правилам видимости локальных переменных, поэтому вы не можете прыгнуть внутрь блока (поскольку метка внутри блока невидима за его пределами).

Во-вторых, вы не можете выпрыгнуть из функции. (Обратите внимание, что первое правило уже исключает возможность прыгнуть внутрь функции).

В-третьих, вы не можете прыгнуть внутрь области видимости локальной переменной (после ее объявления).

Оператор **break** используется для завершения цикла любого вида. Этот оператор прерывает внутренний цикл (**for**, **repeat** или **while**), который его содержит; он не может быть использован за пределами цикла. После прерывания программа продолжит выполнение с точки, следующей сразу после прерванного цикла.

Оператор ***return*** **[<**Список значений-результатов**>]** возвращает результаты из функции, если они есть, или просто завершает ее (при отсутствии параметров). В конце каждой функции присутствует неявный возврат из нее, поэтому вам необязательно его использовать, если ваша функция завершается естественным образом, не возвращая никакого значения.

return в том блоке, в котором он находится непосредственно, может располагаться *только* в конце такого блока. Поэтому, если, например, при отладке его надо вставить в середину такого блока, то его можно использовать, создав дополнительную вложенность в блок, в виде:

do return [<Список результатов**>] end**

Функции Lua

Функции **Lua** при своем вызове выполняют операторы своего <тела> .

Создание функции является оператором **Lua**. **<**Тело функции**>**, начиная с ее параметров, является блоком Lua, возникшем в месте ее создания.

<Функция> ::=

function ([<Список параметров>]) -- блок, начинающийся с параметров функции --

<Тело функции>

end

| ***<Создание функции из текстового файла>***

| ***<Создание функции из строки Lua>***

<Список параметров> это локальные переменные функции, которым могут присаиваться значения при вызове (в том числе с помощью выражений **Lua**). Последним параметром может быть три точки **...** , которые обозначают переменный список параметров функции.

<Тело функции> ::=

<Список операторов Lua>

[return [<Список значений-результатов функции >]]

Операторов **return** в теле функции может быть несколько, в разных ее блоках с учетом того, что они должны располагаться в их конце (иначе ошибка – недостижимый код).

!! ***return*** может возвращать несколько значений. Переменный список параметров (**...**) и список результатов можно вставлять в качестве параметров вызова других функций или вставлять в таблицу.

Элементами таблицы (ключами и полями) могут быть функции **Lua**.

В некоторых случаях (например, при программировании в стиле ООП), удобно определять и вызывать функции внутри таблицы, используя синтаксис ***<Функция таблицы>***.

<Функция таблицы> ::= *function* ***<Таблица>:<Переменная-ключ>***

([<Список параметров>])

<Тело функции>

end

При этом, в ***<Таблице>*** создается поле с именем ***<Переменная-ключ>*** *и значением* функция, определенной ***<Телом функции>***. При этом в функции *автоматически* дополнительно создаётся первый параметр с именем ***self***.

При вызове *любой* функций таблицы с использованием *двоеточия*, первым параметром вызываемой функции передается *сама* таблица. Использование в определении ***<Функции таблицы>*** переменной ***self*** это обращение к ***<Таблице>***.

Пример функции.

```
local  function f (p1. p2, …)
          -- блок 1 ---
          local p_tbl = {…}  --- таблица параметров списка, заданного тремя точками
           local a, b   --- по умолчанию значения локальных nil
                .....            -- какие то вычисления ---
            if p1 then
              -- блок 2 (вложен в блок 1) ---
              return  b     --  первый  return (в блоке 2)
            end
          return    --  второй  return (в блоке 1)
   end
   -------
```

<Присвоение функции переменной> ::=

[local] <Имя переменной хранения ссылки на функцию > = ***<Функция>***

| ***[local] function <Имя переменной хранения ссылки на функцию > ([<Список параметров>]) <Тело функции> end***

<Вызов функции на исполнение> ::= <Функция> ([<Список параметров>]) | <Функция> <Конструктор таблицы ({…..}) или строки ('……')>

<Список> ::= <Это результат вызова функции с несколькими значениями> | <В определении функции это три точки в конце описания формальных параметров>

!! *Внимание*.

1. Задание, функции1 с *несколькими* результатами, при вызове *любой* функции2 в качестве фактического параметра или использование функции1 внутри таблицы это *вставка* соответствующего списка значений результата функции1.
2. Конструкция {…} в теле функции это *вставка* в таблицу списка *фактических* параметров, *соответствующих* переменному списку формальных параметров функции, заданных тремя точками.
3. Если в формальных параметрах функции нет переменного списка, то фактические лишние параметры при вызове в функции *недоступны*.
4. Если, фактических параметров в вызове функции меньше чем формальных, то не заданные при вызове формальные параметры имеют *значение* **nil**.
5. Функция с выдачей результата (возможно с несколькими значениями) может быть вызвана в месте, в котором не требуется результат, например в отдельной строке без присвоения.

Стандартная функция **table.pack** упаковывает список в массив-таблицу, а функция **table.unpack** распаковывает массив-таблицу в список (выдает в качестве списка-значений).

Фукции **Lua** допускают рекурсию, то есть, вызов самой функции в своем теле.

Фунцией (точнее *куском*, описанным далее более детально) является, выполняемый скрипт **Lua**.

Функции могут создаваться внутри любой функции **Lua**.

Функциями являются тексты файлов или строк **Lua**, запускаемые как *куски* **Lua**.

---

*Особенности функций **Lua** (замыкания, устранение хвостовых вызовов)*

Функции являются типом данных **Lua** (**function**). Они отличаются от типа **number** только тем, что являются ссылочными и для них существуют свои операции (действия над ними):

1) операция создания функций;
2) операции присвоения функции переменным;
3) операции сравнения: ==, ~=;
4) логические операции;
5) вызов функций на исполнение.

После создания функции нет возможности для изменения существующей функции.

Как и остальные ссылочные типы **Lua**, функции, на которые нет ссылок из выполняемого скрипта, подпадают под уборку (удалению) мусорщиком **Lua**.

--

В функциях можно использовать неэкранированные (ее локальными переменными) переменные окружения, допустимые в любом месте скрипта.

Но, кроме того, функции имеют "лексическую область видимости". Это значит, что в них, из любого доступного места скрипта (в соответствии с их вложенностью в блоки), можно использовать *внешние локальные* переменные, объявленные вне функций. Эта возможность позволяет реализовать в языке **Lua <**Замыкания**>**.

<Замыкания**>** это возможность создания экземпляров функций **Lua** со своими отдельными внешними локальными переменными и их значениями в момент создания. То есть, имеется возможность удобного создания функций, хранящих свое состояние между их вызовами, во *внешних* переменных их локального окружения.

Пример.

```
local function newFunction (p1)  -- Создание функции замыкания ---
    local i = p1
```

```
    return   -- результат функция-замыкание  ---
        function () -- анонимная функция (не присвоена переменной)
                    i = i +1  --- здесь i внешняя локальная переменная функции --
         return i
      end
   end
 -- Результаты вызова функций, выданных функцией newFunction с разными параметрами
---
    local f1 = newFunction (0)   --- в экземпляре функции, присвоенной переменной f1, хранится состояние ее внешней локальной переменной i, заданоой первоночально 0 при вызове newFunction (0)
   local f2 = newFunction (7)   --- в экземпляре функции, присвоенной переменной f2, хранится состояние ее внешней локальной переменной i, заданоой первоночально 7 при вызове newFunction (7)
         f1() := 1
         f2() := 8
         f1() := 2
         f2() := 9
         f2() := 10
         f1() := 3
 ----
```

*-- Динамика стека **Lua** при выполнении ее функций ---*

Как отмечалось ранее, локальные переменные функции (а это и ее параметры), располагаются в ее стековой памяти.

Стек — данные, представляющие собой динамический список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Пример.

```
    local function f1 (p1, p2, p3  )    -- начало 1 - го блока:   p1, p2, p3  (local)
        local  a  --- 4- я локальная переменная 1-го блока
        do         -- начало 2 - го блока, вложенного в 1- й блок
            local  a      --- 1- я локальная переменная 2-го блока
            if 1 then
                -- начало 3- го блока, вложенного во 2- й блок
               local  a      --- 1- я локальная переменная 3-го блока
                <Операторы 3- го блока, вложенного в 1- й блок >
               -- конец 3- го блока, вложенного во 2- й блок
            else   -- начало 4- го блока, вложенного во 2- й блок
              <Операторы 4 - го блока, вложенного во 2- й блок >
               --  конец 4- го блока, вложенного во 2- й блок
            end
          --   продолжение 2 - го блока, вложенного в 1- й блок
           <Операторы 2 - го блока, вложенного в 1- й блок >
        end     -- конец 2 - го блока, вложенного в 1- й блок
        return f2 (a)   --- a из блока 1
   end    --  конец 1 - го блока
   -----
```

При выполнении функции (а функцией является и сам скрипт), по мере исполнения ее операторов идет динамическое "разворачивание" ее блоков, хранимых в ее стеке.

При выходе вычислений из любого блока, его переменные исчезают.

```
   ------
```

-- Устранение "хвостовых" вызовов в функциях Lua ---

Рассмотрим функцию:

```
  local function f1 (p1)              -- Создание функции замыкания ---
     <Операторы функции f1>
     local function f2 (p1)   <Операторы функции f2>   end
      return f2 (p1)
```

end

Существуют ситуации, когда перед вызовом внути функции ***f1*** функции ***f2***, можно удалять неиспользуемый стек функции ***f1***. Например, это возможно при вызове в ***f1***: *return* ***f2****(<Параметры>)* -- если в ***f2*** нет внешних локальных переменных из ***f1***. В **Lua** реализована, для упомянутых ситуаций, оптимизация вызовов функций, называемая устранением "хвостовых" вызовов (удаление неиспользуемого стека функции). Это обеспечивает для рассмотренного выше случая реализацию рекурсий в скриптах *без роста* его стека.

Любая функция **Lua** может быть вызвана в защищенном режиме, в котором перехватываются все ошибки, с помощью служебных функций **pcall** или **xpcall**. Особенности использования этих функций описаны в разделе «Обработка ошибок в **Lua**».

Lua обеспечивает компиляцию «на лету». Функции Lua могут быть определены в виде строки-кода (**string**) **Lua (**с последующей компиляцией), в том числе, хранящейся на внешнем накопителе.

Компиляция текста-кода Lua

Lua всегда, перед выполнением, компилирует исходный *текст-код* в промежуточную форму (*байт-код*), который обрабатывается виртуальной **Lua**-машиной. Но можно создавать *байт-код* и автономно, используя программу **luac.exe**.

В результате компиляции создается единица трансляции **Lua** – *кусок*.
Кусок это анонимная (без внутреннего имени) функция **Lua**, реализующая следующий интерфейс:

1. У функции *куска* существует внешняя локальная переменная с именем **_ENV**.
2. Все переменные окружения *куска,* без спецификации **local,** синтаксически эквивалентны: **_ENV.<**Переменная окружения *куска***>**. Тоесть, предполагается, что значением **_ENV** может быть таблица (по умолчанию, это глобальная таблица **_G**) и переменные окружения *куска* являются ключами этой таблицы.
   _ENV можно в *куске* присвоить любую таблицу и после этого она станет таблицей окружения *куска*. **_ENV** можно присвоить **Lua**-тип отличный от **table**, но при обращении **_ENV.<**Переменная окружения *куска***>** возникнет соответствующия ошибка времени выполнения.
3. Параметры куска заданы в виде: (**…**). Таким образом, при вызове куска можно задавать переменное количество параметров.
   Обрабатывать параметры внутри куска можно (если это требуется), например, используя следующую идиому:
   Local PRM = {**…**} --- таблица параметров *куска* ---
   --- Далее: первый параметр куска PRM[1] и т.д. ---
4. Как и любая функция **Lua**, *кусок* может выдать при своем вызове результат (возможно несколько значений).

Существенной и удобной *особенностью кусков* (при этом сам основной скрипт также является *куском*) является возможность задания в них своих *отдельных* окружений.

Служебные функции **Lua**, создающие *куски*:

1. **require** - функция поиска и подключения модулей **Lua** (подробно описанная в разделе «Модули и пакеты Lua»).
2. **load (chunk [, chunkname [, mode [, env]]])** - загружает кусок.

Если **chunk** строка, *куском* будет результат компиляции этой строкой. Если **chunk** функция, **load** многократно вызывает её, чобы получить части *куска*. Каждый вызов **chunk** должен возвращать строку, которая присоединяется к предыдущим результатам. Возврат пустой строки, **nil** или ничего сигнализирует о конце куска.

Если в куске нет синтаксических ошибок, возвращает скомпилированный кусок, как функцию; иначе, возвращает **nil** и сообщение об ошибке.

Если результирующая функция имеет **upvalue** (внешние локальные переменные), первое **upvalue** устанавливается равным **env**, если этот параметр передан, или в значение глобального окружения **_G**. Остальные **upvalue** инициализируются значением **nil**. (Когда вы загружаете главный (начальный) кусок, результирующая функция всегда будет иметь только одно **upvalue**, переменную _ENV. Тем не менее, когда вы загружаете бинарный кусок, созданный из функции (см. string.dump), результирующая функция может иметь произвольное количество **upvalue**.) Все **upvalue** доступны только в созданном *куске*.

chunkname - используется как имя куска, для сообщений об ошибках и отладочной информации. Когда отсутствует, его значение по умолчанию это сам **chunk**, если **chunk** это строка, или "=(load)" иначе.

Строка **mode** - управляет тем, каким может быть кусок, текстовым или бинарным (т.е. прекомпилированным). Она может быть "b" (только бинарные куски), "t" (только текстовые куски) или "bt" (текстовые и бинарные куски). Значение по умолчанию это "bt".

Lua не проверяет правильность бинарных кусков при их подключении.

3. **loadfile** (**[filename [, mode [, env]]]**) - аналогично **load**, но получает кусок из файла *filename* или из стандартного потока ввода, если имя файла не передано.
4. **dofile** (**[filename]**)

Открывает файл и запускает его содержимое, как **Lua**-*кусок*. Когда функция вызвана без аргументов, запускает содержимое стандартного ввода (**stdin**). Возвращает все значения, возвращенные куском. **dofile** запускается в незащищенном режиме (в нем возникающие ошибки не перехватываются).

Обработка ошибок в функциях

В функциях могут возникать ошибки двух видов: *обрабатываемые* и *фатальные*.

Обрабатываемые ошибки, устраняемые в функции, являются частью функционала функции и здесь не рассматриваются.

Фатальная ошибка в функии возникает тогда, когда продолжение выполнения функции *невозможно или не имеет смысла*. Причем, фатальная ошибка в вызванной функции, вполне может быть *обрабатываемой* в вызывающей. Например, вызванная функция неспособна обработать какой-то текст, но в вызывающей функции предусмотрено исправление этого текста и повторная его обработка.

Для обработки фатальных ошибок в **Lua** используется известная схема генерации *исключений*. Исключение это специальная схема завершения функции при возникновении фатальной ошибки. При таком завершении выполнение всего стека вызванных функций останавливается с генерацией сигнала исключения для всех остальных функций стека вызова. Если в остальных фунциях стека не установлен режим перехвата исключения, то оно будет перехвачено интерпретатором **Lua**. При этом скрипт будет завершен с выдачей сообщения исключения. Однако, любую функцию мужно вызывать в *защищенном* режиме, в котором неперхваченное исключение в вызванных функциях, будет перехвачено с возможностью его обработки. В любой функции можно программно генерировать свои исключения.

Перехватывать чужие исключения *имеет смысл* только в том случае, если *предполагается* какая то их обработка (хотя бы генерация своего исключения с добавление своего сообщения).

Служебные функции обработки ошибок в **Lua**:

1) Функция генерации (имитации) ошибки (исключения) **error**;

2) Функция проверки условий (предикатов) правильности значений данных скрипта в месте вызова проверки и генерации ошибки **assert**;
3) Функция запуска любых функций в защищенном режиме **pcall**;
4) Функция запуска любых функций в защищенном режиме с возможностью анализа стека вызова **xpcall**.

1. ***error (<**Текст генерируемого исключения**> [, <**Уровень функции источника ошибки**>])***

   Функция генерирует исключение с выдачей **tostring**(**<**Фактическое значения параметра Текста сгенерированного исключения**>**).

   Необязательным параметром **<** Уровень функции источника ошибки**>** можно указать *предполагаемую* функцию источник ошибки. У функции, в которой вызывается **error,** уровень 1. Следующая в стеке вызовов функция уровень 2 и т.д.

2. ***assert (<**Проверяемое условие**>, <**Текст генерируемого исключения**>)***

   <Проверяемое условие**>** - произвольное значение **Lua**. Следует учитывать, что в логических выражениях **Lua** только **false** и **nil** интерпретируются как *ложь,* все остальное как *истина*. Причем истинный результат может быть любым, а не обязательно **true** (смотрите раздел описания логических выражений). Если **<**Проверяемое условие**>** вызов функции с несколькими значениями, то второе значение будет при вызове **assert** вставлено на место **<**Текста сгенерированного исключения**>**. Поэтому для случая, когда результат **<**Проверяемого условие**>** состоит из двух значений и первое является условием, а второе текстом описания исключения, второй параметр **assert** задавать не надо.

   Результатом **assert,** если первое значение **<**Проверяемого условия**>** *истина* (отлично от **false** и **nil**) будет первое значение **<**Проверяемого условия**>**. Иначе будет сгенерировано исключения с выдачей **tostring**(<Фактическое значения параметра Текста генерируемого исключения>).

3. ***pcall(<**Функция вызываемая в защищенном режиме**>, <**Список фактических параметров вызываемой функции**>)***

   Вызывает функцию с параметрами в *защищенном режиме*. Это значит, что любая ошибка внутри нее не распространяется. **pcall** перехватывает ошибку и возвращает код статуса. Её первый результат - код статуса (логическое значение), которое равно true, если вызов успешен. В этом случае **pcall** также возвращает все значения из вызова функции, после первого результата (true – нет ошибки). В случае ошибки **pcall** возвращает **false** и вторым значением сообщение об ошибке.

4. ***xpcall(<**Функция вызываемая в защищенном режиме**>, <**Функция обработки сообщений **>, <**Список фактических параметров вызываемой функции**>)***

   В **xpcall** можно определить *обработчик сообщений*, который будет вызываться в случае ошибок. Эта функция вызывается с оригинальным сообщением об ошибке в ее первом параметре и возвращает новое сообщение об ошибке. Она вызывается до раскрутки стека и сможет получить больше информации об ошибке, например, проверяя стек и создавая историю стека. Этот обработчик сообщений остается защищенным в защищенном вызове и ошибка в обработчике сообщений лишь вызовет его снова. Если этот цикл будет достаточно длинным, **Lua** прервет его и вернет соответствующее сообщение. ***xpcall*** возвращает все значения из вызова функции, после первого результата (true – нет ошибки). В случае ошибки **pcall** возвращает **false** и результат ***<**Функции обработки сообщений **>***.

   debug.traceback при использовании в качестве ***<**Функция обработки сообщений **>*** выдает, в случае ошибки***<**Функции вызванной в защищенном режиме**>,*** строку описания стека ее вызовов.

Хотя исключения в **Lua**, реализованы на *уровне* функций, а не фрагментов кода, любой фрагмент скрипта *легко* оформить как функцию, вызываемую затем в *защищенном* режиме.

Функции итераторы

Итератор (***iterator***) — это любая конструкция, которая позволяет, при многократном вызове ее, перебирать элементы коллекции данных **Lua**.

Любой итератор должен где-то *хранить* свое состояние между последовательными вызовами, чтобы знать, где он находится и как себя вести с этого места. Замыкания предоставляют один из механизмов для этой задачи.

Конструкция замыкания обычно включает в себя: само замыкание и фабрику — функцию, которая создает замыкание вместе с окружающими ее внешними локальными переменными.

Пример простого итератор для таблицы - массива (последовательности). В отличие от **ipairs**, этот итератор возвращает не индекс каждого элемента, а лишь его значение:

```
function values (t)
    local i = 0
    return function () i = i + 1; return t[i] end
end
```

values — это фабрика функций-замыкания.

Использование values в качестве итератора:

```
t = {10, 20, 30}
for element in values(t) do
    print(element)
end
```

Недостатком рассмотренного итератора (замыкания) является то, что нам необходимо создавать новое замыкание для инициализации каждого нового цикла.

Однако, можно использовать сам общий ***for*** для хранения состояния итерации..

Синтаксис общего **for** следующий:

for <список_локальных_переменных for> in <список_выражений итератора> do
<тело>
end

Первую переменную в **<** *списке_локальных_переменных for* **>** называется **<**управляющей переменной**>**. В течение всего цикла ее значение никогда не равно **nil**, поскольку, когда она становится равной **nil**, цикл завершается.

Первое, что делает общий цикл ***for***, — вычисляет выражение после ***in*** и множественно присваивает своим трем внутренним локальным переменным. Это выражение может дать три значения, которые хранит ***for*** : **<**итерирующая функция**>**, **<**переменная состояния (используемая, но необязательно при итерации)**>** и **<**начальное значение**>** управляющей переменной. Когда мы используем простые итераторы, фабрика возвращает только итерирующую функцию, поэтому переменная состояния и управляющая переменная получают значение **nil**.

После этого шага инициализации **for** вызывает итерирующую функцию с двумя аргументами: **<**переменной состояния**>** и **<**управляющей переменной**>**.

Затем ***for*** присваивает значения, возвращенные итерирующей функцией, переменным, объявленным в его **<**списке переменных**>**. Если первое возвращенное значение (присваиваемое **<**управляющей переменной**>**) равно **nil**, то цикл завершается. Иначе ***for*** выполняет свое тело и вновь вызывает итерирующую функцию, повторяя процесс.

Конструкция общего **for**:

```
for var_1, ..., var_n in <список_выражений итератора (1-3 значения)> do <тело цикла>   end
```

эквивалента следующему коду:

```
do
```

```
    local _f, _s, _var = <список_выражений итератора>   --  _f, _s, _var - внутренние переменные хранения состояния итерации
    while true do
      local var_1, ... , var_n = _f(_s, _var)     -- var_1 - управляющая переменная
      _var = var_1
      if _var == nil then break end
      <тело цикла>
    end
  end
```

Поэтому если наша итерирующая функция ***_f,*** состояние ***_s***, а начальное состояние для управляющей переменной (*_var*) **a0**, то управляющая переменная будет пробегать следующие значения **a1** = f (s, **a0**), **a2** = f (_s, **a1**) и т. д., до тех пор, пока **ai** не станет равной **nil**. Если у **for** есть другие переменные, то они просто получат дополнительные значения, возвращаемые при каждом вызове ***f***.

Пример функции интерации без хранения состояния итерации:

```
local function iter (a, i)
      i = i + 1
      local v = a[i]
      if v then return i, v end
end
 -- Использоание iter
   a = {"one", "two", "three"}
   for i, v in iter, a, 0 do print(i, v) end
-------------------------------------
```

Итератор для обхода связанного списка является еще одним интересным примером итератора без состояния.

```
---
local function getnext (list, node)
  if not node then
    return list
  else
    return node.next
  end
end
---
 -- Функция для присвоения <списку_выражений итератора>:
function traverse (list)
  return getnext, list, nil
end
-- Использоание traverse ---
list = nil
for line in <Открытый текстовый файл>.lines() do   -- создание связанного списка --
  list = {val = line, next = list}
end
for node in traverse(list) do print(node.val) end        -- проход по связанному списку --
-----------------------------------------------
```

Для реализации итераторов можно использовать и сопрограммы. Это эффективное решение, хотя и более затратное в реализации, будет рассмотрено далее.

Сопрограммы Lua

Сопрограмма (**coroutine**) похожа на нити в Windows. Это средство выполнения функции со своим отдельным стеком, своими локальными переменными и своим указателем команд. Такая функция разделяет переменные окружения и все остальное, доступное ей в общей области видимости, с другими сопрограммами и функциями скрипта.

Сопрограммы работают совместно (но *последовательно* в одном потоке с функцией, запустившей сопрограммы). В любой момент времени программа с сопрограммами вы-

полняет *только* одну из своих *сопрограмм* или управляющую (сопрограммой) функцию. При этом выполняемая *сопрограмма* приостанавливает свое выполнение, только тогда, когда в ней встретится оператор ее приостановки, и при этом продолжается выполнение приостановленной функции, вызвавшей сопрограмму.

Создание сопрограммы:

co = **coroutine.create**(<Функция>)

Lua пакует все связанные с сопрограммами функции в таблицу ***coroutine***. Функция ***create*** создает новые сопрограммы. У нее есть единственный аргумент — функция с кодом, которую сопрограмма будет выполнять. Она возвращает значение типа **thread**, которое представляет из себя новую сопрограмму.

Сопрограмма может быть в одном из четырех состояний: приостановленное — *suspended*, выполняемое — *running*, завершенное — *dead*, обычное — *normal*.

Можно проверить состояние сопрограммы при помощи функции ***status*:*print***(coroutine.status(co)). Когда мы создаем сопрограмму, она находится в приостановленном состоянии. Функция *coroutine.resume* (пере)запускает/продолжает выполнение сопрограммы, меняя ее состояние из приостановленного в выполняемое: ***coroutine.resume****(co).*

Функция ***yield*** позволяет выполняемой сопрограмме приостановить свое выполнение (иными словами, уступить управление функции-родителю), чтобы она могла быть возобновлена позже. С точки зрения сопрограммы, вся деятельность, которая происходит, пока сопрограмма приостановлена, происходит внутри вызова ***yield***.

Когда мы возобновляем сопрограмму, из вызова ***yield*** возвращается управление, и сопрограмма продолжает свое выполнение до следующего ***yield*** или до своего окончания.

Lua предлагает то, что называется асимметричными сопрограммами. Это значит, что у нее есть одна функция для приостановки выполнения сопрограммы и другая функция для возобновления приостановленной сопрограммы.

resume выполняется в защищенном режиме. Поэтому, если внутри сопрограммы есть какие-либо ошибки, **Lua** не будет показывать сообщение об ошибке, а просто вернет управление вызову resume.

Пара ***resume-yield*** может обмениваться данными. При начальном запуске сопрограммы, *resume*, у которой нет ожидающей ее *yield* (сопрограммы), передает свои дополнительные аргументы функции сопрограммы (через ее параметры). Если ***resume*** возвращает **true**, то нет ошибок и все аргументы переданны соответственной ***yield*** *(*сопрограмме*)*. При последующих вызовах ***resume****, ее* дополнительные аргументы передаются сопрограмме в месте вызова ***yield*** сопрограммы *(*в виде результата ***yield****)*. Результат ***resume***:

1) если ошибка при вызове, то *false* и сообщение об ошибке;
2) иначе вызов сопрограммы, а после возврата из нее *true* и параметры переданные сопрограммой в *yield.*

Аналогично ***yield*** передает все дополнительные свои аргументы соответствующей ***resume*** *(*в виде результата ***resume****)*. Результат ***yield***: параметры,переданные в *resume.*

Наконец, когда сопрограмма завершается, любые значения, возвращенные ее функцией, передаются соответственной ***resume*** *(*в виде результата ***resume****)*.

Есть случаи обработки данных, для которых использование сопрограм может быть хорошим решением. Например, сопрограммы можно использовать для реализации итераторов.

Пример итератора для перебора всех перестановок заданного массива, реализованного с помощью сопрограмы.

Написание подобного итератора напрямую не так легко, но несложно написать рекурсивную функцию, которая генерирует все эти перестановки. Идея проста: по очереди по-

мещать каждый элемент массива на последнюю позицию и рекурсивно генерировать все перестановки оставшихся элементов.

1. Функция печати результата перестановоки:

```
local out =
function printResult (a)
  for i = 1, #a do
     out = out .. tostring ( a[i] ) ..'  '
  end
  out = out .. '\n'
end
---
```

Функция для получения всех перестановок из первых n элементов a (генератор перестановок):

```
function permgen (a, n)
  n = n or #a -- значение 'n' по умолчанию — размер 'a'
  if n <= 1 then -- ничего не изменилось?
    coroutine.yield(a)   --  printResult(a)
  else
    for i = 1, n do
      -- помещает i-ый элемент как последний
      a[n], a[i] = a[i], a[n]
      -- генерирует все преобразования прочих элементов
      permgen(a, n - 1)
      -- восстанавливает i-ый элемент
     a[n], a[i] = a[i], a[n]
  end
end
----
```

Определяем фабрику, которая делает так, чтобы генератор выполнялся внутри сопрограммы, а затем создаем итерирующую функцию. Для получения следующей перестановки итератор просто возобновляет сопрограмму:

```
function permutations (a)
  local co = coroutine.create(function () permgen(a) end)
  return function ()                  --   итератор (замыкание) --
     local code, res = coroutine.resume(co)
     return res
   end
end
---
```

Перебрать всех перестановок массива при помощи общего оператора for :

Вариант 1 (с использованием замыкания):

```
for p in permutations{"a", "b", "c"} do
  printResult(p)
end
-----
```

Вариант 2 (без использования замыкания):.

```
local per_mgen function (co) local code, res = coroutine.resume(co); return res  end  -- итерирующая функция, вызвающая объекта генераци элементов итерации.
function permutations_ (a)  ---- функция выдачи функции - итерации и объекта генераци элементов итерации (сопрограммы) ---
   return per_mgen, coroutine.create(function() permgen(a) end)
end
 ----
for p in permutations_({"a", "b", "c"}) end)  do    --
     printResult (p)
end
----
```

Lua предоставляет особую функцию: ***coroutine.wrap***. Как и *create*, *wrap* создает новую сопрограмму. В отличие от *create*, *wrap* не возвращает саму сопрограмму. Вместо этого она возвращает функцию, которая при вызове возобновляет эту сопрограмму. В отличие от *resume*, она не возвращает код ошибки как свой первый результат; вместо этого при необходимости она вызывает (генерирует) ошибку.

Использование стеков сопрограмм в отдельных потоках в раделяемом режиме (под синхронизацией) будет описано далее в разделе «Конфигурирование Lua для режима запуска сопрограмм в отдельных потоках».

Метатаблицы, слабые таблицы и финализаторы

Общие сведения о *метаблицах* были приведены ранее при описании типа ***table***. Здесь будут рассмотрены более подробно служебные ключи таких таблиц и их интерпретация.

Слабые таблицы и *финализаторы* — это механизмы, которые можно использовать в **Lua**, чтобы управлять сборщиком мусора. Слабые таблицы позволяют сбор объектов **Lua**, на которые есть только *слабые* ссылки (из *слабых* таблиц), в то время как *финализаторы*, позволяют финальную обработку *внешних* объектов, не находящихся под *непосредственным* контролем сборщика мусора. Финализаторы и слабые таблицы задаются с помощью специальных ключей метатаблиц.

Метатаблицы

Каждое значение (в том числе и таблица, подключенная как метатаблица) в **Lua** может иметь *метатаблицу*. *Метатаблица* это таблица **Lua** (подключаемая функцией setmetatable к значению), которая определяет поведение оригинального значения в определенных специальных операциях, предопределенных служебными ключами. Вы можете изменять различные аспекты поведения в операциях, используя служебные поля (начинающиеся с двух подчерков) в метатаблице. Например, когда не цифровое значение является операндом в сложении, **Lua** проверяет есть ли в его метатаблице поле "__add" с функцией. И, если оно существует, Lua вызывает эту функцию для выполнения сложения.

Группа связанных между собой таблиц может совместно использовать общую метатаблицу, которая описывает их общее поведение. Таблица может быть метатаблицей для самой себя, таким образом, описывая свое собственное индивидуальное поведение. Синтаксически допустимо использовать любую схему подключения метатаблиц. У таблицы, подключаемой как метатаблица нет специального признака, что она используется как метатаблица. Таблица, к которой подключена таблица, как метатаблица, ссылается на подключенную таблицу (и это может быть, в том числе, сама исходная таблица).

Служебные ключи в метатаблице это производные от имен *событий*; соответствующие им значения называются *метаметоды*. В предыдущем примере, событие - "add" (добавить) и метаметод - функция, которая выполняет сложение.

Можно запросить метатаблицу любого значения, используя функцию **getmetatable**.

Можно задать/заменить/отключить (заменив на ***nil***) метатаблицу, используя setmetatable. Нельзя изменять метатаблицы предопределенных типов из Lua кода (кроме, как используя библиотеку отладки); для этого надо использовать **C API**.

Любой экземпляр таблицы и пользовательских данных (**userdata**) могут иметь индивидуальные метатаблицы. Значения остальных типов используют одну метатаблицу на каждый тип; т.е, существует одна метатаблица для всех чисел, одна для всех строк и т.д. По умолчанию, значения не имеют метатаблицу, но строковая библиотека создает метатаблицу для строкового типа.

Метатаблица контролирует, как объект ведет себя в арифметических и битовых операциях, сравнениях при сортировке, конкатенации, определении длины, вызовах и индексировании. Метатаблица также может определять функцию, которая будет вызвана для таблицы или пользовательских данных при уничтожении сборщиком мусора.

Детальное описание событий, контролируемых метатаблицами, представлено ниже. Ключ для каждого события это строка начинающаяся с двух подчеркиваний, '**__**'; например, ключ для операции "add" строка "**__add**". Вызовы метаметодов всегда прямые; при вызове некоторого метаметода, остальные метаметоды таблицы не вызываются..

Для двухместных операций Lua определен следующий шаблон соответствующих/вызываемых функций-метаметодов:
function f (S1, S2) …. return ***<****Результат****>*** *end*

Для одноместных операторов (отрицание, длина и битовое отрицание), метаметод вычисляется и вызывается с фиктивным вторым операндом, равным первому. Этот дополнительный операнд нужен лишь для упрощения реализации **Lua**, и может быть убран в следующих версиях.

Список *некоторых* ключей метатаблицы:

- **"__add":** + операция. Если любой операнд при сложении не число (и не строка, которую можно преобразовать в число), Lua попробует вызвать метаметод. Сначала, Lua проверит первый операнд (даже если он правильный). Если этот операнд не определяет метаметод для события "__add", Lua проверит второй операнд. Если Lua найдет метаметод, он будет вызван с двумя операндами в качестве аргументов, и результат вызова (скорректированный до одного значения) будет результатом операции. Иначе будет сгенерирована ошибка.
- **"__sub":** - операция (вычитание). Аналогично операции "add".
- **"__mul":** * операция (умножение). Аналогично операции "add".
- **"__div":** / операция (деление). Аналогично операции "add".
- **"__mod":** % операция (остаток от деления). Аналогично операции "add".
- **"__pow":** ^ операция (возведение в степень). Аналогично операции "add".
- **"__unm":** - операция (одноместный минус). Аналогично операции "add".
- **"__idiv":** // операция (целочисленное деление). Аналогично операции "add".
- **"__band":** & операция (битовое И). Аналогично операции "add", за исключением того, что Lua будет использовать метаметод, если любой из операндов не целое и не значение приводимое к целому (см. §3.4.3).
- **"__bor":** | операция (битовое ИЛИ). Аналогично операции "band".
- **"__bxor":** ~ операция (битовое ИЛИ-НЕ). Аналогично операции "band".
- **"__bnot":** ~ операция (битовое одноместное НЕ). Аналогично операции "band".
- **"__shl":** << операция (битовый сдвиг влево). Аналогично операции "band".
- **"__shr":** >> операция (битовый сдвиг вправо). Аналогично операции "band".
- **"__concat":** .. операция (конкатенация). Аналогично операции "add", за исключением того, что Lua будет использовать метаметод, если любой из операндов не строка и не число (которое всегда приводимо к строке).
- **"__len":** # операция (длина). Если объект не строка, Lua попытается использовать этот метаметод; Если метаметод определен, он будет вызван с объектом в качестве аргумента, и результат вызова (обрезанный до одного значения) будет использован как результат операции. Если метаметод не определен и объект таблица, **Lua** использует операцию длины таблицы (см. §3.4.7). Иначе, **Lua** сгенерирует ошибку.
- **"__eq":** == операция (тождество). Аналогично операции "add", за исключением того, что Lua будет использовать метаметод, только если оба сравниваемых значения таблицы или полные пользовательские данные и они не примитивно равны. Результат вызова всегда преобразуется к логическому (boolean).

- **"__lt":** < операция (меньше). Аналогично операции "add", за исключением того, что Lua будет использовать метаметод, только если оба сравниваемых значения не числа и не строки. Результат вызова всегда преобразуется к логическому (boolean).
- **"__le":** <= операция (меньше или равно). В отличие от других операций, операция "<=" может использовать два разных события. Первое, **Lua** проверяет наличие метаметода "__le" в обоих операндах, также как в операции "lt". Если этот метаметод не найден, **Lua** попытается использовать событие "__lt", предполагая, что a <= b эквивалентно not (b < a). Как и у других операторов сравнения, результат всегда логическое значение. (Это использование события "__lt" может быть убрано в следующих версиях, т.к. оно медленнее, чем реальный вызов метаметода "__le".)
- **"__index":** индексированный доступ table[key]. Это событие случается когда **table** не таблица или когда key не существует в **table**. Метаметод ищется в объекте **table**.

  Несмотря на имя, метаметод для этого события может быть функцией или таблицей. Если это функция, то она вызывается с table и key в качестве аргументов. Если таблица, то конечный результат это результат индексирования этой таблицы с ключом key. Это индексирование рекурсивное, не прямое, и оно также может вызывать срабатывание другого метаметода.

- **"__newindex":** индексированное присваивание table[key] = value. Как и событие "__index", это событие случается когда table не таблица или когда key не существует в table. Как и для индексированного доступа, метаметод для этого события может быть функцией или таблицей. Если это функция, то она вызывается с основной table, key и value в качестве аргументов. Если таблица, **Lua** производит индексированное присваивание в этой таблице с тем же ключом и значением. Это присваивание может вызывать срабатывание другого метаметода (если необходимо, метаметод может самостоятельно вызвать **rawset** для выполнения «сырого» присваивания без учета существования метатабицы).
- **"__call":** операция вызова func(args). Это событие случается когда **Lua** пытается вызвать значение, не являющееся функцией (т.е., func это не функция). Метаметод ищется в объекте func. Если он существует, то он вызывается с func в качестве первого аргумента, следом идут остальные аргументы из оригинального вызова (args).
- **"__*gc*":** операция финализации. Это событие случается при сборке мусора, если удадяется память, занимаемая данным, к которому подключена метатаблица с таким ключем.

Хорошая практика, добавлять все необходимые метаметоды в таблицу перед тем, как назначить её метатаблицей какого-то объекта. В частности, *метаметод "__gc" работает только если была соблюдена эта последовательность.*

Примеры использования метатаблиц

Метаметод _index

Реализация наследования.

Пусть создается несколько таблиц, описывающих окна. Каждая таблица должна описывать различные параметры окна, такие как положение, размер, цветовая схема и т. п. Для всех этих параметров есть значения по умолчанию и поэтому мы хотим строить объекты окон, задавая только те значения, которые отличаются от значений по умолчанию. Первый вариант — предоставить конструктор, заполняющий отсутствующие поля. Второй вариант — сделать так, чтобы новые окна *наследовали* любое отсутствующее поле от прототипа окон. Для начала мы объявим прототип и функцию-конструктор, которая создает новые окна, обладающие общей метатаблицей:

```
-- создает прототип со значениями по умолчанию
```

```
prototype = {x = 0, y = 0, width = 100, height = 100}
mt = {} -- создает метатаблицу
-- объявляет функцию-конструктор
function new (o)
setmetatable(o, mt)
return o
end
```

Теперь мы определим метаметод `__index`:

```
mt.__index = function (_, key)
  return prototype[key]
end
```

После этого кода мы создадим новое окно и обратимся к отсутствующему полю:

```
w = new{x=10, y=20}
print(w.width) --> 100
```

Lua обнаружит, что у w нет требуемого поля, но есть метатаблица с полем __index. Поэтому Lua вызовет этот метаметод с аргументами w (таблица) и "width" (отсутствующий ключ). Затем метаметод индексирует этот прототип заданным ключом и возвращает полученное значение. Использование метаметода __index для наследования в **Lua** настолько распространено, что **Lua** предоставляет сокращенный вариант. Хотя его и зовут *методом*, метаметод __index не обязан быть функцией: например, он может быть таблицей. Когда он является функцией, **Lua** вызывает его, передавая таблицу и отсутствующий ключ в качестве аргументов, как мы только что видели. Когда он является таблицей, **Lua** перенаправляет к ней обращение. Поэтому в нашем предыдущем примере мы могли просто определить `__index` следующим образом:

```
mt.__index = prototype
```

Теперь, когда **Lua** будет искать метаметод __index, он найдет значение `prototype`, которое является таблицей. Соответственно, **Lua** повторит обращение к этой таблице, то есть выполнит аналог `prototype["width"]`. Это обращение и приведет к нужному результату. Использование таблицы в качестве метаметода `__index` дает простой и быстрый способ реализации одиночного наследования.

Когда мы хотим обратиться к таблице без вызова ее метаметода __index, мы используем функцию **rawget**. Вызов `rawget(t,i)` осуществляет *непосредственный* доступ к таблице t, то есть примитивное обращение без использования метатаблиц.

Метаметод __newindex

Метаметод `__newindex` делает то же, что и `__index`, но работает *при обновлениях* таблиц, а не при доступе к ним. Когда вы присваиваете значение отсутствующему индексу в таблице, интерпретатор ищет метаметод `__newindex`: если он есть, то интерпретатор вызывает его вместо выполнения присваивания. Подобно `__index`, если метаметод является таблицей, то интерпретатор выполняет присваивание для этой таблицы вместо исходной. Более того, есть функция с прямым доступом, которая позволяет миновать метаметод: rawset (t, k, v) записывает значение v по ключу k в таблицу t, не вызывая никаких метаметодов. Совместное использование метаметодов `__index` и `__newindex` позволяет реализовать в **Lua** ряд довольно мощных конструкций, таких как таблицы, доступные только для чтения, таблицы со значениями по умолчанию и наследование для объектно-ориентированного программирования.

Таблицы со значениями по умолчанию

Значение по умолчанию для любого поля в обычной таблице — это **nil**. Это легко изменить при помощи метатаблиц:

```
function setDefault (t, d)
  local mt = {__index = function () return d end}
  setmetatable(t, mt)
end
```

```
tab = {x=10, y=20}
print(tab.x, tab.z) --> 10 nil
setDefault(tab, 0)
print(tab.x, tab.z) --> 10 0
```

После вызова *setDefault* любой доступ к отсутствующему полю в *tab* вызовет его метаметод __index, который вернет ноль (значение d для этого метаметода). Функция *setDeafult* создает новое замыкание и новую метатаблицу для каждой таблицы, которой нужно значение по умолчанию., Это может оказаться затратным, если у нас много таблиц, которым нужны значения по умолчанию. У метатаблицы значение по умолчанию d «зашито» в ее метаметод, поэтому мы не можем использовать одну и ту же метатаблицу для всех таблиц. Чтобы можно было использовать одну и ту же метатаблицу для таблиц с разными значениями по умолчанию, мы можем запоминать значение по умолчанию в самой таблице, используя для этого специальное поле. Если мы не беспокоимся о конфликтах имен, мы можем использовать для нашего особого поля ключ вроде "___":

```
local mt = {__index = function (t) return t.___ end}
function setDefault (t, d)
  t.___ = d
  setmetatable(t, mt)
end
```

Обратите внимание, что теперь мы создаем таблицу **mt** лишь один раз, вне функции setDefault. Если мы беспокоимся о конфликтах имен, то можно создать новую таблицу (все таблицы в рамках Lua-инсталяции уникальны) и *использовать ее* в качестве ключа:

```
local key = {} -- уникальный ключ
local mt = {__index = function (t) return t[key] end}
function setDefault (t, d)
  t[key] = d
  setmetatable(t, mt)
end
```

Отслеживание доступа к таблице

И ***__index***, и ***__newindex*** работают только при отсутствии в таблице соответствующего индекса. Поэтому единственный способ отслеживать весь доступ к таблице — это держать ее пустой. Таким образом, если мы хотим отслеживать весь доступ к таблице, нам нужно создать посредника (proxy) для настоящей таблицы. Данный посредник — это пустая таблица с соответствующими метаметодами __index и __newindex для отслеживания доступа к таблице, которые будут перенаправлять
доступ к исходной таблице. Пусть t — это исходная таблица, доступ к которой мы хотим отслеживать. Тогда мы можем написать что-то вроде этого:

```
t = {} -- исходная таблица (созданная где-то еще)
-- хранит закрытый доступ к первоначальной
таблице
local _t = t
-- создает посредника
t = {}
-- создает метатаблицу
local mt = {
__index = function (t, k)
  print("*access to element " .. tostring(k))
  return _t[k] -- доступ к исходной таблице
end,
__newindex = function (t, k, v)
  print("*update of element " .. tostring(k) ..
        " to " .. tostring(v))
```

```
  _t[k] = v -- о б н о в л я е т  и с х о д н у ю  т а б л и ц у
end
}
setmetatable(t, mt)
```

Этот код отслеживает каждое обращение к t:

```
> t[2] = "hello"
*update of element 2 to hello
> print(t[2])
*access to element 2
hello
```

Если нам требуется обойти эту таблицу, мы должны определить в посреднике запись `__pairs`:

```
mt.__pairs = function ()
  return function (_, k) return next(_t, k)  end
end
```

Нам может понадобиться нечто подобное для `__ipairs`. Если мы хотим следить за несколькими таблицами, то нам не нужно для каждой из них создавать отдельную метатаблицу. Вместо этого мы можем как-нибудь связать каждого посредника с его исходной таблицей и разделить одну общую метатаблицу между всеми посредниками. Это похоже на задачу связывания таблицы с ее значениями по умолчанию, которую мы обсуждали в предыдущем разделе. Например, можно хранить исходную таблицу в поле посредника при помощи специального ключа. Результатом является следующий код:

```
local index = {} -- с о з д а е т  з а к р ы т ы й  и н д е к с
local mt = { -- с о з д а е т  м е т а т а б л и ц у
__index = function (t, k)
   print("*access to element " .. tostring(k))
   return t[index][k] -- д о с т у п  к  и с х о д н о й  т а б л и ц е
end,
__newindex = function (t, k, v)
       print("*update of element " .. tostring(k) ..
              " to " .. tostring(v))
      t[index][k] = v -- о б н о в л я е т  и с х о д н у ю  т а б л и ц у
end,
__pairs = function (t)
     return function (t, k) return next(t[index], k)end, t
end
}
function track (t)
  local proxy = {}
  proxy[index] = t
  setmetatable(proxy, mt)
  return proxy
end
```

Теперь, всякий раз, когда нам потребуется отслеживать таблицу t, все, что нам нужно будет сделать, — выполнить `t=track(t)`.

Таблицы, доступные только для чтения

Можно легко адаптировать идею посредников, чтобы реализовать таблицы, доступные только для чтения (`read-only`). Все, что нам нужно, — это вызывать ошибку каждый раз, когда мы ловим попытку обновить таблицу. Для метаметода `__index` мы можем использовать саму исходную таблицу вместо функции, так как нам не нужно отслеживать запросы; проще и эффективнее перенаправлять такие запросы сразу к исходной таблице. Однако,

это потребует новой метатаблицы для каждого доступного только для чтения посредника, с полем __index, указывающим на исходную таблицу:

```
function readOnly (t)
  local proxy = {}
  local mt = { -- с о з д а е т  м е т а т а б л и ц у
      __index = t,
      __newindex = function (t, k, v) error("attempt to update a read-only table", 2)
                   end
        }
   setmetatable(proxy, mt)
   return proxy
end
```

В качестве примера использования таких таблиц мы можем создать таблицу названий дней недели:

```
days = readOnly{"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"}
print(days[1]) --> Sunday
days[2] = "Noday"
stdin:1: attempt to update a read-only table
```

Слабые таблицы

Слабые таблицы используется, чтобы указать **Lua** на то, что ссылка из таблицы не должна препятствовать уничтожению объекта. *Слабая ссылка* (weak reference) — это такая ссылка на *объект* (ссылочное данное), которая не учитывается сборщиком мусора. *Если все ссылки, указывающие на объект, являются слабыми, то данный объект утилизируется мусорщиком, а эти слабые ссылки удаляются. Объектами*/данными, доступными по *ссылкам* являются: ***table***, ***function***, ***userdata*** *и* ***thread***.

В *слабой* таблице и ключи, и значения могут быть *слабыми*. Это значит, что существуют *три* вида слабых таблиц: таблицы со *слабыми ключами*, таблицы со *слабыми значениями* и *полностью слабые* таблицы, где и ключи, и значения (поля) являются слабыми. Независимо от вида таблицы, при уничтожении ключа или значения, из нее *удаляется* вся запись. *Слабость* таблицы задается полем **_mode** ее метатаблицы. Значение этого поля, когда оно присутствует, должно быть строкой; если эта строка равна "**k**", то ключи в этой таблице являются слабыми; если эта строка равна "**v**", то слабыми являются значения в этой таблице; если эта строка равна "**kv**", то и ключи, и значения в данной таблице являются слабыми.

У таблиц есть *ключи* и поля (значения), которые при этом могут содержать любые виды объектов. Сборщик мусора не утилизирует объекты, которые являются ключами и ссылками, используемыми вне слабых таблиц. Такие ключи и значения предотвращают утилизацию тех объектов, на которые указывают.

Числа, строки и логические значения в качестве *слабых* ключей, сборщиком мусора *не собираются*. Например, если мы вставим числовой ключ в таблицу со слабыми ключами, то сборщик мусора никогда его не удалит. Однако, если значение-ссылка на «слабый» объект (не имеющий сильных ссылок на него), соответствующее числовому ключу, хранится в таблице со слабыми значениями, то с удалением такого объекта, мусорщиком, из нее удаляется *вся* соответствующая запись.

Слабые таблицы Lua 5.3 (дополнение) (https://antirek.github.io/luabook/basicConcepts.htm)

Слабые таблицы (weak table) - это таблицы, чьи элементы являются *слабыми ссылками*. Слабая ссылка игнорируется сборщиком мусора. Иначе говоря, если ссылки на объект являются исключительно слабыми ссылками, то сборщик мусора приберет этот объект.

Слабая таблица может иметь слабые ключи, слабые значения или и то, и другое вместе.

Таблица со слабыми значениями позволяет собирать свои значения, но препятствует сбору своих ключей.

Таблица со слабыми и ключами и значениями позволяет собирать как ключи, так и значения. Для такой таблицы, в любом случае, если собраны либо ключи, либо значения, *пара целиком* удаляется из таблицы.

Таблица со слабыми ключами и сильными значениями также называется *эфемерной таблицей*. В эфемерной таблице значение считается доступным, только если доступен его ключ. В частности, если ссылка на ключ *приходит* только через его значение, то пара удаляется.

Любое изменение "слабости" таблицы может вступить в силу только при *следующем* цикле сборки мусора. В особенности, если изменять слабость к *более* сильному режиму, Lua может по-прежнему получать некоторые элементы из этой таблицы, прежде чем изменения вступять в силу.

Из слабых таблиц удаляются только объекты, которые имеют *явную* конструкцию. Ключи, подобные числам, булевым и легким **C**-ишным функциям, не подлежат сборке мусора и поэтому не удаляются из слабых таблиц (пока значения, связанные с ними, не собраны). Несмотря на то, что строки подлежат сборке мусора, они не имеют явной конструкции и, следовательно, не удаляются из слабых таблиц.

Возрожденные объекты (то есть, финализированные объекты и объекты, доступные только через финализированные объекты) имеют особое поведение в слабых таблицах. Они удаляются из слабых значений *перед* запуском их финализаторов, но удаляются из слабых ключей только при последующей сборке мусора, после запуска их финализаторов, когда подобные объекты фактически высвобождены. Такое поведение позволяет финализатору получить доступ к свойствам (рассмотренным ранее), связанными с объектом через слабые таблицы.

Если в цикле сборки мусора слабая таблица появляется среди возрожденных объектов, она не будет вычищена до следующего цикла сбора мусора.

Примеры использования слабых таблиц

---

1. Слабые таблицы могут быть использованы для кеширования данных (при отсутствии которых, они восстанавливаются) с целью обеспечения высокой производительности программы. Объекты, на которые из слабых таблиц есть только слабые ссылки «вытесняются» из таблиц при работе мусорщика автоматически вместе с ссылочными записями.

---

2. Важной областью применения слабых таблиц является связывание атрибутов с объектами. Существует множество ситуаций, в которых нужно прикрепить некоторый атрибут к объекту: *имена* к функциям, *значения по умолчанию* к таблицам, *размеры* к массивам и т.д. внешняя таблица предоставляет хороший способ присоединения атрибутов к объектам. Мы используем объекты как ключи, а их атрибуты — как значения. Обычная таблица может хранить атрибуты объектов любого типа, так как Lua позволяет использовать объекты любого типа в качестве ключей таблицы. Более того, атрибуты, хранящиеся во внешней таблице, не влияют на другие объекты и могут быть закрытыми, так же как и сама таблица. Однако, это на первый взгляд это решение обладает недостатком: как только использовали объект в качестве ключа в таблице, он обречен на вечное существование. **Lua** не может утилизировать объект, который используется в качестве ключа. Если используется обычная таблица, чтобы привязать к функциям их имена, то ни одна из этих функций никогда не будет удалена. Избежать этого недостатка можно при помощи слабых таблиц. Однако, для этого понадобятся слабые ключи. Применение слабых ключей не предотвращает их утилизацию, когда на них не остается больше ссылок. С другой сто-

роны, у такой таблицы не могут быть слабые значения; иначе атрибуты существующих объектов могли бы быть утилизированы.

---

3. Реализация в таблицах значений по умолчанию может быть выполнена с использованием слабых таблиц.

Вариант1. Используется слабая таблица, чтобы связать с каждой таблицей ее значения по умолчанию:

```
local defaults = {}
setmetatable(defaults, {__mode = "k"})
local mt = {__index = function (t) return defaults[t] end}
--- Функция задания значений таблицы t по умолчанию ---
function setDefault (t, d)
  defaults[t] = d
  setmetatable(t, mt)
end
```

Если бы у таблицы **defaults** не было слабых ключей, то все эти таблицы со значениями по умолчанию существовали бы постоянно.

Вариант2. Используются разные метатаблицы для разных значений по умолчанию, но при этом мы многократно используем одну и ту же метатаблицу при каждом повторном использовании значения по умолчанию. Это типичное применение запоминания (кеширование):

```
local metas = {}
setmetatable(metas, {__mode = "v"})
function setDefault (t, d)
  local mt = metas[d]
  if mt == nil then
    mt = {__index = function () return d end}
    metas[d] = mt -- memorize
  end
  setmetatable(t, mt)
end
```

В данном случае мы применяем слабые значения, чтобы разрешить утилизацию уже неиспользуемых метатаблиц.

Какая из этих двух реализаций является лучшей? Как обычно, это зависит от обстоятельств. У обеих вариантов схожая сложность и схожее быстродействие. Первый вариант требует нескольких слов (1 слово = 2 байта) памяти для *каждой* таблицы со значением по умолчанию (для записей в defaults). Второй вариант требует нескольких десятков слов памяти для каждого *отдельного* значения по умолчанию (новая таблица, новое замыкание и запись в metas). Поэтому если в приложении тысячи таблиц всего несколькими различными значениями по умолчанию, то второе решение явно будет лучше. Если же у таблиц много различных значений по умолчанию, то лучше предпочесть первую реализацию.

Финализаторы

Финализатор (finalizer) — это функция, связанная с таблицей, которая вызывается перед тем, как таблица (объект) будет удалена сборщиком мусора. Финализаторы используются для освобождения (закрытия) программных средств, *которые не удаляются мусорщиком* (например, открытые файлы). Конечно, это можно делать вручную, но, финализаторы обеспечивают надежность выполнения таких действий.

Lua реализует финализаторы при помощи метаметода __gc.

Пример:

```
o = {x = "hi"}
setmetatable(o, {__gc = function (o) print(o.x) end})
```

```
o = nil
collectgarbage() --> hi
```
В этом примере мы сперва создаем таблицу и устанавливаем для нее метатаблицу, у которой есть метаметод __gc. Затем мы уничтожаем единственную ссылку на эту таблицу (глобальная переменная o) и запускаем полную сборку мусора при помощи вызова **collectgarbage**. Во время сборки мусора **Lua** обнаруживает, что данная таблица не является доступной и вызывает ее финализатор (метаметод __gc).

У финализаторов Lua есть *нюанс*, связанный с *пометкой* объекта для финализации. Мы помечаем объект для финализации, когда *задаем* (подключаем) для него метатаблицу с *ненулевым* метаметодом **__gc**. Если мы не пометим объект, то он не будет финализирован.

Например:
```
o = {x = "hi"}
mt = {}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage() --> (ничего не печатает)
```
В этом примере метатаблица, которую мы устанавливаем для **o**, не содержит метаметода __gc, поэтому объект и не помечается для финализации. Даже, если потом добавить поле __gc метатаблице, **Lua** *не посчитает* это присваивание пометкой.

Если вы хотите задать метаметод позже, то вы можете использовать любое значение для поля __gc в качестве временного:
```
o = {x = "hi"}
mt = {__gc = true}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage() --> hi
```
Теперь, поскольку метатаблица содержит поле __gc, объект **o** надлежащим образом помечается для финализации. Нет никакой проблемы в том, чтобы задать метаметод позже; **Lua** вызывает финализатор, только если он является *функцией*.

Когда сборщик мусора утилизирует *несколько* объектов в одном и том же цикле, он вызывает их финализаторы в порядке, *обратном* тому, в котором объекты были *помечены* для финализации. *Ссылки* между *утилизируемыми* объектами *не влияют на порядок* их финализации.

Особенностью финализации таблиц, является их *восстановление* (resurrection) для выполнения *финализации*. При своем вызове финализатор получает в качестве параметра финализируемый объект. Таким образом, объект снова становится живым (со всем тем, что достижимо из него), по крайней мере, на время финализации. Это *временное восстановление* (transient resurrection). Во время выполнения финализатора ничто не мешает ему сохранить объект, скажем, в глобальной переменной, чтобы объект (со всем тем, что достижимо из него) остался доступным после возврата из финализатора. Это *перманентное восстановление* (permanent resurrection).

Рассмотрим следующий фрагмент кода:
```
A = {x = "this is A"}
B = {f = A}
setmetatable(B, {__gc = function (o) print(o.f.x) end})
A, B = nil
collectgarbage() --> this is A
```
Финализатор для **B** обращается к **A**, поэтому **A** не может быть удален до финализации **B**. Lua должен восстановить и **A**, и **B** перед вызовом финализатора.

Из-за восстановления объекты с финализаторами собираются в два этапа. Вначале, когда сборщик мусора обнаруживает, что объект с финализатором недостижим, он восстанавливает этот объект и добавляет его к очереди финализации. После выполнения финализатора **Lua** помечает объект как финализированный. В следующий раз, когда сборщик мусора обнаружит, что объект недостижим, он его уничтожит.

Если вы хотите быть уверенными в том, что весь мусор в вашей программе был действительно собран, то вы должны вызвать **collectgarbage** дважды; второй вызов уничтожит объекты, которые были финализированы во время первого вызова.

Финализатор для каждого объекта выполняется *один* раз, поскольку **Lua** ставит пометку на финализированные объекты. Если объект не был собран сборщиком до конца работы программы, то **Lua** вызовет его финализатор при закрытии всего состояния **Lua**. Эта последняя особенность позволяет реализовать в **Lua** аналог функций **atexit**, то есть функций, которые вызываются непосредственно перед *завершением* программы. Все, что для этого нужно, — создать таблицу с финализатором и закрепить ее где-нибудь, например в глобальной переменной:

```
_G.AA = {__gc = function ()
-- поместите здесь ваш 'atexit'-код
print("finishing Lua program")
end}
setmetatable(_G.AA, _G.AA)
```

Другой интересный подход позволяет вызывать заданную функцию каждый раз, когда **Lua** завершает цикл сборки мусора. Поскольку финализатор выполняется ровно *один* раз, то хитрость здесь в том, чтобы финализатор создавал новый *объект* для вызова следующего финализатора:

```
do
local mt = {__gc = function (o)
-- делайте все, что хотите
print("new cycle")
-- создает новый объект для следующего цикла
setmetatable({}, getmetatable(o))
end}
-- создает первый объект
setmetatable({}, mt)
end
collectgarbage() --> новый цикл
collectgarbage() --> новый цикл
collectgarbage() --> новый цикл
```

Во взаимодействии объектов с финализаторами и слабыми таблицами тоже есть нюанс. Сборщик мусора *очищает значения* (если есть для этого условия) в слабой таблице *перед* восстановлением, в то время как ключи очищаются *после восстановления*. Следующий фрагмент кода иллюстрирует это поведение:

```
-- таблица со слабыми ключами
wk = setmetatable({}, {__mode = "k"})
-- таблица со слабыми значениями
wv = setmetatable({}, {__mode = "v"})
o = {} -- объект
wv[1] = o; -- wk[o] = 10 -- добавляем его в обе таблицы
setmetatable(o, {__gc = function (o)    -- o помечен для финализации ---
print(wk[o], wv[1])
end})
o = nil; collectgarbage() --> 10  nil
```

Во время выполнения финализатора, он находит объект в таблице **wk**, но не в таблице **wv**. Для ключей так реализовано потому, что предусмотрено хранение свойств объектов в таблицах со слабыми ключами (как это было рассмотрено ранее), и финализаторам может понадобиться доступ к этим атрибутам, являющимися частью объектов.

ООП в Lua (кратко)

Скрипты в **Lua** можно писать в стиле **ООП**. Такая возможность обеспечивается устройством таблиц **Lua**. *Абстракцию, локализацию-инкапсуляцию*, *полиморфизм* и *наследование*, основные принципы **ООП**- все это можно реализовать, используя фактически единственный тип **Lua** – таблицу с возможностью подключения метатаблицы (в том числе, к

самой себе), а также то, что *функции-замыкания* **Lua** - данные первого класса, допускающие переменное количество параметров динамического типа. *Существенно то*, что в **Lua** реализованы функции-замыкания, так как это обеспечивает создание экземпляров функций, с порождением индивидуального окружения функций, недоступного из вне и хранящего их состояния между их вызовами. Это позволяет реализовать, при необходимости, сокрытие данных.

Стандартные библиотеки **Lua**, например, **string**, **table** и другие - это объекты-библиотеки, представленные служебными таблицами **Lua** с методами/функциями, внутри таких таблиц.

Таблица в **Lua** - это объект во многих отношениях. У таблиц есть состояние, определяемое данными ее элементов. У таблиц есть идентичность (собственное «я» — ***self***), на которую можно ссылаться из ее <Табличных функций>, как это было описано в разделе «Функции Lua».

В **Lua** нет синтаксического понятия класса. Однако, для представления классов можно создать объекты (таблицы), которые будут использованы только в качестве прототипа (места хранения поведения, общего для различных объектов) для других объектов (его экземпляров). Например, если у нас есть два объекта **a** и **b**, то все, что нам нужно сделать, чтобы **b** стал прототипом для **a** – это следующее: **setmetatable**(**a**, {__index = **b**}). После этого **a** будет искать в **b** любое действие, которого у нет.

Во многих случаях программирования на **Lua** в стиле **ООП**, наверное, достаточно использования таблиц как объектов с локализацией методов-функций внутри них. Сами функции могут быть определены вне таблиц и быть *общими* для множества экземпляров объектов.

Более сложные варианты реализации основных принципов **ООП,** с использованием средств языка **Lua**, представленных ранее в данной справке, подробно описаны в книге «Программирование на языке Lua» Р. Иерузалимски в главе 16.

Отладка скриптов Lua

Для отладки скриптов **Lua** можно использовать отладочные функции стандартного пакета **debug.** Кроме того, в разделе «Приложения 1» приведен код функции печати значений **Lua** любого ее типа, в том числе произвольных ее таблиц с учетом их вложенностей и их метатаблиц (если они есть).

Отладочная библиотека **debug** состоит из двух видов функций: интроспективные функции и ловушки. *Интроспективные функции* (introspective function) позволяют изучать различные стороны выполняемой программы, такие как стек ее активных функций, текущая выполняемая строка, значения и имена локальных переменных. *Ловушки* (hook) позволяют нам отслеживать события при выполнении программы.

Важным понятием в отладочной библиотеке является стековый уровень. *Стековый уровень* (stack level) — это число, которое относится к конкретной функции, активной в данный момент: у функции, вызвавшей отладочную библиотеку, уровень 1, у функции, которая вызвала эту функцию, уровень 2 и т. д.

Интроспективные функции

Главной интроспективной функцией в отладочной библиотеке является **debug.getinfo**. Ее первый параметр может быть функцией или стековым уровнем. При вызове **debug.getinfo(foo)** для какой-то функции **foo** вы получите таблицу с некоторыми данными об этой функции. Эта таблица может иметь следующие поля:

source: где была определена функция. Если эта функция была определена в строке (посредством *load*), то значением *source* будет эта строка. Если функция была определена в файле, то значение *source* — имя этого файла с префиксом '@';

short_src: короткая версия source (до 60 символов), полезна для сообщений об ошибках;

linedefined: номер первой строки в source, где функция была определена;

lastlinedefined: номер последней строки в source, где функция былаопределена;

what: что это за функция. Возможные значения: "Lua", если это обычная функция Lua, "C", если это функция C, или "main", если это главная часть куска Lua;

name: подходящее для функции имя;
namewhat: что означает предыдущее поле. Возможные значения: "global", "local", "method", "field" и "" (пустая строка). Пустая строка означает, что Lua не нашел имени для функции;
nups: количество верхних значений для этой функции;
activelines: таблица, представляющая множество активных строк функции. Активная строка (active line) — это строка с каким-то кодом, в отличие от пустых строк и строк, состоящих только изкомментариев. (Типичное использование данной информации — это установка точек прерывания (breakpoint). Большинство отладчиков не позволяет задавать точки прерывания не на активных строках, так как они были бы недостижимы.)
func: сама функция; об этом позже.
Когда **foo** является функцией **C**, у **Lua** о ней почти нет никаких данных. Для таких функций значимы лишь поля *what*, *name* и *namewhat*. Когда вы вызываете **debug.getinfo(n)** для какого-то числа **n**, вы получаете данные о функции, активной на этом уровне стека. Например, если **n** равно 1, то вы получаете данные о функции, совершающей вызов. (Когда n равно 0, вы получаете данные о самой функции **getinfo**, т.е. функцию C.) Если **n** больше числа активных функций в стеке, то **debug.getinfo** возвращает **nil**. Когда вы опрашиваете активную функцию, вызывая **debug.getinfo** с числовым аргументом, у итоговой таблицы будет одно дополнительное поле с именем *currentline*, содержащее номер строки, на которой находится функция в данный момент. Кроме того, *func* содержит функцию, которая активна на этом уровне. Поле *name* непростое. Из-за того, что функции в **Lua** являются значениями первого класса, функция может вообще не иметь имени или иметь несколько имен. **Lua** пытается найти имя функции путем просмотра кода, вызвавшего эту функцию, чтобы увидеть, как он ее вызвал. Этот метод работает лишь при вызове **getinfo** с числовымаргументом, то есть когда мы запрашиваем информацию о конкретном вызове.
Lua содержит отладочную информацию в форме, которая не ухудшает быстродействие программы; эффективный поиск информации здесь вторичен. Чтобы получить большее быстродействие, у **getinfo** есть необязательный второй параметр, который ограничивает круг поиска необходимой информации. Таким образом, данная функция не тратит лишнее время на сбор лишней информации. Формат данного параметра является строкой, где каждая буква служит для выбора группы полей согласно следующей таблице:
'n' - name, namewhat
'f' - func
'S' - source, short_src, what, linedefined, lastlinedefined
'l' - currentline
'L' - activelines
'u' - nup
Следующая функция иллюстрирует использование **debug.getinfo**. Она распечатывает примитивную обратную трассировку активного стека:

```
function traceback ()
  for level = 1, math.huge do
    local info = debug.getinfo(level, "Sl")
    if not info then break end
    if info.what == "C" then -- функция C?
       print(level, "C function")
    else -- функция Lua
      print(string.format("[%s]:%d", info.short_src, info.currentline))
    end
  end
end
```

Эту функцию легко можно улучшить, добавив больше данных из **getinfo**. В действительности в отладочной библиотеке уже есть ее улучшенная версия — функция **traceback**. В отличие от нашей функции, **debug.traceback** не печатает свой результат; вместо этого она возвращает строку с обратной трассировкой.

Доступ к локальным переменным

Мы можем изучать локальные переменные любой активной функции при помощи функции **debug.getlocal**. У этой функции два параметра: стековый уровень опрашиваемой функции и индекс переменной. Она возвращает два значения: имя переменной и ее текущее значение. Если индекс переменной больше числа активных переменных, то **getlocal** возвращает **nil**. Если указан недопустимый стековый уровень, то **getlocal** вызывает ошибку. (Для проверки допустимости уровня стека мы можем воспользоваться **debug.getinfo**.)

Lua нумерует локальные переменные в порядке их появления внутри функции, считая лишь переменные, которые являются активными в текущей области видимости функции. Например, рассмотрим следующую функцию:

```
function foo (a, b)
  local x
  do local c = a - b end
  local a = 1
  while true do
   local name, value = debug.getlocal(1, a)
   if not name then break end
   print(name, value)
   a = a + 1
  end
end
```

Вызов foo(10,20) напечатает следующее:

```
a 10
b 20
x nil
a 4
```

Переменная с индексом 1 — это **a** (первый параметр), с индексом 2 — это **b**, 3 — это **x**, 4 — это другая **a**. В момент вызова **getlocal** переменная **c** уже вышла из области видимости, в то время как **name** и **value** еще в нее не вошли.

Начиная с **Lua 5.2**, отрицательные индексы возвращают информацию о дополнительных аргументах функции: индекс -1 соответствует первому дополнительному аргументу. В этом случае именем переменной всегда будет "(*vararg)".

Вы также можете изменять значения локальных переменных при помощи функции **debug.setlocal**. Ее первые два параметра — это уровень в стеке и индекс переменной, как и в **getlocal**. Ее третий параметр — это новое значение для переменной. Функция возвращает имя переменной или **nil**, если индекс переменной вне области видимости.

Доступ к нелокальным переменным

Отладочная библиотека также позволяет обращаться к нелокальным переменным, используемым функцией **Lua**, при помощи **getupvalue**. В отличие от локальных переменных, нелокальные переменные, используемые функцией, существуют, даже когда функция не активна (в конце концов, в этом суть замыканий). Поэтому первый аргумент для **getupvalue** — это не уровень в стеке, а функция (точнее, замыкание). Второй аргумент — это индекс переменной. **Lua** нумерует нелокальные переменные в том порядке, в котором они впервые встречаются в функции, но этот порядок не важен, поскольку функция не может обратиться сразу к двум нелокальным переменным с одним и тем же именем.
Вы также можете обновлять нелокальные переменные при помощи **debug.setupvalue**. У нее три параметра: замыкание, имя переменной и новое значение. Как и **setlocal**, она возвращает имя переменной или **nil**, если индекс переменной вне допустимого диапазона.

Функция приведенная ниже, показывает, как мы можем получить доступ к значению любой из этих переменных по ее имени. Параметр **level** сообщает, где именно эта функция должна искать; увеличение на единицу нужно, чтобы не включать вызов к самой функции **getvarvalue**. Функция **getvarvalue** сначала проверяет локальную переменную. Если переменных с заданным именем несколько, то она использует переменную с наибольшим

индексом; таким образом, она всегда должна пройти весь цикл. Если функция не может найти ни одной переменной с таким именем, то она проверяет нелокальные переменные. Для этого при помощи **debug.getinfo** она получает вызывающее замыкание, а затем перебирает все его нелокальные переменные. Наконец, если функции не удается найти нелокальную переменную с заданным именем, то она переходит к глобальной переменной: **getvarvalue** вызывает себя рекурсивно для доступа к подходящей переменной **_ENV**, а затем ищет заданное имя в этом окружении.

Получение значения переменной:

```
function getvarvalue (name, level)
  local value
  local found = false
  level = (level or 1) + 1
  -- пробует локальные переменные
  for i = 1, math.huge do
     local n, v = debug.getlocal(level, i)
     if not n then break end
     if n == name then
        value = v
        found = true
     end
   end
   if found then return value end
   -- пробует нелокальные переменные
   local func = debug.getinfo(level, "f").func
   for i = 1, math.huge do
     local n, v = debug.getupvalue(func, i)
     if not n then break end
     if n == name then return v end
   end
   -- не найдено; получает значение из окружения
   local env = getvarvalue("_ENV", level)
   return env[name]
end
```

Доступ к сопрограммам

Все интроспективные функции из отладочной библиотеки могут принимать в качестве первого аргумента сопрограмму, чтобы мы могли изучить сопрограмму извне. Рассмотрим следующий пример:

```
co = coroutine.create(function ()
  local x = 10
  coroutine.yield()
  error("some error")
end)
coroutine.resume(co)
print(debug.traceback(co))
```

Вызов **traceback** обработает сопрограмму co, и результат будет примерно таким:

```
stack traceback:
[C]: in function 'yield'
temp:3: in function <temp:1>
```

Данная трассировка не затрагивает вызов **resume**, поскольку эта сопрограмма и главная программа выполняются в разных стеках. Когда сопрограмма вызывает ошибку, она не раскручивает стек. Это значит, что после ошибки мы можем его изучить. В продолжение нашего примера, если мы вновь возобновим сопрограмму, это приведет кошибке:

```
print(coroutine.resume(co)) --> false temp:4: some error
```

Теперь если мы распечатаем трассировку стека, то получим что-то вроде:

```
stack traceback:
[C]: in function 'error'
```

temp:4: in function <temp:1>
При этом мы можем изучать локальные переменные из сопрограммы даже после ошибки:
print(debug.getlocal(co, 1, 1)) --> x 10

Ловушки

Механизм *ловушек* (hook) из отладочной библиотеки позволяет зарегистрировать функцию, которая будет вызвана при наступлении определенных событий во время выполнения программы. Существует четыре вида событий, которые могут заставить сработать ловушки:
call (событие вызова) происходит каждый раз, когда **Lua** вызывает функцию;
return (событие возврата) происходит каждый раз при возврате из функции;
line (событие строки) происходит, когда **Lua** начинает выполнение следующей строки кода;
count (событие счетчика) происходит после заданного количества команд.

Lua вызывает ловушки с единственным аргументом — строкой, описывающей событие, которое привело в вызову: "call" (или "tail call"), "return", "line" или "count". Для события строки также передается второй аргумент — новый номер строки. Для получения дополнительной информации внутри ловушки следует использовать **debug.getinfo**.

Чтобы зарегистрировать ловушку, мы вызываем функцию **debug.sethook** с двумя или тремя аргументами: первый аргумент — это функция ловушки; второй аргумент — это фильтрующая строка, которая описывает, какие именно события мы хотим отслеживать; и необязательный третий аргумент — это число, задающее, с какойчастотой мы хотим получать события счетчика. Чтобы отслеживать события вызова, возврата и строки, мы добавляем их первые буквы ('c','r' или 'l') к фильтрующей строке. Для отслеживания событий счетчика мы просто передаем счетчик как третий аргумент. Для отключения всех ловушек нужно вызвать **sethook** без аргументов.

В качестве простого примера следующий код устанавливает примитивный трассировщик, который печатает каждую строку кода, выполняемую интерпретатором:

```
debug.sethook(print, "l")
```

Этот вызов устанавливает **print** как функцию ловушки и приказывает **Lua** вызывать ее только при событиях строки. Более проработанный трассировщик может использовать **getinfo**, чтобы добавить к трассировке имя текущего файла:

```
function trace (event, line)
  local s = debug.getinfo(2).short_src
  print(s .. ":" .. line)
end
debug.sethook(trace, "l")
```

Для ловушек удобна функция **debug.debug**. Эта простая функция печатает приглашение ввода, которое выполняет любые команды **Lua**.
Она примерно эквивалентна следующему коду:

```
function debug1 ()
  while true do
    io.write("debug> ")
    local line = io.read()
    if line == "cont" then break end
    assert(load(line))()
  end
end
```

Когда пользователь вводит «команду» *cont*, эта функция завершается. Стандартная реализация очень проста и выполняет команды в глобальном окружении, вне области видимости отлаживаемого кода.

Стандартные библиотеки Lua

В **Lua** существую стандартные библиотеки и прежде чем разрабатывать свою функцию, полезно (*необходимо*), посмотреть, нет ли аналогичной в стандартных библиотеках **Lua** (или пакетах **Lua**, «гуляющих» в интернете).

Все стандартные функции в ***Lua*** можно разбить на следующие несколько групп - основные (*basic*), работа с модулями (*package*), работа со строками, работа с таблицами, математические функции, ввод/вывод, доступ к ОС, средства отладки. Для ряда групп соответствующие функции собраны в таблицы, соответствующие группам, например *string*, *math* и т.п.

Ниже приводятся основные функции для этих групп.

Таблица 1. Основные функции.

Название	Комментарий
assert (v **[**, message**]**)	Вызывает ошибки, когда значение *v* ложно. При этом печатает сообщение *message* или "assertion failed!", если параметр *message* не задан
collectgarbage (opt **[**, arg**]**)	Интерфейс к сборщику мусора. Действия определяются переданным первым параметром. "stop": остановка сборки мусора. "restart": снова запустить сборщик мусора. "collect": осуществить полный цикл сборки мусора. "count": возвращает полный объем используемой ***Lua*** памяти в Кбайтах. "step": выполняет один шаг сборки мусора. Параметр *size* управляет величиной шага. "setpause": устанавливает *arg/100* в качестве нового значения для остановки сборщика. "setstepmul": Устанавливает *arg/100* как новое значение для умножителя шага в сборщике (подробнее в документации §2.10).
dofile (filename)	Открывает и выполняет файл с заданным именем.
getmetatable (object)	Возвращает метатаблицу заданного объекта.
setmetatable (object, mt)	Устанавливает у объекта метатаблицу. Возвращает сам объект.
ipairs (t)	Возвращает итератор для обхода таблицы по целочисленным индексам. Останавливается на первом отсутствующем индексе
pairs (t)	Возвращает итератор для полного обхода таблицы
print (⋯)	Печать всех переданных аргументов при помощи функции *tostring*.
rawequal (v1, v2)	Возвращает равны ли переданные объекты не используя метаметоды.

rawget (table, index)	Чтение из таблицы без использования метаметодов.
rawset (table, index, value)	Запись значения в таблицу без использования метаметодов.
tonumber (e **[**, base**]**)	Перевод величины в число. Параметр *base* является основанием системы счисления от 2 до 36 включительно.
tostring (e)	Перевод величины в строку. Использует метаметод *__tostring*.
type (v)	Возвращает тип параметра как строку.

Таблица 2. Работа с модулями.

Название	Комментарий
module (name **[**, ⋯**]**)	Создает модуль с заданным именем. Если уже есть таблицы в *package.loaded[name]* то она и является модулем. Если есть глобальная таблица с именем *name*, то она становится модулем. В противном случае создается новая таблица с именем *name*, она заносится в *package.loaded[name]*. В этой таблице в поле *_NAME* записывается имя модуля, в поле *_M* записывается ссылка на себя, а в поле *_PACKAGE* - имя пакета. После этого эта таблица становится таблицей окружения для текущей функции.
require (modname)	Загружает заданный модуль. Если модуль уже загружен, то просто возвращает ссылку на него.
package.seeall(module)	Устанавливает метатаблицу для модуля с полем *__index* указывающим на глобальное окружение. Функция предназначена для передачи в качестве дополнительного параметра в функцию *module*

Все функции для работы со строками находятся в таблице *string*. Индексирование символов начинается с единицы. Допустимы отрицательные индексы (-1 указывает на последний символ, -2 - на предпоследний и т.д.).

Таблица 3. Работа со строками.

Название	Комментарий
string.byte (s **[**, i **[**, j**]]**)	Возвращает массив чисел, соответствующих кодам байт из строки - *s[i],s[i+1],...,s[j]*. Если параметр *i* не задан, то он считается равным 1, если параметр *j* не задан, то он считается равным *i*
string.char (⋯)	Возвращает строку, построенную из байт, числовые значения которых были переданы в качестве параметров, так *string.char(97,98,99)* вернет строку 'abc'

string.dump (function)	Возвращает строку, содержащую бинарное представление переданной функции.
string.find (s, pattern **[**, pos **[**, plain**]]**)	Поиск вхождения шаблона в строку.При успехе возвращает индексы начала и конца первого вхождения. Параметр *pos* задает начальную позицию для поиска, по умолчанию равную 1. Если параметр *plain* равен 1, тогда осуществляется просто поиск вхождения подстроки, без использования шаблонов.
string.format (formatstring, ⋯)	Практически полный аналог *printf*, за исключением поддержки **,l,L,n,p,h*. Кроме того появилась дополнительная опция *q*, позволяющая форматировать строки в безопасном для понимания интерпретатором ***Lua*** виде.

string.gmatch (s, pattern)	Возвращает итератор по вхождениям шаблона в строку.
	
	Так следующий пример напечатает все слова из строки
	
	for w in string.gmatch (s, '%a+') do
	** print (w)**
	end
	
	Следующий пример разбирает строку, содержащую набор присваиваний вида *key=value* и заносит их в таблицу.
	
	t = {}
	for k,w in string.gmatch (s, '(%w+)=(%w+)') do
	** t [k] = w**
	end

string.gsub (s, pattern, repl **[**, n**]**)	Возвращает копию строки *s* в которой все вхождения шаблона *pattern* (или же только первые *n* вхождений) были заменены при помощи параметра *repl*. Параметр *repl* может быть как строкой, так и таблицей или функцией. В случае если это строка, то происходит просто замена вхождения шаблона на эту строку. Если это функция, то происходит вывзов функции с передачей ей в качестве параметра найденного вхождения и возвращаемое ей значения используется для замены в строке. Если это таблица, на найденная строка используется для индексирования в таблицу и полученное при этом значение из таблицы испольуется для замены. Если ни одного вхождения шаблона не было най-

	дено, то тогда вя исходная строка используется при обращении к функции или таблице.
string.len (s)	Возвращает длину строки, с учетом нулевых байт.
string.lower (s)	Возвращает копию строкит, в которой все бкувы были заменены на *lower-case*.
string.upper (s)	Возвращает копию строкит, в которой все бкувы были заменены на *upper-case*.
string.match (s, pattern **[**, pos**]**)	Возвращает первое найденнео вхождение шаблона в строку или **nil**
string.rep (s, n)	Возвращает результат конкатенации *n* копий исходной строки, так string.rep('*', 3) будет '***'.
string.reverse (s)	Возвращает строку, полученную обратной перестановкой байт в исходной строке, так строка 'abcd' переходит в 'dcba'.
string.sub (s, i **[**, j**]**)	Возвращает подстроку *s* с позиции *i* по позицию *j* включительно. По умолчанию *j* равно -1.

Функции для работы с таблицами собраны с таблицу *table*. Большиство из них предназначены для работы с массивами.

Таблица 4. Функции для работы с таблицами.

Название	Комментарий
table.concat (t **[**, sep **[**, i **[**, j**]]]**)	Возвращает *t[i]..t[i+1].. ... ..t[j]*
table.insert (table, **[**pos,**]** value)	Вставляет новое значение в заданное место в массиве со сдвигом эелемента массива. Если параметр *pos* не задан, то происходит просто добавление элемента к концу массива.
table.maxn (table)	Возвращает наибольшийй положительный индекс для которого в массиве есть значение. Если таких индексов нет, то возвращается 0.
table.remove (table **[**, pos**]**)	Удаление элемента в заданном месте со сдвигом остальных эелеменетов. Если параметр *pos* не задан, то удаляется последний элемент таблицы. Функция возвращает удаленное занчение.
table.sort (table **[**, comp**]**)	Сортировка таблицы. Функция сравнения получает на вход два сравниваеммых значение и возвращает **true>**, если первый объект меньше второго.

Таблица 5. Математические функции.

Название	Комментарий
math.abs (x)	Модуль числа
math.acos (x)	Аркосинус (в радианах).
math.asin (x)	Синус угла (в радианах).
math.atan (x)	Арктангенс
math.atan2 (y, x)	
math.ceil (x)	Ниаменьшее целое, большее или равное *x*.
math.cos (x)	Косинус угла
math.cosh (x)	Гиперболический косинус.
math.deg (x)	Перевод угла из радиан в градусы.

math.exp (x)	Фозвращает *e^x*.
math.floor (x)	Наибольшее целое, меньшее или равное *x*.
math.fmod (x, y)	Остаток от деления
math.log (x)	Натуральный логарифм
math.log10 (x)	Десятичный логарифм.
math.max (x, ⋯)	Максимальное занчение из переданных чисел.
math.min (x, ⋯)	Минимальное значение из переданных чисел.
math.pow (x, y)	Возведение *x* в степень *y*.
math.rad (x)	Перевод угла из градусов в радианы.
math.random ([m [, n]])	Случайное число. Если параметры не заданы, то возвращается случайное число в *[0,1]*.. Иначе возвращается целое число либо из *[1,m]*, либо из *[m,n]*
math.randomseed (x)	Инициаолиция генерратора псевдослучайных чисел.
math.sin (x)	Синус угла
math.sinh (x)	Гиперболический синус
math.sqrt (x)	Квадратный корень
math.tan (x)	Тангенс угла
math.tanh (x)	Гиперболический тангенс

Таблица 6. Функции ввода/вывода.

Название	Комментарий
io.close (**[**file**]**)	
io.flush ()	
io.input (**[**file**]**)	
io.lines (**[**filename**]**)	
io.open (filename **[**, mode**]**)	
io.output ([file])	
io.read (⋯)	
io.write (⋯)	
file:close ()	
file:flush ()	
file:lines ()	
file:read (⋯)	
file:seek (**[**whence**] [**, offset**]**)	
file:write (⋯)	

Таблица 7. Функции работы с ОС.

Название	Комментарий
os.clock ()	Возвращает приблизительное время CPU в секундах, потраченнное на выполнение данной прграммы.
os.date (**[**format **[**, time**]]**)	Возвращает строку или таблицу, содержащую заданную дату и время
os.difftime (t2, t1)	Разница в секундах.
os.execute (**[**command**]**)	Выполнить команду, аналог *system*

os.remove (filename)	Удалить файл или каталог (каталог должен быть пуст).
os.rename (oldname, newname)	Переименовать файл или каталог.
os.time ([table])	
os.tmpname ()	Возвращает строку, содержащую имя, пригодное для использования в качестве имени временного файла.

Кроме функций есть также набор глобальных переменных, наиболее важные из которых перечислены в следующей таблице.

Таблица 8. Основные системные перменные.

Название	Комментарий
_G	Глобальная таблица окружение, всегда _G._G==_G
_M	Текущеий модуль
_VERSION	Версия как строка, например "Lua 5.1"
math.pi	Константа pi
math.huge	Наибольшее представимое число
_ENV	Переменная окружения *куска* (по умолчанию _ENV = _G)

Стандартная библиотека отладки в **Lua gebug** была описана ранее.

Здесь будут рассмотрены только те функции стандартных библиотек **Lua**, в использовании которых есть существенные особенности или особенности их выполнения.

Модули и пакеты Lua

Возможность реализации скриптов, с использованием автономно (в отдельных файлах) реализованных функций, обеспечивает эффективность и надежность таких сриптов.

Гибкость использование в **Lua** существующих кодов **Lua** определяется тем, что любую строку в **Lua** можно компилировать и исполнить (после успешной компиляции). Но, кроме того, в **Lua** есть дополнительные средства подключения **Lua**-кодов и внешних **C-dll** к выполняемому скрипту.

Модуль — это некоторый код (*текст/байт-код* на **Lua** или *dll* на **C/C++**), который может быть загружен посредством служебной функцией **require (**создающей и исполняющий модуль**)** и который, при своем исполнении *обычно* возвращает таблицу **Lua** с элементами, используемыми в скрипте. Все, что модуль экспортирует, будь то функции или таблицы, он должен определять внутри этой таблицы, которая выступает в качестве пространства имен. Например, все стандартные библиотеки — это модули.

Если в модуле инициализируются переменные глобального окружения скрипта, то такие переменные *доступны*, после инициализации (если не экранируются локальными переменными), в любом месте скрипта. Так как модуль компилируется в *кусок*, то в нем можно, при необходимости, задать окружение модуля, присвоив, созданную в нем таблицу, его переменнй **_ENV**.

Если модуль выдает результат (возможно несколько значений), то этот результат будет выдан **require** в качестве ее результата.

Для загрузки модуля вызываем: **require** "*имя_модуля*".

Первым шагом **require** является проверка по служебной таблице **package.loaded**, не был ли загружен данный модуль ранее. Если да, **require** возвращает его соответствую-

щее значение. Поэтому, как только модуль загружен, другие вызовы, для которых он требуется, просто вернут то же значение без повтороного выполнения кода модуля. Для удаления модуля выполняется следующее:

1) очищаются (значением **nil**) все переменные, которым были присвоены значения вызова **require***.<имя_модуля>;*

2) ***package.loaded.<имя_модуля> = nil***

require можно использовать в любом месте скрипта (не обязательно в начале).

Под *пакетом* в **Lua** понимается модуль, в котором создается *иерархия* таблиц с элементами, используемыми в скрипте.

Поиск файла модуля

Данные для поиска модулей Lua хранятся в виде строки поиска модулей в **package.path (**для Lua-модулей) и в **package.cpath** (для C-модулей).

Строка поиска **require**, — это список шаблонов, каждый из которых описывает способ преобразования имени модуля (аргумента **require**) в имя файла. Каждый шаблон в пути — это имя файла, содержащее необязательные знаки вопроса. Для каждого шаблона **require** заменяет *каждый* '?' на *имя* модуля и проверяет, есть ли файл с получившимся именем; если нет, она переходит к следующему шаблону. Шаблоны в пути разделены точками с запятой. Например, если путем является
?;?.lua;c:\windows\?;/usr/local/lua/?/?.lua
то вызов require "sql" попробует открыть следующие файлы Lua:
sql
sql.lua
c:\windows\sql
/usr/local/lua/sql/sql.lua

Функция **require** допускает использование лишь точки с запятой (для разделения составляющих) и вопросительного знака; все остальное, включая разделители директорий и расширения файлов, определяется самим путем.

Путь, который **require** использует для поиска файлов **Lua**, — это всегда текущее значение переменной **package.path**. Во время запуска Lua инициализирует эту переменную значением переменной окружения LUA_PATH_5_ …. Если эта переменная окружения не определена, **Lua** пытается использовать переменную окружения LUA_PATH. Если они обе не определены, Lua использует путь по умолчанию, заданный при компиляции (Примечание: В **Lua 5.2** опция командной строки –E предотвращает использование этих переменных окружения и заставляет использовать значение по умолчанию). При использовании значения переменной окружения **Lua** подставляет путь по умолчанию вместо любой подстроки ";;". Например, если вы установите LUA_PATH_5_ …на *"mydir/?.lua;;",* то окончательный путь будет шаблоном "*mydir/?.lua*", за которым следует путь по умолчанию. Путь для поиска библиотек *dll* **C** работает точно так же, но его значение берется из переменной **package.cpath** (вместо **package.path**). Аналогично эта переменная получает свое начальное значение из переменной окружения LUA_CPATH_5_ … или LUA_CPATH. Типичным значением этого пути в Windows будет примерно таким:
.\?.dll;C:\Program Files\Lua504\dll\?.dll

Интерфейс Lua с C/C++ (C API)

В этом разделе будут описаны основные особенности интеграции **Lua** с **C/C++**. Детально **C API** описана в оригинальных документах, например в:
https://lua.org.ru/contents_ru.html .

Разработка dll для Lua (для версий Lua >= 5.3)

В этом разделе будут приведены основные особенности разработкки dll в Windows с помощью VS201…, подключаемых в Lua с помощью **require**.

1. C API **Lua** написано на языке **C** и поэтому его подключение должно быть в стиле **C**:

```
//=== Необходимые для Lua константы
================================================================================//
```

```
#define LUA_LIB
#define LUA_BUILD_AS_DLL

//=== Заголовочные файлы LUA ----
================================================================================
====//
extern "C" {
#include "Lua\lauxlib.h"
#include "Lua\lua.h"
#include "Lua\lualib.h"
}
```

2. Коды подключаемых библиотек должны иметь разрядность (32/64), соответствующую разрядности разрабатываемой dll.
3. Надо учитывать статическое или динамическое подключение библиотеки C API требуется в разрабатываемой dll.
4. Если для dll предусмотрено повторное подключение в одном процессе, но в различных инстанциях **Lua**, то следует обеспечить реентерабельность (корректное использоание в различных потоках) dll например, используя возможность кодирования тела стандартной функции входа DLL:
   extern "C" BOOL WINAPI DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)).
5. Шаблон функций dll, вызываемых в Lua:

```
//  Все параметры из Lua передаются через стек L.
//  Значения функции возвращаются тоже через стек L.
static int forLua_<Имя_функции(латиница)> (lua_State* L)
{
  //  Тело функции  ---

 //  Вариант завершения функции  ----
  lua_settop(L, 0); //  очистка стека  ---
  <Запись значений в стек L>
  return(<Количество возвращаемых значений>);
}
```

6. Описание интерфейса dll со скриптами Lua:

```
/* Хидеры и либу брать с
https://sourceforge.net/projects/luabinaries/files/5.3.5/Windows%20Libraries/Static/   */

 // Для 64-р. поставить в свойствах проекта платформу x64
 //   ----------------------------------------------------
 //=== Регистрация dll функций, чтобы они стали "видимы" для Lua.
 //Где forLua_<Функция_1_в_Lua (латиница)> и т.д. имена определений фу-ций dll
static struct luaL_Reg ls_lib[] = {
{"<Функция_1_в_Lua(латиница)>", forLua_<Функция_1_в_Lua (латиница)>},
{"<Функция_2_в_Lua(латиница)>", forLua_<Функция_2_в_Lua (латиница)>},
     //  ---------- и т.д.   ------
{ NULL, NULL }
};
//=== Регистрация названия библиотеки, видимого в скрипте Lua ---
extern "C" LUALIB_API int luaopen_<Имя_модуля (латиницей)>(lua_State * L) {
      luaL_openlib(L, "<Имя_модуля (латиницей)>", ls_lib, 0);
      return 0;
}
//  ----------------------------------------------------
```

C API Lua

C API **Lua** обеспечивает все возможности взаимодействия из **C/C++** со средой и средствами **Lua**, начиная с инсталляции экземпляров **Lua** (их может быть несколько в различных потоках, полностью автономных друг от друга).

В приложении «Язык **Lua** (C API) и использование скриптов на нем в программах на **C++** (Боресков А.В.)» приведен фрагмент статьи Борескова А.В. коротко в хорошем стиле описывающий использование C API в программах на C/C++.

Приложения

Функция печати данных любого типа Lua (в том числе произвольных таблиц с их метатаблицами, с выделением отступами вложенности их элементв и указанием всех связей между ними).

При печати таблиц, их структура (связи, уровни вложенности, метатаблицы) отображается в виде дерева с указанием всех связей. В этой функции много необязательных параметров (описанных, в комментариях кода), но в простых случаях ее использования, достаточно вызова с одним параметром: **dump_tbl(<Распечатываемое значение>)**. Результат: строка.

Для использования функции: 1) копируется ее текст; 2) вставляется в скрипт и проверяется на любой таблице **Lua**, например, **_G**.

```
-----------------------------------
---   Вывод  любых данных Lua. В том числе любых таблиц, в виде таблицы-массива строк (либо в виде строки)   -------
-------  Параметры:
--    1) t (единственный обязательный) -  любой тип, если таблица, то выводятся все вложения до значения параметра limit;
--    2) i  -  начальная строка формирования отступа при выводе вложений (например, '  ')
-----   3) limit - уровень вложенности до которого просматривается таблица (если = 0, то все уровни)------
-----   4) pr - <Вид выдачи результата: 0 (по умолчанию) – строка (образ таблицы); 1 - таблица-массив строк образа таблицы>
-----   5) list - <Таблица имен элементов (задаваемых в ключах) , не разворачиваемых далее >
---   ! Результат (если t - таблица): таблица-массив строк или строка, в зависимости от параметра pr.
---          Элемент результата вывода таблицы: заголовок таблицы. Остальные элементы - строковые представления  структуры таблицы t
---          с дополнительными служебными описаниями вложенных в t таблиц.
local function dump_tbl(t, i, limit, pr, list)
  if type(t) ~='table' then
    return 'Тип ' .. type(t) .. ': ' .. tostring(t)
  end
  i = i or ''
  list = list or {}
  if type(list) ~='table' then
    return nil,  '! Ошибка. Параметр list не таблица'
  end
  pr = pr or 0
  limit = limit or 0
  local tbl = {};  ---   для сбора результата  -----
  tbl[#tbl +1]  ='====   Таблица (текстовая сортировка по возрастанию индексов таблиц): ' .. tostring(t)
                .. ' \n                ! Количество выводимых уровней вложенности (если = 0, то выводятся все) = ' .. limit .. '\n'
  local seen={}            -- просмотренные --
  local Level = 0
  ----------------------------------
  local function dump(t,i, nm)
     nm = nm or 'T'
              if seen[t] then
                return seen[t]
              end
     ------
     seen[t] = nm
                        local  t_v,  t_k, nm_tm, v_tm
                        Level = Level +1  ---- Уровень вложенности таблицы -----
```

```
                    --- Обработка метатаблицы  -----
                    local mt = debug.getmetatable(t)
                    if  mt  then
                      nm_tm = seen[mt]
      if  nm_tm then
        tbl[#tbl +1]  = i .. 'У таблицы ' ..  nm .. '  есть существующая метатаблица  -> ' .. nm_tm .. '\n'
      else
        nm_tm = tostring(mt)
              tbl[#tbl +1]  = i .. 'Вложенность: ' .. Level .. '  == Содержимое метатаблицы ### :  ' .. nm_tm .. '\n'
                    dump(mt, i..'\t', nm_tm)
                    tbl[#tbl +1]  = i .. 'Вложенность: ' .. Level .. '  == Конец  метатаблицы  ##' .. nm_tm .. '\n'
                        end
    end
                    ----------------------------------
                    if next(t) == nil then  ----- Таблица пустая ---
                      tbl[#tbl +1] = i .. 'Вложенность: ' .. Level .. '  Таблица:  ' ..  nm .. '  пустая ### '.. '\n'
                    Level = Level - 1
                        return ''
                    end

    local s={}  ---  массив хранение имен ключей (строк) для сортировки ---
                    local ss={}  -- массив для хранения самих ключей --
    local n=0
                    local ks=0
    for k, v in next, t do
                      ks = tostring(k)
      n=n+1
                        s[n] = ks
          ss[ks] = k
    end
    table.sort(s)

    for k,v in ipairs(s) do  ---  s - таблица-массив  ---
                      t_k = ss [v]  -- ключ записи в t,   v - имя ключа (tostring(t_k))   --
                      t_v = t [t_k]  -- значение записи в t  --

                        if Level < (limit + 2) or limit == 0 then
                          --- Обработка ключа ---
                          v_tm = '[' .. v .. ']  ('
        if type(t_k) == 'table' then  --- Обработка ключа-таблицы ---
                                    nm_tm = seen[t_k]
            if  nm_tm then
                                                v_tm = '[' .. tostring(t_k) .. ' -  ссылка на существующую таблицу-ключ  -> ' ..
nm_tm .. ']  ('  -- !!  будет вместо v_tm
             else
                                                tbl[#tbl +1]  = i .. 'Вложенность: ' .. Level .. '  == Содержимое ключа-таблица $$ :  ' .. v .. '\n'
                                                dump(t_k, i..'\t', tostring(t_k))
                                          tbl[#tbl +1]  = i .. 'Вложенность: ' .. Level .. '  == Конец  ключа-таблицы  $$ ' .. v .. '\n'
                                              end
        end
        ------  Обработка поля  ------
        if type(t_v) == 'table' then ---  Обработка поля-таблицы  ----
                                    nm_tm = seen[t_v]
          if nm_tm then  -- таблица уже обработана ---
                                    tbl[#tbl +1]  = i .. v_tm .. type(t_v) .. ') = ' .. tostring(t_v) ..  '  ссылка на существующую таблицу  -> ' ..
nm_tm .. '\n'
            else
                                    if list[v] then
                                                tbl[#tbl +1]  = i .. v_tm .. type(t_v) .. ') = ' .. tostring(t_v) .. ' | Терминальная таблица (в печати
задано далее не разворачивать) ### \n'
                                              else
              tbl[#tbl +1]  = i .. v_tm .. type(t_v) .. ') = ' .. tostring(t_v) .. '\n'
                tbl[#tbl +1]  = i .. 'Вложенность: ' .. Level .. '  == Содержимое таблицы :  ' .. v .. '\n'
                                      dump(t_v, i..'\t', tostring(t_k))
                                      tbl[#tbl +1]  = i .. 'Вложенность: ' .. Level .. '  == Конец таблицы  ' .. v .. '\n'
                                          end
          end
```

```
            else
              tbl[#tbl + 1]  = i .. v_tm.. type(t_v) .. ') = ' .. tostring(t_v) .. '\n'
            end
            ------------------------------
                              end
      end
                          Level = Level - 1
                          return ''
    end
            ---------------
    dump(t, i, tostring(t))
    if pr == 1 then         --- Выдача результата в виде массива строк  ---
      return  tbl
    else return table.concat(tbl)    --- Выдача результата в виде строки  ---
    end
  end
```

Конфигурирование Lua для режима запуска сопрограмм в отдельных потоках

Изменения (для обеспечения режима запуска сопрограмм в отдельных потоках) в исходном коде Lua 5.3.5 (5.4.1) приведены в следующем виде:

1) № изменения
2) <Имя файла исходника>
3) <Текст исходника, позволяющий определить место после которого внесено изменение>
4) <Текст изменения>

---

1. Изменение № 1
Файл: ldo.c
Место: #include "ldo.h"
Добавление:

```
#include "DbgHelp.h"
#pragma comment(lib, "Dbghelp.lib")
void lua_lock_MT(lua_State *L) {
global_State *g = G(L);
EnterCriticalSection(&g->Глобальная_критическая_секция);
}

void lua_unlock_MT(lua_State *L) {
global_State *g = G(L);
LeaveCriticalSection(&g->Глобальная_критическая_секция);

}
```

---

2. Изменение № 2
Файл: ldo.h
Место: # LUAI_FUNC int luaD_rawrunprotected (lua_State *L, Pfunc f, void *ud);
Добавление:

```
void lua_lock_MT(lua_State *L);
void lua_unlock_MT(lua_State *L);
```

---

3. Изменение № 3
Файл: lua.h
Место: LUA_API int (lua_gethookcount) (lua_State *L);
Добавление:

```
void lua_lock_MT(lua_State *L);
void lua_unlock_MT(lua_State *L);
```

---

4. Изменение № 4
Файл: lua.h
Место: #define lua_h
Добавление:

```
  #include "windows.h"
```

4. Изменение № 4
Файл: limits.h
Место:

```
/*
** macros that are executed whenever program enters the Lua core
```

** ('lua_lock') and leaves the core ('lua_unlock')
*/
Добавление:
#if !defined(lua_lock)
#define lua_lock(L) lua_lock_MT(L)
#define lua_unlock(L) lua_unlock_MT(L)
#endif
---
5. Изменение № 5
Файл: luaconf.h
Место:
** the libraries, you may want to use the following definition (define
** LUA_BUILD_AS_DLL to get it).
*/
Добавление:
#define LUA_BUILD_AS_DLL
---
6. Изменение № 6
Файл: lstate.c
Место:
g->gcstepmul = LUAI_GCMUL;
Добавление:
InitializeCriticalSection(&g->Глобальная_критическая_секция);
---
6. Изменение № 7
Файл: lstate.h
Место:
TString *strcache[STRCACHE_N][STRCACHE_M]; /* cache for strings in API */
Добавление:
CRITICAL_SECTION Глобальная_критическая_секция;

Язык Lua (C API) и использование скриптов на нем в программах на C++ (Боресков А.В.).

C API для Lua

Все необходимые типы, функции и константы определены в файле *lua.h*, кроме этого также необходима библиотека lua<Версия языка>.lib (либо для статического подключения, либо для динамического). Все это можно скачать с сайта разработчика **Lua**.

Подобно OpenGL и другим библиотекам все функции в ***Lua*** начинаются с префикса *lua_*, а константы - с префикса *LUA_*.

Для работы с ***Lua*** необходимо сначала создать контекст (инсталляцию Lua) для работы (иногда вместо термина контекст используется термин виртуальная машина). У вас может быть несколько независимых контекстов, в каждом из которых происходит работа. Все остальные функции C API получают этот контекст как первый параметр. Для создания контекста служат следующие функции:

```
lua_State *lua_open();
lua_State *lua_newstate( lua_Alloc f, void * ud );
lua_State *luaL_newstate();
```

Первая из этих функция на самом деле сейчас определена как макрос, вызывающий *luaL_newstate* и является устаревшей функцией. Функция *lua_newstate* создает новое состояния и при этом получает в качестве параметров функцию выделения памяти и параметр, передаваемый этой функции при ее вызове. Функция *luaL_newstate* использует стандартную функцию выделения памяти, основанную на функции *realloc* и служит заменой *lua_open*.

При ошибке создания контекста возвращается значение **NULL**.

Тип **lua_Alloc** определен следующим образом:

```
typedef void * (*lua_Alloc) ( void * ud, void * ptr, size_t oldSize, size_t
newSize );
```

Для освобождения контекста служит функция *lua_close*:

```
void lua_close ( lua_State * lua );
```

Одной из особенностей ***Lua*** является то, что передача значений между программой на C/C++ и ***Lua*** происходит через специальный стек, каждый элемент этого стека является значением в ***Lua***.

При вызове функции C/C++ из ***Lua*** вызываемая функция получает свой стек Lua, независимый от предыдущих стеков и стеков других функций, активных на данный момент.

Изначально стек содержит все аргументы для функции в прямом порядке (т.е. первый аргумент помещается на стек первым), на этот же стек, функция заносит возвращаемые значения.

При этом используемый стек (в отличии от традиционного понимания стека) позволяет доступ (как на чтение, так и на запись) к каждому своему элементу при помощи целочисленного *индекса*. Положительный индекс соответствует абсолютному положению со дна стека (индексы начинается с единицы), отрицательные значения соответствуют смещению с вершины стека к его началу.

Так если стек содержит *n* элементов, то индекс 1 соответствует первому элементу (первому аргументу вызываемой функции), индекс 2 - второму, индекс *n* соответствует последнему (верхушке стека). Индекс -1 соответствует последнему элементу, -2 - предпоследнему элементу, индекс -*n* - первому элементу (см. рис). Индекс *i* является валидным (т.е. может использоваться для обращения к элементам стека) если *1<=abs(i)<=n*. Нулевое значение не является валидным индексом.

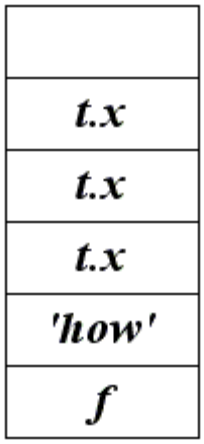

Рис 2. Индексация в стеке.

Вы сами должны отвечать за то, чтобы не произошло переполнение стека (начальный размер стека фиксирован, по умолчанию он равен 20). Для явного задания размера стека служит функция *lua_checkstack*:

```
int lua_checkstack (lua_State * lua, int extra );
```

После этого вызова размер стека будет составлять не менее *extra* элементов. При ошибке (не удалось увеличить размер стека) возвращается ноль. Обратите внимание, что данная функция может только увеличить размер стека, но не уменьшить.

Кроме стандартных индексов для обращения в стек можно использовать и так называемые псевдо-индексы. Псевдо-индексы соответствуют доступным значениям ***Lua***, не находящимся на стеке.

Доступ к глобальным переменным осуществляется через псевдоиндекс LUA_GLOBALSINDEX. Текущее окружение доступно через псевдоиндекс LUA_ENVIRONINDEX.

Для доступа к глобальной переменной используется обычный механизм доступа к полям таблицы. Так для доступа к глобальной переменной *x* используется следующий вызов:

```
lua_getfield ( lua, LUA_GLOBALSINDEX, "x" );
```

Этот вызов возьмет значение глобальной переменной *x* и добавит на вершину стека.

Как и в "маленьких мягких окошках" в ***Lua*** есть свой реестр. Только здесь это просто специальная таблица, в которой код из C/C++ может хранить значения. Для доступа к этой таблице служит псевдоиндекс LUA_REGISTRYINDEX. Не рекомендуется осуществлять запись в таблицу реестра по *целочисленным* ключам - они используются ***Lua***.

Далее мы рассмотрим основные (наиболее часто используемые) функции для работы с ***Lua***, а потом перейдем к практическим примерам использования ***Lua*** в программах на C/C++.

```
void lua_call (lua_State * lua, int numArgs, int numResults );
void lua_pcall (lua_State * lua, int numArgs, int numResults, int errFunc );
```

Эта функция служит для вызова функции. Перед ее вызовом следует поместить вызываемую функцию на стек, а затем все передаваемые аргументы в прямом порядке (первый аргумент первым, затем второй и т.д.).

В параметре *numArgs* передается количество переданных аргументов. После вызова все переданные аргументы автоматически убираются со стека, а вместо них в стек заносятся результаты выполнения функции в естественном порядке.

Если значение параметра *numResults* не равно LUA_MULTRET, то на стеке будет оставлено ровно *numResults* значений. Лишние значения убираются со стека, вместо недостающих, заносятся *nil*.

Функция *lua_pcall* отличается тем, что в случае ошибки просто помещает на стек сообщение об ошибке и возвращает код ошибки. Код ошибки может быть равен нулю (все в порядке) или же принимать одно из следующих значений - LUA_ERRRUN (ошибка во время выполнения), LUA_ERRMEM (ошибка выделения памяти) и LUA_ERRERR (ошибка во время выполнения обработчика ошибок).

Если *errFunc* равно нулю, то возвращается стандартное сообщение об ошибке. В противном случае *errFunc* - это индекс функции для обработки ошибок (в данной реализации это не может быть псевдоиндекс).

```
void lua_getfield (lua_State * lua, int index, const char * key );
```

Данная функция берет значение из стека по заданному индексу, извлекает из него поле с заданным именем и значение этого поля помещается на вершину стека.

Пусть в стеке была таблица *t*. Тогда следующие рисунки иллюстрируют результаты вызовов функции *lua_getfield*.

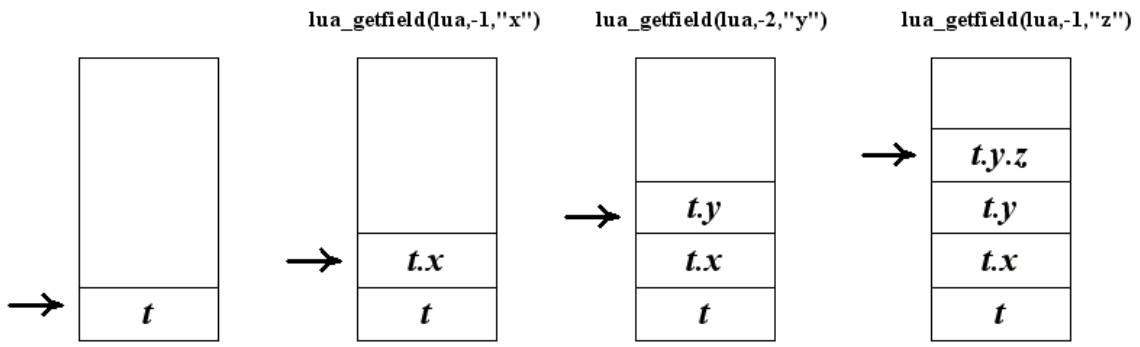

Рис 3. Работа функции *lua_getfield*.

```
void lua_setfield (lua_State * lua, int index, const char * key );
```

Данная функция в таблице, находящуюся в стеке по заданному индексу, устанавливает значение поля с заданным именем значению с вершины стека. При этом значение снимается с вершины стека.

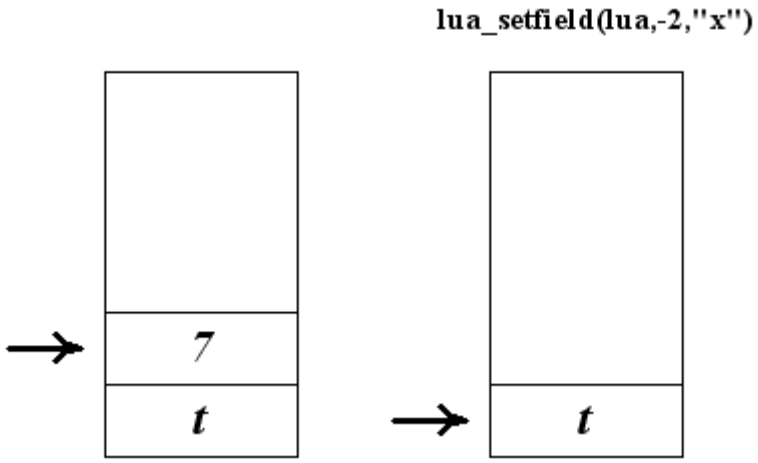

Рис 4. Работа функции *lua_setfield*

```
void lua_pop (lua_State * lua, int n );
```

Данный вызов удаляет с вершины стека *n* значений.

```
void lua_pushboolean (lua_State * lua, int value );
```

Функция добавляет на вершину стека логическое значение *value* (0 или 1).

```
void lua_pushinteger (lua_State * lua, lua_Integer value );
```

Функция добавляет на вершину стека число *value*.

```
void lua_pushlstring (lua_State * lua, const char * str, size_t len );
```

Функция добавляет на вершину стека копию строки с длиной *len* (маркер '\0' игнорируется).

```
void lua_pushnil (lua_State * lua );
```

Функция помещает на вершину стека значение *nil*.

```
void lua_pushnumber (lua_State * lua, lua_Number value );
```

Функция добавляет на вершину стека число *value*.

```
void lua_pushstring (lua_State * lua, const char * str );
```

Функция помещает на вершину стека строку, законченную символом '\0'.

```
void lua_pushvalue (lua_State * lua, int index );
```

Функция помещает копию значения, расположенного по заданному индексу на вершину стека.

```
void lua_pushfstring (lua_State * lua, const char * fmt, ... );
```

Аналогично функции *sprintf* помещает на вершину стека результат форматированного вывода в строку. Функция сама отслеживает необходимый объем памяти под строку. Однако есть ограничения на использование спецификаций форматов - не поддерживается точность и флаги.

```
void lua_remove (lua_State * lua, int index );
```

Удаляет элемент с заданным индексом из стека, сдвигая выше расположенные элементы.

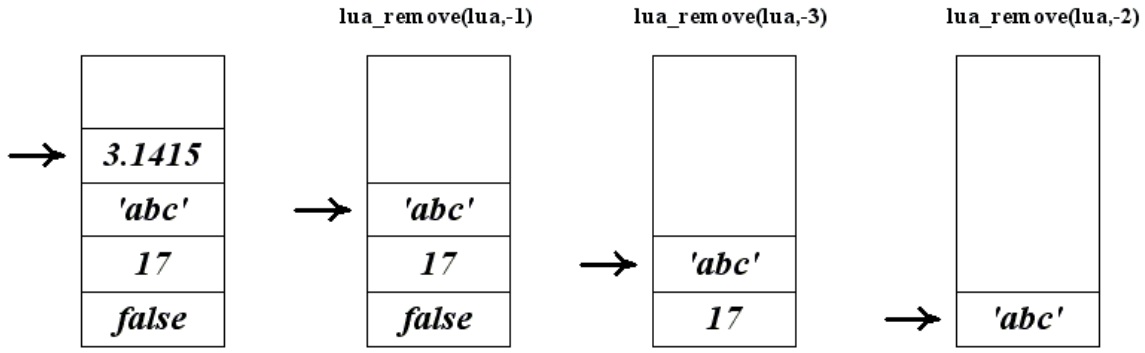

Рис 5. Работы функции *lua_remove*.

```
int lua_gettop (lua_State * lua );
```

Возвращает индекс элемента на вершине стека, т.е. количество элементов в стеке.

```
int lua_settop (lua_State * lua, int index );
```

Устанавливает новое количество элементов в стеке. При этом лишние элементы автоматически удаляются, при необходимости стек дополняется **nil**-ми.

```
int lua_insert (lua_State * lua, int index );
```

Перемещает элемент с вершины стека по заданному индексу сдвигая элементы.

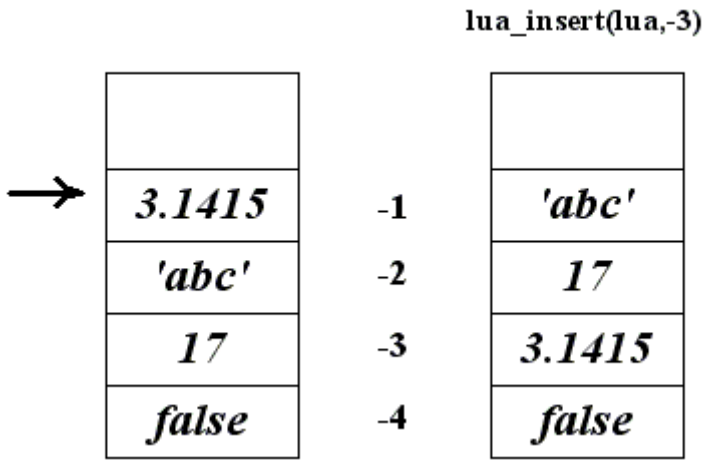

Рис. 6. Работа функции *lua_insert*.

```
void lua_rotate (lua_State *L, int idx, int n);
```

Прокручивает элементы стека между действительным индексом **idx** и вершиной стека. При положительном **n**, элементы проворачиваются на n позиций в направлении к вершине, или на **-n** позиций в направлении вниз, при отрицательномn. Абсолютное значение **n** не должно быть больше размера прокручиваемой части стека. Данная функция не может быть вызвана с псевдоиндексом, так как псевдоиндекс не является фактической позицией стека.

```
int lua_gettable (lua_State * lua, int index );
```

Помещает на вершину стека значение *t[k]*, где *t* - таблица, расположенная по заданному индексу, а ключ *k* - элемент с вершины стека.

```
int lua_createtable (lua_State * lua, int narr, int nass );
```

Создает новую пустую таблицу и помещает ее на вершину стека. При этом в таблице сразу резервируется место для *narr* элементов массива и *nass* для элементов хэша (обычно только один из этих параметров отличен от нуля).

```
int lua_newtable (lua_State * lua );
```

Создает новую таблицу, эквивалентен lua_createtable (lua, 0, 0).

```
void * lua_newuserdata (lua_State * lua, size_t size );
```

Функция выделяет блок памяти заданного размера и создает соответствующий объект типа **userdata**, помещая его на стек. Возвращается указатель на выделенный блок памяти.

```
int lua_equal (lua_State * lua, int index1, int index2 );
```

Возвращает единицу, если элементы по заданным индексам равны и ноль в противном случае.

```
int lua_getmetatable (lua_State * lua, int index );
```

Помещает на вершину стека метатаблицу объекта, расположенного в стеке по заданному индексу. Если индекс не является валидным или у соответствующего объекта нет метатаблицы, то возвращается ноль и на стек ничего не кладется.

```
int lua_setmetatable (lua_State * lua, int index );
```

Снимает таблицу с вершины стека и устанавливает ее в качестве метатаблицы для объекта по заданному индексу

```
int lua_isboolean (lua_State * lua, int index );
```

Возвращает единицу, если значение по заданному индексу является логическим (типа **boolean**) и ноль в противном случае.

```
int lua_isfunction (lua_State * lua, int index );
```

Возвращает единицу, если значение по заданному индексу является функцией (типа **funtion**) и ноль в противном случае.

```
int lua_isnil ( lua_State * lua, int index );
```

Возвращает единицу, если значение по заданному индексу является **nil** и ноль в противном случае.

```
int lua_isnone (lua_State * lua, int index );
```

Возвращает единицу, если переданный индекс является недопустимым.

```
int lua_isoneornil (lua_State * lua, int index );
```

Возвращает единицу, если либо переданный индекс является недопустимым, либо по этому индексу расположен **nil**.

```
int lua_isnumber (lua_State * lua, int index );
```

Возвращает единицу, если значение по заданному индексу является числом (типа **number**) и ноль в противном случае.

```
int lua_isstring (lua_State * lua, int index );
```

Возвращает единицу, если значение по заданному индексу является строкой (типа **string**) и ноль в противном случае.

```
int lua_istable (lua_State * lua, int index );
```

Возвращает единицу, если значение по заданному индексу является таблицей (типа **table**) и ноль в противном случае.

```
int lua_isuserdata (lua_State * lua, int index );
```

Возвращает единицу, если значение по заданному индексу является типа **userdata** и ноль в противном случае.

```
int lua_toboolean (lua_State * lua, int index );
```

Возвращает единицу, если значение, расположенное по заданному индексу истинно и ноль в противном случае.

```
lua_Integer lua_tointeger (lua_State * lua, int index );
```

Преобразует значение по заданному индексу в знаковое целое число и возвращает его. Соответствующее значение должно быть числом или строкой.

```
const char * lua_tolstring (lua_State * lua, int index, size_t * length );
```

Преобразует значение со стека по заданному индексу в строку, если *length* не равен NULL, то в него заносится длина строки. Возвращается указатель на завершенную '\0' строку, расположенную во внутреннем буфере ***Lua***. Соответствующее значение должно быть строкой или числом.

```
lua_Number lua_tonumber (lua_State * lua, int index );
```

Преобразует значение по заданному индексу в число (обычно это **double**) и возвращает его. Значение должно быть строкой или числом.

```
const char * lua_tostring (lua_State * lua, int index );
```

Эквивалентно lua_tolstring (lua, index, NULL).

```
void * lua_tosuserdata (lua_State * lua, int index );
```

Если значение по заданному индексу имеет тип **userdata**, то возвращает указатель на соответствующий блок памяти. Иначе возвращает NULL.

```
int lua_type (lua_State * lua, int index );
```

Возвращает тип объекта по заданному индексу или LUA_TNONE в случае недопустимого индекса. Тип объекта принимает одно из следующих значений - LUA_TNIL, LUA_TNUMBER, LUA_TBOOLEAN, LUA_TSTRING, LUA_TTABLE, LUA_TFUNCTION, LUA_TUSERDATA, LUA_TTHREAD и LUA_TLIGHTUSERDATA.

```
const char * lua_typename (lua_State * lua, int typeCode );
```

Переводит константы типов в соответствующие строковые значения.

```
void lua_register (lua_State * lua, const char * name, lua_CFunction func
);
```

Создает новую функцию с именем *name*, на основе C-функции и делает ее доступной программам на ***Lua***.

Тип *lua_CFunction* определен следующим образом:

```
typedef int (*lua_CFunction) (lua_State * );
```

Фактически каждая функция на C/C++, которую мы хотим экспортировать должна иметь такой вид - все аргументы она получает на стеке, результаты также помещает на стек и возвращает количество результатов, записанных на стек.

```
void lua_rawget (lua_State * lua, int index );
```

Полностью аналогично *lua_gettable*, но для обращения к таблице не использует метаметоды.

```
void lua_rawgeti (lua_State * lua, int index, int n );
```

Если через *t* обозначить таблицу по индексу *index* в стеке, то данный вызов возвращает значение (без использования метаметодов) *t[n]*.

```
void lua_rawset (lua_State * lua, int index );
```

Полностью аналогично *lua_settable*, но для записи в таблицу не использует метаметоды.

```
void lua_rawseti (lua_State * lua, int index, int n );
```

Если через *t* обозначить таблицу по индексу *index* в стеке, а через *v* значение на вершине стека, то данный вызов делает прямую запись (без использования метаметодов) *t[n]=v*.

Пусть у нас есть следующий фрагмент кода на ***Lua***:

```
a = f ( "how", t.x, 14 )
```

Он будет эквивалентен следующему фрагменту кода на C/C++.

```
lua_getfield    (lua, LUA_GLOBALSINDEX, "f" );   // push global function f
on stack
lua_pushstring  (lua, "how" );                   // push first argument on
stack
lua_getfield    (lua, LUA_GLOBALSINDEX, "t" );   // push global table t on
stack
lua_getfield    (lua, -1, "x" );                 // push t.x on stack
lua_remove      (lua, -2 );                      // remove t from stack
lua_pushinteger (lua, 14 );                      // push last argument
lua_call        (lua, 3, 1 );                    // call function taking 3
argsuments and getting one return value
lua_setfield    (lua, LUA_GLOBALSINDEX, "a" );   // store result from top of
stack to global variable a
```

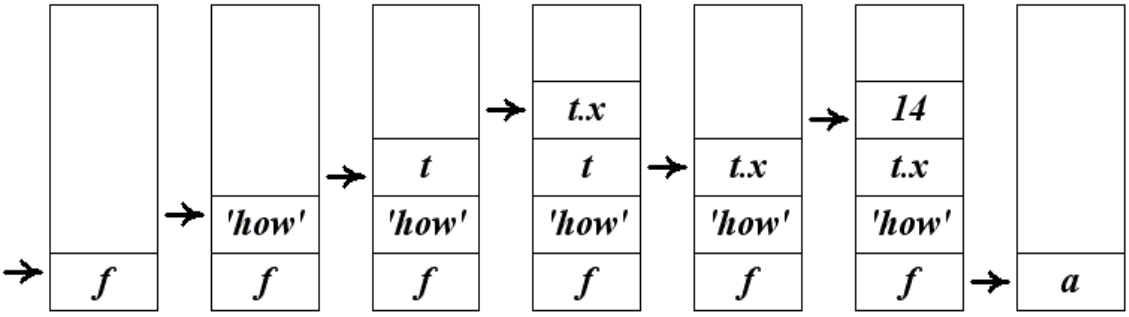

Рис. 7. Стек во время вызова.

Кроме перечисленных функция в C API также в входит дополнительная библиотека. Заголовочным файлом для нее служит файл *lauxlib.h*, а имена всех функций начинаются с префикса *luaL_*.

Все функции дополнительной библиотеки построены на основе основного API, но в ряде случаев более удобны.

```
void luaL_openlibs (lua_State * lua );
```

Открывает и делает доступными в текущем контексте для скриптов на ***Lua*** все стандартные библиотеки.

```
int luaL_callmeta (lua_State * lua, int index, const char * eventName );
```

Если объект по заданному индексу содержит метатаблицу и в ней есть поле с именем *eventName*, то это поле вызывается (как функция) и ей в качестве параметра передается исходный объект (с вершины стека).

```
int luaL_loadfile (lua_State * lua, const char * fileName );
```

Загружает файл с именем *fileName* и помещает его на стек как функцию (поэтому после загрузки необходимо сделать вызов *lua_call* или *lua_pcall*).

При успехе возвращает 0, иначе один из следующих кодов - LUA_ERRSYNTAX, LUA_ERRMEM или LUA_ERRFILE.

```
int luaL_loadstring (lua_State * lua, const char * str );
```

Загружает код на ***Lua*** из строки и помещает его на стек как функцию.

```
int luaL_dofile (lua_State * lua, const char * fileName );
```

Загружает и выполняет заданный файл. Определен как следующий макрос:

```
(luaL_loadfile(lua, filename) || lua_pcall(lua, 0, LUA_MULTRET, 0))
void luaL_register (lua_State * lua, const char * libName, const lua_Reg *
reg );
```

Открывает библиотеку с заданным именем *libName*. Если *libName* равно NULL, то просто регистрирует все функции из массива *reg* (концом массива служит запись с обоими значениями равными NULL).

Если *libName* не равно NULL, то создается новая таблица как глобальная переменная и именем модуля и в ней регистрируются все функции из массива *reg*.

Примеры интеграции скриптов на *Lua* в программы на C/C++

Первый наш пример просто загружает скрипт и выполняет его. Ниже приводится соответствующий скрипт, его задача простейшая - просто вывести сообщение.

```
-- simple test of calling lua from C

print ( 'Hello World !' )
```

Ниже приводится код на C++. Его задача создать контекст, открыть стандартные библиотеки и выполнить заданный файл.

Все это и делает приведенный ниже исходный код.

```
#ifdef__cplusplus
extern "C" {
#endif

#include     "lua.h"
#include     "lauxlib.h"
#include     "lualib.h"

#ifdef__cplusplus
};
#endif

int main ()
{
    lua_State * lua = lua_open ();         // create Lua context
```

```
    if ( lua == NULL )
    {
        printf ( "Error creating Lua context.\n" );

        return 1;
    }

    luaL_openlibs ( lua );                  // open standart librar-
ies
                                            // load and execute a
file
    if ( luaL_dofile ( lua, "test-1.lua" ) )
        printf ( "Error opening test-1.lua\n" );

    lua_close     ( lua );                  // close Lua context

    return 0;
}
```

Следующий пример будет несколько сложнее - нашей задачей будет вызвать определенную в скрипте функцию и получить от нее результат.

Пример функции приведен ниже, он просто печатает входные параметры и возвращает их суммы, хотя это может быть любая требуемая функциональность.

```
function foo ( x, y )
   print ( 'foo: ', x, y )
   return x + y
end
```

Соответствующий код на C++ должен не просто создать констекст и загрузить библиотеки. Также необходимо загрузить файл с нашей функций и вызвать *lua_pcall*, чтобы вызвать этот файл и определенная в нем функция стала доступна.

После этого мы просто помещаем вызываемую функцию и необходимые параметры на стек и делаем вызов *lua_pcall*. После него на стеке останется результат выполнения функции.

```
#ifdef __cplusplus
extern "C" {
#endif

#include     "lua.h"
#include     "lauxlib.h"
#include     "lualib.h"

#ifdef __cplusplus
};
#endif

int main ()
{
    lua_State * lua = lua_open ();                      // create Lua
context

    if ( lua == NULL )
    {
```

```
        printf ( "Error creating Lua context.\n" );

        return 1;
    }

    luaL_openlibs ( lua );                                  // open
standart libraries
                                                            // load and
execute a file
    if ( luaL_loadfile ( lua, "test-2.lua" ) )
        printf ( "Error opening test-2.lua\n" );

    lua_pcall       ( lua, 0, LUA_MULTRET, 0 );
    lua_getfield    ( lua, LUA_GLOBALSINDEX, "foo" );   // push glob-
al function f on stack
    lua_pushstring  ( lua, "17" );                          // push first
argument on stack
    lua_pushinteger ( lua, 3 );                             // push se-
cond argument on stack
    lua_pcall       ( lua, 2, 1, 0 );                   // call func-
tion taking 2 argsuments and getting one return value

                                                            // get return
value and print it
    printf ( "Result: %d\n", lua_tointeger ( lua, -1 ) );

    lua_close       ( lua );                                // close Lua
context

    return 0;
}
```

Наш следующий пример сделает обратное - определит функцию на C++ и проэкспортирует ее в ***Lua***. После этого будет выполнен скрипт, который вызовет нашу функцию.

В качестве примера экспортируемой функции возьмем нахождение среднего из переданных аргументов. При этом пусть возвращается как само среднее значение, так и число переданных аргументов. Ниже приводится скрипт, демонстрирующий использование данной функции.

```
av, n = ave ( 1, 2, 3, 4, 5, 6, 7, 8 )
print ( 'Average ', av )
print ( 'Count ', n )
```

Нам необходимо создать функцию и экспортировать ее в ***Lua***. После этого можно звать сам скрипт. Ниже приводится соответствующий исходный код, обратите внимание, на то что функция возвращает 2, т.к. ровно два выходных значения положены на стек.

```
#ifdef __cplusplus
extern "C" {
#endif

#include     "lua.h"
#include     "lauxlib.h"
#include     "lualib.h"

#ifdef __cplusplus
};
```

```
#endif


int   ave ( lua_State * lua )                                  // function
to be exported to Lua
{
    int     num = lua_gettop ( lua );                          // get # of
arguments
    double  sum = 0;

    for ( int i = 1; i <= num; i++ )
        sum += lua_tonumber ( lua, i );

    lua_pushnumber ( lua, sum / num );                         // return av-
erage
    lua_pushnumber ( lua, num );                               // return #
of arguments

    return 2;                                                  // we re-
turned two vaues
}

int main ()
{
    lua_State * lua = lua_open ();                             // create Lua
context

    if ( lua == NULL )
    {
        printf ( "Error creating Lua context.\n" );

        return 1;
    }

    luaL_openlibs ( lua );                                     // open
standart libraries
    lua_register  ( lua, "ave", ave );                         // register
out function
                                                               // load and
execute a file
    if ( luaL_dofile ( lua, "test-3.lua" ) )
        printf ( "Error opening test-3.lua\n" );

    lua_close     ( lua );                                     // close Lua
context

    return 0;
}
```

Полный формальный синтаксис Lua 5.3 (https://lua.org.ru/manual_ru.html#3.4.3)

В этом разделе приведен полный синтаксис **Lua 5.3** на метаязыке **БНФ**.

1. **<**Name**>** (также называемые идентификаторами переменных) в Lua могут быть любой строкой из букв, цифр и подчеркиваний, не начинающейся с цифры. Идентификаторы используются для именования значений, таблиц, полей таблиц и меток (labels);
2. **<**Numeral**>** - числовая константа (или цифра) может быть записана с опциональной дробной частью и опциональной десятичной экспонентой, обозначенной буквой 'e' или 'E'.

Lua также поддерживает шестнадцатиричные константы, которые начинаются с 0x или 0X. Шестнадцатиричные константы также допускают использование дробной части и бинарной экспоненты, обозначеной буквой 'p' или 'P'. Цифровая константа с разделительной точкой или экспонентой означает вещественное число; иначе она означает целое.

3. **<**LiteralString**>** - литеральные строки могут быть ограничены сочетающимися одинарными или двойными кавычками, и могут содержать C-подобные управляющие последовательности:

'\a' (bell), '\b' (backspace), '\f' (form feed), '\n' (newline), '\r' (carriage return), '\t' (horizontal tab), '\v' (vertical tab), '\\' (backslash), '\"' (двойная кавычка) и '\'' (апостроф [одинарная кавычка]).

обратный слеш, сопровождаемый реальным переходом на новую строку (newline), формирует переход строки (newline) в строке (string). Управляющая последовательность '\z' пропускает следующий за ней диапазон пробелов, включая переходы строки; это особенно полезно, чтобы разбить и отступить длинную литеральную строку на несколько линий без добавления переводов строки и пробелов в содержимое строки.

Символ \0 (0) в строках Lua не является признаком конца строки.

Описания терминалов **<**Name**>**, **<**Numeral**>** и **<**LiteralString**>**, см. §3.1. (https://lua.org.ru/manual_ru.html#3.4.3)

---- *Описание языка **Lua*** (для наглядности, в кавычки ' ' заключены некоторые группы терминальных символов) ---

<chunk**> ::=<** block**>**

<block**> ::= [{<**stat**>}] [<**retstat**>]**

<stat**> ::=** ‘;’ **|**
varlist ‘=’ **<**explist**> |**
<functioncall> |
<label**> |**
break **|**
goto **<**Name**> |**
do **<**block**>** end **|**
while **<**exp**>** do **<**block**>** end **|**
repeat **<**block**>** until **<**exp**> |**
if **<**exp**>** then **<**block**> [{**elseif **<**exp**>** then **<**block**>}] [**else **<**block**>]** end **|**
for **<**Name**>** ‘=’ **<**exp**>** ‘,’ **<**exp**> [**‘,’ <exp>**]** do **<**block**>** end **|**
for **<**namelist**>** in **<**explist**>** do **<**block**>** end **|**
function **<**funcname**>** <funcbody**> |**
local function **<**Name**>** <funcbody**> |**
local **<**namelist**> [**‘=’ **<**explist**>]**

<retstat**> ::=** return **[<**explist**>] [**‘;’**]**

<label**> ::=** ‘::’ **<**Name**>** ‘::’

<funcname**> ::= <**Name**> [{**‘.’ **<**Name**>}] [**‘:’ **<**Name**>]**

<varlist**> ::= <**var**> [{**‘,’ **<**var**>}]**

<var**> ::=** <Name**> |** <prefixexp**>** ‘[’ **<**exp**>** ‘]’ **|** <prefixexp**>** ‘.’ **<**Name**>**

<namelist**> ::= <**Name**> [{**‘,’ **<**Name**>}]**

<explist**> ::= <**exp**> [{**‘,’ **<**exp**>}]**

<exp**> ::=** nil **|** false **|** true **|** <Numeral**> |** <LiteralString**> |** ‘...’ **|** <functiondef**> |**
<prefixexp**> |** <tableconstructor**> |** <exp**>** <binop**>** <exp**> |** <unop**>** <exp**>**

<prefixexp**> ::= <**var**> |** <functioncall**> |** ‘(’ **<**exp**>** ‘)’

<functioncall**> ::=** <prefixexp**>** <args**> |** <prefixexp**>** ‘:’ **<**Name**>** <args**>**

<args**> ::=** ‘(’ **[<**explist**>]** ‘)’ **|** <tableconstructor**> |** <LiteralString**>**

<functiondef**> ::=** function **<**funcbody**>**

<funcbody**> ::=** ‘(’ **[<**parlist>] ‘)’ **<**block**>** end

<parlist**> ::= <**namelist**> [**‘,’ ‘...’**] |** ‘...’

<tableconstructor**> ::=** ‘{’ **[<**fieldlist**>]** ‘}’

<fieldlist**> ::= <**field**> {<**fieldsep**>** <field**>} [<**fieldsep**>]**

<field**> ::=** ‘[’ **<**exp**>** ‘]’ ‘=’ **<**exp**> |** <Name**>** ‘=’ **<**exp**> |** <exp**>**

<fieldsep**> ::=** ‘,’ | ‘;’

<binop**> ::=** ‘+’ **|** ‘-’ **|** ‘*’ **|** ‘/’ **|** ‘//’ **|** ‘^’ **|** ‘%’ **|**
‘&’ **|** ‘~’ **|** ‘|’ **|** ‘>>’ **|** ‘<<’ **|** ‘..’ **|**
‘<’ **|** ‘<=’ **|** ‘>’ **|** ‘>=’ **|** ‘==’ **|** ‘~=’ **|**
and **|** or

<unop**> ::=** ‘-’ **|** not **|** ‘#’ **|** ‘~’
