Lua 5.3 Руководство

Lua · Документация

by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes
перевел Ведерников Николай, Лысьва.

1 – Введение

Lua - язык программирования расширений, разработан для поддержки общего процедурного программирования с возможностью описания данных. Lua также предлагает хорошую поддержку объектно-ориентированного, функционального и управляемого данными (data-driven) программирования. Lua предлагается как мощный и лёгкий встраиваемый скриптовый язык для любой программы, которая в этом нуждается. Lua реализован как библиотека, написан на *чистом C*, общее подмножетво стандартного C и C++.

Как язык расширений, Lua не имеет понятия "главной" программы: он работает только как *встроенный* в основную программу клиент, *встраивающая программа* называется *хост* (host).

Встраивающая программа может вызывать функции для запуска кусочков Lua кода, может писать и читать Lua переменные, может регистрировать C-функции чтобы вызывать их из Lua кода. Через использование C-функций Lua может быть расширен для решения различных задач, таким образом созданные адаптированные языки программирования имеют общую синтаксическую базу. Дистрибутив Lua включает простую хост-программу `lua`, которая использует библиотеку Lua для реализации полного независимого Lua интерпретатора, для интерактивного или пакетного использования.

Lua бесплатное программное обеспечение и предоставляется безо всяких гарантий. Официальный сайт `www.lua.org`.

Как и любое другое руководство, это руководство местами сухо. Описание решений принятых в основе дизайна Lua есть на технических страницах официального сайта. Детальное введение в программирование на Lua представлено в книге Роберто Иерусалимского *Programming in Lua*.

2 – Базовые концепции

Этот раздел описывает базовые концепции языка.

2.1 – Значения и типы

Lua *динамически типизированный язык*. Это означает, что значения не имеют типов; только значения. Язык не имеет определений типов. Все значения несут свой собственный тип.

Все значения в Lua *первоклассные*. Это означает что все значения могут быть сохранены в переменных, переданы как аргументы другим функциям, и возвращены как результаты.

В Lua существует восемь базовых типов: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread* и *table*.

Тип *nil* (нуль) имеет одно единственное значение, **nil**, его главное свойство это отличаться от любых других значений; обычно это означает отсутствие используемого значения.

Тип *boolean* (логический) имеет два значения: **false** (ложь) и **true** (истина). Оба **nil** и **false** означают false; любое другое значение означает true.

Тип *number* (число) представляет целые (integer) и вещественные (float) числа.

Тип *string* (строка) представляет неизменные последовательности байт. Строки в Lua могут содержать любое 8-битное значение, включая нули ('`\0`'). Также Lua противник кодировок; никаких предположений о содержимом строки не делается.

Тип *number* использует два внутренних представления, или два подтипа, один называется *integer* (целое), второй *float* (число с плавающей запятой). Lua имеет явные правила о том, когда какое представление использовать, но при необходимости автоматически конвертирует значение между ними (см. §3.4.3). Следовательно, в большистве случаев программист может игнорировать разницу между целыми и реальными числами, или получить полный контроль над представлением каждого числа. Стандартный Lua использует 64-битные целые (integer) и вещественные числа двойной точности (double 64-bit), но также возможно скомпилировать Lua так, чтобы использовались 32-битные целые и/или вещественные числа одинарной точности (float 32-bit). Эта опция с 32 битами для целых и вещественных чисел особенно актуальна для малых машин и встроенных систем. (Смотри макрос `LUA_32BITS` в файле `luaconf.h`.)

Lua может вызывать и манипулировать функциями написанными на Lua и на C (см. §3.4.10). Оба типа функций в Lua представлены типом *function* (функция).

Тип *userdata* (пользовательские данные) предназначен для хранения произвольных C данных в Lua

переменных. Значение userdata представляет блок памяти (raw memory). Существуют два типа пользовательских данных: *full userdata* (полные пользовательские данные) - объект с блоком памяти, которым управляет Lua, и *light userdata* (лёгкие пользовательские данные) - простой C указатель. Пользовательские данные не имеют предопределенных операторов в Lua, кроме оператора присвоения и сравнения на идентичность. Используя *метатаблицы*, программист может определить операции для значений full userdata (см. §2.4). Значения userdata не могут быть созданы или изменены в Lua, это возможно только через C API. Это гарантирует целостность данных, которыми владеет хост-программа.

Тип *thread* (поток) представляет независимый поток выполнения и используется для реализации сопрограмм (coroutine) (см. §2.6). Lua потоки это не реальные потоки операционной системы. Lua поддерживает сопрограммы на всех системах, даже на тех, где это не поддерживается операционной системой.

Тип *table* (таблица) реализует ассоциативные массивы, это значит, что массив может быть проиндексирован не только числами, но и любым Lua значением, кроме **nil** и NaN. (*Not a Number* специальное значение для представления неопределенных и непредставимых числовых результатов, таких как `0/0`.) Таблицы могут быть *гетерогенными* (разнородными); т.е. могут содержать значения всех типов (кроме **nil**). Любой ключ со значением **nil** не считается частью таблицы. И наоборот, любой ключ, не являющийся частью таблицы, имеет ассоциированное значение **nil**.

Таблицы единственный механизм структурирования данных в Lua; они могут использоваться для представления обычных массивов, последовательностей, таблиц символов, множеств, записей, графов, деревьев и т.п. Для представления записей, Lua использует имена полей как индекс. Язык поддерживает представление `a.name`, как синтаксическое украшение `a["name"]`. Существуют различные пути создания таблиц в Lua (см. §3.4.9).

Мы используем термин *последовательность* (sequence) чтобы обозначить таблицу, где все ключи это натуральные числа {1..*n*} (1,2,3,...), где *n* - длина последовательности (см. §3.4.7).

Как и индексы, значения полей в таблице могут быть любого типа. В частности, т.к. функции это первоклассные значения, поля таблицы могут содержать функции. Такие таблицы также могут содержать *методы* (см. §3.4.11).

Индексирование в таблицах следует принципу "сырого" (raw) равенства в языке. Выражения `a[i]` и `a[j]` определяют один и тот же элемент таблицы, если и только если `i` и `j` равны (raw equal) (значит, равны без метаметодов). В частности, вещественные числа (float) с целыми значениями равны соответствующим целым (integer), т.е., `1.0 == 1`. Во избежание неоднозначностей, любое реальное число с целым значением, которое испльзуется как ключ, конвертируется в соответствующее ему целое число. Например, если написать `a[2.0] = true`, фактически в таблицу будет вставлен целочисленный (integer) ключ `2`. С другой стороны, 2 and "2" разные Lua значения и следовательно обозначают разные данные в таблице.

Таблицы, функции, потоки и пользовательские данные (userdata) - это *объекты*: переменные фактически не *содержат* их значений, только *ссылки* на них. Присвоение, передача параметров и возврат из функций всегда манипулируют ссылками на эти значения; эти операции не подразумевают никакого типа копирования.

Библиотечная функция `type` возвращает строку с названием переданного ей типа (см. §6.1).

2.2 – Окружения и глобальное окружение

Как будет описано в §3.2 и §3.3.3, любая ссылка на свободное имя (т.е., имя не связанное ни с каким определением) `var` синтаксически транслируется в `_ENV.var`. Кроме того, каждый кусок (chunk) компилируется в области внешней локальной переменной, называемой `_ENV` (см. §3.3.2), таким

образом `_ENV` само никогда не бывает свободным именем в куске.

Несмотря на существование этой внешней переменной `_ENV` и трансляцию свободных имен, `_ENV` полностью регулярное имя. В частности, вы можете определить новые переменные и параметры с этим именем. Каждая ссылка на свободное имя использует `_ENV`, которая видима в данной точке программы, следуя обычным правилам видимости Lua (см. §3.5).

Любая таблица, используемая как значение переменной `_ENV`, называется *окружение* (environment).

Lua хранит особое окружение, называемое *глобальное окружение*. Это значение хранится по специальному индексу в C реестре (см. §4.5). В Lua, глобальная переменная `_G` инициализируется тем же значением. (`_G` никогда не используется непосредственно.)

Когда Lua загружает кусок (chunk), по умолчанию его `_ENV` upvalue присваивается значение глобального окружения (см. `load`). Следовательно, по умолчанию, свободные имена в Lua коде ссылаются на элементы в глобальном окружении (и, следовательно, они также называются *глобальными переменными*). Кроме того, все стандартные библиотеки загружаются в глобальное окружение и некоторые их функции действуют в этом окружении. Вы можете использовать `load` (или `loadfile`) для загрузки куска с другим окружением. (В C, вы загружаете кусок и затем изменяете его первое upvalue.)

2.3 – Обработка ошибок

Так как Lua встроенный язык расширений, все Lua действия начинаются с C кода; код в хостовой программе вызывает функцию в Lua библиотеке. (При использовании автономного интерпретатора Lua, программа `lua` выступает в качестве хоста.) Всякий раз, когда происходит ошибка компиляции или выполнения куска Lua кода, управление возвращается хостовой программе, которая может предпринять соответствующие меры (такие как печать сообщения об ошибке).

Lua код может явно сгенерировать ошибку, вызвав функцию `error`. Если вы нуждаетесь в перехвате ошибок в Lua, вы можете использовать `pcall` или `xpcall` для вызова функции в *защищенном режиме*.

При каждой ошибке, создается *объект ошибки* (также называемый *сообщением об ошибке*) с информацией об ошибке. Самостоятельно Lua генерирует только те ошибки, где объект ошибки содержит строку, но программы могут генерировать ошибки с любым значением в объекте ошибки. Объекты ошибки (исключения) пробрасываются вверх в Lua или в хост для обработки.

Когда вы используете `xpcall` или `lua_pcall`, вы можете определить *обработчик сообщений*, который будет вызываться в случае ошибок. Эта функция вызывается с оригинальным сообщением об ошибке и возвращает новое сообщение об ошибке. Она вызывается до раскрутки стека, так она сможет получить больше информации об ошибке, например, проверяя стек и создавая историю стека (stack traceback). Этот обработчик сообщений остается защищенным в защищенном вызове; так, ошибка в обработчике сообщений лишь вызовет его снова. Если этот цикл будет достаточно длинным, Lua прервет его и вернет соответствующее сообщение.

2.4 – Метатаблицы и метаметоды

Каждое значение в Lua может иметь *метатаблицу*. Эта *метатаблица* обычная Lua таблица, которая определяет поведение оригинального значения в определенных специальных операциях. Вы можете изменять различные аспекты поведения в операциях со значением, изменяя специфические поля 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.

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.

Вы можете запросить метатаблицу любого значения, используя функцию `getmetatable`.

Вы можете заменить метатаблицу таблицы, используя `setmetatable`. Нельзя изменять метатаблицы других типов из Lua кода (кроме, как используя библиотеку отладки (§6.10)); для этого вы должны использовать C API.

Таблицы и полные пользовательские данные (full userdata) имеют индивидуальные метатаблицы (хотя таблицы и пользовательские данные могут совместно использовать свои метатаблицы). Значения остальных типов используют одну метатаблицу на 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 (см. §6.4).

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 (§2.5).

Детальное описание событий, контролируемых метатаблицами, представлено ниже. Каждая операция идентифицируется соответсвующим именем события. Ключ для каждого события это строка начинающаяся с двух подчеркиваний, '__'; например, ключ для операции "add" строка "__add". Имейте ввиду, что запросы метаметодов всегда прямые; доступ к метаметоду не запускает других метаметодов

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. (Для большинства применений этот дополнительный операнд несущественнен.)

- **"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 (равенство). 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 регулярное, 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`. Метаметод ищется в объекте `table`.

Как и для индексированного доступа, метаметод для этого события может быть функцией или таблицей. Если это функция, то она вызывается с `table`, `key` и `value` в качестве аргументов. Если таблица, Lua производит индексированное присваивание для этой таблицы с тем же ключом и значением. Это присваивание регулярное, не прямое, и оно также может вызывать срабатывание другого метаметода.

Всякий раз, когда срабатывает метаметод "newindex", Lua не выполняет примитивное присваивание. (Если необходимо, метаметод может самостоятельно вызвать `rawset` для выполнения присваивания.)

- **"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`).

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 (см. §2.5.1).

2.5 – Сборка мусора

Lua выполняет автоматическое управление памятью. Это означает, что вы не должны беспокоиться о выделении памяти новым объектам или об освобождении памяти, когда объекты больше не нужны. Lua управляет памятью, запуская *сборщик мусора* для сборки всех *мёртвых объектов* (т.е., объектов более не доступных из Lua). Вся память, используемая Lua, подлежит автоматическому управлению: строки, таблицы, пользовательские данные, функции, потоки, внутренние структуры и т.д.

Lua реализует пошаговый отмечающий-и-очищающий сборщик. Для контроля циклов очистки мусора используются два числа: *пауза сборщика мусора* и *множитель шагов сборщика мусора*. Оба числа 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. Чем больше значение, тем менее агрессивен сборщик. Значения меньше 100 означают, что сборщик не останавливается перед началом нового цикла. Значение 200 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 по отношению к скорости выделения памяти. Большие значения делают сборщик более агрессивным, но также увеличивают размер каждого шага. Вы не должны 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.

Если вы установите множитель шагов очень большым (больше чем 10% от максимального числа байт, которые может использовать программа), сборщик поведет себя как останавливающий мир. Если вы установите паузу 200, сборщик будет вести себя как в старых версиях Lua, производя полную очистку каждый раз, когда Lua удваивает использование памяти.

Вы можете изменять эти числа, вызывая `lua_gc` в C или `collectgarbage` в Lua. Вы также можете использовать эти функции для прямого контроля сборщика (т.е., его остановки и перезапуска).

2.5.1 – Метаметоды сборки мусора

Вы можете установить метаметоды сборки мусора для таблиц и, используя C API, для полных пользовательских данных (см. §2.4). Эти метаметоды также называются *деструкторы* (finalizers).

Деструкторы позволяют координировать сборку мусора в Lua с внешним управлением ресурсами (таким как закрытие файлов, сетевый подключения, подключения к базам данных или освобождение вашей памяти).

Для объекта (таблицы или пользовательских данных) чтобы быть уничтоженным при сборке мусора, вы должны *отметить* его на уничтожение. Вы отмечаете объект на уничтожение, когда устанавливаете для него метатаблицу, содержащую поле "__gc". Имейте ввиду, что если вы установите метатаблицу без поля __gc и затем создадите это поле в метатаблице, то объект не будет отмечен на уничтожение.

Когда отмеченный объект становится мусором, он не уничтожается напрямую сборщиком мусора. Вместо этого, Lua ложит его в список. После завершения сборки, Lua проходит по этому списку. Для каждого объекта в списке проверяется метаметод __gc: если это функция, Lua вызывает её с объектом в качестве единственного аргумента; если метаметод не функция, Lua просто игнорирует его.

В конце каждого цикла сборки мусора, деструкторы объектов вызываются в порядке обратном порядку их отметки на уничтожение; т.е., первым деструктор будет вызван для последнего отмеченного на уничтожение объекта. Выполнение каждого деструктора может произойти в любое время при выполнении основного кода.

Так как объект подлежащий уничтожению должен быть использован в деструкторе, этот объект (и остальные объекты доступные через него) должны быть *воскрешены* Lua. Обычно, это воскрешение нерезидентно, и память объекта освобождается при следующем цикле сборки мусора. Тем не менее, если деструктор сохраняет объект в каком-то глобальном месте (т.е. глобальной переменной), воскрешение постоянно. Более того, если деструктор отмечает уничтожаемый объект для уничтожения снова, его деструктор будет вызван снова в следующем цикле, где объект не доступен. В любом случае, память объекта освобождается только в цикле сборки мусора, где объект

недоступен и не отмечен на уничтожение через деструктор.

Когда вы закрываете контекст (см. `lua_close`), Lua вызывает деструкторы всех объектов отмеченных на уничтожение, следуя порядку обратному порядку их отметки на уничтожение. Если любой деструктор отмечает объекты для уничтожения в этой фазе, эти отметки не имеют никакого эффекта.

2.5.2 – Слабые таблицы

Слабая таблица (weak table) - это таблица, элементы которой это *слабые ссылки* (weak references). Слабая ссылка игнорируется сборщиком мусора. Другими словами, если на объект существуют только слабые ссылки, то объект будет уничтожен сборщиком мусора.

Слабая таблица может иметь слабые ключи, слабые значения или и то и другое. Таблица со слабыми значениями позволяет уничтожать её значения, но препятствует уничтожению её ключей. Таблица со слабыми значениями и ключами позволяет уничтожать и ключи и значения. В любом случае, если ключ или значение уничтожены, эта пара удаляется из таблицы. Слабость таблицы контролируется полем `__mode` в её метатаблице. Если поле `__mode` это строка содержащая символ 'k', в таблице слабые ключи. Если поле `__mode` содержит 'v', в таблице слабые значения.

Таблица со слабыми ключами и сильными значениями называется *эфемерной таблицей* (ephemeron table). В эфемерной таблице, значение достижимо только если его ключ достижим. В частности, если ссылка на ключ приходит через его значение, пара удаляется.

Любое изменение слабости таблицы будет иметь эффект только в следующем цикле сборки мусора. В частности, если вы сменили слабый на сильный режим, Lua может продолжить сбор некоторых элементов из этой таблицы, пока изменения не будут иметь эффект.

Только объекты, имеющие явную конструкцию, удаляются из слабых таблиц. Значения, такие как числа и легкие C функции, не являются субъектами для сборки мусора, и следовательно не

удаляются из слабых таблиц (пока их ассоциированные значения не удалены). Хотя строки субъекты для сборки мусора, они не имеют явную конструкцию, и следовательно не удаляются из слабых таблиц.

Воскрешенные объекты (т.е., объекты подлежащие уничтожению и объекты достпупные через них) имеют специальное поведение в слабых таблицах. Они удаляются из слабых значений перед запуском их деструкторов, но удаляются из слабых ключей только в следующем цикле сборки, после запуска их деструкторов, когда эти объекты действительно освобождены. Это поведение позволяет деструктору получить доступ к свойствам, ассоциированным с объектом через слабые таблицы.

Если слабая таблица среди воскрешенных объектов в цикле сборки, она не может быть правильно очищена до следующего цикла сборки.

2.6 – Сопрограммы

Lua поддерживает сопрограммы, так называемую *совместную многопоточность*. Сопрограмма в Lua представляет независимый поток выполнения. В отличе от потоков в многопоточных системах, сопрограммма прерывает свое исполнение только явным вызовом функции yield (уступить).

Сопрограмма создается функцией `coroutine.create`. Единственный аргумент функции это главная функция сопрограммы. Функция `create` только создает сопрограмму и возвращает её описатель (объект типа *thread* - поток); она не запускает сопрограмму.

Сопрограмма запускается вызовом `coroutine.resume`. При первом вызове `coroutine.resume`, в качестве первого аргумента передается поток, возвращенный функцией `coroutine.create`, сопрограмма начинает свое исполнение с вызова своей главной функции. Дополнительные аргументы, переданные в `coroutine.resume`, передаются как аргументы этой функции. После запуска сопрограммы, она выполняется пока не будет завершена или не *уступит* (yield).

Сопрограмма может завершить свое исполнение двумя путями: нормально, когда её главная функция вернет управление (явно или не явно, после последней инструкции); и ненормально, если произойдет незащищенная ошибка. В случае нормального завершения, `coroutine.resume` вернет **true** и значения, возвращенные главной фунцией сопрограммы. В случае ошибок, `coroutine.resume` вернет **false** и сообщение об ошибке.

Сопрограмма уступает, вызывая `coroutine.yield`. Когда сопрограмма уступает, соответсвующая `coroutine.resume` немедленно возвращает управление, даже если уступка случилась внутри вложенной функции (т.е., не в главной функции, а в функции прямо или косвенно вызванной из главной функции). В случае уступки, `coroutine.resume` также возвращает **true** и значения, переданные в `coroutine.yield`. В следующий раз, возобновление этой же сопрограммы продолжает её выполнение с точки, где она уступила вызовом `coroutine.yield`, возвращающим дополнительные аргументы, переданные в `coroutine.resume`.

Как и `coroutine.create`, функция `coroutine.wrap` создает сопрограмму, но вместо сопрограммы возвращает функцию, вызов которой возобновляет сопрограмму. Аргументы, переданные этой функции, идут как дополнительные аргументы в `coroutine.resume`. `coroutine.wrap` возвращает все значения полученные от `coroutine.resume`, кроме первого (логический код ошибки). В отличие от `coroutine.resume`, `coroutine.wrap` не перехватывает ошибки; все ошибки передаются вызывающей стороне.

Для примера работы сопрограммы, рассмотрите следующий код:

```
function foo (a)
  print("foo", a)
  return coroutine.yield(2*a)
end
```

```
co = coroutine.create(function (a,b)
      print("co-body", a, b)
      local r = foo(a+1)
      print("co-body", r)
      local r, s = coroutine.yield(a+b, a-b)
      print("co-body", r, s)
      return b, "end"
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))
```

Запустив его, вы получите следующий результат:

```
co-body 1       10
foo     2
main    true    4
co-body r
main    true    11      -9
co-body x       y
main    true    10      end
main    false   cannot resume dead coroutine
```

Вы также можете создавать и манипулировать сопрограммами через C API: см. функции `lua_newthread`, `lua_resume` и `lua_yield`.

3 – Язык

Этот раздел описывает лексику, синтаксис и семантику Lua. Другими словами, этот раздел описывает какие лексемы (tokens) правильны, как они могут быть скомбинированы, и что их комбинации означают.

Языковые конструкции будут объясняться используя обычную расширенную БНФ нотацию, в которой {*a*} означает 0 или больше *a*, и [*a*] означает опциональную *a*. Нетерминалы показаны как non-terminal, ключевые слова показаны как **kword**, и другие терминальные символы показаны как '**=**'. Полный синтасис Lua описан в §9 в конце руководства.

3.1 – Лексические соглашения

Lua - это язык свободной формы. Он игнорирует пробелы (включая переходы на новую строку) и комментарии между лексическими элементами, кроме разделителей между именами и ключевыми словами.

Имена (также называемые *идентификаторами*) в Lua могут быть любой строкой из букв, цифр и подчеркиваний, не начинающейся с цифры. Идентификаторы используются для именования значений, полей таблиц и меток (labels).

Следующие *ключевые слова* зарезервированы и не могут использоваться как имена:

```
and       break     do        else      elseif    end
false     for       function  goto      if        in
local     nil       not       or        repeat    return
```

```
then      true      until     while
```

Язык Lua чувствителен к регистру символов: `and` - зарезервированное слово, но `And` и `AND` - два разных, допустимых имени. Как соглашение, программы должны избегать создания имен, которые начинаются с подчеркивания и следующими за ним одной или несколькими прописными буквами (например, `_VERSION`).

Следующие строки разделяют другие лексемы:

```
+     -     *     /     %     ^     #
&     ~     |     <<    >>    //
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]     ::
;     :     ,     .     ..    ...
```

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.

Строки в Lua могут содержать любое 8-битное значение, влючая встроенные нули, которые могут быть записаны как '`\0`'. Более того, возможно описать любой байт в литеральной строке его числовым значением. Это может быть сделано с помощью управляющей последовательности `\x`*XX*, где *XX* - это пара шестнадцатиричных цифр, или с помощью `\`*ddd*, где *ddd* - последовательность до

трех десятичных цифр. (Обратите внимание, что если десятичная управляющая последовательность сопровождается цифрой, то она должна содержать ровно три цифры.)

Unicode-символ в кодировке UTF-8 может быть вставлен в литеральную строку с помощью последовательности `\u{`*XXX*`}` (обратите внимание на обязательные фигурные скобки), где *XXX* - это одна или больше шестнадцатиричных цифр описывающих код символа.

Литеральные строки также могут быть определены используя *длинные скобки*. Мы определяем *открывающую длинную скобку уровня n*, как открывающую квадратную скобку, следующие за ней *n* знаков = и ещё одну открывающую квадратную скобку. Так, открывающая длинная скобка уровня 0 запишется так: `[[`, для уровня 1 - `[=[`, и так далее. *Закрывающая длинная скобка* определяется аналогично; например, закрывающая длинная скобка уровня 4 запишется так: `]====]`. *Длинный литерал* начинается с открывающей длинной скобки любого уровня и завершается на первой закрывающей длинной скобке того же уровня. Это позволяет содержать любой текст, кроме закрывающей скобки того же уровня. Литералы в такой скобочной форме могут занимать несколько линий (строк), управляющие последовательности в таких строках не работают, длинные скобки других уровней игнорируются. Любой вид последовательности завершения строки (\r, \n, \r\n или \n\r) конвертируется в простой перевод строки \n.

Любой байт в литеральной строке не подвержен влиянию правил своего представления. Тем не менее, Lua открывает файлы для парсинга в текстовом режиме, и функции системы могут иметь проблемы с некоторыми управляющими символами. Поэтому, для безопасного представления не текстовых данных в строке, следует использовать управляющие последовательности.

Для удобства, когда открывается длинная скобка с непосредственным переводом строки за ней, перевод строки не включается в результирующую строку. Например, в системе использующей ASCII (в которой 'a' кодируется как 97, newline кодируется как 10 и '1' кодируется как 49), пять литеральных строк ниже кодируют одинаковые строки:

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

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. Примеры допустимых целых чисел:

```
3   345   0xff   0xBEBADA
```

Примеры допустимых вещественных чисел:

```
3.0     3.1416     314.16e-2     0.31416E1     34e1
0x0.1E  0xA23p-4   0X1.921FB54442D18P+1
```

Комментарии начинаются с двойного тире (`--`) в любом месте за пределами литеральной строки. Если текст, непосредственно следующий за `--`, не открывающая длинная скобка, то это *короткий комментарий*, который продолжается до конца строки. Иначе, это *длинный комментарий*, который продолжается до соответствующей закрывающей длинной скобки. Длинные комментарии часто используются для временного отключения кода.

3.2 – Переменные

Переменные - это место где хранятся значения. Существует три вида переменных: глобальные переменные, локальные переменные и поля таблиц.

Одно имя может означать глобальную или локальную переменную (или формальный параметр функции, который является частным случаем локальной переменной):

```
var ::= Name
```

Name означает идентификаторы, как описано в §3.1.

Любое имя переменной предполагается глобальным, пока явно не определено локальным (см. §3.3.7). Локальные переменные *лексически ограниченные*: локальные переменные свободно доступны функциям, определенным внутри их области видимости (см. §3.5).

Перед первым присваиванием переменной, её значение равно **nil**.

Квадратные скобки используются для индексирования в таблице:

```
var ::= prefixexp ‘[’ exp ‘]’
```

Значение доступа к полям таблицы может быть изменено через метатаблицы. Доступ к индексированной переменной `t[i]` эквивалентно вызову `gettable_event(t,i)`. (См. §2.4 для полного описания функции `gettable_event`. Эта функция не определена и не используется в Lua. Мы используем её только для пояснения.)

Синтаксис `var.Name` - это лишь семантическое украшение для `var["Name"]`:

```
var ::= prefixexp '.' Name
```

Доступ к глобальной переменной x эквивалентен `_ENV.x`. в силу того, что блоки компилируются, `_ENV` никогда не является глобальным именем (см. §2.2).

3.3 – Выражения

Lua поддерживает почти стандартный набор выражений, подобный наборам в Pascal или C. Этот набор включает в себя присваивания, управляющие структуры, вызовы функций и определения переменных.

3.3.1 – Блоки

Блок - это список выражений, которые выполняются последовательно:

```
block ::= {stat}
```

Lua допускает *пустые выражения*, что позволяет вам разделять выражения с помощью точки с запятой (;), начинать блок с точки с запятой или писать две точки с запятой подряд:

```
stat ::= ';'
```

Вызовы функций и присваивания могут начинаться с открывающейся скобки. Эта возможность ведет к неоднозначности в грамматике Lua. Рассмотрим следующий фрагмент:

```
a = b + c
(print or io.write)('done')
```

Грамматика может рассматривать это двумя путями:

```
a = b + c(print or io.write)('done')

a = b + c; (print or io.write)('done')
```

Текущий парсер всегда рассматривает такие констркции первым путем, интерпретируя открывающуюся скобку, как начало аргументов для вызова. Для избежания этой неоднозначности, лучше всегда ставить точку с запятой (;) перед выражениями начинающимися со скобок:

```
;(print or io.write)('done')
```

Блок может быть явно выделен для создания единственного выражения:

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

Явные блоки полезны для контроля за областью видимости переменных. Явные блоки также иногда используются для добавления выражения **return** в середину другого блока (см. §3.3.4).

3.3.2 – Куски

Единица компиляции в Lua называется *куском* (chunk). Синтаксически, кусок это простой блок:

```
chunk ::= block
```

Lua обрабатывает кусок, как тело анонимной функции с переменным числом аргументов (см. §3.4.11). Как таковой, кусок может определять локальные переменные, получать аргументы и возвращать значения. Более того, такая анонимная функция компилируется в области видимости внешней локальной переменной `_ENV` (см. §2.2). Результирующая функция всегда имеет `_ENV`, как только свою upvalue, даже если не использует эту переменную.

Кусок может храниться в файле или в строке внутри хостовой программы. Для запуска куска, Lua сперва *загружает* его (load), прекомпилируя код куска в инструкции виртуальной машины, и затем запускает скомпилированный код.

Кусок также может быть прекомпилирован в бинарную форму; смотри программу `luac` и функцию `string.dump`. Программы в исходном коде и в скомпилированной форме взаимозаменяемы; Lua автоматически определяет тип файла и действует соответственно (см. `load`).

3.3.3 – Присваивание

Lua позволяет множественные присваивания. Синтаксис присваивания определяет список переменных с левой строны и список выражений с правой стороны. Элементы в обоих списках разделяются запятыми:

```
stat ::= varlist ‘=’ explist
varlist ::= var {‘,’ var}
explist ::= exp {‘,’ exp}
```

Выражения обсуждаются в §3.4.

Перед присваиванием, список значений *корректируется* до длины списка переменных. Если значений больше чем нужно, лишние значения отбрасываются. Если значений меньше чем нужно, список расширяется добавлением необходимого числа значений **nil**. Если список выражений заканчивается вызовом функции, все значения, возвращенные этой функцией, попадают в список значений перед присваиванием. (кроме вызова заключенного в скобки; см. §3.4).

Оператор присваивания сперва вычисляет все свои выражения и лишь затем выполняет присваивание. Таким образом код

```
i = 3
i, a[i] = i+1, 20
```

устанавливает a[3] равным 20, не изменяя a[4], т.к. i в a[i] вычисляется (= 3) до присваивания ему 4. Аналогично, строка

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

меняет местами значения x и y, и

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

циклически переставляет значения x, y и z.

Значение присваиваний глобальным переменным и полям таблиц может быть изменено через метатаблицы. Присваивание индексированной переменной `t[i] = val` эквивалентно `settable_event(t,i,val)`. (См. §2.4 для полного описания функции `settable_event`. Эта функция не определена и не используется в Lua. Мы используем её только для пояснения.)

Присванивание глобальной переменной `x = val` эквивалентно присваиванию `_ENV.x = val` (см. §2.2).

3.3.4 – Управляющие конструкции

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

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else block] end
```

Lua также имеет две формы оператора **for** (см. §3.3.5).

Условное выражение в управляющей конструкции может возвращать любое значение. **False** и **nil** рассматриваются как false. Все значения, отличные от **nil** и **false**, рассматриваются как true (в частности, число 0 и пустая строка это тоже true).

В цикле **repeat**–**until**, внутренний блок заканчивается после всего условия, а не на ключевом слове **until**. Так, условие может ссылаться на локальные переменные объявленные внутри цикла.

Оператор **goto** передает управление на метку. По синтаксическим причинам, метки в Lua тоже считаются выражениями:

```
stat ::= goto Name
stat ::= label
label ::= ‘::’ Name ‘::’
```

Метка видна в блоке где объявлена, за исключением вложенных блоков, где определена метка с таким же именем и вложенных функций. Goto может переходить на любую видимую метку, так далеко, пока не войдет в область видимости локальной переменной.

Метки и пустые выражения называются *пустыми выражениями*, так как они не производят дейтсвий.

Выражение **break** завершает исполнение цикла **while**, **repeat** или **for**, пропуская оставшиеся команды цикла:

```
stat ::= break
```

break завершает самый внутренний цикл.

Выражение **return** используется для возврата значений из функции или куска (который является

анонимной функцией). Функции могут возвращать несколько значений, поэтому синтаксис для выражения **return** следующий:

```
stat ::= return [explist] [';']
```

Выражение **return** может быть написано только, как последнее в блоке. Если действительно необходимо иметь **return** в середине блока, тогда можно использовать явный внутренний блок, как идиома `do return end`, теперь **return** это последнее выражение в его (внутреннем) блоке.

3.3.5 – Конструкция For

Конструкция **for** имеет две формы: цифровую и общую.

Цифровой цикл **for** повторяет блок кода, пока управляющая переменная изменяется в арифметической прогрессии. Он имеет следующий синтаксис:

```
stat ::= for Name '=' exp ',' exp [',' exp] do block end
```

block повторяется для *name* начиная с первого значения *exp*, пока не достигнет значения втрого *exp*, шагами равными третьему *exp*. Более точно, выражение **for**

```
for v = e1, e2, e3 do block end
```

эквивалентно следующему коду:

```
do
  local var, limit, step = tonumber(e1), tonumber(e2), tonumber(e3)
  if not (var and limit and step) then error() end
  var = var - step
```

```
    while true do
      var = var + step
      if (step >= 0 and var > limit) or (step < 0 and var < limit) then
        break
      end
      local v = var
      block
    end
  end
```

Имейте ввиду следующее:

- Все три управляющие выражения вычисляются только один раз, перед тем как начнется цикл. Их результаты должны быть числами.
- *var*, *limit* и *step* - невидимые переменные. Имена показаны здесь только для пояснения.
- Если третье выражение (шаг) отсутствует, тогда используется шаг 1.
- Для выхода из цикла **for** вы можете использовать **break** и **goto**.
- Переменная цикла v является локальной для тела цикла. Если необходимо использовать её значение после цикла, сохраните его в другой переменной перед выходом из цикла.

Общий **for** работает через функции, называемые *итераторами*. На каждой итерации, функция-итератор вызывается чтобы выдать новое значение, остановка происходит, когда новое значение равно **nil**. Общий цикл **for** имеет следующий синтаксис:

```
stat ::= for namelist in explist do block end
namelist ::= Name {‘,’ Name}
```

Выражение **for**

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

эквивалентно следующему коду:

```
do
  local f, s, var = explist
  while true do
    local var_1, ···, var_n = f(s, var)
    if var_1 == nil then break end
    var = var_1
    block
  end
end
```

Имейте ввиду следующее:

- `explist` вычисляется только один раз. Его результат это функция *итератор*, *состояние* и начальное значение для первой *итерируемой переменной*.
- `f`, `s` и `var` - невидимые переменные. Имена показаны здесь только для пояснения.
- Для выхода из цикла **for** вы можете использовать **break**.
- Переменная цикла `var_i` является локальной для тела цикла. Если необходимо использовать её значение после цикла, сохраните его в другой переменной перед выходом из цикла.

3.3.6 – Вызовы функций как выражения

Чтобы позволить побочные эффекты, вызовы функций могут запускаться как выражения:

```
stat ::= functioncall
```

В этом случае все возвращенные значения отбрасываются. Вызовы функций рассматриваются в §3.4.10.

3.3.7 – Локальные определения

Локальные переменные могут быть объявлены в любой части блока. Определение может содержать начальное присваивание (инициализацию):

```
stat ::= local namelist [‘=’ explist]
```

Если есть начальное присваивание, то оно имеет семантику схожую с множественным присваиванием (см. §3.3.3). Иначе, все переменные инициализируются значением **nil**.

Кусок также является блоком (см. §3.3.2), и следовательно локальные переменные могут быть объявлены в куске вне всякого явного блока.

Правила видимости локальных переменных рассматриваются в §3.5.

3.4 – Выражения

Базовые выражения в Lua следующие:

```
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Numeral
exp ::= LiteralString
exp ::= functiondef
exp ::= tableconstructor
```

```
exp ::= ‘...’
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | ‘(’ exp ‘)’
```

Числа и литеральные строки рассматриваются в §3.1; переменные рассматриваются в §3.2; определения функций рассматриваются в §3.4.11; вызовы функций рассматриваются в §3.4.10; конструкторы таблиц описаны в §3.4.9. Выражения с переменными аргументами, обозначенными тремя точками ('...'), могут быть использованы только внутри функции с переменным числом аргументов; они описываются в §3.4.11.

Бинарные операторы, включая арифметические (см. §3.4.1), битовые операторы (см. §3.4.2), операторы сравнения (см. §3.4.4), логические операторы (см. §3.4.5), и оператор конкатенации (см. §3.4.6). Одноместные операторы, включая одноместный минус (см. §3.4.1), одноместный битовый НЕ (см. §3.4.2), одноместный логический **not** (см. §3.4.5), и одноместный *оператор длины* (см. §3.4.7).

Функции и выражения с переменным числом аргументов могут возвращать несколько значений. Если вызов функции используется как выражение (см. §3.3.6), он возвращает список скорректированный до нуля элементов, т.е. отброшены все возвращенные значения. Если выражение используется последним элементом в списке выражений (или оно только одно), то корректировка не производится (если выражение не заключено в скобки). Во всех остальных случаях Lua корректирует список результатов до одного элемента, отбрасывая все значения кроме первого, или добавляя один **nil**, если значений нет.

Здесь несколько примеров:

```
f()                -- корректируется до 0 результатов
g(f(), x)          -- f() корректируется до 1 результата
g(x, f())          -- g получает x и все результаты из f()
```

```
a,b,c = f(), x     -- f() корректируется до 1 результата (c получает nil)
a,b = ...          -- a получает первый множественный параметр, b получает
                   -- второй (a и b могут получить nil, если в множестве параметров нет
                   -- соответствующего параметра)

a,b,c = x, f()     -- f() корректируется до 2 результатов
a,b,c = f()        -- f() корректируется до 3 результатов
return f()         -- возвращает все результаты из f()
return ...         -- возвращает все полученные переменные параметры
return x,y,f()     -- возвращает x, y и все результаты из f()
{f()}              -- создает список с результатами из f()
{...}              -- создает список со всеми переменными параметрами
{f(), nil}         -- f() корректируется до 1 результата
```

Любое выражение, заключенное в скобки, всегда возвращает только одно значение. Т.е., `(f(x,y,z))` всегда одно значение, даже если `f` возвращает несколько значений. (Значение `(f(x,y,z))` это первое из значений возвращенных `f` или **nil**, если `f` не вернет ни одного значения.)

3.4.1 – Арифметические операторы

Lua поддерживает следующие арифметические операторы:

- **+:** сложение
- **-:** вычитание
- ***:** умножение
- **/:** деление
- **//:** целочисленное деление
- **%:** модуль

- **^:** возведение в степень
- **-:** одноместный минус

За исключением возведения в степень и деления, арифметические операторы работают следующим образом: Если оба операнда целые, операция производится над целыми и результатом является целое. Иначе, если оба операнда числа или строки, которые могут быть сконвертированы в числа (см. §3.4.3), они конвертируются в числа с плавающей запятой (float), операции производятся согласно обычным правилам арифметики вещественных чисел (обычно стандарт IEEE 754), и результатом будет число с плавающей запятой.

Возведение в степень и деление (`/`) всегда конвертирует операнды в вещественные числа и результат вещественное число. Возведение в степень использует функцию `pow` из ISO C, она работает и для не целых экспонент тоже.

Целочисленное деление (`//`) - это деление, которое округляет частное по отношению к минус беспонечности, т.е. нижнее от деления операндов.

Модуль определено как остаток от деления, которое округляет частное по отношению к минус беспонечности (целочисленное деление).

В случае переполнения целочисленной арифметики, все операции *циклически переходят*, в соответствии с обычными правилами арифметики двоичного дополнения. (Другими словами, они возвращают уникальное представимое целое, которое равно по модулю 2^{64} математическому результату.)

3.4.2 – Битовые операторы

Lua поддерживает следующие битовые операторы:

- **&:** битовое И
- **|:** битовое ИЛИ
- **~:** битовое ИЛИ НЕ
- **>>:** правый сдвиг
- **<<:** левый сдвиг
- **~:** одноместное битовое НЕ

Все битовые операторы преобразуют свои операнды в целые (см. §3.4.3), оперируют всеми битами этих целых и результат тоже целое.

Правый и левый сдвиги заполняют свободные биты нулями. Отрицательное смещение сдвигает в противоположном направлении; смещения с абсолютными значениями большими (или равными), чем число бит в целом, дают результат ноль (все биты сдвинуты наружу).

3.4.3 – Приведения и преобразования

Lua производит некоторые автоматические преобразования между типами и представлениями во время выполнения. Битовые операторы всегда конвертируют вещественные операнды в целые. Возведение в степень и деление всегда конвертируют целые операнды в вещественные. Все остальные арифметические операции, примененные к смешанным числам (целые и вещественные), преобразуют целые числа в вещественные; это называется *обычное правило*. C API также конвертирует целые в вещественные и вещественные в целые, при необходимости. Более того, конкатенация строк принимает числа, как аргументы между строками.

Lua также конвертирует строки в числа, когда ожидается число.

В преобразовании целого в вещественное, если целое значение имеет точное представление как вещественное, это представление и будет результатом. Иначе, преобразование получает ближайшее

большее или ближайшее меньшее представимое значение. Этот тип преобразования всегда успешен.

Преобразование вещественного в целое проверяет, что вещественное имеет точное представление как целое (т.е., вещественное имеет целое значение и в диапазоне представления целых). Если это так, то это представление будет результатом. Иначе, преобразование терпит неудачу.

Преобразование строк в числа идет следующим образом: Первое, строка преобразуется в целое или вещественное, следуя синтаксису и правилам лексера Lua. (Строка может содержать начальные и конечные пробелы и знак.) Затем, получившееся число (целое или вещественное) преобразуется в тип (целое или вещественное), требуемый в данном контексте (т.е., требуемый операцией которая конвертирует принудительно).

Преобразование чисел в строки использует не определенный читабельный формат. Для полного контроля над тем, как числа конвертируются в строки, используйте функцию `format` из строковой библиотеки (см. `string.format`).

3.4.4 – Операторы сравнения

Lua поддерживает следующие операторы сравнения:

- **==:** равенство
- **~=:** неравенство
- **<:** меньше
- **>:** больше
- **<=:** меньше или равно
- **>=:** больше или равно

Эти операторы всегда возвращают **false** или **true**.

Равенство (==) сперва сравнивает типы операндов. Если типы разные, то результат **false**. Иначе, сравниваются значения операндов. Строки сравниваются очевидным способом. Числа равны, если они обозначают одинаковые математические значения.

Таблицы, пользовательские данные и потоки сравниваются по ссылке: два объекта считаются равными только, если это один и тот же объект. Всегда, когда вы создаете новый объект (таблицу, пользовательские данные или поток), этот новый объект отличен от любого существующего объекта. Замыкания (closure) с одинаковыми ссылками всегда равны. Замыкания с любыми обнаруживаемыми различиями (разное поведение, разное определение) всегда различны.

Используя метаметод "eq", вы можете изменить способ, которым Lua сравнивает таблицы и пользовательские данные (см. §2.4).

Проверка на равенство не конвертирует строки в числа и наоборот. Так, `"0"==0` вернет **false**, `t[0]` и `t["0"]` означают разные элементы в таблице.

Оператор ~= это явное отрицание равенства (==).

Операторы упорядочения работают следующим образом. Если оба аргумента числа, то они сравниваются согласно их математическим значениям. (независимо от их подтипов). Иначе, если оба аргумента строки, то их значения сравниваются согласно текущей локали. Иначе, Lua пытается вызвать метаметоды "lt" или "le" (см. §2.4). Сравнение `a > b` транслируется в `b < a` и `a >= b` транслируется в `b <= a`.

Следуя стандарту IEEE 754, NaN рассматривается как никакой: не меньший, не больший, не равный ни одному значению (включая себя самого).

3.4.5 – Логические операторы

В Lua существуют следующие логические операторы: **and** (И), **or** (ИЛИ) и **not** (НЕ). Как и управляющие структуры (см. §3.3.4), все логические операторы считают **false** и **nil** как false, а все остальное как true.

Оператор отрицания **not** всегда возвращает **false** или **true**. Оператор конъюнкции **and** возвращает свой первый аргумент, если его значение **false** или **nil**; иначе, **and** возвращает второй аргумент. Оператор дизъюнкции **or** возвращает свой первый аргумент, если его значение отлично от **nil** и **false**; иначе, **or** возвращает второй аргумент. Оба **and** и **or** используют ленивое вычисление; т.е., второй операнд вычисляется только если необходимо. Несколько примеров:

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

(В этом руководстве, `-->` означает результат предыдущего выражения.)

3.4.6 – Конкатенация

Конкатенация строк в Lua обозначается двумя точками (`'..'`). Если оба операнда строки или числа, они конвертируются в строки согласно правилам, описанным в §3.4.3. Иначе, вызывается метаметод `__concat` (см. §2.4).

3.4.7 – Оператор длины

Оператор длины обозначается одноместным префиксом #. Длина строки это число байт в ней (т.е., обычное значение длины строки, когда каждый символ размером 1 байт).

Программа может модифицировать поведение оператора длины для любого значения, кроме строк, используя метаметод `__len` (см. §2.4).

Пока метаметод `__len` не определен, длина таблицы `t` определена только, если таблица `t` *последовательность*, т.е., набор её положительных числовых ключей равен *{1..n}* для какого-то положительного целого *n*. В этом случае, *n* это длина. Имейте ввиду, таблица как

```
{10, 20, nil, 40}
```

это не последовательность, т.к. она имеет ключ `4`, но не имеет ключ `3`. (Так, там нет *n*, который в множестве *{1..n}* равен набору положительных числовых ключей этой таблицы.) Тем не менее, эти не числовые ключи не создают помех с независимой последовательностью в таблице.

3.4.8 – Приоритет

Приоритет операторов в Lua представлен далее, с меньшего к большему приоритету:

```
or
and
<     >     <=    >=    ~=    ==
|
~
&
<<    >>
```

```
..
+     -
*     /     //    %
unary operators (not   #     -     ~)
^
```

Как обычно, чтобы изменить приоритет в выражении, вы можете использовать скобки. Операторы конкатенации ('`..`') и возведения в степень ('^') имеют правую ассоциативность. Все остальные бинарные операторы имеют левую ассоциативность.

3.4.9 – Конструкторы таблиц

Конструкторы таблиц - это выражения, которые создают таблицы. При каждом запуске конструктора создается новая таблица. Конструктор может использоваться для создания пустой таблицы или для создания таблицы с инициализацией некоторых её полей. Общий синтаксис конструкторов следующий:

```
tableconstructor ::= ‘{’ [fieldlist] ‘}’
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp
fieldsep ::= ‘,’ | ‘;’
```

Каждое поле вида `[exp1] = exp2` добавляет новый элемент таблицы с ключем `exp1` и значением `exp2`. Поле вида `name = exp` эквивалентно записи `["name"] = exp`. Поля вида `exp` эквивалентны `[i] = exp`, где `i` последовательные целые, начинающиеся с 1; поля в других форматах не влияют на этот счет. Например,

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

эквивалентно

```
do
  local t = {}
  t[f(1)] = g
  t[1] = "x"         -- первое exp
  t[2] = "y"         -- второе exp
  t.x = 1            -- t["x"] = 1
  t[3] = f(x)        -- третье exp
  t[30] = 23
  t[4] = 45          -- четвертое exp
  a = t
end
```

Порядок присваиваний в конструкторе не определен. (Этот порядок важен только для повторяющихся ключей.)

Если последнее поле в списке имеет вид `exp` и выражение это вызов функции или обозначение множества аргументов (...), то все значения, возвращенные этим выражением, последовательно включаются в список (см. §3.4.10).

Список полей может иметь опциональный конечный разделитель, для удобства машинно-сгенерированного кода.

3.4.10 – Вызовы функций

Вызов функции в Lua имеет следующий синтаксис:

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

При вызове функции, сперва вычисляются prefixexp и args. Если значение prefixexp имеет тип *function*, то вызывается эта функция с данными аргументами. Иначе, для объекта prefixexp вызывается метаметод "call", в качестве первого параметра передается значение prefixexp, затем остальные аргументы вызова. (см. §2.4).

Форма

```
functioncall ::= prefixexp ‘:’ Name args
```

может быть использована для вызова "методов". Вызов `v:name(`*`args`*`)` это синтаксическое украшение для `v.name(v,`*`args`*`)`, за исключением того, что `v` вычисляется только один раз.

Аргументы имеют следующий синтаксис:

```
args ::= ‘(’ [explist] ‘)’
args ::= tableconstructor
args ::= LiteralString
```

Все выражения в аргументах вычисляются перед вызовом.

Вызов вида `f{`*`fields`*`}` это синтаксическое украшение для `f({`*`fields`*`})`; т.е. в качестве аргумента передается одна новая таблица.

Вызов вида `f'`*`string`*`'` (или `f"`*`string`*`"` или `f[[`*`string`*`]]`) это синтаксическое украшение для `f('`*`string`*`')`; т.е. в качестве аргумента передается одна литеральная строка.

Вызов вида `return` *`functioncall`* называется *хвостовым вызовом* (tail call). Lua реализует *соответствующие хвостовые вызовы* (proper tail call) (или *соответствующую хвостовую рекурсию*): в хвостовом вызове, вызванная функция повторно использует элементы стека

вызывающей функции. Следовательно, не существует лимита на число вложенных хвостовых вызовов, выполняемых программой. Тем не менее, хвостовой вызов затирает любую отладочную информацию о вызывающей функции. Имейте ввиду, что хвостовой вызов происходит только с конкретным синтаксисом, где **return** имеет только один вызов функции в качестве аргумента; при таком синтаксисе вызывающая функция возвращает тоже, что и вызванная функция. Так, в следующих примерах хвостовой вызов не происходит:

```
return (f(x))        -- результаты корректируются до 1
return 2 * f(x)
return x, f(x)       -- дополнительные результаты
f(x); return         -- результаты отброшены
return x or f(x)     -- результаты корректируются до 1
```

3.4.11 – Определения функций

Синтаксис для определения функции:

```
functiondef ::= function funcbody
funcbody ::= ‘(’ [parlist] ‘)’ block end
```

Следующее синтаксическое украшение упрощает определение функций:

```
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name {‘.’ Name} [‘:’ Name]
```

Выражение

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

транслируется в

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

Выражение

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

транслируется в

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

Выражение

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

транслируется в

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

но, не в

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

(Это имеет значение, когда тело функции содержит ссылки на `f`.)

Определение функции - это исполняемое выражение, значение которого имеет тип *function*. Когда Lua предкомпилирует кусок, все тела функций в нем прекомпилируются тоже. Затем, всякий раз, когда Lua запускает определение функции, функция *инстанцируется* (или *замыкается*). Этот экземпляр функции (или *замыкание* [closure]) будет финальным значением выражения.

Параметры действуют как локальные переменные, инициализированные значениями аргументов:

```
parlist ::= namelist [‘,’ ‘...’] | ‘...’
```

Когда функция вызывается, список аргументов корректируется до длины списка параметров, пока функция это не *функция с переменным числом аргументов* (vararg function); переменные аргументы указываются тремя точками ('...') в конце списка параметров. Функция с переменным числом аргументов не корректирует свой список аргументов; взамен, она коллекционирует все дополнительные аргументы и передает их в функцию через *vararg-выражение*, которое также обозначается тремя точками. Значение этого выражения это список всех фактических дополнительных аргументов, подобно функции с множественными возвращаемыми значениями. Если vararg-выражение используется внутри другого выражения или в середине списка выражений, то его результат корректируется до одного элемента. Если vararg-выражение используется как последний элемент в списке выражений, то корректировка не производится (если последнее выражение не заключено в скобки).

Например, рассмотрим следующие определения:

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

Мы получим следующее соответствие аргументов параметрам и vararg-выражению:

```
Вызов           Пааметры

f(3)            a=3, b=nil
f(3, 4)         a=3, b=4
f(3, 4, 5)      a=3, b=4
```

```
f(r(), 10)       a=1, b=10
f(r())           a=1, b=2

g(3)             a=3, b=nil, ... -->  (пусто)
g(3, 4)          a=3, b=4,   ... -->  (пусто)
g(3, 4, 5, 8)    a=3, b=4,   ... -->  5  8
g(5, r())        a=5, b=1,   ... -->  2  3
```

Результаты возвращаются выражением **return** (см. §3.3.4). Если управление достигает конца функции без выражения **return**, то функция не возвращает результатов.

Существует системозависимый предел числа значений, которые может вернуть функция. Этот предел гарантированно больше 1000.

Синтаксис *двоеточия* используется для определения *методов*, т.е. функций, которые имеют явный дополнительный параметр `self`. Таким образом, выражение

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

это синтаксическое украшение для

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

3.5 – Правила видимости

Lua - язык с лексическими областями видимости. Область видимости локальной переменной начинается с первого выражения после её определения и продолжается до последнего не пустого выражения внутреннего блока, включающего определение. Рассмотрим следующий пример:

```
x = 10                -- глобальная переменная
do                    -- новый блок
  local x = x         -- новый 'x', со значением 10
  print(x)            --> 10
  x = x+1
  do                  -- другой блок
    local x = x+1     -- другой 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (глобальная переменная)
```

Учтите, что в определении вида `local x = x`, новый x пока вне зоны видимости и, таким образом, второй x ссылается на внешнюю переменную.

Согласно правилам видимости, локальные переменные могут быть свободно доступны функциям, определенным в их зоне видимости. Локальная переменная, которая используется внутренней функцией, внутри данной функции называется *upvalue* или *внешняя локальная переменная*.

Обратите внимание, каждое выполнение выражения **local** определяет новую локальную переменную. Рассмотрим следующий пример:

```
a = {}
local x = 20
for i=1,10 do
  local y = 0
  a[i] = function () y=y+1; return x+y end
```

```
end
```

Цикл создает 10 замыканий (т.е., 10 экземпляров анонимной функции). Каждое из этих замыканий использует разные переменные y и общую для всех x.

4 – Программный интерфейс (API)

Этот раздел описывает C API для Lua, т.е. набор C функций, доступный хостовой программе для взаимодействия с Lua. Все API функции и связанные типы и константы определены в заголовочном файле `lua.h`.

Каждый раз, когда мы используем термин "функция", любая возможность в API может быть реализована как макрос. Везде, где не оговорено обратное, все такие макросы используют каждый свой аргумент только один раз (исключая первый аргумент, который всегда является контекстом Lua) и не генерируют ни каких скрытых побочных эффектов.

Как и в большинстве C библиотек, функции Lua API не проверяют свои аргументы на корректность и согласованность. Тем не менее, вы можете изменить это поведение, скомпилировав Lua с включенным определением `LUA_USE_APICHECK`.

4.1 – Стек

Чтобы передавать значения из и в C Lua использует *виртуальный стек*. Каждый элемент этого стека представляет Lua значение (**nil**, number, string и др.).

Кажды раз, когда Lua вызывает C, вызванная функция получает новый стек, который независим от

предыдущих стеков и стеков активных в данный момент C функций. Этот стек изначально содержит все аргументы для C функции, туда же C функция ложит свои результаты, чтобы вернуть вызывающей стороне (см. `lua_CFunction`).

Для удобства, большинство операций запроса в API не следуют строгой стековой дисциплине. Напротив, они могут ссылаться на любой элемент в стеке, используя его *индекс*: Положительный индекс представляет абсолютную позицию в стеке (начиная с 1); отрицательный индекс представляет смещение относительно вершины стека. Точнее, если стек имеет *n* элементов, индекс 1 представляет первый элемент (т.е., элемент, который был положен на стек первым) и индекс *n* представляет последний элемент; индекс -1 также представляет последний элемент (т.е., элемент на вершине стека) и индекс *-n* представляет первый элемент.

4.2 – Размер стека

Когда вы работаете с Lua API, вы ответственны за гарантирование согласованности. В частности, ***вы ответственны за контроль над переполнением стека***. Вы можете использовать функцию `lua_checkstack` чтобы проверять, что стек имеет достаточно места для новых элементов.

Каждый раз, когда Lua вызывает C, он убеждается, что стек имеет место для как минимум `LUA_MINSTACK` дополнительных слотов. `LUA_MINSTACK` определен равным 20, так обычно вы не должны беспокоиться о размере стека, пока ваш код не содержит циклы, помещающие элементы в стек.

Когда вы вызываете Lua функцию без определенного числа результатов (см. `lua_call`), Lua гарантирует, что стек имеет достаточно места для помещения всех результатов, но не гарантирует любое дополнительное пространство. Так, перед тем как положить что-либо на стек после вызова функции, вы должны использовать `lua_checkstack`.

4.3 – Правильные и допустимые индексы

Любая API функция, получающая стековые индексы, работает только с *правильными индексами* или *допустимыми индексами*.

Правильный индекс - это индекс, который ссылается на позицию, содержащую модифицируемое Lua значение. Допустимый диапазон включает стековые индексы между 1 и вершиной стека (`1 ≤ abs(index) ≤ top`) плюс *псевдоиндексы*, которые представляют некоторые позиции, доступные для C кода, но не находящиеся в стеке. Псевдоиндексы используются для доступа к реестру (см. §4.5) и к внешним локальным переменным (upvalue) C функции (см. §4.4).

Функции, не нуждающиеся в специфической изменяемой позиции, а нуждающиеся только в значении (т.е., функции запросов), могут быть вызваны с допустимыми индексами. *Допустимым индексом* может быть любой правильный индекс, но это также может быть любой положительный индекс после вершины стека внутри пространства, выделенного для стека, т.е., индексы выше размера стека. (Учтите, что 0 это всегда недопустимый индекс.) Если не оговорено иное, API функции работают с допустимыми индексами.

Допустимые индексы служат для избежания дополнительных тестов вершины стека при запросах. Например, C функция может запросить свой третий аргумент без проверки его существования, т.е., без проверки того, что индекс 3 является правильным.

Для функций, которые могут быть вызваны с допустимыми индексами, любой не правильный индекс, трактуется как индекс к значению виртуального типа `LUA_TNONE`, которое ведет себя как значение nil.

4.4 – C замыкания

Когда C функция создана, можно ассоциировать с ней несколько значений, таким образом создав *C замыкание* (closure) (см. `lua_pushcclosure`); эти значения называются *upvalue* и доступны функции во время вызова.

При каждом вызове C функции, её upvalue располагаются по специфическим псевдоиндексам. Псевдоиндексы создаются макросом `lua_upvalueindex`. Первое upvalue, ассоциированное с функцией, располагается по индексу `lua_upvalueindex(1)`, и так далее. Любое обращение к `lua_upvalueindex(`*n*`)`, где *n* больше количества upvalue текущей функции (но не больше 256), создает допустимый, но не правильный индекс.

4.5 – Реестр

Lua предоставляет *реестр*, предопределенную таблицу, которая доступна C коду для хранения любых Lua значений. Таблица рееста всегда расположена по псевдоидексу `LUA_REGISTRYINDEX`. Любая C библиотека может хранить данные в этой таблице, но она должна заботиться о выборе уникальных ключей, чтобы избежать коллизий с другими библиотеками. Обычно, вы должны использовать в качестве ключа строку содержащую имя библиотеки, или легкие пользовательские данные (light userdata) с адресом C объекта в вашем коде, или любой Lua объект созданный вашим кодом. Как и имена переменных, ключи, начинающиеся с подчеркивания со следующими за ним прописными буквами, зарезервированы для Lua.

Целочисленные ключи в реестре используются механизмом ссылок (см. `luaL_ref`) и некоторыми предопределенными значениями. Следовательно, целочисленные ключи не должны использоваться для других целей.

Когда вы создаете новый Lua контекст, его реестр содержит некоторые предопределенные значения. Эти предопределенные значения индексируются целочисленными ключами, определенными как константы в `lua.h`. Определены следующие константы:

- **`LUA_RIDX_MAINTHREAD`:** По этому индексу в реестре расположен главный поток контекста. (Главный поток создается при создании контекста.)
- **`LUA_RIDX_GLOBALS`:** По этому индексу в реестре расположено глобальное окружение.

4.6 – Обработка ошибок в C

Внутри, для обработки ошибок Lua использует C `longjmp`. (Lua будет использовать исключения, если вы скомпилируете его как C++; для подробностей ищите `LUAI_THROW` в исходном коде.) Когда Lua сталкивается с любой ошибкой (такой как ошибка выделения памяти, ошибки типов, синтаксические ошибки и ошибки времени выполнения), он *возбуждает* ошибку; т.е., делает длинный переход (long jump). *Защищенное окружение* использует `setjmp` для установки точки восстановления; любая ошибка переходит к самой последней точке восстановления.

Если ошибка случается за пределами защищенного окружения, Lua вызывает *функцию паники* (см. `lua_atpanic`) и затем вызывает `abort`, завершая хостовое приложение. Ваша функция паники может избежать завершения приложения, если не вернет управление (т.е., сделает длинный переход [long jump] на вашу точку восстановления вне Lua).

Функция паники запускается как обработчик сообщений (см. §2.3); в частности, сообщение об ошибке лежит на вершине стека. Тем не менее, это не гарантирует стековое пространство. Чтобы положить что-либо на стек, функция паники должна сперва проверить доступное место (см. §4.2).

Большинство API функций могут возбуждать ошибки, например при выделении памяти. Описание для каждой функции предупреждает, что она может возбуждать ошибки.

Внутри C функции вы можете сгенерировать ошибку вызовом `lua_error`.

4.7 – Обработка уступок в C

Внутри, для приостановки сопрограммы Lua использует C `longjmp`. Следовательно, если C функция `foo` вызывает функцию API и эта API функция уступает (прямо или опосредованно, вызывая другую функцию, которая уступает), Lua больше не может вернуться к `foo`, потому что `longjmp` удаляет свой фрейм из C стека.

Чтобы избежать такой тип проблем, Lua возбуждает ошибку каждый раз, когда происходит уступка через вызов API, за исключением трех функций: `lua_yieldk`, `lua_callk` и `lua_pcallk`. Все эти функции получают *функцию продолжения* (как параметр `k`) чтобы продолжить исполнение после уступки.

Чтобы рассмотреть продолжения установим некоторую терминологию. Мы имеем C функцию вызванную из Lua, назовем её *оригинальной функцией*. Эта оригинальная функция затем вызывает одну из этих трех API функций, которые мы будем называть *вызываемой функцией*, которая затем уступает текущий поток. (Это может случиться, когда вызываемая функция это `lua_yieldk`, или когда вызываемая функция любая из `lua_callk` или `lua_pcallk` и вызванная ими функция уступает.)

Допустим, выполняющийся поток уступает во время выполнения вызываемой функции. Затем поток продолжается, это в конечном счете завершит вызываемую функцию. Тем не менее, вызываемая функция не может вернуться в оригинальную, т.к. её фрейм в C стеке был уничтожен уступкой. Взамен, Lua вызывает *функцию продолжения*, которая передается как аргумент вызываемой функции. Как подразумевается, продолжающая функция должна продолжить выполнение задачи оригинальной функции.

Как иллюстрацию, рассмотрим следующую функцию:

```
int original_function (lua_State *L) {
```

```
  ...     /* code 1 */
  status = lua_pcall(L, n, m, h);  /* calls Lua */
  ...     /* code 2 */
}
```

Сейчас мы хотим позволить Lua коду, запущенному через `lua_pcall`, уступать. Сначала, перепишем нашу функцию как показано ниже:

```
int k (lua_State *L, int status, lua_KContext ctx) {
  ...  /* code 2 */
}

int original_function (lua_State *L) {
  ...     /* code 1 */
  return k(L, lua_pcall(L, n, m, h), ctx);
}
```

В этом коде, новая функция k это *функция продолжения* (с типом `lua_KFunction`), которая должна делать всю работу, которую оригинальная функция делала после вызова `lua_pcall`. Мы должны проинформировать Lua о том, что если Lua код, запущенный `lua_pcall`, будет прерван (ошибки или уступка), то необходимо вызвать k; переписываем код, заменяя `lua_pcall` на `lua_pcallk`:

```
int original_function (lua_State *L) {
  ...     /* code 1 */
  return k(L, lua_pcallk(L, n, m, h, ctx2, k), ctx1);
}
```

Обратите внимание на внешний явный вызов продолжения: Lua будет вызывать продолжение только при необходимости, т.е. в случае ошибок или возобновления после уступки. Если вызванная функция

возвратится нормально, без уступок, `lua_pcallk` (и `lua_callk`) также возвратятся нормально. (Конечно, в этом случае вместо вызова продолжения, вы можете выполнить эквивалентную работу прямо внутри оригинальной функции.)

Рядом с Lua состоянием, функция продолжения имеет два других параметра: конечный статус вызова и контекстное значение (`ctx`), которое изначально передается в `lua_pcallk`. (Lua не использует это контекстное значение; только передает его из оригинальной функции в функцию продолжения.) Для `lua_pcallk`, статус это тоже значение, которое будет возвращено функцией `lua_pcallk`, за исключением того, что после выполнения уступки это будет `LUA_YIELD`, (взамен `LUA_OK`). Для `lua_yieldk` и `lua_callk`, когда Lua вызывает продолжение, статус всегда будет равен `LUA_YIELD`. (Для этих двух функций в случае ошибок Lua не вызовет продолжение, т.к. они не обрабатывают ошибки.) Аналогично, когда используется `lua_callk`, вы должны вызывать функцию продолжения со статусом `LUA_OK`. (Для `lua_yieldk`, нет смысла в прямом вызове функции продолжения, т.к. `lua_yieldk` обычно не возвращается.)

Lua обрабатывает функцию продолжения так, как будто это оригинальная функция. Функция продолжения получает тот же Lua стек из оригинальной функции, в том же состоянии, как если бы вызываемая функция вернулась. (Например, после `lua_callk` функция и её аргументы удалены из стека и заменены результатами вызова.) Она также будет иметь те же upvalue. Результаты, которые она вернет, будут обработаны Lua так, как будто их вернула оригинальная функция.

4.8 – Функции и типы

Здесь представлен список всех функций и типов из C API в алфавитном порядке. Каждая функция имеет индикатор вида: [-o, +p, *x*]

Первое поле, `o`, показывает сколько элементов функция снимает со стека. Второе поле, `p`, показывает сколько элементов функция ложит на стек. (Функции всегда ложат свои результаты после

того, как снимут аргументы со стека.) Поле вида `x|y` означает, что функция может ложить (или снимать) x или y элементов, в зависимости от ситуации; знак вопроса '?' означает, что по аргументам функции невозможно определить сколько элементов функция ложит/снимает (т.е. функция может зависеть от того, что лежит в стеке). Третье поле, `x`, показыает может ли функция генерировать ошибки: '-' означает, что функция никогда не генерирует ошибки; 'e' означает, что функция может генерировать ошибки; 'v' означает, что фунция предназначена для генерации ошибки.

lua_absindex

```
int lua_absindex (lua_State *L, int idx);
```

[-0, +0, –]

Конвертирует допустимый индекс `idx` в эквивалентный абсолютный индекс (который не зависит от вершины стека).

lua_Alloc

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

Тип функции выделения памяти, которая используется состоянием Lua. Функция выделения памяти должна предоставлять функциональность подобную функции `realloc`, но не точно такую же. Её аргументы `ud` - непрозрачный указатель, передаваемый в `lua_newstate`; `ptr` - указатель на блок для выделения/перераспределения/освобождения; `osize` - оригинальный размер блока или код, определяющий под что выделяется память; `nsize` - новый размер блока.

Когда `ptr` не `NULL`, `osize` это размер блока по указателю `ptr`, т.е. размер полученный при выделении

или перераспределении.

Когда `ptr = NULL`, `osize` кодирует тип объекта, для которого выделяется память. `osize` может быть равен `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA` и `LUA_TTHREAD`, когда (и только когда) Lua создает новый объект данного типа. Когда `osize` какое-то другое значение, Lua выделяет память для чего-то другого.

Lua предполагает следующее поведение функции выделения памяти:

Когда `nsize = 0`, функция должна действовать как `free` и возвращать `NULL`.

Когда `nsize != 0`, функция должна действовать как `realloc`. Функция возвращает `NULL` только, если не может выполнить запрос. Lua предполагает, что функция выделения памяти не может выдать ошибку при `osize >= nsize`.

Здесь представлена простая реализация функции выделения памяти. Она используется во вспомогательной библиотеке функцией `luaL_newstate`.

```
static void *l_alloc (void *ud, void *ptr, size_t osize,
                                           size_t nsize) {
  (void)ud;  (void)osize;  /* not used */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

Учтите, что стандартный C гарантирует, что `free(NULL)` не производит действий и `realloc(NULL,size)` эквивалентно `malloc(size)`. Этот код предполагает, что `realloc` не вызывает ошибок при уменьшении блока. (Хотя стандартный C не предполагает такого поведения, кажется, это безопасное предположение.)

lua_arith

```
void lua_arith (lua_State *L, int op);
```

[-(2|1), +1, *e*]

Выполняет арифметическую или битовую операцию над двумя значениями (или одним, в случае отрицаний), находящимися на вершине стека, значение на вершине является вторым операндом, удаляет эти значения со стека и ложит результат операции. Функция следует семантике соответствующего оператора (т.е., может вызывать метаметоды).

Значение `op` должно быть одной из следующих констант:

- **`LUA_OPADD`:** сложение (+)
- **`LUA_OPSUB`:** вычитание (-)
- **`LUA_OPMUL`:** умножение (*)
- **`LUA_OPDIV`:** деление (/)
- **`LUA_OPIDIV`:** целочисленное деление (//)
- **`LUA_OPMOD`:** модуль (%)
- **`LUA_OPPOW`:** возведение в степень (^)
- **`LUA_OPUNM`:** математическое отрицание (одноместный -)
- **`LUA_OPBNOT`:** битовое отрицание (~)
- **`LUA_OPBAND`:** битовое И (&)
- **`LUA_OPBOR`:** битовое ИЛИ (|)
- **`LUA_OPBXOR`:** битовое ИЛИ-НЕ (~)

- **LUA_OPSHL:** левый сдвиг (<<)
- **LUA_OPSHR:** правый сдвиг (>>)

lua_atpanic

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);
```

[-0, +0, –]

Устанавливает новую функцию паники и возвращает старую (см. §4.6).

lua_call

```
void lua_call (lua_State *L, int nargs, int nresults);
```

[-(nargs+1), +nresults, *e*]

Вызывает функцию.

Для вызова функции вы должны использовать следующий протокол: первое, вызываемая функция должна быть помещена в стек; затем, на стек ложатся аргументы в прямом порядке; т.е., первый аргумент ложится первым. Наконец, производится вызов `lua_call`; `nargs` - количество аргументов на стеке. Когда функция вызывается, все аргументы и функция снимаются со стека. Результаты ложатся на стек, когда функция возвращается. Количество результатов корректируется до `nresults`, если `nresults` не равно `LUA_MULTRET`. В этом случае, все результаты из функции ложатся на стек. Lua заботится, чтобы возвращенным значениям хватило места в стеке. Результаты функции ложатся в стек в прямом порядке (первый результат ложится первым), так что после вызова последний результат будет на вершине стека.

Любая ошибка внутри вызванной функции распространяется вверх (с `longjmp`).

Следующий пример показывает, как хостовая программа может выполнить действия эквивалентные

этому Lua коду:

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

Это же в C:

```
lua_getglobal(L, "f");          /* функция для вызова */
lua_pushliteral(L, "how");      /* 1-й аргумент */
lua_getglobal(L, "t");          /* таблица для индексирования */
lua_getfield(L, -1, "x");       /* положить результат t.x (2-й аргумент) */
lua_remove(L, -2);              /* удаление 't' из стека */
lua_pushinteger(L, 14);         /* 3-й аргумент */
lua_call(L, 3, 1);              /* вызов 'f' с 3 аргументами и 1 результатом */
lua_setglobal(L, "a");          /* установить глобальную 'a' */
```

Обратите внимание, что код выше *сбалансирован*: после его выполнения стек возвращается в первоначальное состояние. Это хорошая практика програмирования.

lua_callk

```
void lua_callk (lua_State *L,
                int nargs,
                int nresults,
                lua_KContext ctx,
                lua_KFunction k);
```

[-(nargs + 1), +nresults, *e*]

Эта функция аналогична `lua_call`, но позволяет вызываемой функции уступать (см. §4.7).

lua_CFunction

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

Тип для C функций.

Для правильного взаимодействия с Lua, C функция должна использовать следующий протокол, который определяет как передаются параметры и результаты: C функция получает аргументы из Lua в своём стеке в прямом порядке (первый аргумент ложится первым). Так, когда функция стартует, `lua_gettop(L)` возвращает количество аргументов, полученных функцией. Первый аргумент (если существует) находится по индексу 1, последний аргумент находится по индексу `lua_gettop(L)`. Для возврата значений в Lua, C функция просто ложит их на стек в прямом порядке (первый результат ложится первым) и возвращает число результатов. Любое другое значение в стеке ниже результатов будет сброшено Lua. Как и Lua функция, C функция, вызванная Lua, может возвращать несколько результатов.

Например, следующая функция получает переменное количество числовых аргументов и возвращает их среднее и их сумму:

```
static int foo (lua_State *L) {
  int n = lua_gettop(L);          /* количество аргументов */
  lua_Number sum = 0.0;
  int i;
  for (i = 1; i <= n; i++) {
    if (!lua_isnumber(L, i)) {
      lua_pushliteral(L, "incorrect argument");
      lua_error(L);
    }
    sum += lua_tonumber(L, i);
  }
```

```
  lua_pushnumber(L, sum/n);       /* 1-й результат */
  lua_pushnumber(L, sum);         /* 2-й результат */
  return 2;                       /* количество результатов */
}
```

lua_checkstack

```
int lua_checkstack (lua_State *L, int n);
```

[-0, +0, –]

Проверяет, что стек имеет место для как минимум n дополнительных слотов. Функция возвращает false, если не может выполнить запрос, потому что размер стека больше фиксированного максимума (обычно несколько тысяч элементов) или поскольку не может выделить память для дополнительных слотов. Эта функция никогда не уменьшает стек; если стек уже больше чем новый размер, он остаётся без изменений.

lua_close

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

[-0, +0, –]

Уничтожает все объекты в Lua состоянии (вызываются соответствующие метаметоды сборки мусора, если существуют) и освобождает всю динамическую память, использованную этим состоянием. На некоторых платформах, вы можете не вызывать эту функцию, т.к. все ресурсы освобождаются при завершении хостовой программы. С другой стороны, долго работающие программы, которые создают множество состояний, такие как демоны или веб-серверы, вероятно будут нуждаться в закрытии состояний, когда они станут ненужными.

lua_compare

```
int lua_compare (lua_State *L, int index1, int index2, int op);
```
[-0, +0, *e*]

Сравнивает два Lua значения. Возвращает 1, если значение по индексу `index1` удовлетворяет `op`, когда сравнивается со значением по индексу `index2`, следуя семантике соответствующего оператора Lua (т.е., могут вызываться метаметоды). Иначе возвращает 0. Также, возвращает 0, если любой из индексов не правильный.

Значение `op` должно равняться одной из констант:

- **`LUA_OPEQ`:** равенство (`==`)
- **`LUA_OPLT`:** меньше (`<`)
- **`LUA_OPLE`:** меньше или равно (`<=`)

lua_concat

```
void lua_concat (lua_State *L, int n);
```
[-n, +1, *e*]

Производит конкатенацию `n` значений на вершине стека, снимает их со стека и оставляет результат на вершине стека. Если `n` = 1, результат это одно значение на стеке (т.е., функция ничего не делает); если `n` = 0, результат пустая строка. Конкатенация выполняется следуя обычной семантике Lua (см. §3.4.6).

lua_copy

```
void lua_copy (lua_State *L, int fromidx, int toidx);
```
[-0, +0, –]

Копирует элемент по индексу `fromidx` в правильный индекс `toidx`, заменяя значение на этой позиции. Значения в других позициях не изменяются.

lua_createtable

```
void lua_createtable (lua_State *L, int narr, int nrec);
```
[-0, +1, *e*]

Создает новую пустую таблицу и ложит её на стек. Параметр `narr` - подсказка, сколько элементов в таблице будет как последовательность; параметр `nrec` - подсказка, сколько других элементов будет в таблице. Lua может использовать эти подсказки для предварительного выделения памяти для таблицы. Это предварительное выделение памяти полезно для производительности, когда вы заранее знаете сколько элементов будет в таблице. Иначе вы можете использовать функцию `lua_newtable`.

lua_dump

```
int lua_dump (lua_State *L,
                        lua_Writer writer,
                        void *data,
                        int strip);
```
[-0, +0, *e*]

Сохраняет функцию как бинарный кусок. Получает Lua функцию на вершине стека и выдает её скомпилированной, при повторной загрузке куска, результаты эквивалентны результатам компилируемой функции. По мере выдачи частей куска, для его записи, `lua_dump` вызывает функцию `writer` (см. `lua_Writer`) с полученной переменной `data`.

Если `strip` = true, бинарное представление может не включать всю отладочную информацию о функции, для уменьшения размера.

Возвращаемое значение: код ошибки, возвращенный при последнем вызове функции writer; 0 означает нет ошибок.

Эта функция не убирает Lua функцию со стека.

lua_error

```
int lua_error (lua_State *L);
```

[-1, +0, *v*]

Генерирует ошибку Lua, используя значение на вершине стека, как объект ошибки. Эта функция производит длинный переход (long jump) и никогда не возвращается (см. `luaL_error`).

lua_gc

```
int lua_gc (lua_State *L, int what, int data);
```

[-0, +0, *e*]

Управляет сборщиком мусора.

Эта функция производит различные действия, в зависимости от параметра `what`:

- **`LUA_GCSTOP`:** останавливает сборщик мусора.
- **`LUA_GCRESTART`:** перезапускает сборщик мусора.
- **`LUA_GCCOLLECT`:** производит полный цикл сборки мусора.
- **`LUA_GCCOUNT`:** возвращает текущий объем памяти (в килобайтах), используемый Lua.
- **`LUA_GCCOUNTB`:** возвращает остаток от деления текущего объема памяти, используемой Lua, в байтах на 1024.
- **`LUA_GCSTEP`:** производит шаг очистки мусора.
- **`LUA_GCSETPAUSE`:** устанавливает `data`, как новое значение для *паузы* сборщика (см. §2.5), и возвращает предыдущее значение паузы.
- **`LUA_GCSETSTEPMUL`:** устанавливает `data`, как новое значение для *множителя шагов* сборщика (см. §2.5), и возвращает предыдущее значение множителя шагов.

- **LUA_GCISRUNNING:** возвращает логическое значение, которое говорит, что сборщик запущен (т.е., не остановлен).

Для более подробного описания опций, см. `collectgarbage`.

lua_getallocf

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```
[-0, +0, –]

Возвращает функцию выделения памяти для данного Lua состояния. Если `ud` не `NULL`, Lua записывает в `*ud` непрозрачный указатель, полученный когда устанавливалась функция выделения памяти.

lua_getfield

```
int lua_getfield (lua_State *L, int index, const char *k);
```
[-0, +1, *e*]

Ложит в стек значение `t[k]`, где `t` - значение по индексу index. Как и в Lua, эта функция может запускать метаметод для события "index" (см. §2.4).

Возвращает тип положенного на стек значения.

lua_getextraspace

```
void *lua_getextraspace (lua_State *L);
```
[-0, +0, –]

Возвращает указатель на область "сырой" (raw) памяти, ассоциированную с данным Lua состоянием. Приложение может использовать эту область для любых целей; Lua не использует эту область.

Каждый новый поток имеет такую область, инициализированную копией области главного потока.

По умолчанию, эта область имеет размер пустого указателя [sizeof(void*)], но вы можете перекомпилировать Lua с другим размером для этой области (см. `LUA_EXTRASPACE` в `luaconf.h`).

lua_getglobal

```
int lua_getglobal (lua_State *L, const char *name);
```

[-0, +1, *e*]

Ложит в стек значение глобальной переменной `name`. Возвращает тип этого значения.

lua_geti

```
int lua_geti (lua_State *L, int index, lua_Integer i);
```

[-0, +1, *e*]

Ложит в стек значение `t[i]`, где `t` - значение по индексу index. Как и в Lua, эта функция может запускать метаметод для события "index" (см. §2.4).

Возвращает тип положенного на стек значения.

lua_getmetatable

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

[-0, +(0|1), –]

Если значение по индексу index имеет метатаблицу, функция ложит эту метатаблицу на стек и возвращает 1. Иначе, функция возвращает 0 и ничего не ложит на стек.

lua_gettable

```
int lua_gettable (lua_State *L, int index);
```
[-1, +1, *e*]

Ложит на стек значение `t[k]`, где `t` - значение по индексу index и `k` - значение на вершине стека.

Эта функция снимает ключ со стека и ложит результирующее значение на его место. Как и в Lua, эта функция может запускать метаметод для события "index" (см. §2.4).

Возвращает тип положенного на стек значения.

lua_gettop

```
int lua_gettop (lua_State *L);
```
[-0, +0, –]

Возвращает индекс элемента на вершине стека. Т.к. индексы начинаются с 1, результат равен количеству элементов в стеке; в частности, 0 означает, что стек пуст.

lua_getuservalue

```
int lua_getuservalue (lua_State *L, int index);
```
[-0, +1, –]

Ложит на стек Lua значение ассоциированное с пользовательскими данными (userdata), находящимися по индексу index.

Возвращает тип положенного на стек значения.

lua_insert

```
void lua_insert (lua_State *L, int index);
```
[-1, +1, –]

Перемещает верхний элемент стека в полученный правильный индекс, сдвигая вверх элементы выше этого индекса. Эта функция не может быть использована с псевдоиндексом, т.к. псевдоиндекс не обозначает актуальной позиции на стеке.

lua_Integer

```
typedef ... lua_Integer;
```

Тип целых в Lua.

По умолчанию, это тип `long long`, (обычно 64-битное целое), но это может быть изменено в `long` или `int` (обычно 32-битное целое). (см. `LUA_INT_TYPE` в `luaconf.h`.)

Lua также определяет константы `LUA_MININTEGER` и `LUA_MAXINTEGER`, с минимальным и максимальным значениями для этого типа.

lua_isboolean

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

[-0, +0, –]

Возвращает 1, если значение по индексу index имеет тип boolean, иначе возвращает 0.

lua_iscfunction

```
int lua_iscfunction (lua_State *L, int index);
```

[-0, +0, –]

Возвращает 1, если значение по индексу index это C функция, иначе возвращает 0.

lua_isfunction

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

[-0, +0, –]

Возвращает 1, если значение по индексу index это функция (любая C или Lua), иначе возвращает 0.

lua_isinteger

```
int lua_isinteger (lua_State *L, int index);
```

[-0, +0, –]

Возвращает 1, если значение по индексу index это целое (т.е., это число и оно представлено как целое [integer]), иначе возвращает 0.

lua_islightuserdata

```
int lua_islightuserdata (lua_State *L, int index);
```

[-0, +0, –]

Возвращает 1, если значение по индексу index это легкие пользовательские данные (light userdata), иначе возвращает 0.

lua_isnil

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

[-0, +0, –]

Возвращает 1, если значение по индексу index это **nil**, иначе возвращает 0.

lua_isnone

```
int lua_isnone (lua_State *L, int index);
```
[-0, +0, –]

Возвращает 1, если полученный индекс не правильный, иначе возвращает 0.

lua_isnoneornil

```
int lua_isnoneornil (lua_State *L, int index);
```
[-0, +0, –]

Возвращает 1, если полученный индекс не правильный или значение по индексу это **nil**, иначе возвращает 0.

lua_isnumber

```
int lua_isnumber (lua_State *L, int index);
```
[-0, +0, –]

Возвращает 1, если значение по индексу index это число или строка конвертируемая в число, иначе возвращает 0.

lua_isstring

```
int lua_isstring (lua_State *L, int index);
```
[-0, +0, –]

Возвращает 1, если значение по индексу index это строка или число (которое всегда можно преобразовать в строку), иначе возвращает 0.

lua_istable

```
int lua_istable (lua_State *L, int index);
```
[-0, +0, –]

Возвращает 1, если значение по индексу index это таблица, иначе возвращает 0.

lua_isthread

```
int lua_isthread (lua_State *L, int index);
```
[-0, +0, –]

Возвращает 1, если значение по индексу index это поток (thread), иначе возвращает 0.

lua_isuserdata

```
int lua_isuserdata (lua_State *L, int index);
```
[-0, +0, –]

Возвращает 1, если значение по индексу index это пользовательские данные (любые полные или легкие), иначе возвращает 0.

lua_isyieldable

```
int lua_isyieldable (lua_State *L);
```
[-0, +0, –]

Возвращает 1, если данная сопрограмма может уступить (yield), иначе возвращает 0.

lua_KContext

```
typedef ... lua_KContext;
```

Тип для контекста функции продолжения. Он должен быть числовым типом. Этот тип определен как `intptr_t`, когда `intptr_t` доступен, так что может содержать и указатели. Иначе, тип определен как `ptrdiff_t`.

lua_KFunction

```
typedef int (*lua_KFunction) (lua_State *L, int status, lua_KContext ctx);
```

Тип для функций продолжения (см. §4.7).

lua_len

```
void lua_len (lua_State *L, int index);
```
[-0, +1, *e*]

Возвращает длину значения по индексу index. Это эквивалент Lua оператору '#' (см. §3.4.7) и может вызывать метаметод для события "length" (см. §2.4). Результат ложится на стек.

lua_load

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname,
              const char *mode);
```
[-0, +1, –]

Загружает Lua кусок без запуска его на исполнение. Если ошибок нет, `lua_load` ложит на стек скомпилированный кусок, как Lua функцию. Иначе, ложит на стек сообщение об ошибке.

Возвращаемые значения `lua_load`:

- **`LUA_OK`:** нет ошибок;
- **`LUA_ERRSYNTAX`:** синтаксическая ошибка при компиляции;

- **LUA_ERRMEM:** ошибка выделения памяти;
- **LUA_ERRGCMM:** ошибка при выполнении метаметода `__gc`. (Эта ошибка не имеет отношения к загружаемому куску. Она генерируется сборщиком мусора.)

Функция `lua_load` использует пользовательскую функцию `reader` для чтения куска (см. `lua_Reader`). Аргумент `data` - это непрозрачный указатель, передаваемый функции reader.

Аргумент `chunkname` дает куску имя, которое используется для сообщений об ошибках и в отладочной информации (см. §4.9).

`lua_load` автоматически определяет, когда кусок текст или бинарные данные, и соответственно загружает его (см. программу `luac`). Строка `mode` работает как и в функции `load`, с добавлением того, что значение `NULL` эквивалентно строке "`bt`".

`lua_load` использует стек непосредственно, поэтому функция reader всегда должна оставлять его неизменным после возвращения.

Если результирующая функция имеет внешние локальные переменные (upvalue), её первое upvalue устанавливается в значение глобального окружения, записанное в реестре по индексу `LUA_RIDX_GLOBALS` (см. §4.5). При загрузке главных кусков, это upvalue будет переменной `_ENV` (см. §2.2). Остальные upvalue инициализируются значением **nil**.

lua_newstate

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```
[-0, +0, –]

Создает новый поток, выполняющийся в новом независимом состоянии. Возвращает `NULL`, если не может создать поток или состояние (из-за нехватки памяти). Аргумент `f` - функция выделения памяти; Lua выполняет всё выделение памяти для данного состояния через эту функцию. Второй аргумент,

`ud` - это непрозрачный указатель, который Lua передает в функцию выделения памяти при каждом вызове.

lua_newtable

```
void lua_newtable (lua_State *L);
```

[-0, +1, *e*]

Создает новую пустую таблицу и ложит её на стек. Эквивалентно вызову `lua_createtable(L, 0, 0)`.

lua_newthread

```
lua_State *lua_newthread (lua_State *L);
```

[-0, +1, *e*]

Создает новый поток, ложит его на стек и возвращает указатель на `lua_State`, который представляет этот новый поток. Новый поток использует одно глобальное окружение с оригинальным потоком, но имеет независимый стек исполнения.

Не существует явной функции для закрытия или уничтожения потока. Потоки это субъект для сборки мусора, как и любой Lua объект.

lua_newuserdata

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

[-0, +1, *e*]

Функция выделяет новый блок памяти размером size, ложит на стек новый объект full userdata (полные пользовательские данные) с адресом этого блока, и возвращает этот адрес. Хостовая программа может свободно использовать эту память.

lua_next

`int lua_next (lua_State *L, int index);` [-1, +(2|0), *e*]

Снимает ключ со стека, ложит пару ключ-значение из таблицы, находящейся по индексу index ("следующая" пара после данного ключа). Если в таблице больше нет элементов, `lua_next` возвращает 0 (и ничего не ложит на стек).

Типичный перебор элементов таблицы выглядит так:

```
/* таблица находится в стеке по индексу 't' */
lua_pushnil(L);  /* первый ключ */
while (lua_next(L, t) != 0) {
  /* используем 'ключ' (по индексу -2) и 'значение' (по индексу -1) */
  printf("%s - %s\n",
         lua_typename(L, lua_type(L, -2)),
         lua_typename(L, lua_type(L, -1)));
  /* удалаяем 'значение', сохраняя 'ключ' для следующей итерации */
  lua_pop(L, 1);
}
```

Когда производится перебор таблицы, не вызывайте `lua_tolstring` прямо на ключе, пока не убедитесь, что ключ действительно строка. Помните, что `lua_tolstring` может изменить значение на полученном индексе; это собьет следующий вызов `lua_next`.

Смотри функцию `next` для предостережений о модификации таблицы во время её перебора.

lua_Number

```
typedef ... lua_Number;
```

Тип вещественных чисел в Lua.

По умолчанию, это тип double, но он может быть изменен в single float или long double (см. `LUA_FLOAT_TYPE` в `luaconf.h`).

lua_numbertointeger

```
int lua_numbertointeger (lua_Number n, lua_Integer *p);
```

Преобразует вещественное Lua в целое Lua. Этот макрос предполагает, что `n` имеет целое значение. Если это значение входит в диапазон целых Lua, оно конвертируется в целое и назначается в `*p`. Макрос возвращает логическое значение, указывающее, что преобразование успешно. (Учтите, что из-за округлений эта проверка диапазона может быть мудрёной, чтобы сделать её корректно без этого макроса.)

Этот макрос может вычислять свои аргументы несколько раз.

lua_pcall

```
int lua_pcall (lua_State *L, int nargs, int nresults, int msgh);
```
[-(nargs + 1), +(nresults|1), –]

Вызывает функцию в защищенном режиме.

Оба `nargs` и `nresults` имеют тоже значение, что и в `lua_call`. Если в течение вызова нет ошибок, `lua_pcall` действует точно как `lua_call`. Тем не менее, в случае любых ошибок, `lua_pcall` перехватывает их, ложит одно значение на стек (собщение об ошибке) и возвращает код ошибки. Как и `lua_call`, `lua_pcall` всегда удаляет функцию и её аргументы со стека.

Если `msgh = 0`, сообщение об ошибке, возвращенное на стек, это оригинальное сообщение об ошибке. Иначе, `msgh` - это стековый индекс *обработчика ошибок*. (Этот индекс не может быть псевдоиндексом.) В случае ошибок выполнения, эта функция будет вызвана с сообщением об ошибке и значение, которое она вернет, будет возвращено на стеке функцией `lua_pcall`.

Обычно обработчик ошибок используется чтобы добавить больше отладочной информации в сообщение об ошибке, такой как трассировка стека. Эта информация не может быть получена после возврата из `lua_pcall`, поскольку стек будет раскручен.

Функция `lua_pcall` возвращает одну из следующих констант (определенны в `lua.h`):

- **`LUA_OK (0)`:** успех.
- **`LUA_ERRRUN`:** ошибка выполнения.
- **`LUA_ERRMEM`:** ошибка выделения памяти. Для таких ошибок Lua не вызывает обработчик ошибок.
- **`LUA_ERRERR`:** ошибка при выполнении обработчика ошибок.
- **`LUA_ERRGCMM`:** ошибка при выполнении метаметода `__gc`. (Эта ошибка обычно не имеет отношения к вызываемой функции.)

lua_pcallk

```
int lua_pcallk (lua_State *L,                                [-(nargs + 1), +(nresults|1), –]
                int nargs,
                int nresults,
                int msgh,
                lua_KContext ctx,
                lua_KFunction k);
```

Функция аналогична `lua_pcall`, но позволяет вызванной функции уступать (см. §4.7).

lua_pop

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

[-n, +0, –]

Снимает `n` элементов со стека.

lua_pushboolean

```
void lua_pushboolean (lua_State *L, int b);
```

[-0, +1, –]

Ложит на стек логическое значение (boolean) равное `b`.

lua_pushcclosure

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```

[-n, +1, *e*]

Ложит на стек новое C замыкание.

Когда C функция создана, можно ассоциировать с ней несколько значений, таким образом создав *C замыкание* (closure) (см. §4.4); эти значения доступны функции во время вызова. Чтобы ассоциировать эти значениея с C функцией, сначала эти значения нужно положить на стек (первое значение ложится первым). Затем вызвать `lua_pushcclosure` для создания C функции на стеке, с аргументом `n`, сообщающим сколько значений будет ассоциировано с функцией. `lua_pushcclosure` снимает эти значения со стека.

Максимальное значение для `n` это 255.

Когда `n = 0`, эта функция создает *лёгкую C функцию*, которая является лишь указателем на C функцию. В этом случае, она никогда не вызывает ошибку памяти.

lua_pushcfunction

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

[-0, +1, –]

Ложит C функцию на стек. Эта функция принимает указатель на C функцию и ложит в стек Lua значение типа `function`, которое при вызове запускает соответствующую C функцию.

Любая функция, чтобы быть работоспособной в Lua, должна следовать корректному протоколу приема параметров и возврата результатов (см. `lua_CFunction`).

lua_pushfstring

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

[-0, +1, *e*]

Ложит на стек форматированную строку и возвращает указатель на эту строку. Функция подобна ISO C функции `sprintf`, но имеет несколько важных отличий:

- Вы не выделяете место для результата: результат Lua строка и Lua заботится о выделении памяти (и её освобождении, через сборку мусора).
- Спецификаторы преобразований ограничены. Здесь нет флагов, ширины или точности. Спецификаторы могут быть только следующими: '`%%`' (вставляет символ '`%`'), '`%s`' (вставляет строку завершаемую нулем, без ограничений по размеру), '`%f`' (вставляет `lua_Number`), '`%I`' (вставляет `lua_Integer`), '`%p`' (вставляет указатель как шестнадцатиричное число), '`%d`' (вставляет `int`), '`%c`' (вставляет `int` как однобайтовый символ), '`%U`' (вставляет `long int` как байтовую последовательность UTF-8).

lua_pushglobaltable

```
void lua_pushglobaltable (lua_State *L);
```

[-0, +1, –]

Ложит глобальное окружение в стек.

lua_pushinteger

```
void lua_pushinteger (lua_State *L, lua_Integer n);
```

[-0, +1, –]

Ложит на стек целое равное n.

lua_pushlightuserdata

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

[-0, +1, –]

Ложит на стек легкие пользовательские данные (light userdata).

Пользовательские данные представляют C значения в Lua. *Легкие пользовательские данные* предсталяют указатель, `void*`. Это значение (как число): вы не создаете его, оно не имеет индивидуальной метатаблицы, и не уничтожается сборщиком (т.к. никогда не создаётся). Легкие пользовательские данные равны "любым" легким пользовательским данным с тем же C адресом.

lua_pushliteral

```
const char *lua_pushliteral (lua_State *L, const char *s);
```

[-0, +1, *e*]

Этот макрос аналогичен `lua_pushstring`, но должен использоваться только, когда `s` это литеральная

строка.

lua_pushlstring

```
const char *lua_pushlstring (lua_State *L, const char *s, size_t len);
```
[-0, +1, *e*]

Ложит на стек строку, указанную `s`, размером `len`. Lua создает (или использует снова) внутреннюю копию данной строки, так что после завершения функции память по адресу `s` может быть освобождена или использована снова. Строка может содержать любые бинарные данные, в том числе встроенные нули.

Возвращает указатель на внутреннюю копию строки.

lua_pushnil

```
void lua_pushnil (lua_State *L);
```
[-0, +1, –]

Ложит на стек значение nil.

lua_pushnumber

```
void lua_pushnumber (lua_State *L, lua_Number n);
```
[-0, +1, –]

Ложит на стек вещественное число равное `n`.

lua_pushstring

```
const char *lua_pushstring (lua_State *L, const char *s);
```
[-0, +1, *e*]

Ложит на стек завершаемую нулем строку, указанную `s`. Lua создает (или использует снова) внутреннюю копию данной строки, так что после завершения функции память по адресу `s` может быть освобождена или использована снова.

Возвращает указатель на внутреннюю копию строки.

Если `s` = `NULL`, ложит **nil** и возвращает `NULL`.

lua_pushthread

```
int lua_pushthread (lua_State *L);
```
[-0, +1, –]

Ложит на стек поток, представленный Lua состоянием `L`. Возвращает 1, если поток является главным потоком состояния.

lua_pushvalue

```
void lua_pushvalue (lua_State *L, int index);
```
[-0, +1, –]

Ложит на стек копию элемента по индексу index.

lua_pushvfstring

```
const char *lua_pushvfstring (lua_State *L,
                              const char *fmt,
                              va_list argp);
```
[-0, +1, *e*]

Аналогично `lua_pushfstring`, за исключением того, что фунцкция получает `va_list` вместо переменного числа аргументов.

lua_rawequal

```
int lua_rawequal (lua_State *L, int index1, int index2);
```
[-0, +0, –]

Возвращает 1, если два значения по индексам `index1` и `index2` примитивно равны (т.е. без вызова метаметодов). Иначе возвращает 0. Также возвращает 0, если любой из индексов не правильный.

lua_rawget

```
int lua_rawget (lua_State *L, int index);
```
[-1, +1, –]

Аналогично `lua_gettable`, но производит "сырой" доступ (т.е. без вызова метаметодов).

lua_rawgeti

```
int lua_rawgeti (lua_State *L, int index, lua_Integer n);
```
[-0, +1, –]

Ложит на стек значение `t[n]`, где `t` - таблица по индексу index. Доступ "сырой", т.е. без вызова метаметодов.

Возвращает тип положенного на стек значения.

lua_rawgetp

```
int lua_rawgetp (lua_State *L, int index, const void *p);
```
[-0, +1, –]

Ложит на стек значение `t[k]`, где `t` - таблица по индексу index, `k` - указатель `p`, представленный как лёгкие пользовательские данные. Доступ "сырой", т.е. без вызова метаметодов.

Возвращает тип положенного на стек значения.

lua_rawlen

```
size_t lua_rawlen (lua_State *L, int index);
```
[-0, +0, –]

Возвращет сырую "длину" значения по индексу index: для строк это длина строки; для таблиц это результат оператора длины ('#') без метаметодов; для пользовательских данных это размер выделенного блока памяти; для остальных значений это 0.

lua_rawset

```
void lua_rawset (lua_State *L, int index);
```
[-2, +0, *e*]

Аналогично `lua_settable`, но производит "сырое" присваивание (т.е. без вызова метаметодов).

lua_rawseti

```
void lua_rawseti (lua_State *L, int index, lua_Integer i);
```
[-1, +0, *e*]

Действие аналогично `t[i] = v`, где `t` - таблица по индексу index, `v` - значение на вершине стека.

Эта функция снимает значение со стека. Присваивание "сырое", т.е. без вызова метаметодов.

lua_rawsetp

```
void lua_rawsetp (lua_State *L, int index, const void *p);
```
[-1, +0, *e*]

Действие аналогично `t[p] = v`, где `t` - таблица по индексу index, `p` - кодируется как легкие пользовательские данные, `v` - значение на вершине стека.

Эта функция снимает значение со стека. Присваивание "сырое", т.е. без вызова метаметодов.

lua_Reader

```
typedef const char * (*lua_Reader) (lua_State *L,
                                    void *data,
                                    size_t *size);
```

Функция reader используется в `lua_load`. Каждый раз, когда требуется следующая часть куска, `lua_load` вызывает функцию reader, передавая ей свой параметр `data`. Функция reader должна возвращать указатель на блок памяти с новой частью куска и устанавливать `size` равным размеру блока. Блок должен существовать, пока функция reader не будет вызвана снова. Для сигнала о конце куска, функция должна вернуть `NULL` или установить `size` равным 0. Функция reader может возвращать части любого размера больше нуля.

lua_register

```
void lua_register (lua_State *L, const char *name, lua_CFunction f);
```
[-0, +0, *e*]

Устанавливает C функцию `f` в качестве нового значения глобальной переменной `name`. Функция определена как макрос:

```
#define lua_register(L,n,f) \
       (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

lua_remove

```
void lua_remove (lua_State *L, int index);
```
[-1, +0, –]

Удаляет элемент по данному правильному индексу, сдвигая вниз элементы выше этого индекса, чтобы заполнить промежуток. Эта функция не может быть вызвана с псевдоиндексом, т.к. псевдоиндекс не является действительной позицией в стеке.

lua_replace

```
void lua_replace (lua_State *L, int index);
```
[-1, +0, –]

Перемещает элемент с вершины стека в позицию по правильному индексу index без сдвига элементов (следовательно, заменяя значение по данному индексу) и затем снимает вехний элемент со стека.

lua_resume

```
int lua_resume (lua_State *L, lua_State *from, int nargs);
```
[-?, +?, –]

Запускает и продолжает сопрограмму в данном потоке `L`.

Для запуска подпрограммы, вы ложите в стек потока главную функцию и её аргументы; затем вызываете `lua_resume`, `nargs` - количество аргументов. Этот вызов возвращается, когда сопрограмма приостанавливается или завершается. Когда функция возвращается, стек содержит все значения, переданные в `lua_yield`, или все значения, возвращенные телом функции. `lua_resume` возвращает `LUA_YIELD` - если сопрограмма уступила, `LUA_OK` - если сопрограмма завершила свое исполнение без ошибок, или код ошибки в случае ошибок (см. `lua_pcall`).

В случае ошибок, стек не раскручивается, так вы можете использовать отладочные API на нём. Сообщение об ошибке находится на вершине стека.

Чтобы продолжить сопрограмму, вы удаляете все результаты из последнего `lua_yield`, ложите в её стек только значения, передаваемые в качестве результатов из `yield`, и затем вызываете `lua_resume`.

Параметр `from` представляет сопрограмму, которая продолжает `L`. Если такой сопрограммы нет, этот параметр может быть равен `NULL`.

lua_rotate

```
void lua_rotate (lua_State *L, int idx, int n);
```
[-0, +0, –]

Вращает элементы стека между правильным индексом `idx` и вершиной стека. Элементы вращаются на `n` позиций по направлению к вершине стека, для позитивного `n`; или `-n` позиций в направлении дна стека, для отрицательного `n`. Абсолютное значение `n` должно быть не больше чем размер вырезки для вращения. Эта функция не может быть вызвана с псевдоиндексом, т.к. псевдоиндекс не является действительной позицией в стеке.

lua_setallocf

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```
[-0, +0, –]

Заменяет функцию выделения памяти данного состояния на функцию `f` с пользовательскими данными `ud`.

lua_setfield

```
void lua_setfield (lua_State *L, int index, const char *k);
```
[-1, +0, *e*]

Действие аналогично `t[k] = v`, где `t` - таблица по индексу index, `v` - значение на вершине стека.

Эта функция снимает значение со стека. Как и в Lua, эта функция может запускать метаметод для события "newindex" (см. §2.4).

lua_setglobal

```
void lua_setglobal (lua_State *L, const char *name);
```
[-1, +0, *e*]

Снимает значение со стека и устанавливает его в качестве нового значения глобальной переменной `name`.

lua_seti

```
void lua_seti (lua_State *L, int index, lua_Integer n);
```
[-1, +0, *e*]

Действие аналогично `t[n] = v`, где `t` - значение по индексу index, `v` - значение на вершине стека.

Эта функция снимает значение со стека. Как и в Lua, эта функция может запускать метаметод для события "newindex" (см. §2.4).

lua_setmetatable

```
void lua_setmetatable (lua_State *L, int index);
```
[-1, +0, –]

Снимает таблицу со стека и устанавливает её в качестве новой метатаблицы для значения по индексу index.

lua_settable

```
void lua_settable (lua_State *L, int index);
```

[-2, +0, *e*]

Действие аналогично `t[k] = v`, где `t` - значение по индексу index, `v` - значение на вершине стека, `k` - значение следующее за верхним.

Эта функция снимает со стека ключ и значение. Как и в Lua, эта функция может запускать метаметод для события "newindex" (см. §2.4).

lua_settop

```
void lua_settop (lua_State *L, int index);
```

[-?, +?, –]

Принимает любой индекс или 0, и устанавливает вершину стека равной этому индексу. Если новая вершина больше предыдущей, новые элементы заполняются значением **nil**. Если `index = 0`, то все элементы стека удаляются.

lua_setuservalue

```
void lua_setuservalue (lua_State *L, int index);
```

[-1, +0, –]

Снимает значение со стека и устанавливает его в качестве нового значения, ассоциированного с пользовальскими данными (userdata) по индексу index.

lua_State

```
typedef struct lua_State lua_State;
```

Непрозрачная структура, которая указывает на поток и косвенно (через поток) на целое состояние интерпретатора Lua. Библиотека Lua полностью повторно входимая: она не имеет глобальных переменных. Вся информация о состоянии доступна через эту структуру.

Указатель на эту структуру должен передаваться, как первый аргумент, в каждую функцию в библиотеке, за исключением `lua_newstate`, которая создает Lua состояние.

lua_status

```
int lua_status (lua_State *L);
```

[-0, +0, –]

Возвращает состояние потока `L`.

Статус может быть 0 (`LUA_OK`) - для нормального потока, код ошибки - если поток завершил выполнение `lua_resume` с ошибкой, или `LUA_YIELD` - если поток приостановлен.

Вы можете вызывать функции только в потоках со статусом `LUA_OK`. Вы можете продолжать потоки со статусом `LUA_OK` (для запуска новой сопрограммы) или `LUA_YIELD` (для продолжения сопрограммы).

lua_stringtonumber

```
size_t lua_stringtonumber (lua_State *L, const char *s);
```

[-0, +1, –]

Преобразует завершаемую нулем строку `s` в число, ложит это число на стек и возвращает общий размер строки, т.е. её длину + 1. Результатом преобразования может быть целое или вещественное число, в соответствии с лексическими соглашениями Lua (см. §3.1). Строка может иметь начальные и конечные пробелы и знак. Если строка не правильное число, функция возвращает 0 и ничего не ложит на стек. (Имейте ввиду, что результат может быть использован как логическое значение, если преобразование успешно, то результат true.)

lua_toboolean

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

[-0, +0, –]

Преобразует Lua значение по индексу index в C boolean (0 или 1). Как и все проверки в Lua, `lua_toboolean` возвращает true для любого Lua значения отличного от **false** и **nil**; иначе, возвращает false. (Если вы хотите принимать только значения типа boolean, используйте `lua_isboolean` для проверки.)

lua_tocfunction

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

[-0, +0, –]

Преобразует Lua значение по индексу index в C функцию. Это значение должно быть C функцией; иначе, возвращает `NULL`.

lua_tointeger

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

[-0, +0, –]

Эквивалент `lua_tointegerx` с `isnum` = `NULL`.

lua_tointegerx

```
lua_Integer lua_tointegerx (lua_State *L, int index, int *isnum);
```

[-0, +0, –]

Преобразует Lua значение по индексу index в целое со знаком типа `lua_Integer`. Lua значение должно быть целым, или числом или строкой, преобразуемой в целое (см. §3.4.3); иначе,

`lua_tointegerx` возвращает 0.

Если `isnum` != `NULL`, он указывает на логическое значение, которое показывает успешность операции.

lua_tolstring

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```

[-0, +0, *e*]

Преобразует Lua значение по индексу index в C строку. Если `len` != `NULL`, она также устанавливает `*len` равным длине строки. Lua значение должно быть строкой или числом; иначе, функция возвращает `NULL`. Если значение число, то `lua_tolstring` также *изменяет действительное значение в стеке в строку*. (Это изменение сбивает `lua_next`, когда `lua_tolstring` применяется к ключам при переборе таблицы.)

`lua_tolstring` возвращает полностью выровненный указатель на строку внутри Lua состояния. Эта строка всегда имеет завершающий ноль ('`\0`') после последнего символа (как в C), но может содержать другие нули в своем теле.

Т.к. в Lua есть сборка мусора, нет гарантии, что указатель, возвращенный `lua_tolstring`, будет действителен после удаления соответствующего Lua значения из стека.

lua_tonumber

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

[-0, +0, –]

Эквивалент `lua_tonumberx` с `isnum` = `NULL`.

lua_tonumberx

```
lua_Number lua_tonumberx (lua_State *L, int index, int *isnum);
```
[-0, +0, –]

Преобразует Lua значение по индексу index в C тип `lua_Number` (см. `lua_Number`). Lua значение должно быть числом или строкой, преобразуемой в число (см. §3.4.3); иначе, `lua_tonumberx` возвращает 0.

Если `isnum` != `NULL`, он указывает на логическое значение, которое показывает успешность операции.

lua_topointer

```
const void *lua_topointer (lua_State *L, int index);
```
[-0, +0, –]

Преобразует Lua значение по индексу index в пустой C указатель (`void*`). Значение может быть пользовательскими данными, таблицей, потоком или функцией; иначе, `lua_topointer` возвращает `NULL`. Разные объекты дадут разные указатели. Не существует способа сконвертировать указатель обратно в оригинальное значение.

Обычно эта функция используется для хеширования и отладочной информации.

lua_tostring

```
const char *lua_tostring (lua_State *L, int index);
```
[-0, +0, *e*]

Эквивалент `lua_tolstring` с `len` = `NULL`.

lua_tothread

```
lua_State *lua_tothread (lua_State *L, int index);
```
[-0, +0, –]

Преобразует Lua значение по индексу index в Lua поток (представленный как `lua_State*`). Это значение должно быть потоком; иначе, функция возвращает `NULL`.

lua_touserdata

```
void *lua_touserdata (lua_State *L, int index);
```

[-0, +0, –]

Если значение по индексу index это полные пользовательские данные, возвращает адрес их блока. Если значение это легкие пользовательские данные, возвращает их указатель. Иначе, возвращает `NULL`.

lua_type

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

[-0, +0, –]

Возвращает тип значения по правильному индексу index, или `LUA_TNONE` для не правильного (но допустимого) индекса. Типы, возвращаемые `lua_type`, кодируются следующими константами (определены в `lua.h`): `LUA_TNIL` (0), `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD` и `LUA_TLIGHTUSERDATA`.

lua_typename

```
const char *lua_typename (lua_State *L, int tp);
```

[-0, +0, –]

Возвращает имя типа, кодированного значением `tp`, которое должно быть одним из возвращаемых `lua_type`.

lua_Unsigned

```
typedef ... lua_Unsigned;
```

Беззнаковая версия `lua_Integer`.

lua_upvalueindex

```
int lua_upvalueindex (int i);
```
[-0, +0, –]

Возвращает псевдоиндекс, который представляет `i`-е upvalue запущенной функции (см. §4.4).

lua_version

```
const lua_Number *lua_version (lua_State *L);
```
[-0, +0, *v*]

Возвращает адрес номера версии, хранящейся в ядре Lua. Когда вызвана с правильным `lua_State`, возвращает адрес версии, использованной для создания этого состояния. Когда вызвана с `NULL`, возвращает адрес версии выполнившей вызов.

lua_Writer

```
typedef int (*lua_Writer) (lua_State *L,
                           const void* p,
                           size_t sz,
                           void* ud);
```

Тип функции writer, используемой в `lua_dump`. Каждый раз, когда производится очередная часть куска, `lua_dump` вызывает функцию writer, передавая ей буфер для записи (`p`), его размер (`sz`) и параметр `data`, переданный в `lua_dump`.

Функция writer возвращает код ошибки; 0 означает нет ошибок; любое другое значение означает ошибку и останавливает `lua_dump` от последующих вызовов writer.

lua_xmove

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```
[-?, +?, –]

Обмен значениями между разными потоками одного состояния.

Эта функция снимает со стека `from` `n` значений и ложит их в стек `to`.

lua_yield

```
int lua_yield (lua_State *L, int nresults);
```
[-?, +?, *e*]

Функция аналогична `lua_yieldk`, но она не имеет продолжения (см. §4.7). Следовательно, когда поток продолжается, он продолжает функцию, которая вызвала `lua_yield`.

lua_yieldk

```
int lua_yieldk (lua_State *L,
                int nresults,
                lua_KContext ctx,
                lua_KFunction k);
```
[-?, +?, *e*]

Устапает сопрограмму (поток).

Когда C функция вызывает `lua_yieldk`, вызывающая сопрограмма приостанавливает свое

исполнение, и вызов `lua_resume`, который запустил эту сопрограмму, возвращается. Параметр `nresults` - это количество значений в стеке, которые будут переданы как результаты в `lua_resume`.

Когда сопрограмма снова возобновится, Lua вызывает переданную функцию продолжения `k`, для продолжения выполнения C функции, которая уступила (см. §4.7). Эта функция продолжения получает тот же стек из предыдущей функции, но `n` результатов будут удалены и заменены аргументами, переданными в `lua_resume`. Кроме того, функция продолжения получает значение `ctx`, переданное в `lua_yieldk`.

Обычно эта функция не возвращается; когда сопрограмма в конечном счете возобновляется, она выполняет фукнцию продолжения. Тем не менее, есть один специальный случай, который случается когда функция вызвана из построчного перехватчика (см. §4.9). В этом случает, `lua_yieldk` должна быть вызвана без продолжения (вероятно в форме `lua_yield`), и перехватчик должен вернуться непосредственно после вызова. Lua уступит и, когда сопрограмма вновь возобновится, она продолжит нормальное исполнение (Lua) функции, которая инициировала перехват.

Эта функция может вызывать ошибку, если вызвана из потока с незаконченным C вызовом без функции продолжения, или если вызвана из потока, который не запущен внутри сопрограммы (т.е., главный поток).

4.9 – Отладочный интерфейс

Lua не имеет встроенных отладочных средств. Взамен, он предлагает специальный интерфейс функций и *перехватчиков* (hook). Этот интерфейс позволяет строить различные типы отладчиков, профилировщиков и других инструментов, нуждающихся во "внутренней информации" интерпретатора.

lua_Debug

```
typedef struct lua_Debug {
  int event;
  const char *name;           /* (n) */
  const char *namewhat;       /* (n) */
  const char *what;           /* (S) */
  const char *source;         /* (S) */
  int currentline;            /* (l) */
  int linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  unsigned char nups;         /* (u) количество upvalue */
  unsigned char nparams;      /* (u) количество параметров */
  char isvararg;              /* (u) */
  char istailcall;            /* (t) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* private part */
  other fields
} lua_Debug;
```

Структура, используемая для хранения различных частей информации о функции или о записи активации. `lua_getstack` заполняет только приватную часть этой структуры, для последующего использования. Для заполнения остальных полей `lua_Debug` полезной информацией, вызовите `lua_getinfo`.

Поля `lua_Debug` имеют следующие значения:

- **`source`:** имя куска, который создал функцию. Если `source` начинается с '@', это означает, что функция была определена в файле, где имя файла следует за символом '@'. Если `source` начинается с '=', остаток его содержимого описывает источник в определяемой пользователем

форме. Иначе, функция была определена в строке, где `source` - эта строка.

- **`short_src`:** "выводимая" (printable) версия `source`, для использования в сообщениях об ошибках.
- **`linedefined`:** номер строки, где начинается определение функции.
- **`lastlinedefined`:** номер строки, где заканчивается определение функции.
- **`what`:** строка `"Lua"` - если функция это Lua функция, `"C"` - если это C функция, `"main"` - если это главная часть куска.
- **`currentline`:** текущая строка, где полученная функция исполняется. Когда информация о строках не доступна, `currentline` устанавливается равным -1.
- **`name`:** разумное имя полученной функции. Т.к. функции в Lua это первоклассные значения, они не имеют фиксированных имен: некоторые функции могут быть значениями множества глобальных переменных, тогда как другие могут храниться только в полях таблицы. Чтобы найти подходящее имя `lua_getinfo` проверяет как функция была вызвана. Если она не может найти имя, то `name` устанавливается равным `NULL`.
- **`namewhat`:** поясняет поле `name`. Значение `namewhat` может быть `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"` или `""` (пустая строка), в соответствии с тем как была вызвана функция. (Lua использует пустую строку, когда не может применить ни какую другую опцию.)
- **`istailcall`:** истина, если этот вызов функции хвостовой (tail call). В этом случае, вызывающий этого уровня не на стеке.
- **`nups`:** количество upvalue у функции.
- **`nparams`:** количество фиксированных параметров у функции (всегда 0 для C функций).
- **`isvararg`:** истина, если функция имеет переменное число аргументов (всегда истина для C функций).

lua_gethook

```
lua_Hook lua_gethook (lua_State *L);
```

[-0, +0, –]

Возвращает текущую функцию перехватчик.

lua_gethookcount

```
int lua_gethookcount (lua_State *L);
```
[-0, +0, –]

Возвращает количество перехватчиков.

lua_gethookmask

```
int lua_gethookmask (lua_State *L);
```
[-0, +0, –]

Возвращает текущую маску перехватов.

lua_getinfo

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```
[-(0|1), +(0|1|2), *e*]

Предоставляет информацию о специфической функции или о вызове функции.

Для получения информации о вызове функции, параметр `ar` должен быть правильной записью активации, которая заполняется вызовом `lua_getstack` или передается как аргумент в перехватчик (см. `lua_Hook`).

Для получения информации о функции, нужно положить её в стек и начать строку `what` символом '>'. (В этом случае, `lua_getinfo` снимет функцию с вершины стека.) Например, чтобы узнать в которой строке определена функция `f`, вы можете написать следующий код:

```
lua_Debug ar;
```

```
lua_getglobal(L, "f");  /* получить глобальную 'f' */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

Каждый символ в строке `what` выбирает определенные поля в структуре `ar` для заполнения или значение, передаваемое через стек:

- **'n':** заполняет поля `name` и `namewhat`;
- **'S':** заполняет поля `source`, `short_src`, `linedefined`, `lastlinedefined` и `what`;
- **'l':** заполняет поле `currentline`;
- **'t':** заполняет поле `istailcall`;
- **'u':** заполняет поля `nups`, `nparams` и `isvararg`;
- **'f':** ложит в стек функцию, которая запущена на данном уровне;
- **'L':** ложит в стек таблицу, индексы которой - это номера значимых строк функции. (*Значимая строка* - это строка с некоторым кодом, т.е. строка, где можно поставить точку останова [break point]. Не значащие строки - это пустые строки и комментарии.)

  Если эта опция получена вместе с опцией 'f', таблица ложится в стек после функции.

Эта функция возвращает 0 при ошибке (например, при неправильной опции в строке `what`).

lua_getlocal

```
const char *lua_getlocal (lua_State *L, const lua_Debug *ar, int n);
```
[-0, +(0|1), –]

Предоставляет информацию о локальной переменной данной записи активации или функции.

В первом случае, параметр `ar` должен быть правильной записью активации, которая заполнена предыдущим вызовом `lua_getstack` или передана как аргумент перехватчику (см. `lua_Hook`). Индекс

`n` выбирает локальную переменную для проверки; см. `debug.getlocal` для подробностей о индексах и именах переменных.

`lua_getlocal` ложит в стек значение переменной и возвращает её имя.

Во втором случае, `ar` должен быть `NULL`, и функция для проверки должна находиться на вершине стека. В этом случае, видны только параметры Lua функций (т.к. здесь нет информации о том, какие переменные активны) и значения в стек не ложатся.

Возвращает `NULL` (и ничего не ложит на стек), когда индекс больше чем количество активных локальных переменных.

lua_getstack

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```
[-0, +0, –]

Предоставляет информацию о исполняемом (runtime) стеке интерпретатора.

Эта функция заполняет части структуры `lua_Debug` с индентификацией *записи активации* функции, выполняемой на данном уровне (level). Уровень 0 - это текущая выполняющаяся функция, уровень *n+1* - это функция которая вызывает уровень *n* (за исключением хвостовых вызовов, которые не считаются на стеке). Когда ошибок нет, `lua_getstack` возвращает 1; когда вызвана с уровнем больше чем глубина стека, возвращает 0.

lua_getupvalue

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```
[-0, +(0|1), –]

Предоставляет информацию о `n`-ном upvalue замыкания по индексу `funcindex`. Функция ложит на

стек значение upvalue и возвращает его имя. Возвращает `NULL` (и ничего не ложит на стек), когда индекс `n` больше количества upvalue.

Для C функций, эта функция использует пустую строку `""` в качестве имени для всех upvalue. (Для Lua функций, upvalue - это внешние локальные переменные, которые использует функция, и которые, следовательно, включены в её замыкание.)

Upvalue не имеют конкретного порядка, как они активны внутри функции. Они нумеруются в произвольном порядке.

lua_Hook

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

Тип для отладочных функций перехватчиков.

Всякий раз, когда вызывается перехватчик, его аргумент `ar` имеет поле `event`, установленное согласно событию, вызвавшему срабатывание перехватчика. Lua идентифицирует эти события следующими константами: `LUA_HOOKCALL`, `LUA_HOOKRET`, `LUA_HOOKTAILCALL`, `LUA_HOOKLINE` и `LUA_HOOKCOUNT`. Более того, для событий строк устанавливается поле `currentline`. Для получения значений остальных полей в `ar`, перехватчик должен вызвать `lua_getinfo`.

Для событий вызова, `event` может быть `LUA_HOOKCALL` - нормальное значение, или `LUA_HOOKTAILCALL` - для хвостового вызова; в этом случае, соответствующего события возврата не будет.

Пока Lua выполняет перехватчик, все остальные перехватчики отключаются. Следовательно, если перехватчик вызывает Lua для выполнения функции или куска, это выполнение произойдет без вызова перехватчиков.

Функции перехватчики не могут иметь продолжений, т.е. они не могут вызывать `lua_yieldk`, `lua_pcallk` или `lua_callk` с не нулевым k.

Функции перехватчики могут уступать при следующих условиях: только счетные и события строк могут уступать; для уступки, функция перехватчик должна завершить свое исполнение, вызвав `lua_yield` с `nresults` = 0 (т.е. без значений).

lua_sethook

```
void lua_sethook (lua_State *L, lua_Hook f, int mask, int count);
```
[-0, +0, –]

Устанавливает отладочную функцию-перехватчик.

Аргумент `f` - это функция перехватчик. `mask` - определяет на каких событиях будет вызван перехватчик: формируется битовым ИЛИ констант `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE` и `LUA_MASKCOUNT`. Аргумент `count` - имеет значение только, когда маска включает `LUA_MASKCOUNT`. Перехватчик для каждого события объясняется ниже:

- **Перехватчик вызова (call hook):** вызывается, когда интерпретатор вызывает функцию. Перехватчик вызывается сразу после того, как Lua войдет в новую функцию, перед тем как функция получит свои аргументы.
- **Перехватчик возврата (return hook):** вызывается, когда интерпретатор возвращается из функции. Перехватчик вызывается перед тем, как Lua покинет функцию. Здесь нет стандартного пути для доступа к возвращаемым функцией значениям.
- **Перехватчик строки (line hook):** вызывается, когда интерпретатор начинает выполнение новой строки кода, или когда он переходит назад в код (даже в ту же строку). (Это событие случается только пока Lua выполняет Lua функцию.)
- **Счетный перехватчик (count hook):** вызывается после того, как интерпретатор запустит `count`

инструкций. (Это событие случается только пока Lua выполняет Lua функцию.)

Перехват отключается установкой `mask` = 0.

lua_setlocal

```
const char *lua_setlocal (lua_State *L, const lua_Debug *ar, int n);
```
[-(0|1), +0, –]

Устанавливает значение локальной переменной в данной записи активации (ar). Устанавливает переменной значение с вершины стека и возвращает её имя, затем удаляет значение из стека.

Возвращает `NULL` (и ничего не удаляет со стека), когда индекс больше количества активных локальных переменных.

Параметры `ar` и `n` такие же, как в функции `lua_getlocal`.

lua_setupvalue

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```
[-(0|1), +0, –]

Устанавливает значение для upvalue замыкания. Устанавливает upvalue равным значению с вершины стека и возвращает её имя, затем удаляет значение из стека

Возвращает `NULL` (и ничего не удаляет со стека), когда индекс больше количества upvalue.

Параметры `funcindex` и `n` такие же, как в функции `lua_getupvalue`.

lua_upvalueid

```
void *lua_upvalueid (lua_State *L, int funcindex, int n);
```
[-0, +0, –]

Возвращает уникальный идентификатор для upvalue под номером `n` из замыкания по индексу `funcindex`.

Эти уникальные идентификаторы позволяют программе проверить, когда замыкания совместно используют upvalue. Lua замыкания, которые совместно используют upvalue (т.е., обращаются к одной внешней локальной переменной), вернут одинаковые идентификаторы для индексов своих upvalue.

Параметры `funcindex` и `n` такие же, как в функции `lua_getupvalue`, но `n` не может быть больше количества upvalue.

lua_upvaluejoin

```
void lua_upvaluejoin (lua_State *L, int funcindex1, int n1,
                                    int funcindex2, int n2);
```
[-0, +0, –]

Делает `n1`-ое upvalue Lua замыкания по индексу `funcindex1` ссылкой на `n2`-ое upvalue Lua замыкания по индексу `funcindex2`.

5 – Вспомогательная библиотека

Вспомогательная библиотека предоставляет различные удобные функции для взаимодействия C с Lua. Когда базовые API предоставляют примитивные функции для всех взаимодействий между C и Lua, вспомогательная библиотека предоставляет высокоуровневые функции для некоторых общих задач.

Все функции и типы вспомогательной библиотеки определены в заголовочном файле `lauxlib.h` и имеют префикс `luaL_`.

Все функции вспомогательной библиотеки строятся из базовых API, и так они не предоставляют ничего, чего нельзя было бы сделать с помощью этих API. Тем не менее, использование вспомогательной библиотеки гарантирует большую стабильность вашему коду.

Различные функции вспомогательной библиотеки используют внутри несколько дополнительных слотов. Когда функция вспомогательной библиотеки использует меньше пяти слотов, она не проверяет размер стека; она просто предполагает, что там достаточно слотов.

Различные функции вспомогательной библиотеки используются для проверки аргументов C функций. Так как сообщение об ошибке формируется для аргументов (т.е., "`bad argument #1`"), вы не должны использовать эти функции для других значений стека.

Функции `luaL_check*` всегда генерируют ошибку, если проверка не удовлетворена.

5.1 – Функции и типы

Здесь представлен список всех функций и типов вспомогательной библиотеки в алфавитном порядке.

luaL_addchar

```
void luaL_addchar (luaL_Buffer *B, char c);
```

[-?, +?, *e*]

Добавляет байт `c` в буфер `B` (см. `luaL_Buffer`).

luaL_addlstring

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```
[-?, +?, *e*]

Добавляет строку по указателю `s` длиной `l` в буфер `B` (см. `luaL_Buffer`). Строка может содержать встроенные нули.

luaL_addsize

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```
[-?, +?, *e*]

Добавляет в буфер `B` (см. `luaL_Buffer`) строку длиной `n`, прежде скопированную в область буфера (см. `luaL_prepbuffer`).

luaL_addstring

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```
[-?, +?, *e*]

Добавляет завершаемую нулем строку по указателю `s` в буфер `B` (см. `luaL_Buffer`).

luaL_addvalue

```
void luaL_addvalue (luaL_Buffer *B);
```
[-1, +?, *e*]

Добавляет значение с вершины стека в буфер `B` (см. `luaL_Buffer`). Снимает значение со стека.

Только эта функция для строковых буферов может (и должна) вызываться с дополнительным аргументом на стеке, являющимся значением для добавления в буфер.

luaL_argcheck

```
void luaL_argcheck (lua_State *L,                                    [-0, +0, v]
                    int cond,
                    int arg,
                    const char *extramsg);
```

Проверяет когда `cond` истина. Если это не так, генерирует ошибку со стандартным сообщением (см. `luaL_argerror`).

luaL_argerror

```
int luaL_argerror (lua_State *L, int arg, const char *extramsg);     [-0, +0, v]
```

Генерирует ошибку, сообщающую о проблеме с аргументом `arg` для C функции, которая вызвала luaL_argerror. Используется стандартное сообщение, включающее `extramsg` как комментарий:

```
bad argument #arg to 'funcname' (extramsg)
```

Эта функция никогда не возвращается.

luaL_Buffer

```
typedef struct luaL_Buffer luaL_Buffer;
```

Тип для *строкового буфера*.

Строковый буфер позволяет C коду строить Lua строки по частям. Обычно он используется следующим образом:

- Сначала объявляется переменная `b` типа `luaL_Buffer`.
- Затем она инициализируется вызовом `luaL_buffinit(L, &b)`.
- Затем части строки добавляются в буфер вызовами функций `luaL_add*`.
- Завершается вызовом `luaL_pushresult(&b)`. Этот вызов оставляет финальную строку на вершине стека.

Если вы заранее знаете размер результирующей строки, вы можете использовать буфер так:

- Сначала объявляется переменная `b` типа `luaL_Buffer`.
- Затем вызовом `luaL_buffinitsize(L, &b, sz)` она инициализируется, и для неё выделяется память размером `sz`.
- Затем в эту область копируется строка.
- Завершается вызовом `luaL_pushresultsize(&b, sz)`, где `sz` - размер результирующей строки, скопированной в эту область.

При нормальном управлении, строковый буфер использует переменное число слотов стека. Так, пока используется буфер, вы не можете предполагать, что вы знаете где вершина стека. Вы можете использовать стек между успешными вызовами буферных операций, пока он сбалансирован; т.е., когда вы вызываете буферную операцию, стек должен быть на том же уровне, что и непосредственно после предыдущей буферной операции. (Существует только одно исключение, это `luaL_addvalue`.) После вызова `luaL_pushresult` стек возвращается на тот уровень, когда был инициализирован буфер, плюс результирующая строка на вершине.

luaL_buffinit

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```
[-0, +0, –]

Инициализирует буфер `B`. Эта функция не выделяет память; буфер должен быть определен как

переменная (см. `luaL_Buffer`).

luaL_buffinitsize

```
char *luaL_buffinitsize (lua_State *L, luaL_Buffer *B, size_t sz);
```

[-?, +?, *e*]

Эквивалентно последовательности `luaL_buffinit`, `luaL_prepbuffsize`.

luaL_callmeta

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

[-0, +(0|1), *e*]

Вызывает метаметод.

Если объект по индексу `obj` имеет метатаблицу и его метатаблица имеет поле `e`, эта функция вызывает это поле, передавая объект в качестве первого аргумента. В этом случае функция возвращает true и ложит в стек значение, возвращенное вызовом метаметода. Если метатаблицы или метаметода не существует, функция возвращает false (и ничего не ложит на стек).

luaL_checkany

```
void luaL_checkany (lua_State *L, int arg);
```

[-0, +0, *v*]

Проверяет, что функция имеет аргумент любого типа (включая **nil**) по индексу `arg`.

luaL_checkinteger

```
lua_Integer luaL_checkinteger (lua_State *L, int arg);
```

[-0, +0, *v*]

Проверяет, что аргумент функции `arg` это целое (или может быть преобразован в целое), и возвращает это целое, как `lua_Integer`.

luaL_checklstring

```
const char *luaL_checklstring (lua_State *L, int arg, size_t *l);
```
[-0, +0, *v*]

Проверяет, что аргумент функции `arg` это строка, и возвращает эту строку; если `l` не `NULL`, пишет в `*l` длину строки.

Эта функция использует `lua_tolstring` для получения результата, так все преобразования и предостережения этой функции относятся и сюда.

luaL_checknumber

```
lua_Number luaL_checknumber (lua_State *L, int arg);
```
[-0, +0, *v*]

Проверяет, что аргумент функции `arg` это число, и возвращает это число.

luaL_checkoption

```
int luaL_checkoption (lua_State *L,
                      int arg,
                      const char *def,
                      const char *const lst[]);
```
[-0, +0, *v*]

Проверяет, что аргумент функции `arg` это строка, и ищет эту строку в массиве `lst` (который должен завершаться NULL-символом). Возвращает индекс в массиве, где найдена строка. Генерирует

ошибку, если аргумент не строка или строка не найдена в массиве.

Если `def` не `NULL`, функция использует `def` как значение по умолчанию, когда нет аргумента `arg` или он равен **nil**.

Эта функция полезна для отображения строк в C перечисления (enum). (Обычно, для выбора опций в Lua библиотеках принято использовать строки вместо чисел.)

luaL_checkstack

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

[-0, +0, *v*]

Увеличивает размер стека до `top + sz` элементов, вызывает ошибку, если стек не может быть увеличен до этого размера. `msg` - это дополнительный текст для сообщения об ошибке (или `NULL`, если его нет).

luaL_checkstring

```
const char *luaL_checkstring (lua_State *L, int arg);
```

[-0, +0, *v*]

Проверяет, что аргумент функции `arg` это строка, и возвращает эту строку.

Эта функция использует `lua_tolstring` для получения результата, так все преобразования и предостережения этой функции относятся и сюда.

luaL_checktype

```
void luaL_checktype (lua_State *L, int arg, int t);
```

[-0, +0, *v*]

Проверяет, что аргумент функции `arg` имеет тип `t`. См. `lua_type` для кодировок типов `t`.

luaL_checkudata

```
void *luaL_checkudata (lua_State *L, int arg, const char *tname);
```
[-0, +0, *v*]

Проверяет, что аргумент функции `arg` это пользовательские данные типа `tname` (см. `luaL_newmetatable`), и возвращает адрес пользовательских данных (см. `lua_touserdata`).

luaL_checkversion

```
void luaL_checkversion (lua_State *L);
```
[-0, +0, –]

Проверяет, что ядро, выполняющее вызов, ядро, создавшее Lua состояние, и код, производящий вызов, все используют одну версию Lua. Также проверяет, что ядро, выполняющее вызов, и ядро, создавшее Lua состояние, используют одно адресное пространство.

luaL_dofile

```
int luaL_dofile (lua_State *L, const char *filename);
```
[-0, +?, *e*]

Загружает и запускает данный файл. Определена как макрос:

```
(luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

Возвращает false, если ошибок нет, или true в случае ошибок.

luaL_dostring

```
int luaL_dostring (lua_State *L, const char *str);
```
[-0, +?, –]

Загружает и запускает переданную строку. Определена как макрос:

```
(luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

Возвращает false, если ошибок нет, или true в случае ошибок.

luaL_error

```
int luaL_error (lua_State *L, const char *fmt, ...);
```
[-0, +0, *v*]

Генерирует ошибку. Сообщение об ошибке форматируется строкой `fmt` и дополнительными аргументами, следует тем же правилам, что и `lua_pushfstring`. Также добавляет в начало сообщения имя файла и номер строки, где произошла ошибка, если эта информация доступна.

Эта функция никогда не возвращается, но её идиома для использования в C функциях: `return luaL_error(`*`args`*`)`.

luaL_execresult

```
int luaL_execresult (lua_State *L, int stat);
```
[-0, +3, *e*]

Эта функция выдает возвращаемые значения для связанных с процессами функций стандартной библиотеки (`os.execute` и `io.close`).

luaL_fileresult

```
int luaL_fileresult (lua_State *L, int stat, const char *fname);
```
[-0, +(1|3), *e*]

Эта функция выдает возвращаемые значения для связанных с файлами функций стандартной библиотеки (`io.open`, `os.rename`, `file:seek` и др.).

luaL_getmetafield

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```

[-0, +(0|1), *e*]

Ложит на стек поле `e` из метатаблицы объекта по индексу `obj` и возвращает тип положенного на стек значения. Если объект не имеет метатаблицы или в метатаблице нет этого поля, ничего не ложит на стек и возвращает `LUA_TNIL`.

luaL_getmetatable

```
int luaL_getmetatable (lua_State *L, const char *tname);
```

[-0, +1, –]

Ложит на стек метатаблицу, ассоциированную с именем `tname` в реестре (см. `luaL_newmetatable`) (**nil**, если метатаблицы с таким именем нет). Возвращает тип положенного на стек значения.

luaL_getsubtable

```
int luaL_getsubtable (lua_State *L, int idx, const char *fname);
```

[-0, +1, *e*]

Гарантирует, что значение `t[fname]` это таблица, и ложит эту таблицу на стек; `t` - это значение по индексу `idx`. Возвращает true, если находит существующую таблицу, и false, если создает новую.

luaL_gsub

```
const char *luaL_gsub (lua_State *L,
```

[-0, +1, *e*]

```
                        const char *s,
                        const char *p,
                        const char *r);
```

Создает копию строки `s`, заменяя все вхождения строки `p` на строку `r`. Ложит на стек результирующую строку и возвращает её.

luaL_len

```
lua_Integer luaL_len (lua_State *L, int index);
```
[-0, +0, *e*]

Возвращает "длину" значения по индексу index как число; эквивалентна Lua оператору '#' (см. §3.4.7). Генерирует ошибку, если результат операции не целое. (Это может случиться только через метаметоды.)

luaL_loadbuffer

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
                     const char *name);
```
[-0, +1, –]

Эквивалентно `luaL_loadbufferx` с `mode` = `NULL`.

luaL_loadbufferx

```
int luaL_loadbufferx (lua_State *L,
                      const char *buff,
```
[-0, +1, –]

```
                        size_t sz,
                        const char *name,
                        const char *mode);
```

Загружает буфер как Lua кусок. Эта функция использует `lua_load` для загрузки куска в буфер `buff` размером `sz`.

Функция возвращает те же результаты, что и `lua_load`.

`name` - имя куска, используемое для отладочной информации и сообщений об ошибках. Строка `mode` работает как в функции `lua_load`.

luaL_loadfile

```
int luaL_loadfile (lua_State *L, const char *filename);
```
[-0, +1, *e*]

Эквивалентно `luaL_loadfilex` с `mode` = `NULL`.

luaL_loadfilex

```
int luaL_loadfilex (lua_State *L, const char *filename,
                                                const char *mode);
```
[-0, +1, *e*]

Загружает файл как Lua кусок. Эта функция использует `lua_load` для загрузки куска из файла `filename`. Если `filename` = `NULL`, загружает кусок из потока стандартного ввода. Первая линия файла игнорируется, если она начинается с #.

Строка `mode` работает как в функции `lua_load`.

Эта функция возвращает те же результаты, что и `lua_load`, но имеет дополнительный код ошибки `LUA_ERRFILE`, если не может открыть/прочитать файл или файл имеет неверный режим.

Как и `lua_load`, эта функция только загружает кусок; но не запускает его.

luaL_loadstring

```
int luaL_loadstring (lua_State *L, const char *s);
```
[-0, +1, –]

Загружает строку как Lua кусок. Эта функция использует `lua_load` для загрузки завершаемой нулем строки `s`.

Эта функция возвращает те же результаты, что и `lua_load`.

Как и `lua_load`, эта функция только загружает кусок; но не запускает его.

luaL_newlib

```
void luaL_newlib (lua_State *L, const luaL_Reg l[]);
```
[-0, +1, *e*]

Создает новую таблицу и регистрирует в ней функции из списка `l`.

Реализована как макрос:

```
(luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))
```

Массив `l` должен быть фактическим массивом, не указателем на него.

luaL_newlibtable

```
void luaL_newlibtable (lua_State *L, const luaL_Reg l[]);
```
[-0, +1, *e*]

Создает новую таблицу с размером, оптимизированным для хранения всех записей массива `l` (но фактически не сохраняет их). Она предполагается для использования в связке с `luaL_setfuncs` (см. `luaL_newlib`).

Реализована как макрос. Массив `l` должен быть фактическим массивом, не указателем на него.

luaL_newmetatable

```
int luaL_newmetatable (lua_State *L, const char *tname);
```
[-0, +1, *e*]

Если реестр уже содержит ключ `tname`, возвращает 0. Иначе, создает новую таблицу для использования в качестве метатаблицы пользовательских данных, добавляет в эту новую таблицу пару `__name = tname`, добавляет в реестр пару `[tname] = new table`, и возвращает 1. (Элемент `__name` используется некоторыми функциями, сообщающими об ошибках.)

В обоих случаях ложит на стек финальное значение ассоциированное с `tname` в реестре.

luaL_newstate

```
lua_State *luaL_newstate (void);
```
[-0, +0, –]

Создает новое Lua состояние. Вызывает `lua_newstate` с функцией выделения памяти, базирующейся на стандартной C функции `realloc`, и затем устанавливает функцию паники (см. §4.6), которая в случае фатальных ошибок печатает сообщение об ошибке в стандартный поток вывода.

Возвращает новое состояние, или `NULL`, если не удалось выделить память.

luaL_openlibs

```
void luaL_openlibs (lua_State *L);
```
[-0, +0, *e*]

Открывает все стандартные библиотеки Lua в данное состояние.

luaL_optinteger

```
lua_Integer luaL_optinteger (lua_State *L,
                             int arg,
                             lua_Integer d);
```
[-0, +0, *v*]

Если аргумент функции `arg` это целое (или может быть преобразован в целое), возвращает это целое. Если аргумент отсутствует или равен **nil**, возвращает `d`. Иначе, генерирует ошибку.

luaL_optlstring

```
const char *luaL_optlstring (lua_State *L,
                             int arg,
                             const char *d,
                             size_t *l);
```
[-0, +0, *v*]

Если аргумент функции `arg` это строка, возвращает эту строку. Если аргумент отсутствует или равен **nil**, возвращает `d`. Иначе, генерирует ошибку.

Если `l` не `NULL`, записывает в `*l` длину результата.

luaL_optnumber

```
lua_Number luaL_optnumber (lua_State *L, int arg, lua_Number d);
```
[-0, +0, *v*]

Если аргумент функции `arg` это число, возвращает это число. Если аргумент отсутствует или равен **nil**, возвращает `d`. Иначе, генерирует ошибку.

luaL_optstring

```
const char *luaL_optstring (lua_State *L,
                            int arg,
                            const char *d);
```
[-0, +0, *v*]

Если аргумент функции `arg` это строка, возвращает эту строку. Если аргумент отсутствует или равен **nil**, возвращает `d`. Иначе, генерирует ошибку.

luaL_prepbuffer

```
char *luaL_prepbuffer (luaL_Buffer *B);
```
[-?, +?, *e*]

Эвивалентно `luaL_prepbuffsize` с предопределенным размером `LUAL_BUFFERSIZE`.

luaL_prepbuffsize

```
char *luaL_prepbuffsize (luaL_Buffer *B, size_t sz);
```
[-?, +?, *e*]

Возвращает адрес пространства размером `sz`, куда вы можете скопировать строку, чтобы добавить в буфер `B` (см. `luaL_Buffer`). После копирования строки в это пространство, вы должны вызвать `luaL_addsize` с размером строки, для фактического добавления её в буфер.

luaL_pushresult

```
void luaL_pushresult (luaL_Buffer *B);
```
[-?, +1, *e*]

Завершает использование буфера `B`, оставляя финальную строку на вершине стека.

luaL_pushresultsize

```
void luaL_pushresultsize (luaL_Buffer *B, size_t sz);
```
[-?, +1, *e*]

Эквивалентно последовательности `luaL_addsize`, `luaL_pushresult`.

luaL_ref

```
int luaL_ref (lua_State *L, int t);
```
[-1, +0, *e*]

Создает и возвращает *ссылку* (reference) в таблице по индексу `t`, для объекта на вершине стека (и снимает этот объект со стека).

Ссылка это уникальный целый ключ. Пока вы не добавляете целые ключи в таблицу `t`, `luaL_ref` гарантирует уникальность возвращаемых ключей. Вы можете извлечь объект по ссылке `r`, вызвав `lua_rawgeti(L, t, r)`. Функция `luaL_unref` освобождает ссылку и ассоциированный с ней объект.

Если объект на вершине стека равен **nil**, `luaL_ref` возвращает константу `LUA_REFNIL`. Константа `LUA_NOREF` всегда отлична от любой другой ссылки, возвращаемой `luaL_ref`.

luaL_Reg

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

Тип для массивов функций для регистрации функцией `luaL_setfuncs`. `name` - имя функции, `func` - указатель на функцию. Любой массив `luaL_Reg` должен завершаться ключевым элементом, в котором `name` и `func` = `NULL`.

luaL_requiref

```
void luaL_requiref (lua_State *L, const char *modname,
                    lua_CFunction openf, int glb);
```
[-0, +1, *e*]

Если `modname` ещё не присутствует в `package.loaded`, вызывает функцию `openf` со строкой `modname` в качестве аргумента и пишет результат вызова в `package.loaded[modname]`, как если бы эта функция была бы вызвана через `require`.

Если `glb` = true, также сохраняет модуль в глобальную `modname`.

Оставляет копию модуля на стеке.

luaL_setfuncs

```
void luaL_setfuncs (lua_State *L, const luaL_Reg *l, int nup);
```
[-nup, +0, *e*]

Регистрирует все функции в массиве `l` (см. `luaL_Reg`) в таблицу на вершине стека (ниже опциональных upvalue, см. далее).

Когда `nup != 0`, все функции создаются, разделяющими `nup` upvalue, которые должны быть предварительно положены на стек поверх таблицы библиотеки. После регистрации эти значения снимаются со стека.

luaL_setmetatable

```
void luaL_setmetatable (lua_State *L, const char *tname);
```
[-0, +0, –]

Устанавливает метатаблицей объекта на вершине стека метатаблицу, ассоциированную с именем `tname` в реестре (см. `luaL_newmetatable`).

luaL_Stream

```
typedef struct luaL_Stream {
  FILE *f;
  lua_CFunction closef;
} luaL_Stream;
```

Стандартное представление для описателей файлов (handle), которые используются стандартной библиотекой ввода-вывода.

Описатель файла реализован как полные пользовательские данные, с метатаблицей называемой `LUA_FILEHANDLE` (где `LUA_FILEHANDLE` - это макрос с фактическим именем метатаблицы). Метатаблица создается библиотекой ввода-вывода (см. `luaL_newmetatable`).

Эти пользовательские данные должны начинаться со структуры `luaL_Stream`; они могут содержать другие данные после этой начальной структуры. Поле `f` - указывает на соответствующий C поток (ими может быть `NULL` для индикации не полностью созданного описателя). Поле `closef` - указывает

на Lua функцию, которая будет вызвана для закрытия потока, при закрытии или сборке описателя; эта функция получает описатель файла в качестве единственного аргумента и должна вернуть **true** (в случае успеха) или **nil** с сообщением об ошибке (в случае ошибки). Как только Lua вызывает это поле, его значение меняется на `NULL` для сигнализации, что описатель закрыт.

luaL_testudata

```
void *luaL_testudata (lua_State *L, int arg, const char *tname);
```
[-0, +0, *e*]

Эта функция работает как `luaL_checkudata`, за исключением того, что когда проверка неудачна, она возвращает `NULL`, вместо генерирования ошибки.

luaL_tolstring

```
const char *luaL_tolstring (lua_State *L, int idx, size_t *len);
```
[-0, +1, *e*]

Конвертирует любое Lua значение по индексу idx в C строку в разумном формате. Результирующая строка ложится на стек и также возвращается функцией. Если `len` не `NULL`, функция пишет в `*len` длину строки.

Если значение имеет метатаблицу с полем `"__tostring"`, то `luaL_tolstring` вызывает соответствующий метаметод со значением в качестве аргумента и использует результат вызова, как свой результат.

luaL_traceback

```
void luaL_traceback (lua_State *L, lua_State *L1, const char *msg,
                     int level);
```
[-0, +1, *e*]

Создает и ложит на стек трассировку стека `L1`. Если `msg` не `NULL`, оно добавляется в начало трассировки. Параметр `level` указывает на каком уровне начинать трассировку.

luaL_typename

```
const char *luaL_typename (lua_State *L, int index);
```
[-0, +0, –]

Возвращает имя типа для значения по индексу index.

luaL_unref

```
void luaL_unref (lua_State *L, int t, int ref);
```
[-0, +0, –]

Освобождает ссылку `ref` из таблицы по индексу `t` (см. `luaL_ref`). Элемент удаляется из таблицы, так что ассоциированный объект может быть собран сборщиком мусора. Ссылка `ref` также освобождается для повторного использования.

Если `ref` = `LUA_NOREF` или `LUA_REFNIL`, `luaL_unref` ничего не делает.

luaL_where

```
void luaL_where (lua_State *L, int lvl);
```
[-0, +1, *e*]

Ложит на стек строку, идентифицирующую текущую позицию на управляющем уровне `lvl` в стеке вызовов. Обычно эта строка имеет следующий формат:

chunkname:*currentline*:

Уровень 0 - это текущая выполняемая функция, уровень 1 - это функция, которая вызвала текущую, и

так далее.

Эта функция используется для формирования префикса сообщений об ошибках.

6 – Стандартные библиотеки

Стандартные библиотеки Lua предоставляют полезные функции, реализованные напрямую через C API. Некоторые из этих функций предоставляют существенные службы языка (например, `type` и `getmetatable`); другие предоставляют доступ к "внешним" сервисам (например, I/O); и другие могут быть реализованы в Lua самостоятельно, но очень полезны и имеют критические требования к производительности, что заслуживает реализации в C (например, `table.sort`).

Все библиотеки реализованы через официальный C API и предоставляются, как отдельные C модули. Lua имеет следующие стандартные библиотеки:

- базовая библиотека (§6.1);
- библиотека сопрограмм (§6.2);
- библиотека пакетов (§6.3);
- манипуляции со строками (§6.4);
- базовая поддержка UTF-8 (§6.5);
- манипуляции с таблицами (§6.6);
- математические функции (§6.7) (sin, log и др.);
- ввод и вывод (§6.8);
- средства операционной системы (§6.9);
- отладочные средства (§6.10).

За исключением базовой и пакетной библиотек, каждая библиотека предоставляет все свои функции, как поля глобальной таблицы или методы её объектов.

Для получения доступа к этим библиотекам хоствая C программа должна вызвать функцию `luaL_openlibs`, которая открывает все стандартные библиотеки. Либо, хостовая программа может открыть их индивидуально, используя `luaL_requiref` для вызова `luaopen_base` (для базовой библиотеки), `luaopen_package` (для библиотеки пакетов), `luaopen_coroutine` (для библиотеки сопрограмм), `luaopen_string` (для строковой библиотеки), `luaopen_utf8` (для библиотеки UTF8), `luaopen_table` (для табличной библиотеки), `luaopen_math` (для математической библиотеки), `luaopen_io` (для библиотеки ввода-вывода), `luaopen_os` (для библиотеки операционной системы) и `luaopen_debug` (для отладочной библиотеки). Все эти функции определены в `lualib.h`.

6.1 – Базовые функции

Базовая библиотека предоставляет функции ядра для Lua. Если вы не включите эту библиотеку в ваше приложение, вы должны проявлять осторожность, когда будете нуждаться в предоставлении реализаций для некоторых её возможностей.

assert (v [, message])

Вызывает `error`, если значение аргумента `v` = false (т.е., **nil** или **false**); иначе, возвращает все свои аргументы. В случае ошибки, `message` - это объект ошибки; когда он отсутствует, его значение по умолчанию `"assertion failed!"`

collectgarbage ([opt [, arg]])

Эта функция общий интерфейс к сборщику мусора. Она выполняет различные действия в

соответствии с её аргументом `opt`:

- **`"collect"`:** выполняет полный цикл очистки мусора. Это опция по умолчанию.
- **`"stop"`:** останавливает автоматическое выполнение сборщика мусора. Сборщик будет запущен, только когда явно вызван, до вызова его перезапуска.
- **`"restart"`:** перезапускает автоматическое выполнение сборщика мусора.
- **`"count"`:** возвращает общее количество используемой Lua памяти в килобайтах. Это значение имеет дробную часть, так что его произведение на 1024 дает точное количство байт, используемых Lua (за исключением переполнений).
- **`"step"`:** выполняет шаг сборки мусора. "Размер" шага контролируется аргументом `arg`. С нулевым значением сборщик выполнит один базовый (неделимый) шаг. Для не нулевых значений, сборщик выполнит, как если это количество памяти (в килобайтах) было выделено Lua. Возвращает **true**, если шаг завершил цикл сборки.
- **`"setpause"`:** устанавливает `arg` в качестве нового значения для *паузы* сборщика (см. §2.5). Возвращает предыдущее значение *паузы*.
- **`"setstepmul"`:** устанавливает `arg` в качестве нового значения для *множителя шагов* сборщика (см. §2.5). Возвращает предыдущее значение *шага*.
- **`"isrunning"`:** возвращает логическое значение, говорящее запущен ли сборщик (т.е. не остановлен).

dofile ([filename])

Открывает файл и запускает его содержимое, как Lua кусок. Когда вызвана без аргументов, запускает содержимое стандартного ввода (`stdin`). Возвращает все значения, возвращенные куском. В случае ошибок, `dofile` распространяет ошибку вызывающему коду (т.е. `dofile` не запускается в защищенном режиме).

error (message [, level])

Завершает последнюю защищенно вызванную функцию и возвращает `message`, как объект ошибки. Функция `error` никогда не возвращается.

Обычно `error` добавляет информацию о позиции ошибки в начало сообщения, если сообщение (message) это строка. Аргумент `level` определяет как получить позицию ошибки. Когда level = 1 (по умолчанию) - позиция ошибки там, где вызвана функция `error`. Level = 2 - указывает ошибку там, где вызвана функция, вызвавшая `error`; и так далее. Передача level = 0 - позиция ошибки не влючается в сообщение.

_G

Глобальная переменная (не функция), которая хранит глобальное окружение (см. §2.2). Lua сама не использует эту переменную; её изменение не влияет на окружение, ни наоборот.

getmetatable (object)

Если `object` не имеет метатаблицы, возвращает **nil**. Иначе, если метатаблица объекта имеет поле `"__metatable"`, возвращает ассоциированное с этим полем значение. Иначе, возвращает метатаблицу переданного объекта.

ipairs (t)

Возвращает три значения (функцию итератор, таблицу `t` и 0), так что конструкция

```
for i,v in ipairs(t) do body end
```

будет перебирать пары ключ–значение (`1,t[1]`), (`2,t[2]`), ..., до первого значения nil.

load (chunk [, chunkname [, mode [, env]]])

Загружает кусок.

Если `chunk` это строка, кусок будет этой строкой. Если `chunk` это 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, первое upvalue устанавливается равным `env`, если этот параметр передан, или в значение глобального окружения. Остальные upvalue инициализируются значением **nil**. (Когда вы загружаете главный кусок, результирующая функция всегда будет иметь только одно upvalue, переменную `_ENV` (см. §2.2). Тем не менее, когда вы загружаете бинарный кусок, созданный из функции (см. `string.dump`), результирующая функция может иметь произвольное количество upvalue.) Все upvalue свежие, т.е. они не разделяются между любыми другими функциями.

`chunkname` - используется как имя куска, для сообщений об ошибках и отладочной информации (см. §4.9). Когда отсутствует, его значение по умолчанию это сам `chunk`, если `chunk` это строка, или "`=(load)`" иначе.

Строка `mode` - управляет 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. Злонамеренно созданный бинарный кусок может разрушить интерпретатор.

loadfile ([filename [, mode [, env]]])

Аналогично `load`, но получает кусок из файла `filename` или из стандартного потока ввода, если имя файла не передано.

next (table [, index])

Позволяет программе просмотреть все поля таблицы. Её первый аргумент - это таблица, второй аргумент - индекс в этой таблице. `next` возвращает следующий индекс таблицы и ассоциированное с ним значение. Когда вызывается с **nil** в качестве второго аргумента, `next` возвращает начальный индекс и ассоциированое с ним значение. Когда вызывается с последним индексом, или с **nil** для пустой таблицы, `next` возвращает **nil**. Если второй аргумет отсутствует, то он интерпретируется как **nil**. В частности, вы можете использовать `next(t)` для проверки, что таблица пустая.

Порядок перечисления индексов не определен, *даже для числовых индексов*. (Для просмотра таблицы в числовом порядке, используйте числовой **for**.)

Поведение `next` неопределено, если во время просмотра вы присваиваете любое значение несуществующему полю в таблице. Тем не менее вы можете модифицировать существующие поля. В частности, вы можете очищать существующие поля.

pairs (t)

Если `t` имеет метаметод `__pairs`, вызывает его с аргументом `t` и возвращает первые три результата вызова.

Иначе, возвращает три значения: функцию `next`, таблицу `t` и **nil**, так что конструкция

```
for k,v in pairs(t) do body end
```

будет перебирать все пары ключ–значение в таблице `t`.

См. функцию `next` для предостережений о модификации таблицы во время просмотра.

pcall (f [, arg1, ···])

Вызывает функцию `f` с переданными аргументами в *защищенном режиме*. Это значит, что любая ошибка внутри `f` не распространяется; взамен, `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 результаты из вызова, после первого результата. В случае ошибки `pcall` возвращает false и сообщение об ошибке.

print (···)

Получает произвольное количество аргументов и печатает их значения в `stdout`, используя функцию `tostring` для преобразования каждого аргумента в строку. `print` не предназначен для форматированного вывода, только как быстрый способ показать значение, например, для отладки. Для полного контроля над выводом используйте `string.format` и `io.write`.

rawequal (v1, v2)

Проверяет, что `v1` равен `v2`, без вызова метаметодов. Возвращает логическое значение.

rawget (table, index)

Возвращает реальное значение `table[index]`, без вызова метаметодов. `table` должен быть таблицей, `index` может быть любым значением.

rawlen (v)

Возвращает длину объекта `v`, который должен быть таблицей или строкой, без вызова метаметодов. Возвращает целое.

rawset (table, index, value)

Устанавливает реальное значение `table[index]` равным `value`, без вызова метаметодов. `table` должен быть таблицей, `index` - любое значение отличное от **nil** и NaN, `value` - любое Lua значение.

Эта функция возвращает `table`.

select (index, ···)

Если `index` это число, возвращает все аргументы после аргумента номер `index`; негативное число индексируется с конца (-1 последний аргумент). Иначе, если `index` строка `"#"`, `select` вернет общее количество дополнительных аргументов.

setmetatable (table, metatable)

Устанавливает метатаблицу для данной таблицы. (Вы не можете изменять метатаблицы других типов

из Lua, только из C.) Если `metatable` = **nil**, удаляет метатаблицу переданной таблицы. Если оригинальная метатаблица имеет поле "`__metatable`", генерирует ошибку.

Эта функция возвращает `table`.

tonumber (e [, base])

Когда вызвана без аргумента `base`, `tonumber` пытается конвертировать аргумент в число. Если аргумент уже число или строка, которую можно преобразовать в число, то `tonumber` возвращает это число; иначе, возвращает **nil**.

Преобразование строк может выдавать в результате целые или вещественные числа, в соответствии с лексическими соглашениями Lua (см. §3.1). (Строка может иметь начальные и конечные пробелы и знак.)

Когда вызвана с аргументом `base`, то аргумент `e` должен быть строкой, которая интерпретируется как целое в данной системе счисления. base может быть любым целым от 2 до 36, включительно. При base > 10, символ 'A' (в верхнем или нижнем регистре) представляет 10, 'B' представляет 11 и так далее, с 'Z' представляющим 35. Если строка `e` не правильное число в данной системе счисления, функция возвращает **nil**.

tostring (v)

Получает значение любого типа и преобразует его в строку в понятном человеку формате. (Для полного контроля над преобразованием чисел используйте `string.format`.)

Если метатаблица `v` имеет поле "`__tostring`", то `tostring` вызывает соответствующее значение с `v` в качестве аргумента и использует результат вызова, как свой результат.

type (v)

Возвращает тип любого аргумента, представленный строкой. Возможные результаты этой функции: "`nil`" (строка, не значение **nil**), "`number`", "`string`", "`boolean`", "`table`", "`function`", "`thread`" и "`userdata`".

_VERSION

Глобальная переменная (не функция), которая содержит строку с текущей версией интерпретатора. Текущее значение этой переменной "`Lua 5.3`".

xpcall (f, msgh [, arg1, ···])

Эта функция похожа на `pcall`, но она устанавливает новый обработчик сообщений `msgh`.

6.2 – Работа с сопроцессами

Эта библиотека содержит операции для манипуляции с сопроцессами, которые хранятся в таблице `coroutine`. См. §2.6 для общего описания сопроцессов.

coroutine.create (f)

Создает новый сопроцесс, с телом `f`. `f` должен быть функцией. Возвращает этот новый сопроцесс, как объект с типом "`thread`".

coroutine.isyieldable ()

Возвращает true, когда запущенный сопроцесс может уступить.

Запущенный сопроцесс может уступать, если это не главный поток и он не внутри неприостанавливаемой C функции.

coroutine.resume (co [, val1, ···])

Начинает или продолжает выполнение сопроцесса `co`. При первом возобновлении сопроцесса запускает его тело. Значение `val1`, ... передаются как аргументы телу сопроцесса (его функции). Если сопроцесс был приостановлен, `resume` перезапускает его; значнинния `val1`, ... передаются как результаты из yield.

Если сопроцесс запущен без ошибок, `resume` возвращает **true** и все значения, переданные в `yield` (когда сопроцесс уступает) или все значения, возвращенные функцией сопроцесса (когда сопроцесс завершается). В случае ошибок, `resume` возвращает **false** и сообщение об ошибке.

coroutine.running ()

Возвращает запущенный сопроцесс и логическое значение; true, если сопроцесс это главный поток.

coroutine.status (co)

Возвращает статус сопроцесса `co`, как строку: `"running"` - сопроцесс запущен (т.е. он вызвал `status`); `"suspended"` - сопроцесс приостановлен в вызове `yield` или еще не запущен; `"normal"` - сопроцесс активен, но не выполняется (т.е. он был продолжен другим сопроцессом); `"dead"` - сопроцесс завершил своё тело или был остановлен с ошибкой.

coroutine.wrap (f)

Создает новый сопроцесс с телом `f`. `f` должен быть функцией. Возвращает функцию, которая возобновляет сопроцесс при каждом её вызове. Все аргументы, переданные этой функции, ведут себя как дополнительные аргументы в `resume`. Возвращает те же значения, что и `resume`, за исключением первого логического значения. В случае ошибки, распространяет ошибку.

coroutine.yield (···)

Приостанавливает выполнение вызывающего сопроцесса. Все аргументы `yield` передаются, как дополнительные результаты в `resume`.

6.3 – Модули

Эта пакетная библиотека предоставляет базовые возможности для загрузки модулей в Lua. Она экспортирует одну функцию напрямую в глобальное окружение: `require`. Всё остальное экспортируется в таблице `package`.

require (modname)

Загружает переданный модуль. Функция начинает с просмотра таблицы `package.loaded` для определения, что модуль `modname` уже загружен. Если это так, то `require` возвращает значение, хранящееся в `package.loaded[modname]`. Иначе, пробует найти *загрузчик* для модуля.

При поиске загрузчика `require` руководствуется последовательностью `package.searchers`. Изменяя эту последовательность, вы можете изменить то, как `require` ищет модуль. Следующее объяснение базируется на конфигурации по умолчанию для `package.searchers`.

Сначала `require` запрашивает `package.preload[modname]`. Если это значение существует, то это

значение (которое может быть функцией) это загрузчик. Иначе `require` ищет Lua загрузчик, используя путь, хранящийся в `package.path`. Если это тоже неудачно, она ищет C загрузчик, используя путь, хранящийся в `package.cpath`. Если и это неудачно, пробует *все в одном* загрузчик (см. `package.searchers`).

Как только загрузчик найден, `require` вызывает загрузчик с двумя аргументами: `modname` и дополнительное значение, зависящее от того, как получен загрузчик. (Если загрузчик пришел из файла, это дополнительное значение будет именем файла.) Если загрузчик возвращает не-nil значение, `require` записывает возвращенное значение в `package.loaded[modname]`. Если загрузчик не вернул не нулевое значение и `package.loaded[modname]` не присвоено никакое значение, то `require` пишет туда **true**. В любом случае, `require` возвращает финальное значение `package.loaded[modname]`.

Если во время загрузки или запуска модуля происходит ошибка, или если не найден ни один загрузчик для модуля, `require` генерирует ошибку.

package.config

Строка описывающая некоторые конфигурации времени компиляции для пакетов. Эта строка (string) состоит из последовательности строк (line):

- Первая строка - это строка разделитель директорий. По умолчанию, это '\' для Windows и '/' для всех остальных систем.
- Вторая строка - символ, который разделяет шаблоны в пути. По умолчанию это '**;**'.
- Третья строка - это строка, которая обозначает точки замены в шаблоне. По умолчанию это '**?**'.
- Четвертая строка - это строка, которая в пути в Windows, заменяется на директорию исполняемого файла. По умолчанию это '**!**'.
- Пятая строка - это метка для игнорирования всего текста после неё, когда формируется имя

функции `luaopen_`. По умолчанию это '-'.

package.cpath

Этот путь используется `require` для поиска C загрузчика.

Lua инициализирует C путь `package.cpath` также, как и Lua путь `package.path`, используя переменную окружения `LUA_CPATH_5_3` или переменную окружения `LUA_CPATH` или путь по умолчанию, определенный в `luaconf.h`.

package.loaded

Таблица, используемая `require` для контроля за тем, какие модули уже загружены. Когда вы загружаете модуль `modname` и `package.loaded[modname]` не false, `require` просто возвращает хранящееся там значение.

Эта переменная только ссылка на реальную таблицу; присваивание этой переменной не изменяет таблицу, используемую функцией `require`.

package.loadlib (libname, funcname)

Динамически связывает (link) хостовую программу с C библиотекой `libname`.

Если `funcname = "*"`, то она только связывает с библиотекой, делая экспортируемые библиотекой символы доступными для других динамических библиотек. Иначе, она ищет функцию `funcname` внутри библиотеки и возвращает эту функцию, как C функцию. Так, `funcname` должна соответствовать прототипу `lua_CFunction` (см. `lua_CFunction`).

Это низкоуровневая функция. Она полностью обходит систему пакетов и модулей. В отличие от `require`, она не производит поисков пути и не добавляет расширения автоматически. `libname` должен быть завершенным именем файла C библиотеки, включая, если необходимо, путь и расширение. `funcname` должен быть точным именем, экспортируемым C библиотекой (которое может зависеть от используемого C компилятора и компоновщика).

Эта функция не поддерживается стандартным C. Т.к., она доступна только для некоторых платформ (Windows, Linux, Mac OS X, Solaris, BSD и другие Unix системы, которые поддерживаю стандарт `dlfcn`).

package.path

Этот путь используется `require` для поиска Lua загрузчика.

Во время запуска Lua инициализирует эту переменную значением переменной окружения `LUA_PATH_5_3` или переменной окружения `LUA_PATH`, или значением по умолчанию, определенным в `luaconf.h`, если эти переменные окружения не определены. Любое `";;"` в значении переменной окружения заменяется путем по умолчанию.

package.preload

Таблица для хранения загрузчиков для специфических модулей (см. `require`).

Эта переменная только ссылается на реальную таблицу; присваиваение этой переменной не изменяет таблицу, используемую функцией `require`.

package.searchers

Таблица, используемая `require` для контроля над тем, как загружать модули.

Каждый элемент в этой таблице это *функция искатель (searcher)*. Когда ищется модуль, `require` вызывает каждую из этих функций в возрастающем порядке, с именем модуля (аргумент переданный в `require`) в качестве единственного параметра. Функция может вернуть другую функцию (*загрузчик* модуля) и дополнительное значение, которое будет передано загрузчику, или строку, объясняющую, почему модуль не найден (или **nil** если нечего сказать).

Lua инициализирует эту таблицу четырьмя функциями искателями.

первый искатель просто ищет загрузчик в таблице `package.preload`.

Втрой искатель ищет загрузчик для Lua библиотеки, используя путь `package.path`. Поиск производится как описано в функции `package.searchpath`.

Третий искатель ищет загрузчик для C библиотеки, используя путь `package.cpath`. Поиск производится как описано в функции `package.searchpath`. Например, если C путь это строка

```
"./?.so;./?.dll;/usr/local/?/init.so"
```

искатель для модуля `foo` будет пробовать открыть файлы `./foo.so`, `./foo.dll`, и `/usr/local/foo/init.so`, в этом порядке. Когда он найдет C библиотеку, сначала он использует средства динамического связывания с библиотекой. Затем попытается найти в библиотеке C функцию для использования в качестве загрузчика. Имя этой C функции это строка "`luaopen_`", склеенная с копией имени модуля, где каждая точка заменена на подчеркивание. Более того, если имя модуля имеет дефис, его суффикс после первого дефиса удаляется (дефис тоже удаляется). Например, если имя модуля `a.b.c-v2.1`, имя функции будет `luaopen_a_b_c`.

Четвертый искатель пробует *все в одном загрузчик*. Он ищет C путь для библиотеки с корневым

путем переданного модуля. Например, когда требуется `a.b.c`, он будет искать C библиотеку `a`. Если существует, он заглянет в нее для открытия функции подмодуля; в нашем примере, это будет `luaopen_a_b_c`. С этой возможностью, пакет может упаковывать различные C подмодули в одной библиотеке, где каждый подмодуль имеет свою оригинальную функцию открытия.

Все искатели, за исключением первого (preload), возвращают имя файла, где найден модуль, как дополнительное значение, возвращаемое `package.searchpath`. Первый искатель не возвращает дополнительное значение.

package.searchpath (name, path [, sep [, rep]])

Ищет имя `name` в пути `path`.

Путь - это строка, содержащая последовательность *шаблонов*, разделенных точкой с запятой (;). Для каждого шаблона, функция заменяет каждый знак вопроса (если существует) копией `name`, где все случаи `sep` (точка, по умолчанию) заменяются на `rep` (системный разделитель директорий, по умолчанию), и пытается открыть результирующее имя файла.

Например, если путь это строка

```
"./?.lua;./?.lc;/usr/local/?/init.lua"
```

поиск имени `foo.a` будет пытаться открыть файлы `./foo/a.lua`, `./foo/a.lc` и `/usr/local/foo/a/init.lua`, в этом порядке.

Возвращает результирующее имя первого файла, который можно открыть в режиме чтения (после закрытия файла), или **nil** и сообщение об ошибке, если ничего не удалось. (Это сообщение об ошибке перечисляет все имена файлов, которые пыталась открыть функция.)

6.4 – Работа со строками

Эта библиотека предоставляет общие функции для работы со строками, такие как поиск, извлечение подстрок и сопоставление шаблонов. Когда индексируются строки в Lua, первый символ находится на позиции 1 (не на 0, как в C). Допускаются негативные индексы, они интерпретируются как индексирование обратно (задом наперед), с конца строки. Таким образом, последний символ находится на позиции -1 и так далее.

Строковая библиотека предоставляет все свои функции в таблице `string`. Она также устанавливает метатаблицу для строк, где поле `__index` указывает на таблицу `string`. Следовательно, вы можете использовать строковые функции в объектно-ориентированном стиле. Например, выражение `string.byte(s,i)` может быть записано как `s:byte(i)`.

Строковая библиотека предполагает кодирование символов одним байтом.

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

Возвращает внутренние цифровые коды символов `s[i]`, `s[i+1]`, ..., `s[j]`. По умолчанию `i = 1`; `j = i`. Эти индексы следуют тем же правилам, что и в функции `string.sub`.

Цифровые коды не обязательно портабельны между платформами.

string.char (···)

Получает ноль или более целых. Возвращает строку длиной равной количеству аргументов, в которой каждый символ имеет внутренний цифровой код равный соответствующему аргументу.

Цифровые коды не обязательно портабельны между платформами.

string.dump (function [, strip])

Возвращает строку содержащую бинарное представление (*бинарный кусок*) переданной функции, так что `load` для этой строки возвращает копию функции (но с новыми upvalue). Если `strip = true`, бинарное представление может не включать всю отладочную информацию о функции, для уменьшения размера.

Функции с upvalue сохраняют только количество upvalue. При загрузке, эти upvalue получают свежие экземпляры, содержащие **nil**. (Вы можете использовать отладочную библиотеку, чтобы сохранить и перезагрузить upvalue функции в том виде, как вам нужно.)

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

Ищет первое совпадение шаблона `pattern` (см. §6.4.1) в строке `s`. Если совпадение найдено, то `find` возвращает индексы `s`, где совпадение начинается и заканчивается; иначе, возвращает **nil**. Третий опциональный цифровой аргумент `init` определяет, где начинать поиск; по умолчанию он равен 1 и может быть отрицательным. Значение **true** в качестве четвертого опционального аргумента `plain` выключает возможности поиска шаблонов, так функция выполняет плоский поиск подстроки, без магических символов в `pattern`. Учтите, что если передан `plain`, то должен быть передан и `init`.

Если шаблон имеет захваты (capture), то при успешном совпадении захваченные значения также возвращаются, после двух индексов.

string.format (formatstring, ···)

Возвращает форматированную версию переменного количества аргументов, следуя описанию в первом аргументе (должен быть строкой). formatstring - следует тем же правилам, что и в функции `sprintf` в ISO C. Только отличается тем, что опции/модификаторы `*`, `h`, `L`, `l`, `n` и `p` не

поддерживаются, и тем, что имеет дополнительную опцию q. Опция q форматирует строку между двойными кавычками и использует управляющие символы, когда необходимо гарантировать, что строка может быть прочитана Lua интерпретатором обратно. Например, вызов

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

может выдать строку:

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

Опции A, a, E, e, f, G и g - все ожидают цифровой аргумент. Оции c, d, i, o, u, X и x - ожидают целое. Опция q - ожидает строку. Опция s - ожидает строку без встроенных нулей; если аргумент не строка, он конвертируется следуя тем же правилам, что и в tostring.

Когда Lua скомпилирована с не C99 компилятором, опции A и a (шестнадцатиричные вещественные числа) не поддерживают модификаторы (флаги, ширина, длина).

string.gmatch (s, pattern)

Возвращает функцию-итератор, которая при каждом вызове возвращает следующие захваченные значения из pattern (см. §6.4.1) по строке s. Если pattern не определяет захватов, то в каждом вызове возвращается целое совпадение.

Например, следующий цикл будет перебирать все слова из строки s, печатая по одному в строке:

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

```
end
```

Следующий пример собирает в таблице все пары `key=value` из строки:

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

Для этой функции, символ '^' в начале шаблона не работает как якорь, т.к. это мешает итерации.

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

Возвращает копию `s`, в которой все (или первые `n`, если передано) совпадения шаблона `pattern` (см. §6.4.1) заменены на замещающую строку, определенную параметром `repl`, который может быть строкой, таблицей или функцией. `gsub` также возвращает общее число совпадений, как второе значение. Имя `gsub` происходит от *Global SUBstitution* (глобальная подстановка).

Если `repl` это строка, то её значение используется для замены. Символ `%` работает, как управляющий символ: любая последовательность в `repl` в виде *%d*, с *d* между 1 и 9, соответствует *d*-ой захваченной подстроке. Последовательность *%0* соответствует полному совпадению. Последовательность %% соответствует одному символу %.

Если `repl` это таблица, то таблица запрашивается для каждого совпадения, используя первое захваченное значение, как ключ.

Если `repl` это функция, то эта функция вызывается для каждого совпадения, все захваченные подстроки передаются в качестве аргументов, по порядку.

В любом случае, если шаблон не имеет захватов, то он ведет себя так, будто весь шаблон находится в захвате.

Если значение, возвращенное из табличного запроса или из функции, это строка или число, то оно используется, как замещающая строка; иначе, если это **false** или **nil**, то замена не производится (т.е. оригинальное содержимое совпадения сохраняется в строке).

Несколько примеров:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return load(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.3"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
```

```
--> x="lua-5.3.tar.gz"
```

string.len (s)

Получает строку и возвращает её длину. Пустая строка `""` имеет длину 0. Встроенные нули считаются, так `"a\000bc\000"` имеет длину 5.

string.lower (s)

Получает строку и возвращает её копию с заменой всех символов в верхнем регистре на символы в нижнем регистре. Остальные символы не изменяются. Определение какие символы в верхнем регистре зависит от текущей локали.

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

Ищет первое *совпадение* шаблона `pattern` (см. §6.4.1) в строке `s`. Если находит, то `match` возвращает захваченные значения из шаблона; иначе возвращает **nil**. Если `pattern` не описывает захватов, то возвращается целое совпадение. Третий опциональный цифровой аргумент `init` определяет, где начинать поиск; по умолчанию он равен 1, и может быть отрицательным.

string.pack (fmt, v1, v2, ···)

Возвращает бинарную строку, содержащую значения `v1`, `v2` и т.д. упакованными (т.е. записанными в бинарной форме) согласно форматной строке `fmt` (см. §6.4.2).

string.packsize (fmt)

Возвращает размер результирующей строки из `string.pack` с переданным форматом. Форматная строка не может иметь опций переменной длины 's' или 'z' (см. §6.4.2).

string.rep (s, n [, sep])

Возвращает строку, которая состоит из слияния `n` копий строки `s`, разделенных строкой `sep`. Значение по умолчанию для `sep` это пустая строка (т.е. нет разделителя). Возвращает пустую строку, если `n` отрицательное значение.

string.reverse (s)

Возвращает строку, в которой символы `s` идут в обратном порядке.

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

Возвращает подстроку `s`, начинающуюся на `i` и продолжающуюся до `j`; `i` и `j` могут быть отрицательными. Если `j` отсутствует, то он подразумевается равным -1 (тоже что и длина строки). В частности, вызов `string.sub(s,1,j)` возвращает префикс `s` длиной `j`, и `string.sub(s, -i)` возвращает суффикс `s` длиной `i`.

Если после трансляции отрицательных индексов `i` < 1, он корректируется до 1. Если `j` больше длины строки, он корректируется до этой длины. Если после этих преобразований `i` > `j`, функция возвращает пустую строку.

string.unpack (fmt, s [, pos])

Возвращает значения, упакованные в строке `s` (см. `string.pack`) согласно форматной строке `fmt` (см. §6.4.2). Опциональный параметр `pos` отмечает, где начинать чтение в `s` (по умолчанию 1). После

чтения значений, эта функция также возвращает индекс первого не прочитанного байта в `s`.

string.upper (s)

Получает строку и возвращает её копию с заменой всех символов в нижнем регистре на символы в верхнем регистре. Остальные символы не изменяются. Определение какие символы в нижнем регистре зависит от текущей локали.

6.4.1 – Шаблоны

Шаблоны в Lua описываются регулярными строками, которые интерпретируются, как шаблоны, функциями сопоставления шаблонов `string.find`, `string.gmatch`, `string.gsub` и `string.match`. Этот раздел описывает синтаксис и значение (сопоставление) этих строк.

Символьный класс:

Символьный класс используется для представления набора символов. В описании символьного класса допустимы следующие комбинации:

- ***x*:** (где *x* - не один из *магических символов* `^$()%.[]*+-?`) представляет символ *x* непосредственно.
- **`.`:** (точка) представляет все символы.
- **`%a`:** представляет все буквы.
- **`%c`:** представляет все управляющие символы.
- **`%d`:** представляет все цифры.
- **`%g`:** представляет все печатаемые символы, кроме пробела.
- **`%l`:** представляет все буквы в нижнем регистре.

- **%p:** представляет все знаки пунктуации.
- **%s:** представляет все пробельные символы.
- **%u:** представляет все буквы в верхнем регистре.
- **%w:** представляет все алфавитно-цифровые символы.
- **%x:** представляет все шестнадцатиричные символы.
- **%*x*:** (где *x* - не алфавитно-цифровой символ) представляет символ *x*. Это стандартный способ кодирования магических символов. Любой не алфавитно-цифровой символ (включая все знаки пунктуации, даже не магические) могут предваряться '%', когда используются для представления себя в шаблоне.
- **[*набор*]:** представляет класс, который является объединением всех символов в *наборе*. Диапазон символов может быть определен отделением конечного символа диапазона, в восходящем порядке, символом '-'. Все классы *%x*, описанные выше, также могут быть включены в *набор*, как компоненты. Все остальные символы в *наборе* представляют непосредственно себя. Например, `[%w_]` (или `[_%w]`) представляет все алфавитно-цифровые символы и подчеркивание, `[0-7]` представляет восьмиричные цифры, `[0-7%l%-]` представляет восьмиричные цифры, буквы в нижнем регистре и символ '-'.

  Взаимодействие между диапазонами и классами не определено. Следовательно, шаблоны как `[%a-z]` или `[a-%%]` не имеют значения.

- **[^*набор*]:** представляет отрицание *набора*, где *набор* интерпретируется, как описано выше.

Для всех классов, представленных одним символом (`%a`, `%c` и др.), соответствующие представления с буквой в верхнем регистре отрицают класс. Например, `%S` представляет все не пробельные символы.

Определения букв, пробелов и других групп символов зависят от текущей локали. В частности, класс `[a-z]` может не быть эквивалентом `%l`.

Элемент шаблона:

элементом шаблона может быть

- односимвольный класс, который соответствует одному символу в классе;
- односимвольный класс с последующим символом '*', который соответствует нулю или более повторам символов в классе. Этот повтор элементов всегда будет соответствовать самой длинной возможной последовательности;
- односимвольный класс с последующим символом '+', который соответствует одному или более повторам символов в классе. Этот повтор элементов всегда будет соответствовать самой длинной возможной последовательности;
- односимвольный класс с последующим символом '-', который соответствует нулю или более повторам символов в классе. В отличие от '*', этот повтор элементов всегда будет соответствовать самой короткой возможной последовательности;
- односимвольный класс с последующим символом '?', который соответствует нулю или одному случаю символа в классе. Он всегда соответсвует одному вхождению, если возможно;
- %*n*, для *n* между 1 и 9; этот элемент соответствует подстроке равной *n*-й захваченной строке (см. ниже);
- %b*xy*, где *x* и *y* два четких символа; этот элемент соответствует строкам, которые начинаются с *x* и заканчиваются на *y*, и где *x* и *y* *сбалансированы*. Это значит, что если при чтении строки слева направо, считать *+1* для *x* и *-1* для *y*, завершающий *y* это первый *y*, где счет достигнет 0. Например, элемент %b() соответсвует выражениям в сбалансированных скобках.
- %f[*набор*], *граничный шаблон*; этот элемент соответствует пустой строке в любой позиции такой, что следующий символ принадлежит *набору* и предыдущий символ не принадлежит *набору*. *Набор* интерпретируется, как описано выше. Начало и конец субъекта обрабатывается, как если там символ '\0'.

Шаблон:

Шаблон - это последовательность элементов шаблона. Символ '^' в начале шаблона фиксирует его в начале строки. Символ '$' в конце шаблона фиксирует его в конце строки. В остальных позициях '^' и '$' не имеют специального значения и представляют сами себя.

Захваты (capture):

Шаблон может содержать подшаблоны заключенные в скобки; они описывают *захваты*. Когда совпадение успешно, подстроки, которые соответсвуют захватам (*захваченные*), сохраняются для последующего использования. Захваты нумеруются соответственно их левым скобкам. Например, в шаблоне `"(a*(.)%w(%s*))"`, часть строки, соответствующая `"a*(.)%w(%s*)"`, сохраняется как первый захват (и следовательно имеет номер 1); соответствие `"."` захватывается с номером 2, и часть соответствующая `"%s*"` имеет номер 3.

Специальный случай, пустой захват `()` захватывает текущую позицию в строке (число). Например, если мы применим шаблон `"()aa()"` к строке `"flaaap"`, будет сделано два захвата: 3 и 5.

6.4.2 – Формат строк для string.pack и string.unpack

Первый аргумент в `string.pack`, `string.packsize` и `string.unpack` это строка формата, которая описывает формат создаваемой или читаемой структуры.

Строка формата - это последовательность опций преобразования. Опции преобразования следующие:

- **<:** устанавливает прямой порядок байт (little endian)
- **>:** устанавливает обратный порядок байт (big endian)

- **=:** устанавливает исконный порядок байт
- **![*n*]:** устанавливает максимальное выравнивание равным `n` (по умолчанию, исконное выравнивание)
- **b:** знаковый байт (`char`)
- **B:** беззнаковый байт (`char`)
- **h:** знаковый `short` (исконный размер)
- **H:** беззнаковый `short` (исконный размер)
- **l:** знаковый `long` (исконный размер)
- **L:** беззнаковый `long` (исконный размер)
- **j:** `lua_Integer`
- **J:** `lua_Unsigned`
- **T:** `size_t` (исконный размер)
- **i[*n*]:** знаковый `int` с `n` байт (по умолчанию, исконный размер)
- **I[*n*]:** беззнаковый `int` с `n` байт (по умолчанию, исконный размер)
- **f:** `float` (исконный размер)
- **d:** `double` (исконный размер)
- **n:** `lua_Number`
- **c*n*:** строка фиксированного размера с `n` байт
- **z:** завершаемая нулем строка
- **s[*n*]:** строка с предваряющим её размером, кодированным как беззнаковое целое с `n` байт (по умолчанию, это `size_t`)
- **x:** один байт заполнения
- **X*op*:** пустой элемент, который выравнивает в соответствии с опцией `op` (которая в противном случае игнорируется)
- **' ':** (пустое пространство) игнорируется

("[*n*]" означает опциональную целую цифру.) Кроме заполнения, пробелов и конфигураций (опции

"xX <=>!"), каждая опция соответствует аргументу (в `string.pack`) или результату (в `string.unpack`).

Для опций "`!`*n*", "`s`*n*", "`i`*n*" и "`I`*n*", `n` может быть целым от 1 до 16. Все целые опции проверяют переполнения; `string.pack` проверяет, что переданное значение поместится в переданный размер; `string.unpack` проверяет, что прочитанное значение поместится в целое Lua.

Любая строка формата начинается так, будто содержит префикс "`!1=`", т.е. с максимальным выравниванием 1 (без выравнивания) и исконным порядком байт.

Выравнивание работает так: для каждой опции формат получает дополнительное заполнение, пока не начнутся данные по смещению, которое равно произведению минимума между размером опции и максимального выравнивания; этот минимум должен быть степенью 2. Опции "`c`" и "`z`" не выравниваются; опция "`s`" следует выравниванию её начального целого.

Все заполнения заполняются нулями в `string.pack` (и игнорируются в `string.unpack`).

6.5 – Поддержка UTF-8

Эта библиотека предоставляет базовую поддержку для кодировки UTF-8. Она предоставляет все свои функции в таблице `utf8`. Эта библиотека не предоставляет другой поддержки для Unicode, кроме обработки кодировки. Все операции, нуждающиеся в значении символа, такие как классификация символов, не входят в эту область.

Пока не установлено иначе, все функции, которые ожидают позицию байта, как параметр, предполагают, что переданная позиция является также началом последовательности байт или позиция плюс длина строки. Как и в строковой библиотеке, отрицательные индексы отсчитываются с конца строки.

utf8.char (···)

Получает ноль или более целых, конвертирует каждое из них в соответствующую последовательность байт UTF-8 и возвращает строку со слиянием всех этих последовательностей.

utf8.charpattern

Шаблон (строка, не функция) `"[\0-\x7F\xC2-\xF4][\x80-\xBF]*"` (см. §6.4.1), который соответствует точно одной последовательности байт UTF-8, предполагается, что субъект это правильная UTF-8 строка.

utf8.codes (s)

Возвращает значения такие, что конструкция

```
for p, c in utf8.codes(s) do body end
```

будет перебирать все символы в строке `s`, где `p` - позиция (в байтах) и `c` - кодовая точка для каждого символа. Функция вызывает ошибку, если встретит неправильную последовательность байт.

utf8.codepoint (s [, i [, j]])

Возвращает кодовые точки (как целые) для всех символов в `s`, начиная с байта на позиции `i` и заканчивая `j` (включительно). По умолчанию, `i` = 1 и `j` = `i`. Функция вызывает ошибку, если встретит неправильную последовательность байт.

utf8.len (s [, i [, j]])

Возвращает количество символов UTF-8 в строке `s`, начиная с байта на позиции `i` и заканчивая `j` (включительно). По умолчанию, `i = 1` и `j = -1`. Если функция встречает неправильную последовательность байт, она возвращает false и позицию первой неправильной последовательности байт.

utf8.offset (s, n [, i])

Возвращает позицию (в байтах), где начинается `n`-й закодированный символ строки `s` (счет идет с позиции `i`). При отрицательном значении `n`, получает символы перед позицией `i`. По умолчанию, `i = 1`, когда `n` положительное значение, и `i = #s + 1` иначе, так `utf8.offset(s, -n)` получает смещение `n`-го символа с конца строки. Если определенный символ ни в субъекте, ни справа после его конца, функция возвращает **nil**.

Специальный случай, когда `n = 0`, функция возвращает начало закодированного символа, который содержит `i`-й байт строки `s`.

Эта функция предполагает, что `s` это правильная строка UTF-8.

6.6 – Работа с таблицами

Эта библиотека предоставляет базовые функции для работы с таблицами. Она предоставляет все свои функции в таблице `table`.

Помните, что когда операция нуждается в длине таблицы, таблица должа содержать соответсвующую последовательность или иметь метаметод `__len` (см. §3.4.7). Все функции игнорируют не цифровые ключи в таблицах, полученных как аргументы.

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

Получает список list, где все элементы строки или числа, возвращает строку `list[i]..sep..list[i+1] ··· sep..list[j]`. По умолчанию, `sep` это пустая строка, `i = 1` и `j = #list`. Если `i > j`, возвращает пустую строку.

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

Вставляет элемент `value` на позицию `pos` в список `list`, сдвигая элементы вверх `list[pos], list[pos+1], ···, list[#list]`. По умолчанию, `pos = #list+1`, так вызов `table.insert(t,x)` вставляет `x` в конец списка `t`.

table.move (a1, f, e, t [,a2])

Перемещает элементы из таблицы `a1` в таблицу `a2`. Эта функция эквивалентна множественному присваиванию: `a2[t],··· = a1[f],···,a1[e]`. По умолчанию, `a2 = a1`. Целевой диапазон может перекрываться с диапазоном источником. Количество элементов для перемещения должно помещаться в целое Lua.

table.pack (···)

Возвращает новую таблицу, в которой все параметры сохранены с ключами 1, 2 и т.д. и поле "`n`" содержит количество параметров. Учтите, что результирующая таблица может не быть последовательностью.

table.remove (list [, pos])

Удаляет из списка `list` элемент на позиции `pos`, возвращая значение этого элемента. Когда `pos` это целое между 1 и `#list`, функция сдвигает элементы вниз `list[pos+1], list[pos+2], ···, list[#list]` и стирает элемент `list[#list]`; Индекс `pos` может быть 0, когда `#list` = 0, или `#list + 1`; в этих случаях функция стирает элемент `list[pos]`.

По умолчанию, `pos` = `#list`, так вызов `table.remove(l)` удаляет последний элемент списка `l`.

table.sort (list [, comp])

Сортирует элементы полученного списка, *на месте*, от `list[1]` до `list[#list]`. Если передан параметр `comp`, он должен быть функцией, которая получает два элемента списка и возвращает true, когда первый элемент должен находиться перед вторым в финальном упорядочении (так выражение `not comp(list[i+1],list[i])` будет истинным после сортировки). Если параметр `comp` не передан, то взамен Lua использует стандартный оператор `<`.

Алгоритм сортировки не стабилен; т.е. элементы, считающиеся равными в этом упорядочении, в результате сортировки могут изменить свои относительные позиции.

table.unpack (list [, i [, j]])

Возвращает элементы из полученного списка. Эта функция эквивалентна

```
return list[i], list[i+1], ···, list[j]
```

По умолчанию, `i` = 1 и `j` = `#list`.

6.7 – Математические функции

Эта библиотека предоставляет базовые математические функции. Она предоставляет все свои функции и константы в таблице `math`. Функции с комментарием `"integer/float"` выдают целые результаты для целых аргументов и вещественные результаты для вещественных (или смешанных) аргументов. Функции округления (`math.ceil`, `math.floor` и `math.modf`) возвращают целое, когда результат помещается в диапазон целых, или вещественное иначе.

math.abs (x)

Возвращает абсолютное значение x. (integer/float)

math.acos (x)

Возвращает арккосинус x (в радианах).

math.asin (x)

Возвращает арксинус x (в радианах).

math.atan (y [, x])

Возвращает арктангенс `y/x` (в радианах), но использует знаки обоих параметров для поиска квадранта результата. (Также корректно обрабатывает случай, когда x = 0.)

По умолчанию x = 1, так вызов `math.atan(y)` возвращает арктангенс y.

math.ceil (x)

Возвращает наименьшее целое значение, которое больше или равно x.

math.cos (x)

Возвращает косинус x (в радианах).

math.deg (x)

Преобразует угол x из радиан в градусы.

math.exp (x)

Возвращает значение e^x (где e - основание натурального логарифма).

math.floor (x)

Возвращает наибольшее значение, которое меньше или равно x.

math.fmod (x, y)

Возвращает остаток от деления x на y, который округляет частное к нулю. (integer/float)

math.huge

Вещественное значение `HUGE_VAL`, которое больше любого другого числового значения.

math.log (x [, base])

Возвращает логарифм x по основанию `base`. По умолчанию, `base = e` (так функция возвращает

натуральный логарифм x).

math.max (x, ···)

Возвращает аргумент с максимальным значением, в соответствии с Lua оператором <. (integer/float)

math.maxinteger

Целое с максимальным значением для целого.

math.min (x, ···)

Возвращает аргумент с минимальным значением, в соответствии с Lua оператором <. (integer/float)

math.mininteger

Целое с минимальным значением для целого.

math.modf (x)

Возвращает целую и дробную часть x. Второй результат всегда вещественное число.

math.pi

Значение *π*.

math.rad (x)

Преобразует угол x из градусов в радианы.

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

Когда вызвана без аргументов, возвращает псевдослучайное вещественное число с однородным распределением в диапазоне *[0,1)*. Когда вызвана с двумя целыми `m` и `n`, `math.random` возвращает псевдослучайное целое с однородным распределением в диапазоне *[m, n]*. (Значение *m-n* не может быть отрицательным и должно помещаться в целое Lua.) Вызов `math.random(n)` эквивалентен вызову `math.random(1,n)`.

Эта функция является интерфейсом к генератору псевдослучайных чисел, предоставляемому C. Нет гарантий для его статистических свойств.

math.randomseed (x)

Устанавливает x как "затравку" (seed) для генератора псевдослучайных чисел: одинаковые затравки производят одинаковые последовательности чисел.

math.sin (x)

Возвращает синус x (в радианах).

math.sqrt (x)

Возвращает квадратный корень x. (Для вычисления этого значения вы также можете использовать выражение `x^0.5`.)

math.tan (x)

Возвращает тангенс x (в радианах).

math.tointeger (x)

Если значение x можно преобразовать в целое, возвращает целое. Иначе, возвращает **nil**.

math.type (x)

Возвращает "`integer`" - если x целое, "`float`" - если x вещественное, или **nil** - если x не число.

math.ult (m, n)

Возвращает логическое значение, true, если целое `m` ниже целого `n`, когда они сравниваются как беззнаковые целые.

6.8 – Средства ввода-вывода

Библиотека ввода-вывода предоставляет два разных стиля для файловых манипуляций. Первый использует подразумевающиеся описатели файлов (handle); т.е. там есть операции установки файла ввода и файла вывода по умолчанию, и все операции ввода-вывода используют эти файлы. Второй стиль использует явные описатели файлов.

При использовании неявных описателей файлов, все операции предоставляются в таблице `io`. При использовании явных описателей, операция `io.open` возвращает описатель файла и затем все операции предоставляются, как методы этого описателя.

Таблица `io` также предоставляет три предопределенных файловых описателя с обычными

значениями из C: `io.stdin`, `io.stdout` и `io.stderr`. Библиотека ввода-вывода никогда не закрывает эти файлы.

Пока не установлено иначе, все функции ввода-вывода возвращают **nil** при ошибке (и сообщение об ошибке, как второй результат, и зависящий от системы код ошибки, как третий) и отличное от **nil** значение при успехе. На не POSIX системах, формирование сообщения об ошибке и кода ошибки может не быть потокобезопасным, т.к. они полагаются на глобальную C переменную `errno`.

io.close ([file])

Эквивалентно `file:close()`. Без `file`, закрывает выходной файл по умолчанию.

io.flush ()

Эквивалентно `io.output():flush()`.

io.input ([file])

Когда вызвана с именем файла, открывает данный файл (в текстовом режиме), и устанавливает его описатель, как файл ввода по умолчанию. Когда вызвана с описателем файла, просто устанавливает этот описатель, как файл ввода по умолчанию Когда вызвана без параметров, возвращает текущий файл ввода по умолчанию.

В случае ошибок эта функция генерирует ошибку, вместо возвращения кода ошибки.

io.lines ([filename ···])

Открывает переданный файл в режиме чтения и возвращает функцию итератор, которая работает

подобно `file:lines(···)` для открытого файла. Когда итератор доходит до конца файла, он ничего не возвращает (для завершения цикла) и автоматически закрывает файл.

Вызов `io.lines()` (без имени файла) эквивалентно `io.input():lines("*l")`; т.е. он перебирает линии файла ввода по умолчанию. В этом случае он не закрывает файл при завершении цикла.

В случае ошибок эта функция генерирует ошибку, вместо возвращения кода ошибки.

io.open (filename [, mode])

Функция открывает файл в режиме, определяемым строкой `mode`. Возвращает новый описатель файла, или, в случае ошибок, возвращает **nil** и сообщение об ошибке.

Строка `mode` может быть следующей:

- **"r":** режим чтения (по умолчанию);
- **"w":** режим записи;
- **"a":** режим добавления;
- **"r+":** режим обновления, все предыдущие данные сохраняются;
- **"w+":** режим обновления, все предыдущие данные стираются;
- **"a+":** режим добавления и обновления, предыдущие данные сохраняются, запись разрешена только в конец файла.

Строка `mode` также может содержать 'b' в конце, это нужно на некоторых системах для открытия файла в бинарном режиме.

io.output ([file])

Подобно `io.input`, но для файла вывода по умолчанию.

io.popen (prog [, mode])

Эта функция зависит от системы и не доступна на некоторых платформах.

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

io.read (···)

Эквивалентно `io.input():read(···)`.

io.tmpfile ()

Возвращает описатель для временного файла. Этот файл открывается в режиме обновления и автоматически удаляется по завершении программы.

io.type (obj)

Проверяет, что независимый `obj` это правильный описатель файла. Возвращает строку `"file"` - если `obj` открытый описатель файла, `"closed file"` - если `obj` закрытый описатель файла, или **nil** - если `obj` не является описателем файла.

io.write (···)

Эквивалентно `io.output():write(···)`.

file:close ()

Закрывает `file`. Учтите, что файлы закрываются автоматически, когда их описатели собраны сборщиком мусора, но это может занять неопределенное количество времени.

Когда закрываемый описатель файла создан функцией `io.popen`, `file:close` возвращает те же значения, что и `os.execute`.

file:flush ()

Сохраняет все записанные данные в `file`.

file:lines (···)

Возвращает функцию итератор, которая при каждом вызове читает файл в соответствии с переданными форматами. Когда формат не передан, использует "`l`", по умолчанию. Например, конструкция

```
for c in file:lines(1) do body end
```

будет перебирать все символы файла, начиная с текущей позиции. В отличие от `io.lines`, эта функция не закрывает файл после завершения цикла.

В случае ошибок эта функция генерирует ошибку, вместо возвращения кода ошибки.

file:read (···)

Читает файл `file`, в соответствии с переданными форматами, которые определяют, что читать. Для

каждого формата, функция возвращает строку или число с прочитанными символами, или **nil**, если не может прочитать данные в этом формате. (В этом последнем случае, функция не читает последующие форматы.) Когда вызвана без форматов, использует по умолчанию формат, читающий следующую строку (см. ниже).

Доступны следующие форматы

- **"n":** читает число и возвращает его, как вещественное или целое, следуя лексическим соглашениям Lua. (Число может содержать начальные пробелы и знак.) Этот формат всегда читает самую длинную входную последовательность, которая является правильным префиксом для числа; если префикс не правильный (пустая строка, "`0x`" или "`3.4e-`"), он отбрасывается и функция возвращает **nil**.
- **"a":** читает весь файл, начиная с текущей позиции. В конце файла возвращает пустую строку.
- **"l":** читает следующую строку, пропуская символ конца строки, возвращает **nil** в конце файла. Это формат по умолчанию.
- **"L":** читает следующую строку, сохраняя символ конца строки (если есть), возвращает **nil** в конце файла.
- ***число*:** читает строку этой длины в байтах, возвращает **nil** в конце файла. Если `number = 0`, ничего не читает и возвращает пустую строку, или **nil** в конце файла.

Форматы "`l`" и "`L`" должны использоваться только для текстовых файлов.

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

Устанавливает и возвращает позицию в файле, измеряемую от начала файла до позиции переданной в `offset` плюс база, определенная строкой `whence`:

- **"set":** база на позиции 0 (начало файла);

- **"cur":** база на текущей позиции;
- **"end":** база в конце файла;

В случае успеха, `seek` возвращает окончательную позицию в файле, измеряемую в байтах от начала файла. При ошибке, возвращает **nil** и сообщение об ошибке.

По умолчанию `whence` = `"cur"`, `offset` = 0. Так вызов `file:seek()` возвращает текущую позицию, не изменяя её; вызов `file:seek("set")` устанавливает позицию в начало файла (и возвращает 0); вызов `file:seek("end")` устанавливает позицию в конец файла и возвращает размер файла.

file:setvbuf (mode [, size])

Устанавливает режим буферизации для выходного файла. Доступны три режима:

- **"no":** без буферизации; результат любой операции вывода проявляется непосредственно.
- **"full":** полная буферизация; операция вывода выполняется только, когда буфер полон или когда вы явно сбрасываете (flush) файл (см. `io.flush`).
- **"line":** строчная буферизация; выход буферизуется до новой строки в выводе или до любого ввода из специальных файлов (таких как терминал).

Для двух последних случаев `size` определяет размер буфера, в байтах. По умолчанию это подходящий размер.

file:write (···)

Записывает значение каждого аргумента в `file`. Аргументы должны быть строками или числами.

В случае успеха функция возвращает `file`. Иначе, возвращает **nil** и сообщение об ошибке.

6.9 – Средства операционной системы

Эта библиотека реализована через таблицу `os`.

os.clock ()

Возвращает апроксимацию количества секунд времени процессора, использованного программой.

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

Возвращает строку или таблицу, содержащую дату и время, отформатированную в соответствии с переданной строкой `format`.

Если аргумент `time` передан, то он является временем для форматирования (см. функцию `os.time` для описания этого значения). Иначе, `date` форматирует текущее время.

Если `format` начинается с символа '!', то дата форматируется во всемирном координированном времени (UTC). После этого опционального символа, если `format` это строка "`*t`", то `date` возвращает таблицу со следующими полями: `year` - год (четыре цифры), `month` - месяц (1–12), `day` - день (1–31), `hour` - час (0–23), `min` - минута (0–59), `sec` - секунда (0–61), `wday` - день недели (воскресенье = 1), `yday` - день в году и `isdst` - летнее время (boolean). Это последнее поле может отсутствовать, если информация недоступна.

Если `format` не "`*t`", то `date` возвращает дату как строку, форматированную согласно правилам C функции `strftime`.

Когда вызвана без аргументов, `date` возвращает разумное представление даты и времени, зависящее от хостовой системы и текущей локали (т.е. `os.date()` эквивалентно `os.date("%c")`).

На не POSIX системах эта функция может не быть потокобезопасной, т.к. она использует C функции `gmtime` и `localtime`.

os.difftime (t2, t1)

Возвращает разницу, в секундах, от времени `t1` до времени `t2` (где значения времени возвращены `os.time`). В POSIX, Windows и некоторых других системах это значение точно `t2-t1`.

os.execute ([command])

Эта функция эквивалентна ISO C функции `system`. Она передает `command` для запуска оболочкой операционной системы. Её первый результат равен **true**, если команда завершена успешно, или **nil** иначе. После этого первого результата функция возвращает строку и число:

- **`"exit"`:** команда завершена нормально; следующее число это выходной статус команды.
- **`"signal"`:** команда была завершена сигналом; следующее число это сигнал, завершивший команду.

Когда вызвана без `command`, `os.execute` возвращет логическое значение, которое равно true, если оболочка (shell) доступна.

os.exit ([code [, close]])

Вызывает ISO C функцию `exit` для завершения хостовой программы. Если `code` = **true**, возвращается статус `EXIT_SUCCESS`; если `code` = **false**, возвращается статус `EXIT_FAILURE`; если `code` это число, возвращается статус равный этому числу. По умолчанию, `code` = **true**.

Если опциональный второй аргумент `close` = true, закрывает Lua состояние перед выходом.

os.getenv (varname)

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

os.remove (filename)

Удаляет файл (или пустую директорию, на POSIX системах) с переданным именем. Если функция терпит неудачу, она возвращает **nil**, сообщение об ошибке и код ошибки.

os.rename (oldname, newname)

Переименовывает файл или директорию `oldname` в `newname`. Если функция терпит неудачу, она возвращает **nil**, сообщение об ошибке и код ошибки.

os.setlocale (locale [, category])

Устанавливает текущую локаль для программы. `locale` - системозависимая строка, определяющая локаль; `category` - опциональная строка, описывающая какую категорию изменять: `"all"`, `"collate"`, `"ctype"`, `"monetary"`, `"numeric"` или `"time"`; по умолчанию category = `"all"`. Функция возвращает имя новой локали, или **nil**, если запрос не может быть выполнен.

Если `locale` это пустая строка, текущая локаль устанавливается в засимую от реализации родную локаль. Если `locale` это строка `"C"`, текущая локаль устанавливается в стандартную C локаль.

Когда вызвана с **nil** в качестве первого аргумента, эта функция только возвращает имя текущей локали для данной категории.

Эта функция может не быть потокобезопасной, т.к. использует C функцию `setlocale`.

os.time ([table])

Возвращает текущее время, когда вызвана без аргументов, или время, представляющее локальную дату и время определенные в переданной таблице. Эта таблица должна иметь поля `year`, `month` и `day`, и может иметь поля `hour` (по умолчанию, 12), `min` (по умолчанию, 0), `sec` (по умолчанию, 0) и `isdst` (по умолчанию, **nil**). Остальные поля игнорируются. Для описания этих полей, см. функцию `os.date`.

Значения этих полей могут не быть в своих правильных диапазонах. Например, если `sec` = -10, то это означает -10 секунд от времени, определенного другими полями; если `hour` = 1000, это означает +1000 часов от времени, определенного другими полями.

Возвращенное значение это число, значение которого зависит от вашей системы. В POSIX, Windows и некоторых других системах это количество секунд, прошедших с какого-то определенного времени ("эпоха"). В других системах, значение не определено, и число, возвращенное функцией `time`, может использоваться только, как аргумент для `os.date` и `os.difftime`.

os.tmpname ()

Возвращает строку с именем файла, который может быть использован, как временный. Файл должен быть явно открыт перед использованием и явно удален, когда больше не нужен.

На POSIX системах эта функция также создает файл с этим именем, для избежания рисков беззопасности. (Кто-нибудь другой может создать файл с неправильными разрешениями в промежуток времени между получением имени файла и его созданием.) Вы по прежнему должны открыть файл для его использования и удалить его (даже если не использовали).

Когда возможно, предпочтительно использовать функцию `io.tmpfile`, которая автоматически удаляет файл при завершении программы.

6.10 – Библиотека отладки

Эта библиотека предоставляет Lua программам функциональность отладочного интерфейса (§4.9). Используя эту библиотеку, вы должны проявлять внимательность. Различные функции этой билиотеки нарушают базовые предположения о Lua коде (например, что локальные переменные функции не могут быть доступны снаружи; что метатаблицы пользовательских данных не могут изменяться Lua кодом; что Lua программы не падают) и следовательно могут скомпрометировать защищенный код. Кроме того, некоторые функции в этой библиотеке могут быть медленными.

Все функции в этой библиотеке предоставляются в таблице `debug`. Все функции, которые оперируют с потоком, имеют опциональный первый аргумент, определяющий поток. По умолчанию, это всегда текущий поток.

debug.debug ()

Входит в интерактивный режим с пользователем, запуская каждую строку, вводимую пользователем. Используя простые команды и другие отладочные возможности, пользователь может проверять глобальные и локальные переменные, изменять их значения, вычислять выражения и т.д. Строка, содержащая только слово `cont`, завершает эту функцию, так что вызывающий продожает своё исполнение.

Учтите, что команды для `debug.debug` не являются лексически вложенными ни в одну функцию и не имеют прямого доступа к локальным переменным.

debug.gethook ([thread])

Возвращает настройки текущего перехватчика потока, как три значения: текущая функция-перехватчик, текущая маска перехвата и текущий счетчик перехвата (как установлено функцией `debug.sethook`).

debug.getinfo ([thread,] f [, what])

Возвращает таблицу с информацией о функции. Вы можете передать функцию напрямую или можете передать число, как значение `f`, которое означает функцию, выполняющуюся на уровне `f` стека вызовов данного потока thread: уровень 0 - текущая функция (непосредственно `getinfo`); уровень 1 - функция, которая вызвала `getinfo` (за исключением хвостовых вызовов, которые не считаются на стеке); и так далее. Если число `f` больше количества активных функций, то `getinfo` возвращает **nil**.

Возвращенная таблица может содержать все поля возвращаемые `lua_getinfo`, со строкой `what`, описывающей какие поля заполнены. По умолчанию, `what` установлена для получения всей доступной информации, кроме таблицы значимых строк. Если передана, опция '`f`' добавляет поле `func` с функцией непосредственно. Если передана, опция '`L`' добавляет поле `activelines` с таблицей значимых строк.

Например, выражение `debug.getinfo(1,"n").name` возвращает имя текущей функции, если разумное имя существует, и выражение `debug.getinfo(print)` возвращает таблицу со всей доступной информацией о функции `print`.

debug.getlocal ([thread,] f, local)

Эта функция возвращает имя и значение локальной переменной с индексом `local` функции на уровне `f` стека вызовов. Эта функция получает доступ не только к явным локальным переменным, но

также к параметрам, временным переменныи и др.

Первый параметр или локальная переменная имеет индекс 1, и так далее, следуя порядку определения в коде, считаются только активные переменные в текущей области функции. Отрицательные индексы ссылаются на переменные (vararg) параметры; -1 первый переменный параметр. Функция возвращает **nil**, если нет переменной по данному индексу, и вызывает ошибку, когда вызвана с уровнем за пределами диапазона. (Вы можете вызвать `debug.getinfo` чтобы проверить, какой уровень допустим.)

Имена переменных, начинающиеся с '(' (открывающая скобка) представляют переменные с неизвестными именами (внутренние переменные, такие как переменные управления циклом, и переменные из кусков, сохраненных без отладочной информации).

Параметр `f` также может быть функцией. В этом случае `getlocal` возвращает только имена параметров функции.

debug.getmetatable (value)

Возвращает метатаблицу переданного значения `value` или **nil**, если у значения нет метатаблицы.

debug.getregistry ()

Возвращает таблицу реестра (см. §4.5).

debug.getupvalue (f, up)

Эта функция возвращает имя и значение upvalue с индексом `up` функции `f`. Функция возвращает **nil**, если по данному индексу нет upvalue.

Имена переменных, начинающиеся с символа '(' (открывающая скобка) , представляют переменные с неизвестными именами (переменные из кусков, сохраненных без отладочной информации).

debug.getuservalue (u)

Возвращает Lua значение, ассоциированное с `u`. Если `u` не пользовательские данные, возвращает **nil**.

debug.sethook ([thread,] hook, mask [, count])

Устанавливает переданную функцию, как перехватчик. Строка `mask` и число `count` описывают, когда будет вызыван перехватчик. Строка mask может содержать комбинацию следующих символов:

- **'c':** перехватчик вызывается каждый раз, когда Lua вызывает функцию;
- **'r':** перехватчик вызывается каждый раз, когда Lua возвращается из функции;
- **'l':** перехватчик вызывается каждый раз, когда Lua входит на новую линию кода.

Кроме того, с `count` не равным нулю, перехватчик вызывается также через каждые `count` инструкций (count в значении "количество").

Когда вызвана без аргументов, `debug.sethook` выключает перехват.

Когда вызван перехватчик, его первый параметр - это строка, описывающая событие, которое стало причиной вызова перехватчика: `"call"` (или `"tail call"`), `"return"`, `"line"` и `"count"`. Для событий строк, перехватчик также получает номер строки во втором параметре. Внутри перехватчика вы можете вызвать `getinfo` с уровнем 2 для получения информации о запущенной функции (уровень 0 - это функция `getinfo`, уровень 1 - это функция-перехватчик).

debug.setlocal ([thread,] level, local, value)

Эта функция присваивает значение `value` локальной переменной по индексу `local` функции на уровне `level` в стеке вызовов. Функция возвращает **nil**, если локальная переменная с данным индексом не существует, и генерирует ошибку, когда вызвана с `level` вне диапазона. (Вы можете использовать `getinfo` чтобы проверить какой уровень допустим.) Иначе, возвращает имя локальной переменной.

См. `debug.getlocal` для дополнительной информации о именах и индексах переменных.

debug.setmetatable (value, table)

Устанавливает `table` метатаблицей для `value` (`table` может быть равно **nil**). Возвращает `value`.

debug.setupvalue (f, up, value)

Эта функция присваивает значение `value` в upvalue с индексом `up` функции `f`. Функция возвращает **nil**, если по данному индексу нет upvalue. Иначе, возвращает имя upvalue.

debug.setuservalue (udata, value)

Устанавливает переданное значение `value`, как Lua значение, ассоциированное с `udata`. `udata` должно быть полными пользовательскими данными.

Возвращает `udata`.

debug.traceback ([thread,] [message [, level]])

Если `message` передано, но не строка и не **nil**, эта функция возвращет `message` без дальнейшей обработки. Иначе, возвращает строку с трассировкой стека вызовов. Опциональная строка `message` добавляется в начале трассировки стека. Опциональное число `level` говорит, на каком уровне начинать трассировку (по умолчанию равен 1 - функция, вызвавшая `traceback`).

debug.upvalueid (f, n)

Возвращает уникальный идентификатор (как лёгкие пользовательские данные) для upvalue под номером `n` из переданной функции.

Эти уникальные идентификаторы позволяют программе проверить, когда разные замыкания совместно используют одни upvalue. Lua замыкания, которые совместно используют upvalue (т.е. имеют доступ к одной внешней локальной переменной) вернут одинаковые идентификаторы для этих upvalue.

debug.upvaluejoin (f1, n1, f2, n2)

Заставляет `n1`-е upvalue Lua замыкания `f1` ссылаться на `n2`-е upvalue Lua замыкания `f2`.

7 – Интерпретатор Lua

Хотя Lua был разработан, как язык расширений, чтобы встраивать в хостовую C программу, он также часто используется, как автономный язык. Интерпретатор Lua, как автономного языка, называется просто `lua` и предоставляется в стандартном дистрибутиве. Автономный интерпретатор включает все стандартные библиотеки, в том числе отладочную библиотеку. Синтаксис командной строки:

```
lua [options] [script [args]]
```

Опции:

- **`-e` *`stat`*:** запускает строку *stat*;
- **`-l` *`mod`*:** загружает (require) *mod*;
- **`-i`:** входит в интерактивный режим после запуска *script*;
- **`-v`:** печатает информацию о версии;
- **`-E`:** игнорирует переменные окружения;
- **`--`:** останавливает обработку опций;
- **`-`:** запускает `stdin` как файл и останавливает обработку опций.

После обработки опций, `lua` запускает переданный *скрипт* (script). Когда вызван без аргументов, `lua` ведет себя, как `lua -v -i`, когда стандартный ввод (`stdin`) это терминал; и как `lua -` иначе.

Когда вызван без опции `-E`, интерпретатор проверяет переменную окружения `LUA_INIT_5_3` (или `LUA_INIT`, если предыдущая не определена) перед запуском аргументов. Если содержимое переменной имеет формат *`@filename`*, то `lua` запускает это файл. Иначе, `lua` запускает непосредственно эту строку.

Когда запущен с опцией `-E`, вместе с игнорированием `LUA_INIT`, Lua также игнорирует значения `LUA_PATH` и `LUA_CPATH`, устанавливая значения `package.path` и `package.cpath` путями по умолчанию, определенными в `luaconf.h`.

Все опции обрабатываются по порядку, кроме `-i` и `-E`. Например, вызов

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

сначала установит `a = 1`, затем напечатает значение `a`, и наконец запустит файл `script.lua` без

аргументов. (Здесь $ - это приглашение командной строки. Ваше приглашение может отличаться.)

Перед запуском любого кода `lua` собирает все аргументы командной строки в глобальной таблице `arg`. Имя скрипта идет под индексом 0, первый аргумент после имени скрипта идет по индексу 1, и т.д. Все аргументы перед именем скрипта (т.е имя интерпретатора и его опции) находятся по отрицательным индексам. Например, вызов

```
$ lua -la b.lua t1 t2
```

даст таблицу:

```
arg = { [-2] = "lua", [-1] = "-la",
        [0] = "b.lua",
        [1] = "t1", [2] = "t2" }
```

Если в вызове нет скрипта, имя интерпретатора идет по индексу 0, далее идут другие аргументы. Например, вызов

```
$ lua -e "print(arg[1])"
```

напечатает "`-e`". Если есть скрипт, то он вызывается с параметрами `arg[1]`, ⋯, `arg[#arg]`. (Как все куски в Lua, скрипт компилируется, как функция с переменным числом аргументов.)

В интерактивном режиме, Lua многократно выдает приглащение и ждет ввода строки. После ввода строки, Lua сначала пробует интерпретировать строку как выражение. В случае успеха, печатает её значение. Иначе, интерпретирует строку как оператор. Если вы напишете незавершенное выражение, интерпретатор будет ждать его завершения, выдывая приглашение.

В случае незащищенных ошибок в скрипте, интерпретатор пишет ошибку в стандартный поток

ошибок. Если объект ошибки это не строка, но имеет метаметод `__tostring`, интерпретатор вызывает этот метаметод для выдачи финального сообщения. Иначе, интерпретатор конвертирует объект ошибки в строку и добавляет к нему трассировку стека.

При нормальном завершении интерпретатор закрывает своё главное Lua состояние (см. `lua_close`). Скрипт может избежать этого шага, вызвав `os.exit` для завершения.

Чтобы использовать Lua, как интерпретатор скриптов в Unix системах, автономный интерпретатор пропускает первую линию куска, если он начинается с символа #. Следовательно, Lua скрипты могут быть сделаны исполняемыми программами, используя `chmod +x` и `#!` форму, например

```
#!/usr/local/bin/lua
```

(Конечно, расположение Lua интерпретатора может быть другим. Если `lua` в вашей переменной `PATH`, то

```
#!/usr/bin/env lua
```

более портабельное решение.)

8 – Несовместимости с предыдущей версией

Здесь приведен список несовместимостей, которые вы можете встретить при портировании программы с Lua 5.2 в Lua 5.3. Вы можете избежать некоторые несовместимости, скомпилировав Lua с соответствующими опциями (см. файл `luaconf.h`). Тем не менее, все эти опции совметимости

будут убраны в будущем.

Версии Lua всегда могут изменить C API способами, которые не подразумевают изменение исходного кода программы, такие, как цифровые значения констант или реализация функций через макросы. Следовательно, вы не должны ожидать бинарной совметимости между разными версиями Lua. Всегда перекомпилируйте клиентов Lua API, когда используете новую версию.

Аналогично, версии Lua всегда могут изменить внутреннее представление скомпилированных кусков; скомпилированные куски не совместимы между разными версиями Lua.

Стандартные пути в официальном дистрибутиве могут меняться между версиями.

8.1 – Изменения в языке

- Главное различие между Lua 5.2 и Lua 5.3 это введение подтипа целых для чисел. Хотя это изменение не влияет на "нормальные" вычисления, некоторые вычисления (главным образом это некоторые типы переполнений) могут давать разные результаты.

  Вы можете убрать эти отличия, принудительно сделав числа вещественными (в Lua 5.2 все числа вещественные), в частности записывая константы с завершающим `.0` или используя `x = x + 0.0` для преобразования переменных. (Эта рекомендация только для быстрого исправления редких несовместимостей; это не общая рекомендация для хорошего программирования. Для хорошего программирования, используйте вещественные числа там, где нужны вещественные, и целые там, где нужны целые.)

- Преобразование вещественных чисел в строку теперь добавляет суффикс `.0` в результат, если он выглядит как целое. (Например, вещественное 2.0 будет напечатано как `2.0`, не как `2`.) Вы всегда должны использовать явный фомат, когда нуждаетесь в специфическом формате для

чисел.

(Формально, это не несовместимость, т.к. Lua не определяет, как числа форматируются в строки, но некоторые программы предполагают что это специфический формат.)

- Генерационный режим работы сборщика мусора был удален. (Это была экспериментальная возможность в Lua 5.2.)

8.2 – Изменения в библиотеках

- Библиотека `bit32` стала нежелательной. Легко загрузить совместимую внешнюю библиотеку или лучше заменить её функции соответствующими битовыми операторами. (Помните, что `bit32` оперирует с 32-битными целыми, а битовые операторы Lua 5.3 оперируют с целыми Lua, которые по умолчанию имеют 64 бита.)
- Табличная библиотека теперь уважает метаметоды для установки и получения элементов.
- Итератор `ipairs` теперь уважает метаметоды и его метаметод `__ipairs` стал нежелательным.
- Имена опций в `io.read` больше не имеют начального символа '*'. Для совместимости Lua продолжит принимать (и игнорировать) этот символ.
- Следующие функции стали нежелательными в математической библиотеке: `atan2`, `cosh`, `sinh`, `tanh`, `pow`, `frexp` и `ldexp`. Вы можете заменить `math.pow(x,y)` на `x^y`; вы можете заменить `math.atan2` на `math.atan`, который теперь принимает один или два параметра; вы можете заменить `math.ldexp(x,exp)` на `x * 2.0^exp`. Для других операций вы можете также использовать внешнюю библиотеку или реализовать их в Lua.
- Искатель для C загрузчиков, используемый `require`, изменил способ обработки имен с версиями. Сейчас версия должна идти после имени модуля (как обычно в большинстве других инструментов). Для совместимости, этот искатель все еще пытается использовать старый формат, если не может найти функцию открытия соответствующую новому стилю. (Lua 5.2 уже

работает таким способом, но это не документировано.)
- Вызов `collectgarbage("count")` сейчас возвращает только один результат. (Вы можете вычислить второй результат из дробной части первого результата.)

8.3 – Изменения в API

- Функции продолжения сейчас принимают как параметры то, что им нужно было получать через `lua_getctx`, так функция `lua_getctx` была удалена. Откорректируйте ваш код соответственно.
- Функция `lua_dump` имеет дополнительный параметр `strip`. Используйте 0 для этого параметра, чтобы получить старое поведение.
- Функции для вставки/получения беззнаковых целых (`lua_pushunsigned`, `lua_tounsigned`, `lua_tounsignedx`, `luaL_checkunsigned`, `luaL_optunsigned`) нежелательны. Используйте их знаковые эквиваленты с преобразованием типа.
- Макросы для получения нестандартных целых типов (`luaL_checkint`, `luaL_optint`, `luaL_checklong`, `luaL_optlong`) нежелательны. Используйте их эквиваленты с `lua_Integer` с преобразованием типов (или, когда возможно, используйте `lua_Integer` в вашем коде).

9 – Полный синтаксис Lua

Здесь приведен полный синтаксис Lua в БНФ. Как обычно в расширенной БНФ, {A} означает 0 или более A, и [A] означает опциональное A. (Для приоритета операторов, см. §3.4.8; для описания терминалов Name, Numeral и LiteralString, см. §3.1.)

```
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 | ‘#’ | ‘~’
```

last update: wed jun 10 18:31:15 brt 2015 last change: revised for lua 5.3.1