Lua.org

Программирование на языке Lua

РОБЕРТУ ИЕРУЗАЛИМСКИ

третье издание

Программирование на языке Lua

Третье издание

Роберту Иерузалимски

Programming in Lua

Third Edition

Roberto Ierusalimschy

Посвящается Иде, Ноэми и Анне Лучии.

перевод и оформление: N1cke

Оглавление

Введение
Аудитория
О третьем издании
Другие ресурсы
Некоторые типографские соглашения
Запуск примеров
Благодарности
I. Язык
1. Начало работы
1.1. Куски
1.2. Некоторые лексические соглашения
1.3. Глобальные переменные
1.4. Автономный интерпретатор
Упражнения
2. Типы и значения
2.1. Отсутствие значения (nil)
2.2. Логические значения (boolean)
2.3. Числа (number)
2.4. Строки (string)
Строковые литералы
Дпинные строки
Приведения типов
2.5. Таблицы (table)
2.6. Функции (function)
2.7. Пользовательские данные (userdata)
2.8. Нити (thread)
Упражнения
3. Выражения
3.1. Арифметические операции

3.2. Операции сравнения
3.3. Логические операции
3.4. Конкатенация
3.5. Операция длины
3.6. Приоритеты операций
Конструкторы таблиц
Упражнения
4. Операторы
4.1. Операторы присваивания
4.2. Локальные переменные и блоки
4.3. Управляющие конструкции
if then else
while
repeat
Числовой for
Общий for
4.4. break, return и goto
Упражнения
5. Функции
5.1. Множественные результаты
5.2. Вариадические функции
5.3. Именованные аргументы
Упражнения
6. Еще раз о функциях
6.1. Замыкания
6.2. Неглобальные функции
6.3. Корректные хвостовые вызовы
Упражнения
7. Итераторы и общий for
7.1. Итераторы и замыкания
7.2. Семантика общего for
7.3. Итераторы без состояния
7.4. Итераторы со сложным состоянием

7.5. Подлинные итераторы
Упражнения
8. Компиляция, выполнение и ошибки
8.1. Компиляция
8.2. Предкомпилированный код
8.3. Код C
8.4. Ошибки
8.5. Обработка ошибок и исключений
8.6. Сообщения об ошибках и обратные трассировки
Упражнения
9. Сопрограммы
9.1. Основы сопрограмм
9.2. Каналы и фильтры
9.3. Сопрограммы как итераторы
9.4. Невытесняющая многонитевость
Упражнения
10. Законченные примеры
10.1. Задача о восьми королевах
10.2. Самые часто встречающиеся слова
10.3. Алгоритм цепи Маркова
Упражнения
II. Таблицы и объекты
11. Структуры данных
11.1. Массивы
11.2. Матрицы и многомерные массивы
11.3. Связанные списки
11.4. Очереди и двойные очереди
11.5. Множества и мультимножества
11.6. Строковые буферы
11.7. Графы
Упражнения
12. Файлы с данными и сохраняемость
12.1. Файлы с данными

12.2. Сериализация
Сохранение таблиц без циклов
Сохранение таблиц с циклами
Упражнения
13. Метатаблицы и метаметоды
13.1. Арифметические метаметоды
13.2. Метаметоды сравнения
13.3. Библиотечные метаметоды
13.4. Метаметоды доступа к таблице
Метаметод __index
Метаметод __newindex
Таблицы со значениями по умолчанию
Отслеживание доступа к таблице
Таблицы, доступные только для чтения
Упражнения
14. Окружение
14.1. Глобальные переменные с динамическими именами
14.2. Объявления глобальных переменных
14.3. Неглобальные окружения
14.4. Использование _ENV
14.5. _ENV и load
Упражнения
15. Модули и пакеты
15.1. Функция require
Переименование модуля
Поиск пути
Искатели
15.2. Основной подход к написанию модулей на Lua
15.3. Использование окружений
15.4. Подмодули и пакеты
Упражнения
16. Объектно-ориентированное программирование
16.1. Классы

16.2. Наследование
16.3. Множественное наследование
16.4. Конфиденциальность
16.5. Подход с единственным методом
Упражнения
17. Слабые таблицы и финализаторы
17.1. Слабые таблицы
17.2. Функции с запоминанием
17.3. Атрибуты объекта
17.4. Вновь таблицы со значениями по умолчанию
17.5. Эфемерные таблицы
17.6. Финализаторы
Упражнения
III. Стандартные библиотеки
18. Математическая библиотека
Упражнения
19. Побитовая библиотека
Упражнения
20. Табличная библиотека
20.1. Функции insert и remove
20.2. Сортировка
20.3. Конкатенация
Упражнения
21. Строковая библиотека
21.1. Основные строковые функции
21.2. Функции сопоставления с образцом
Функция string.find
Функция string.match
Функция string.gsub
Функция string.gmatch
21.3. Образцы
21.4. Захваты
21.5. Замены

Кодировка URL
Разложение символов табуляции на пробелы
21.6. Специфические приемы
21.7. Юникод
Упражнения
22. Библиотека ввода-вывода
22.1. Простая модель ввода-вывода
22.2. Полная модель ввода-вывода
Небольшой прием для увеличения быстродействия
Бинарные файлы
22.3. Прочие операции над файлами
Упражнения
23. Библиотека операционной системы
23.1. Дата и время
23.2. Прочие системные вызовы
Упражнения
24. Отладочная библиотека
24.1. Интроспективные средства
Доступ к локальным переменным
Доступ к нелокальным переменным
Доступ к другим сопрограммам
24.2. Ловушки
24.3. Профилирование
Упражнения
IV. C API
25. Обзор C API
25.1. Первый пример
25.2. Стек
Заталкивание элементов
Обращение к элементам
Другие стековые операции
25.3. Обработка ошибок в C API
Обработка ошибок в прикладном коде

Обработка ошибок в библиотечном коде
Упражнения
26. Расширение вашего приложения
26.1. Основы
26.2. Работа с таблицами
26.3. Вызовы функций Lua
26.4. Обобщенный вызов функции
Упражнения
27. Вызываем C из Lua
27.1. Функции C
27.2. Продолжения
27.3. Модули C
Упражнения
28. Приемы написания функций C
28.1. Обработка массивов
28.2. Обработка строк
28.3. Хранение состояния в функциях C
Реестр
Верхние значения
Общие верхние значения
Упражнения
29. Задаваемые пользователем типы в C
29.1. Пользовательские данные (userdata)
29.2. Метатаблицы
29.3. Объектно-ориентированный доступ
29.4. Доступ как к массиву
29.5. Облегченные пользовательские данные
Упражнения
30. Управление ресурсами
30.1. Итератор по директории
30.2. Парсер XML
Упражнения
31. Нити и состояния

31.1. Многонитевость
31.2. Состояния Lua
Упражнения
32. Управление памятью
32.1. Выделяющая функция
32.2. Сборщик мусора
API сборщика мусора
Упражнения
Приложения. Lua 5.3
А. Переход на Lua 5.3
А.1. Изменения в языке
А.2. Изменения в библиотеках
А.3. Изменения в API
Б. Новое в Lua 5.3
Б.1. Язык
Целые числа
Официальная поддержка 32-битных чисел
Побитовые операции
Базовая поддержка UTF-8
Целочисленное деление
Б.2. Библиотеки
Опция strip в string.dump
Функция table.move
Функции string.pack, string.unpack, string.packsize
Б.3. C API
Опция strip в lua.dump
Функции
Б.4. Автономный интерпретатор Lua

Введение

Когда Вальдемар, Луис и я начали разработку Lua в 1993 году, мы с трудом могли себе представить, что Lua сможет так распространиться. Начавшись как домашний язык для двух специфичных проектов, сейчас Lua широко используется во всех областях, которые могут получить выигрыш от простого, расширяемого, переносимого и эффективного скриптового языка, таких как встроенные системы, мобильные устройства и, конечно, игры.

С самого начала мы разрабатывали Lua для интеграции с программным обеспечением, написанным на C/C++ и других общепринятых языках. Эта интеграция дает много преимуществ. Lua — это крошечный и простой язык, частично из-за того, что он не пытается превзойти C в том, в чем он уже хорош, например: высочайшее быстродействие, низкоуровневые операции и взаимодействие со сторонним программным обеспечением. Для этих задач Lua полагается на C. Lua предлагает как раз то, для чего C недостаточно хорош: высокая степень независимости от аппаратного обеспечения, динамические структуры, отсутствие избыточности и легкость тестирования и отладки. Для этих целей Lua располагает безопасным окружением, автоматическим управлением памятью и хорошими средствами для работы со строками и другими видами данных с динамически изменяемым размером.

Часть силы Lua идет от его библиотек, и это не случайно. В конце концов, одной из самых сильных сторон Lua является его расширяемость. Многие качества языка вносят в это свой вклад. Динамическая типизация обеспечивает высокую степень полиморфизма. Автоматическое управление памятью упрощает создание интерфейсов, поскольку нет необходимости решать, кто отвечает за выделение и освобождение памяти, или как обрабатывать ее переполнение. Функции высшего порядка и

анонимные функции обеспечивают высокую степень параметризации, делая функции более универсальными.

Lua является не только расширяемым, но и *связующим языком*. Lua поддерживает подход для разработки программного обеспечения на основе компонентов, когда мы создаем приложение, связывая вместе существующие высокоуровневые компоненты. Эти компоненты пишутся на компилируемом языке со статической типизацией, таком как C или C++; Lua является «клеем», который мы используем для компоновки и соединения этих компонентов. Обычно, компоненты (или объекты) представляют более конкретные низкоуровневые сущности (такие как виджеты и структуры данных), которые почти не меняются во время разработки программы, и которые расходуют большую часть процессорного времени итоговой программы. Lua придает окончательную форму приложению, которая, вероятно, будет неоднократно изменяться во время жизненного цикла данной программы. Однако, в отличие от других связующих технологий, Lua при этом является полноценным языком программирования. Поэтому мы можем использовать Lua не только для связывания компонентов, но и для их адаптации и настройки, а также для создания полностью новых компонентов.

Разумеется, Lua — не единственный скриптовый язык. Существуют другие языки, которые вы можете использовать примерно для тех же целей. Тем не менее, Lua предоставляет набор возможностей, которые делают его лучшим выбором для многих ваших задач, и которые придают ему свой уникальный профиль:

- *Расширяемость*. Расширяемость Lua настолько значительна, что многие рассматривают Lua не как язык, а как набор для построения предметно-ориентированных языков (domain-specific language — DSL). Мы с самого начала разрабатывали Lua так, чтобы он был расширяемым как через код Lua, так и через код C. В

качестве доказательства этой концепции Lua реализует большую часть своей базовой функциональности через внешние библиотеки. Обеспечить взаимодействие Lua с C/C++ действительно просто, и Lua был успешно интегрирован со многими другими языками, такими как Fortran, Java, Smalltalk, Ada, C#, и даже со скриптовыми языками, такими как Perl и Python.

- *Простота*. Lua — это простой и маленький язык. У него мало концепций (зато они эффективные). Эта простота облегчает изучение Lua и помогает сохранять его размер небольшим. Полный дистрибутив (исходный код, руководство, бинарные файлы для некоторых платформ) спокойно размещается на одном флоппи-диске.
- *Эффективность*. Lua обладает весьма эффективной реализацией. Независимые тесты показывают, что Lua является одним из самых быстрых среди скриптовых языков.
- *Переносимость*. Когда мы говорим о переносимости, то имеем в виду запуск Lua на всех платформах, о которых вы только слышали: все версии Unix и Windows, PlayStation, Xbox, Mac OS X и iOS, Android, Kindle Fire, NOOK, Haiku, QUALCOMM Brew, мейнфреймы IBM, RISC OS, Symbian OS, процессоры Rabbit, Raspberry Pi, Arduino и многие другие. Исходный код для каждой из этих платформ практически одинаков. Lua не использует условную компиляцию для адаптации своего кода под различные машины; вместо этого он придерживается стандартного ANSI (ISO) C. Таким образом, вам обычно не нужно адаптировать его под новую среду: если у вас есть компилятор ANSI C, то вам всего лишь нужно скомпилировать Lua без каких-либо предварительных настроек.

Аудитория

Пользователи Lua обычно относятся к одной из трех широких групп: те, кто используют Lua, уже встроенный в приложение, те, кто используют Lua автономно, и те, кто используют Lua и C вместе.

Многие используют Lua, встроенный в какую-либо прикладную программу, например, Adobe Lightroom, Nmap или World of Warcraft. Эти приложения используют Lua-C API для регистрации новых функций, создания новых типов и изменения поведения некоторых операций языка, конфигурируя Lua под свою специфическую область. Часто пользователи таких приложений даже не знают, что Lua является независимым языком, адаптированным под данную область. Например, многие разработчики плагинов для Lightroom не знают о других способах использования этого языка; пользователи Nmap, как правило, рассматривают Lua как язык Nmap Scripting Engine; игроки в World of Warcraft могут считать Lua языком исключительно для данной игры.

Lua также полезен как самостоятельный язык, не только для обработки текста и одноразовых маленьких программ, но также и для различных проектов от среднего до большого размера. При данном применении основную функциональность Lua дают его библиотеки. Стандартные библиотеки, например, предлагают базовое сопоставление с образцом и другие функции для работы со строками. Улучшение поддержки своих библиотек у Lua привело к быстрому увеличению количества внешних пакетов. Lua Rocks, система внедрения и управления модулями для Lua, на данный момент насчитывает более 150 пакетов.

Наконец, есть программисты, которые работают на другой стороне скамьи, и которые пишут приложения, использующие Lua как библиотеку C. Такие люди больше пишут на C, чем на Lua,

хотя им требуется хорошее понимание Lua для создания интерфейсов, которые будут простыми, легкими в использовании и хорошо интегрированными с языком.

Данной книге есть что предложить всем этим людям. Первая часть охватывает сам язык, показывая, как мы можем раскрыть весь его потенциал. Мы фокусируемся на различных конструкциях языка и используем многочисленные примеры и упражнения, чтобы показать, как их использовать для практических задач. Некоторые главы этой части охватывают базовые понятия, такие как управляющие структуры, в то время как остальные главы охватывают более продвинутые темы, такие как итераторы и сопрограммы.

Вторая часть полностью посвящена таблицам, единственной структуре данных в Lua. Главы этой части обсуждают структуры данных, их сохраняемость, пакеты и объектно-ориентированное программирование. Именно там мы раскроем настоящую мощь языка.

Третья часть представляет стандартные библиотеки. Эта часть особенно полезна для тех, кто использует Lua как самостоятельный язык, хотя многие другие приложения также частично или полностью включают в себя стандартные библиотеки. Данная часть отводит по одной главе для каждой стандартной библиотеки: математическая библиотека, побитовая библиотека, табличная библиотека, строковая библиотека, библиотека ввода-вывода, библиотека операционной системы и отладочная библиотека.

Наконец, последняя часть книги охватывает API между Lua и C для тех, кто использует C, чтобы овладеть полной мощью Lua. Подача материала данной части в силу необходимости весьма отличается от всей остальной книги. Здесь мы будем программировать на C, а не на Lua, так что нам придется сменить амплуа. Для некоторых читателей обсуждение C API может представлять незначительный интерес; для других эта часть может оказаться самой важной.

О третьем издании

Эта книга является обновленной и расширенной версией второго издания книги *«Programming in Lua»* (также известной как *PiL 2*). Хотя структура книги практически та же самая, это издание включает в себя изрядное количество нового материала.

Во-первых, я обновил всю книгу до Lua 5.2. Особую значимость представляет глава об окружениях, которая была практически полностью переписана. Я также переписал несколько примеров, чтобы показать преимущества от использования новых возможностей, предоставляемых Lua 5.2. Тем не менее, я четко обозначил отличия от Lua 5.1, поэтому вы можете использовать книгу и для этой версии языка.

Во-вторых, что более важно, я добавил упражнения во все главы книги. Эти упражнения варьируются от простых вопросов о языке до небольших полноценных проектов. Некоторые примеры иллюстрируют важные аспекты программирования на Lua и так же важны, как примеры, которые расширяют ваш набор полезных приемов.

Как и в случае с первым и вторым изданиями «Programming in Lua», мы опубликовали это третье издание самостоятельно. Несмотря на ограниченные возможности распространения, этот подход обладает рядом преимуществ: мы сохраняем полный контроль над содержимым книги; мы сохраняем все права на предложение книги в других формах; мы свободны выбирать, когда выпустить следующее издание; мы можем быть уверены, что выпуск данной книги не будет прекращен.

Другие ресурсы

Справочник (reference manual) по языку необходим всем, кто действительно хочет его освоить. Эта книга не заменяет справочник

Lua. Напротив, они дополняют друг друга. Справочник лишь описывает Lua. Он не показывает ни примеров, ни объяснений для конструкций языка. С другой стороны, он полностью описывает язык: эта книга опускает некоторые редко используемые «темные углы» Lua. Более того, справочник является официальным документом о Lua. Там, где эта книга расходится со справочником, доверяйте справочнику. Чтобы получить справочник и дополнительную информацию о Lua, посетите веб-сайт `http://www.lua.org`.

Вы также можете найти полезную информацию на сайте пользователей Lua, поддерживаемом их сообществом: `http://lua-users.org`. Среди прочих ресурсов он предлагает учебное пособие (tutorial), список сторонних пакетов и документации, и архив официальной рассылки по Lua.

Эта книга описывает Lua 5.2, хотя большая часть ее содержимого также применима к Lua 5.1 и Lua 5.0. Некоторые отличия Lua 5.2 от предыдущих версий Lua 5 четко обозначены в тексте книги. Если вы используете более новую версию (выпущенную после этой книги), обратитесь к соответственному руководству по поводу отличий между версиями. Если вы используете версию старше 5.2, то пришла пора подумать об обновлении.

Некоторые типографские соглашения

В данной книге "`строковые литералы`" заключены в двойные кавычки, а одиночные символы, например '`a`', заключены в одинарные кавычки. Строки, которые применяются как образцы, также заключены в одинарные кавычки, например '`[%w_]*`'. Книга использует моноширинный шрифт как для `кусков кода`, так и для `идентификаторов`. Для **зарезервированных слов** используется полужирный шрифт. Крупные куски кода показаны с применением

следующего стиля:

```
-- программа "Hello World"
print ("Hello World")    --> Hello World
```

Обозначение `-->` показывает результат выполнения оператора или, изредка, результат выражения:

```
print(10)     --> 10
13+3          --> 16
```

Поскольку двойной дефис (`--`) начинает комментарий в Lua, вы можете без проблем применять данные обозначения в ваших программах. Наконец, в книге используется обозначение `<-->` для указания на то, что что-то эквивалентно чему-то другому:

```
this    <-->    that
```

(Примечание: так выглядят примечания от автора книги.) (А так выглядят примечания от переводчика.)

Запуск примеров

Вам понадобится интерпретатор Lua для запуска примеров из этой книги. В идеале вам следует использовать Lua 5.2, но большинство примеров без каких-либо изменений будет работать и на Lua 5.1.

Сайт Lua (`http://www.lua.org`) хранит исходный код этого интерпретатора. Если у вас есть компилятор C и знание того, как скомпилировать код C на вашем компьютере, то вам стоит попробовать установить Lua из его исходного кода; это действительно легко. Сайт *Lua Binaries* (поищите `luabinaries`) предлагает заранее скомпилированные интерпретаторы для большинства основных платформ. Если вы используете Linux или другую UNIX-подобную систему, вы можете проверить репозиторий

вашего дистрибутива; некоторые дистрибутивы уже предлагают готовые пакеты с Lua. Для Windows хорошим выбором является *Lua for Windows* (поищите `luaforwindows`), являющийся «заряженной средой» для Lua. Он включает в себя интерпретатор, интегрированный текстовый редактор и множество библиотек.

Если вы используете Lua, встроенный в приложение, такое как WoW или Nmap, то вам может понадобиться руководство по данному приложению (или помощь «местного гуру»), чтобы разобраться, как запускать ваши программы. Тем не менее, Lua остается все тем же языком; большинство вещей, которое мы увидим в этой книге, применимо независимо то того, как вы используете Lua. Однако, я рекомендую начать изучение Lua с автономного интерпретатора для запуска ваших первых примеров и экспериментов.

Благодарности

Прошло уже почти десять лет с тех пор, как я опубликовал первое издание этой книги. Ряд друзей и организаций помогал мне на этом пути.

Как всегда, Луиг Хенрик де Фигуредо и Вальдемар Селес, соавторы Lua, предложили все виды помощи. Андре Карригал, Аско Кауппи, Бретт Капилик, Диего Мехаб, Эдвин Морагас, Фернандо Джефферсон, Гэвин Врес, Джон Д. Рамсделл и Норман Ремси предоставили неоценимые замечания и полезную аналитическую информацию для различных изданий этой книги.

Луиза Новаэс, глава отдела искусства и дизайна в PUC-Rio, смогла найти время в своем занятом графике, чтобы создать идеальную обложку для данного издания.

Lightning Source, Inc. предложило надежный и эффективный вариант для печати и распространения данной книги. Без них о самостоятельной публикации этой книги не могло бы быть и речи.

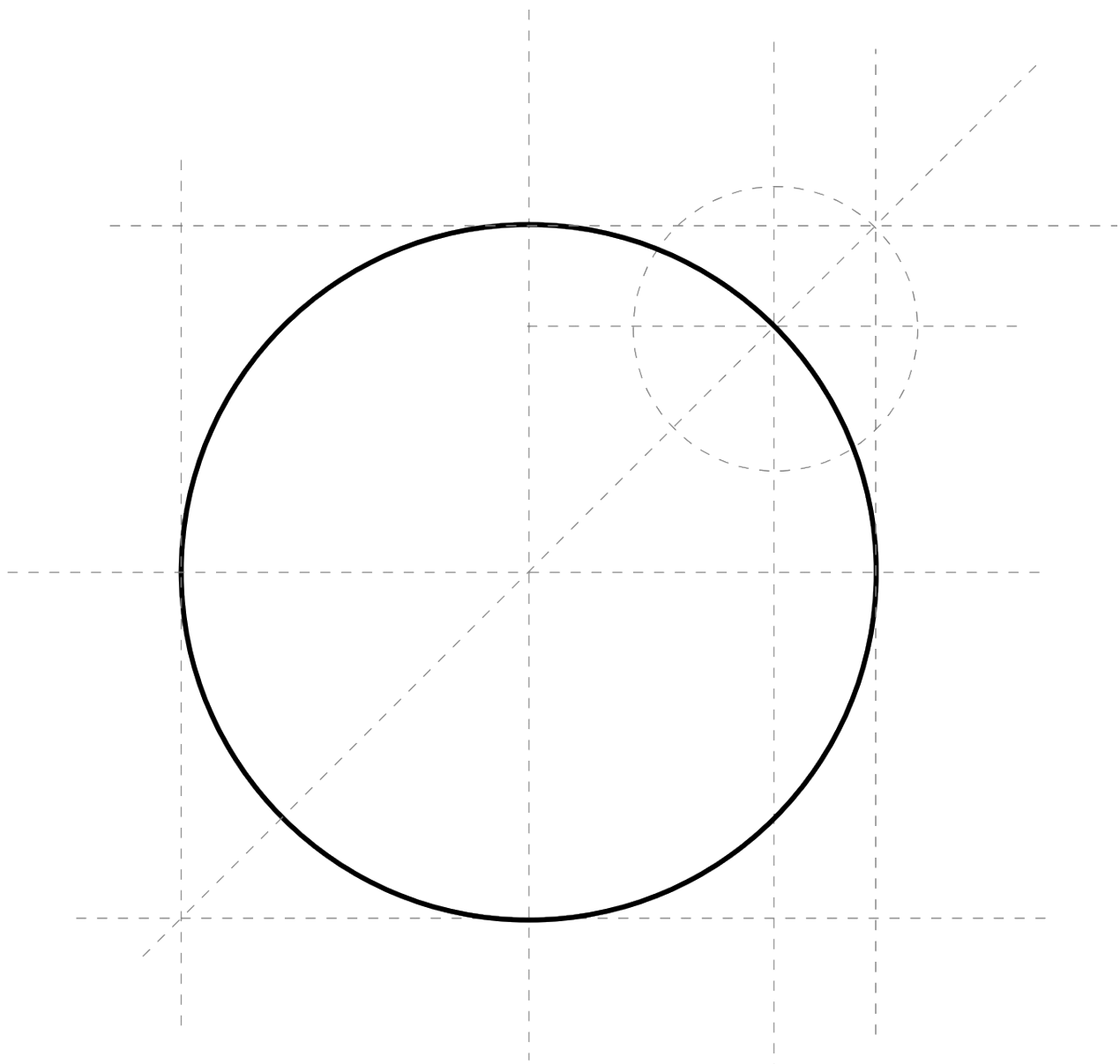
Центр латиноамериканских исследований в Стендфордском университете предоставил мне крайне необходимый перерыв от регулярной работы в очень стимулирующем окружении, во время которого я и сделал большую часть работы над третьим изданием.

Я также хотел бы поблагодарить Папский католический университет Рио де Жанейро (PUC-Rio) и Бразильский национальный исследовательский совет (CNPq) за их постоянную поддержку моей работы.

Наконец, я должен выразить мою глубокую благодарность Ноэми Родригес за все виды помощи (в том числе и технической) и скрашивание моей жизни.

Часть I

Язык



ГЛАВА 1

Начало работы

Продолжая сложившуюся традицию, наша первая программа на Lua всего лишь печатает "Hello World":

```
print("Hello World")
```

Если вы используете автономный интерпретатор Lua, то все, что вам требуется для запуска вашей первой программы — это вызвать интерпретатор (обычно он называется `lua` или `lua5.2`) с именем текстового файла, содержащего вашу программу. Если вы сохраните вышеприведенную программу в файл `hello.lua`, то следующая команда должна его запустить:

```
% lua hello.lua
```

В качестве более сложного примера следующая программа определяет функцию для вычисления факториала заданного числа, запрашивает у пользователя число и печатает его факториал:

```
-- определяет функцию факториала
function fact (n)
  if n == 0 then
    return 1
  else
    return n * fact(n-1)
  end
end

print("enter a number:")
a = io.read("*n")        -- считывает число
print(fact(a))
```

1.1. Куски

Каждый выполняемый Lua фрагмент кода, такой как файл или отдельная строка в интерактивном режиме, называется куском. *Кусок* (chunk) — это просто последовательность команд (или операторов).

Lua не нужен разделитель между идущими подряд операторами, но вы можете использовать точку с запятой, если хотите. Мое личное правило — использовать точки с запятой только для разделения двух или более операторов, записанных в одной строке. Переводы строк не играют никакой роли в синтаксисе Lua; например, следующие четыре куска допустимы и эквивалентны:

```
a = 1
b = a*2

a = 1;
b = a*2;

a = 1; b = a*2

a = 1 b = a*2     -- уродливо, но допустимо
```

Кусок может быть просто одиночным оператором, как в примере «Hello World», или состоять из набора операторов и определений функций (которые на самом деле являются присваиваниями, как мы увидим позже), как в примере с факториалом. Кусок может быть настолько большим, насколько вы захотите. Поскольку Lua также используется как язык для описания данных, куски в несколько мегабайт не являются редкостью. У интерпретатора Lua не возникает никаких проблем при работе с большими кусками.

Вместо записи вашей программы в файл вы можете запустить автономный интерпретатор в интерактивном режиме. Если вы запустите `lua` без аргументов, то увидите его приглашение ввода:

```
% lua
Lua 5.2 Copyright (C) 1994-2012 Lua.org, PUC-Rio
>
```

С этого момента каждая команда, которую вы наберете (как, например, `print "Hello World"`), выполняется сразу после ее ввода. Для выхода из интерактивного режима и интерпретатора просто наберите управляющий символ конца файла (`ctrl-D` в UNIX, `ctrl-Z` в Windows) или вызовите функцию `exit` из библиотеки операционной системы — для этого нужно набрать `os.exit()`.

В интерактивном режиме Lua обычно интерпретирует каждую строку, которую вы набираете, как законченный кусок. Однако, если он обнаруживает, что строка не образует законченный кусок, то он ждет продолжения ввода до тех пор, пока этот кусок не будет закончен. Таким образом, вы можете вводить многострочные определения наподобие функции `factorial` прямо в интерактивном режиме. Тем не менее, обычно более удобно помещать подобные определения в файл и затем вызывать Lua для его выполнения.

Вы можете использовать опцию `-i`, чтобы дать указание Lua начать интерактивный сеанс после выполнения заданного куска:

```
% lua -i prog
```

Строка с командой вроде этой выполнит кусок в файле `prog` и затем выведет приглашение ввода интерактивного режима. Это особенно удобно для отладки и тестирования вручную. В конце данной главы мы рассмотрим другие опции автономного интерпретатора.

Другой способ выполнять куски состоит в применении функции `dofile`, которая немедленно выполняет файл. Например, допустим, у вас есть файл `lib1.lua` со следующим кодом:

```
function norm (x, y)
  return (x^2 + y^2)^0.5
end

function twice (x)
  return 2*x
end
```

Тогда в интерактивном режиме вы можете набрать

```
> dofile("lib1.lua")    -- загружает вашу библиотеку
> n = norm(3.4, 1.0)
> print(twice(n))       --> 7.0880180586677
```

Функция `dofile` также удобна, когда вы тестируете фрагмент кода. Вы можете работать с двумя окнами: одно будет текстовым редактором с вашей программой (скажем, в файле `prog.lua`), а другое консолью, выполняющей Lua в интерактивном режиме. После сохранения изменений в вашей программе, вы выполняете `dofile("prog.lua")` в консоли Lua для загрузки нового кода; затем вы проверяете этот новый код, вызывая его функции и печатая результаты.

1.2. Некоторые лексические соглашения

Идентификаторы (или имена) в Lua могут быть любой последовательностью из букв, цифр и символов подчеркивания, не начинающейся с цифры, например:

```
i       j      i10      _ij
aSomewhatLongName    _INPUT
```

Вы должны избегать идентификаторов, начинающихся с символа подчеркивания, за которым следует одна или несколько заглавных букв (как, например, `_VERSION`); они зарезервированы в Lua для особых целей. Обычно я использую идентификатор `_` (одиночный символ подчеркивания) для пустых переменных.

В старых версиях Lua понятие того, что является буквой, зависело от локали. Однако, подобные буквы делают вашу программу непригодной для выполнения на системах, которые не поддерживают данную локаль. Поэтому Lua 5.2 разрешает использовать в идентификаторах только буквы из диапазонов `A-Z` и `a-z`.

Следующие слова зарезервированы; мы не можем использовать их в качестве идентификаторов:

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

Lua учитывает регистр букв: **and** — это зарезервированное слово, но `And` и `AND` — это два отличных от него и друг от друга идентификатора.

Комментарий начинается в любом месте с двойного дефиса (`--`) и длится до конца строки кода. Lua также поддерживает блочный комментарий, который начинается с `-[[` и длится до ближайших `]]`. (Примечание: Блочные комментарии могут быть более сложными, как мы увидим в разделе 2.4). Распространенным приемом для закомментирования фрагмента кода является заключение этого кода между `--[[` и `--]]`, как показано ниже:

```
--[[
  print(10)         -- ничего не происходит
(закомментировано)
--]]
```

Для восстановления работоспособности этого кода мы добавляем один дефис к первой строке:

```
---[[
  print(10)         --> 10
--]]
```

В первом примере `--[[` в первой строке начинает блочный комментарий, а двойной дефис в последней строке по-прежнему находится внутри этого комментария. Во втором примере последовательность `---[[` начинает обычный однострочный комментарий, поэтому первая и последняя строки становятся независимыми комментариями. В этом случае `print` находится вне комментариев.

1.3. Глобальные переменные

Глобальным переменным не нужны объявления; вы их просто используете. Обратиться к неинициализированной переменной не является ошибкой; вы всего лишь получите значение nil в качестве результата:

```
print(b)    --> nil
b = 10
print(b)    --> 10
```

Если вы присвоите nil глобальной переменной, то Lua поведет себя так, как если бы эта переменная никогда не использовалась:

```
b = nil
print(b)    --> nil
```

После данного присваивания Lua может со временем высвободить память, выделенную под эту переменную.

1.4. Автономный интерпретатор

Автономный интерпретатор (также называемый `lua.c` в связи с названием его исходного файла или просто `lua` из-за имени его выполнимого файла) — это небольшая программа, которая позволяет использовать Lua непосредственно. В данном разделе представлены ее основные опции.

Когда интерпретатор загружает файл, то он пропускает первую строку кода, если она начинается с октоторпа ('#'). Это свойство позволяет использовать Lua как скриптовый интерпретатор в UNIX-системах. Если вы начнете ваш скрипт с чего-нибудь вроде

```
#!/usr/local/bin/lua
```

(при условии, что автономный интерпретатор находится в

/usr/local/bin) или

```
#!/usr/bin/env lua
```

то вы можете вызвать ваш скрипт напрямую, без явного вызова интерпретатора Lua.

Использование lua следующее:

```
lua [options] [script [args]]
```

Все параметры необязательны. Как мы уже видели, когда мы запускаем lua без аргументов, интерпретатор переходит в интерактивный режим.

Опция -e позволяет нам вводить код прямо в командной строке, как показано ниже:

```
% lua -e "print (math.sin(12))"    -->
-0.53657291800043
```

(UNIX требует двойных кавычек, чтобы командная оболочка не стала интерпретировать круглые скобки).

Опция -l загружает библиотеку. Как мы уже видели ранее, -i вызывает интерактивный режим после выполнения остальных аргументов. Таким образом, следующий вызов загрузит библиотеку lib, затем выполнит присваивание x=10 и выведет приглашение ввода для взаимодействия.

```
% lua -i -llib -e "x = 10"
```

В интерактивном режиме вы можете напечатать значение любого выражения, набрав строку, начинающуюся со знака равенства, за которым следует выражение:

```
> = math.sin(3)    --> 0.14112000805987
> a = 30
> = a              --> 30
```

Это особенность позволяет использовать Lua как калькулятор.

Перед выполнением своих аргументов интерпретатор ищет переменную окружения с именем `LUA_INIT_5_2` или, если такой переменной нет, `LUA_INIT`. Если одна из этих переменных присутствует и содержит *@имя_файла*, то интерпретатор выполнит заданный файл. Если `LUA_INIT_5_2` (или `LUA_INIT`) определена, но ее содержимое не начинается с '@', то интерпретатор считает, что она содержит код Lua и выполняет его. `LUA_INIT` предоставляет огромные возможности по конфигурированию автономного интерпретатора, поскольку при конфигурировании нам доступна вся функциональность Lua. Мы можем предварительно загружать пакеты, изменять текущий путь, определять наши собственные функции, переименовывать или уничтожать функции и т. д.

Скрипт может получать свои аргументы из предопределенной глобальной переменной `arg`. При таком вызове, как `%lua script a b c`, интерпретатор перед выполнением скрипта создаст таблицу `arg` со всеми аргументами командной строки. Имя скрипта располагается по индексу 0, первый аргумент (в примере это `a`) располагается по индексу 1 и т. д. Предшествующие опции располагаются по отрицательным индексам, поскольку они находятся перед скриптом. Например, рассмотрим этот вызов:

```
% lua -e "sin=math.sin" script a b
```

Интерпретатор собирает аргументы следующим образом:

```
arg[-3] = "lua"
arg[-2] = "-e"
arg[-1] = "sin=math.sin"
arg[0]  = "script"
arg[1]  = "a"
arg[2]  = "b"
```

Чаще всего скрипт использует только положительные индексы (в примере это `arg[1]` и `arg[2]`).

Начиная с Lua 5.1 скрипт также может получить свои аргументы посредством выражения с переменным числом аргументов. В

главном теле скрипта выражение `...` (три точки) передает в скрипт эти аргументы (мы обсудим выражения с переменным числом аргументов в разделе 5.2).

Упражнения

Упражнение 1.1. Запустите пример с факториалом. Что случится с вашей программой, если вы введете отрицательное число? Измените пример, чтобы избежать этой проблемы.

Упражнение 1.2. Запустите пример `twice`, один раз загружая файл при помощи опции `-l`, а другой раз через `dofile`. Что для вас предпочтительнее?

Упражнение 1.3. Можете ли вы назвать другой язык, использующий `--` для комментариев?

Упражнение 1.4. Какие из следующих строк являются допустимыми идентификаторами?

```
___   _end   End   end   until?   nil   NULL
```

Упражнение 1.5. Напишите простой скрипт, который печатает свое имя, не зная его заранее.

ГЛАВА 2

Типы и значения

Lua — язык с динамической типизацией. В нем нет определений типов; каждое значение содержит в себе свой собственный тип.

В Lua существует восемь базовых типов: *nil*, *boolean*, *number*, *string*, *table*, *function*, *userdata* и *thread*. Функция `type` возвращает имя типа любого заданного значения:

```
print(type("Hello world"))    --> string
print(type(10.4*3))           --> number
print(type(print))            --> function
print(type(type))             --> function
print(type(true))             --> boolean
print(type(nil))              --> nil
print(type(type(X)))          --> string
```

Последняя строка вернет `string` вне зависимости от значения `X`, поскольку результат `type` всегда является строкой.

У переменных нет предопределенных типов; любая переменная может содержать значения любого типа:

```
print(type(a))    --> nil ('a' не инициализирована)
a = 10
print(type(a))    --> number
a = "a string!!"
print(type(a))    --> string
a = print         -- да, это допустимо!
a(type(a))        --> function
```

Обратите внимание на последние две строки: в Lua функции являются значениями первого класса; ими можно манипулировать, как и любыми другими значениями. (Более подробно мы рассмотрим данные средства в главе 6.)

Обычно, когда вы используете одну и ту же переменную для

значений разных типов, это приводит к запутанному коду. Однако, иногда разумное использование этой возможности оказывается полезным, например, при использовании nil, чтобы отличать нормальное возвращаемое значение от непредусмотренного состояния.

2.1. Отсутствие значения (`nil`)

Тип **nil** — это тип с единственным значением, ***nil***, основная задача которого состоит в том, чтобы отличаться от всех остальных значений. Lua использует nil как нечто, не являющееся значением, чтобы изобразить отсутствие подходящего значения. Как мы уже видели, глобальные переменные по умолчанию имеют значение nil до своего первого присваивания, и вы можете присвоить nil глобальной переменной, чтобы удалить ее.

2.2. Логические значения (`boolean`)

Тип **boolean** обладает двумя значениями — ***true*** и ***false***, которые представляют традиционные логические (или булевы) значения. Однако, не только булевы значения могут выражать условие: в Lua условие может быть представлено любым значением. Проверки условий (например, условий в управляющих структурах) считают nil и булево false ложными, а все прочие значения истинными. В частности, при проверках условий Lua считает ноль и пустую строку истинными значениями.

На протяжении данной книги «ложными» будут называться значения nil и false, а «истинными» — все остальные. Когда речь идет именно о булевых значениях, они указываются явно как false или true.

2.3. Числа (number)

Тип **number** представляет вещественные числа, т.е. числа двойной точности с плавающей точкой (тип double в C). В Lua нет целочисленного типа (тип integer в C).

Некоторые опасаются, что даже простые операции инкремента или сравнения могут некорректно работать с числами с плавающей точкой. Однако, на самом деле это не так. Практически все платформы в наше время поддерживают стандарт IEEE 754 для представления чисел с плавающей точкой. Согласно этому стандарту, единственным возможным источником ошибок является ошибка представления, которая происходит, когда число не может быть точно представлено. Операция округляет свой результат, только если у него нет точного представления. Любая операция, результат которой может быть точно представлен, должна выдавать его без округления.

На самом деле любое целое число вплоть до 2^{53} (приблизительно 10^{16}) имеет точное представление в виде числа двойной точности с плавающей точкой. Когда вы используете вещественные числа для представления целых, никаких ошибок округления не происходит, пока модуль числа не превышает 2^{53}. В частности, число Lua может представлять любые 32-битовые целые числа без проблем с округлением.

Разумеется, у дробных чисел могут быть ошибки представления. Эта ситуация не отличается от той, когда вы пользуетесь ручкой и бумагой. Если мы хотим записать `1/7` в десятичном виде, то мы должны будем где-то остановиться. Если мы используем десять цифр для представления числа, то `1/7` будет округлено до `0.142857142`. Если мы вычислим `1/7*7` при помощи десяти цифр, то получим `0.999999994`, что отличается от `1`. Более того, дробные числа, которые имеют конечное представление в десятичном виде, могут иметь бесконечное представление в

двоичном виде. Например, `12.7-20+7.3` не равно нулю, поскольку у обоих чисел, `12.7` и `7.3`, нет точного конечного представления в двоичном виде (см. упражнение 2.3).

Прежде чем мы продолжим, запомните: у целых чисел есть точное представление и потому нет ошибок округления.

Большинство современных процессоров выполняет операции с плавающей точкой так же быстро, как и с целыми числами (или даже быстрее). Тем не менее, легко скомпилировать Lua так, чтобы он использовал для числовых значений другой тип, например, длинные целые числа (тип long в C) или числа одинарной точности с плавающей точкой (тип float в C). Это особенно удобно для платформ без аппаратной поддержки чисел с плавающей точкой, например, для встраиваемых систем. Подробности смотрите в файле `luaconf.h` из дистрибутива.

Мы можем записывать числовые константы вместе с необязательной десятичной дробной частью и необязательным десятичным порядком. Примеры допустимых числовых констант:

```
4    0.4    4.57e-3    0.3e12    5E+20
```

Более того, мы также можем использовать шестнадцатеричные константы, применив префикс `0x`. Начиная с Lua 5.2, шестнадцатеричные константы также могут иметь дробную часть и двоичный порядок (с префиксом '`p`' или '`P`'), как в следующих примерах:

```
0xff (255)    0x1A3 (419)    0x0.2 (0.125)    0x1p-1
(0.5)
0xa.bp2 (42.75)
```

(Для каждой константы мы добавили в круглых скобках ее десятичное представление.)

2.4. Строки (`string`)

Тип **string** в Lua имеет обычный смысл: последовательность символов. Lua поддерживает все 8-битовые символы, и его строки могут содержать символы с любыми числовыми кодами, включая нуль-символы. Это значит, что вы можете хранить в виде строк любые бинарные данные. Вы также можете хранить строки Юникода в любом представлении (UTF-8, UTF-16 и т. д.). Стандартная библиотека, которая идет вместе с Lua, не предлагает явную поддержку этих представлений. Тем не менее, вы можете работать со строками UTF-8 вполне привычным образом, что будет рассмотрено в разделе 21.7.

Строки в Lua являются неизменяемыми значениями. Вы не можете изменить символ внутри строки, как вы делаете это в C; вместо этого вы создаете новую строку с необходимыми изменениями, как показано в следующем примере:

```
a = "one string"
b = string.gsub(a, "one", "another")   -- меняет
части строки
print(a)    --> строка "one"
print(b)    --> строка "another"
```

Строки в Lua подвержены автоматическому управлению памятью, как и любые другие объекты Lua (таблицы, функции и т. д.). Это значит, что вам не нужно беспокоиться о выделении и освобождении строк; этим за вас займется Lua. Строка может состоять из одного символа или целой книги. Программы, которые обрабатывают строки из 100K или 10M символов, — не редкость в Lua.

Вы можете получить длину строки, используя префиксную операцию '#' (называемую *операцией длины*):

```
a = "hello"
print(#a)                --> 5
print(#"good\0bye")      --> 8
```

Строковые литералы

Мы можем определять границы строковых литералов при помощи пар одинарных или двойных кавычек:

```
a = "a line"
b = 'another line'
```

Эти виды записи эквиваленты; единственное отличие состоит в том, что внутри одного вида кавычек вы можете использовать другой, не применяя экраны (т.е. экранированные/управляющие последовательности).

Это дело вкуса, но большинство программистов всегда использует кавычки одного вида для одних и тех же видов строк, где «виды» строк зависят от программы. Например, библиотека, которая работает с XML, может зарезервировать строки в одинарных кавычках для фрагментов XML, поскольку эти фрагменты часто содержат двойные кавычки.

Строки в Lua могут содержать следующие C-образные экранированные последовательности:

`\a`	звонок
`\b`	возврат на одну позицию (backspace)
`\f`	перевод страницы
`\n`	перевод строки
`\r`	возврат каретки
`\t`	горизонтальная табуляция
`\v`	вертикальная табуляция
`\\`	обратный слеш
`\"`	двойная кавычка
`\'`	одинарная кавычка

Следующие примеры иллюстрируют их использование:

```
> print("one line\nnext line\n\"in quotes\", 'in
quotes'")
one line
next line
"in quotes", 'in quotes'
```

```
> print('a backslash inside quotes: \'\\\'')
a backslash inside quotes: '\'

> print("a simpler way: '\\'")
a simpler way: '\'
```

Мы также можем задать символ в строке при помощи его числового значения через экранированные последовательности `\`*ddd* и `\x\`*hh*, где *ddd* — это последовательность не более чем из трех десятичных цифр, а *hh* — последовательность ровно из двух шестнадцатеричных цифр. В качестве примера посложнее возьмем два литерала: "`alo\n123\`" и '`\971o\10\04923`'. Они обладают одним и тем же значением в системе, использующей ASCII: 97 — это код ASCII для '`a`', 10 — это код для символа перевода строки, а 49 — это код для цифры '`1`'. (В этом примере мы должны записать значение 49 при помощи трех десятичных цифр как `\049`, поскольку за ним следует другая цифра; иначе Lua прочтет это число как 492). Мы также можем записать эту строку как '`\x61\x6c\x6f\x0a\x31\x32\x33\x22`', представляя каждый символ его шестнадцатеричным кодом.

Дпинные строки

Мы можем задавать строковые литералы при помощи пар из двойных квадратных скобок, как мы делали это с длинными комментариями. Литералы в этой скобочной форме могут занимать несколько строк, а экранированные последовательности в этих строках не будут интерпретироваться. Более того, эта форма игнорирует первый символ строки, если это перевод строки. Эта форма особенно удобна для написания строк, содержащих большие фрагменты кода, как в следующем примере:

```
page = [[
<html>
```

```
<head>
  <title>An HTML Page</title>
</head>
<body>
  <a href="http://www.lua.org">Lua</a>
</body>
</html>
]]

write(page)
```

Иногда вам может потребоваться заключить в квадратные скобки фрагмент кода, который содержит нечто вроде `a = b[c[i]]` (обратите внимание на `]]` в этом коде) или содержит уже закомментированный фрагмент. Чтобы справиться с подобными ситуациями, вы можете добавить любое количество знаков равенства между двумя открывающими квадратными скобками, например, так: `[===[`. После этого изменения строковый литерал завершится только на следующих закрывающих квадратных скобках с тем же самым количеством знаков равенства между ними (`]===]` для нашего примера). Сканер Lua игнорирует пары скобок с разным количеством знаков равенства. Путем подбора подходящего количества знаков равенства вы можете заключить в квадратные скобки любой строковый литерал без необходимости добавлять в него экраны.

Такая же возможность действует и для комментариев. Например, если вы начнете длинный комментарий с `--[=[`, то он будет продолжаться вплоть до следующей пары `]=]`. Эта возможность позволяет запросто закомментировать фрагмент кода, который содержит уже закомментированные части.

Длинные строки — это идеальный формат для включения текста в виде литерала в ваш код, но вам не следует использовать их для нетекстовых литералов. Хотя строковые литералы в Lua могут содержать любые символы, использовать такие символы в своем коде — не очень хорошая идея: вы можете столкнуться с проблемами при работе с вашим текстовым редактором; более того,

последовательности для завершения строк наподобие "\r\n" могут при чтении измениться на "\n". Вместо этого лучше кодировать произвольные бинарные данные при помощи числовых (десятичных и шестнадцатеричных) экранированных последовательностей, таких как "\x13\x01\xA1\xBB". Однако, это представляет проблему для длинных строк, поскольку может привести к строкам довольно большого размера.

Для подобных ситуаций Lua 5.2 предлагает экранированную последовательность \z: она пропускает все последующие символы в строке вплоть до первого непробельного символа. Следующий пример иллюстрирует ее применение:

```
data = "\x00\x01\x02\x03\x04\x05\x06\x07\z

        \x08\x09\xOMxOB\xOC\xOD\xOE\xOF"
```

Находящаяся в конце первой строки \z пропускает последующий конец строки и отступ следующей строки, поэтому в итоговой строке за байтом \x07 сразу же следует байт \x08.

Приведения типов

Lua обеспечивает автоматическое преобразование между числами и строками во время выполнения программ. Любая числовая операция, примененная к строке, пытается преобразовать эту строку в число:

```
print("10" + 1)          --> 11
print("10 + 1")          --> 10 + 1
print("-5.3e-10"*"2")    --> -1.06e-09
print("hello" + 1)       -- ОШИБКА (невозможно
преобразовать "hello")
```

Lua применяет подобные преобразования не только в арифметических операциях, но и в других местах, где ожидается число, таких как аргумент math.sin.

Верно и обратное — каждый раз, когда Lua находит число там, где ожидает строку, он преобразует это число в строку:

```
print(10 .. 20)          --> 1020
```

(Операция `..` служит в Lua для конкатенации строк. Когда вы записываете ее сразу после числа, вы должны отделить их друг от друга при помощи пробела; иначе Lua решит, что первая точка — это десятичная точка числа.)

Сегодня мы не уверены, что эти автоматические приведения типов были хорошей идеей в дизайне Lua. Как правило, лучше на них не рассчитывать. Они удобны в некоторых местах; но добавляют сложности как языку, так и программам, которые их используют. В конце концов, строки и числа — это разные вещи, несмотря на все эти преобразования. Сравнение вроде `10="10"` дает в результате false, поскольку `10` — это число, а `"10"` — это строка.

Если вам нужно явно преобразовать строку в число, то вы можете воспользоваться функцией `tonumber`, которая возвращает nil, если строка не является правильным числом Lua:

```
line = io.read()            -- читает строку
n = tonumber(line)          -- пытается преобразовать
ее в число
if n == nil then
  error(line .. " is not a valid number")
else
  print(n*2)
end
```

Для преобразования числа в строку вы можете использовать функцию `tostring` или конкатенировать число с пустой строкой:

```
print(tostring(10) == "10")     --> true
print(10 .. "" == "10")         --> true
```

Эти преобразования всегда работают.

2.5. Таблицы (`table`)

Тип **table** представляет ассоциативные массивы. Ассоциативный массив — это массив, который может быть индексирован не только числами, но и строками или любым другим значением языка, кроме nil.

Таблицы являются главным (на самом деле единственным) механизмом структурирования данных в Lua, притом очень эффективным. Мы используем таблицы для представления обычных массивов, множеств, записей и других структур данных простым, однородным и эффективным способом. Также Lua использует таблицы для представления пакетов и объектов. Когда мы пишем `io.read`, мы думаем о «функции `read` из модуля `io`». Для Lua это выражение означает «индексировать таблицу `io`, используя строку `read` в качестве ключа».

Таблицы в Lua не являются ни значениями, ни переменными; они *объекты*. Если вы знакомы с массивами в Java или Scheme, то вы понимаете, что я имею в виду. Вы можете рассматривать таблицу как динамически выделяемый объект; ваша программа работает только со ссылками (указателями) на них. Lua никогда не прибегает к скрытому копированию или созданию новых таблиц. Более того, вам не нужно объявлять таблицу в Lua; на самом деле для этого даже не существует способа. Вы создаете таблицы при помощи *выражения-конструктора*, которое в своей простейшей форме записывается как `{}`:

```
a = {}              -- создает таблицу и сохраняет
ссылку на нее в 'a'
k = "x"
a[k] = 10           -- новая запись с ключом "x" и
значением 10
a[20] = "great"     -- новая запись с ключом 20 и
значением "great"
print(a["x"])       --> 10
k = 20
```

```
print(a[k])        --> "great"
a["x"] = a["x"] + 1    -- инкрементирует запись "x"
print(a["x"])      --> 11
```

Таблица всегда анонимна. Не существует постоянной связи между переменной, которая хранит таблицу, и самой таблицей:

```
a = {}
a["x"] = 10
b = a              -- 'b' ссылается на ту же таблицу,
что и 'a'
print(b["x"])      --> 10
b["x"] = 20
print(a["x"])      --> 20
a = nil            -- лишь 'b' по-прежнему ссылается на
ту таблицу
b = nil            -- ссылок на таблицу не осталось
```

Когда в программе больше не остается ссылок на таблицу, сборщик мусора Lua со временем удалит эту таблицу, чтобы повторно использовать ее память.

Каждая таблица может хранить значения с разными типами индексов и растет по мере добавления новых записей:

```
a = {}             -- пустая таблица
-- создает 1000 новых записей
for i = 1, 1000 do a[i] = i*2 end
print(a[9])        --> 18
a["x"] = 10
print(a["x"])      --> 10
print(a["y"])      --> nil
```

Обратите внимание на последнюю строку: как и глобальные переменные, поля таблицы возвращают nil, когда не инициализированы. Так же, как и с глобальными переменными, вы можете присвоить полю таблицы nil, чтобы его удалить. Это не совпадение: Lua хранит глобальные переменные в обыкновенных таблицах. Мы рассмотрим это подробнее в главе 14.

Для представления записей вы используете имя поля как индекс.

Lua поддерживает это представление, предлагая `a.name` в качестве синтаксического сахара для `a["name"]`. Таким образом, мы можем переписать последние строки предыдущего примера следующим, более чистым образом:

```
a.x = 10       -- то же, что и a["x"] = 10
print(a.x)     -- то же, что и print(a["x"])
print(a.y)     -- то же, что и print(a["y"])
```

Для Lua эти две формы эквивалентны и могут свободно использоваться вместе. Для читателя, однако, каждая форма может сообщать о разном намерении. Точечная нотация ясно показывает, что мы используем таблицу как запись, где у нас есть некоторый набор постоянных, предопределенных ключей. Строковая нотация дает представление о том, что у таблицы в качестве ключа может быть любая строка, и что по некоторой причине мы работаем с этим конкретным ключом.

Типичная ошибка новичков — спутать `a.x` с `a[x]`. Первая форма соответствует `a["x"]`, то есть таблица индексирована при помощи строки "x". Вторая форма означает, что таблица индексирована при помощи значения переменной `x`. Взгляните на разницу:

```
a = {}
x = "y"
a[x] = 10                  -- записывает 10 в поле "y"
print(a[x])     --> 10     -- значение поля "y"
print(a.x)      --> nil    -- значение поля "x" (не
определено)
print(a.y)      --> 10     -- значение поля "y"
```

Чтобы представить традиционный массив или список, просто используйте таблицу с целочисленными ключами. Нет ни способа, ни необходимости объявлять размер; вы всего лишь инициализируете те элементы, которые вам нужны:

```
-- считывает 10 строк, сохраняя их в таблице
```

```
a = {}
for i = 1, 10 do
  a[i] = io.read()
end
```

Поскольку вы можете индексировать таблицу любым значением, вы можете начинать индексы массива с любого числа, которое вам нравится. Однако, в Lua принято начинать массивы с единицы (а не с нуля, как в C), и некоторые средства Lua придерживаются этого соглашения.

Обычно, когда вы работаете со списком, вам нужно знать его длину. Она может быть константой или храниться где-то еще. Часто мы храним длину списка в нечисловом поле таблицы; по историческим причинам некоторые программы используют для этих целей поле "`n`".

Однако, зачастую длина не может быть явно определена. Как вы помните, любой неинициализированный индекс равен nil; вы можете использовать это значение как граничную метку для обозначения конца списка. Например, после считывания десяти строк в список легко запомнить, что его длина равна 10, поскольку его числовыми ключами являются `1,2,...,10`. Этот прием работает только тогда, когда у списка нет *дыр*, т.е. элементов nil внутри него. Мы называем такой список *последовательностью* (sequence).

Для последовательностей Lua предлагает операцию длины '`#`'. Она возвращает последний индекс или длину последовательности. Например, вы можете напечатать строки, считанные в предыдущем примере, при помощи следующего кода:

```
-- печатает строки
for i = 1, #a do
  print(a[i])
end
```

Поскольку мы можем индексировать таблицу с помощью любого типа, то при индексировании таблицы возникают те же тонкости, что и при проверке на равенство. Хотя мы можем

индексировать таблицу и с помощью целого числа `0`, и с помощью строки "`0`", эти два значения различны и тем самым соответствуют разным записям таблицы. Аналогично, строки "`+1`", "`01`" и "`1`" также соответствуют разным записям таблицы. Когда вы не уверены насчет действительных типов ваших индексов, для верности используйте явное приведение типов:

```
i = 10; j = "10"; k = "+10"
a = {}
a[i] = "one value"
a[j] = "another value"
a[k] = "yet another value"
print(a[i])                    --> one value
print(a[j])                    --> another value
print(a[k])                    --> yet another value
print(a[tonumber(j)])          --> one value
print(a[tonumber(k)])          --> one value
```

В вашей программе могут появиться трудноуловимые ошибки, если не уделять внимание данному моменту.

2.6. Функции (`function`)

Функции (тип **function**) являются в Lua значениями первого класса: программы могут хранить функции в переменных, передавать функции как аргументы для других функций и возвращать функции как результаты. Подобные возможности придают языку огромную гибкость; программа может переопределить функцию, чтобы добавить новую функциональность, или просто стереть функцию для создания безопасного окружения при выполнении фрагмента ненадежного кода (например, кода, полученного по сети). Более того, Lua предоставляет хорошую поддержку функционального программирования, включая вложенные функции с соответствующим лексическим окружением; просто дождитесь

главы 6. Наконец, функции первого класса играют ключевую роль в объектно-ориентированных возможностях Lua, как мы увидим в главе 16.

Lua может вызывать функции, написанные на Lua, и функции, написанные на C. Обычно мы прибегаем к функциям C для достижения более высокого быстродействия и для доступа к средствам, недоступным непосредственно из Lua, таким как средства операционной системы. Все стандартные библиотеки в Lua написаны на C. Они включают в себя функции обработки строк, обработки таблиц, ввода-вывода, доступа к базовым средствам операционной системы, математические и отладочные функции.

Мы обсудим функции Lua в главе 5, а функции C в главе 27.

2.7. Пользовательские данные (`userdata`)

Тип **userdata** позволяет запоминать произвольные данные C в переменных Lua. У него нет предопределенных операций в Lua, за исключением присваивания и проверки на равенство. Пользовательские данные используются для представления новых типов, созданных прикладной программой или библиотекой, написанной на C; например, стандартная библиотека ввода-вывода использует их для представления открытых файлов. Мы более подробно обсудим этот тип позже, когда перейдем к C API.

2.8. Нити (`thread`)

Тип **thread** будет разобран в главе 9, где мы рассмотрим сопрограммы.

Упражнения

Упражнение 2.1. Что является значением выражения `type(nil)==nil`? (Вы можете использовать Lua для проверки своего ответа.) Можете ли вы объяснить результат?

Упражнение 2.2. Какие из приведенных ниже чисел являются допустимыми в Lua? Каковы их значения?

```
.0e12    .e12    0.0e    0x12    0xABFG    0xA FFFF
0xFFFFFFFF
0x    0x1P10    0.1e1    0x0.1p1
```

Упражнение 2.3. Число `12.7` равно дроби `127/10`, где знаменатель является степенью десяти. Можете ли вы представить его в виде простой дроби, где знаменатель является степенью двойки? Как насчет числа `5.5`?

Упражнение 2.4. Как вы можете вставить следующий фрагмент XML в виде строки в Lua?

```
<![CDATA[
Hello world
]]>
```

Используйте не менее двух разных способов.

Упражнение 2.5. Допустим, вам нужно записать длинную последовательность произвольных байт в виде строкового литерала в Lua. Как вы это сделаете? Обратите внимание на такие моменты, как читаемость, максимальную длину строки и быстродействие.

Упражнение 2.6. Рассмотрите следующий код:

```
a = {}; a.a = a
```

Что будет значением `a.a.a.a`? Какое-либо `a` в этой последовательности как-то отличается от остальных?

Теперь добавьте следующую строку к предыдущему коду:

```
a.a.a.a = 3
```

Что теперь будет значением `a.a.a.a`?

ГЛАВА 3

Выражения

Выражения служат для получения значений. Выражения в Lua включают числовые константы, строковые литералы, переменные, унарные и бинарные операции и вызовы функций. В выражения также могут входить нетрадиционные определения функций и конструкторы таблиц.

3.1. Арифметические операции

Lua поддерживает стандартные арифметические операции: бинарные '+' (сложение), '-' (вычитание), '*' (умножение), '/' (деление), '^' (возведение в степень), '%' (остаток от деления) и унарную '-' (отрицание). Все они работают с вещественными числами. Например, `x^0.5` вычисляет квадратный корень из `x`, а `x^(-1/3)` вычисляет обратно пропорциональное значение его кубического корня.

Следующее правило определяет операцию остатка от деления:

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

Для целочисленных операндов она работает обычным образом и дает результат с тем же знаком, что и у второго аргумента. Для вещественных операндов у нее есть некоторое дополнительное применение. Например, `x%1` дает дробную часть `x`, следовательно, `x-x%1` дает его целую часть. Аналогично, `x-x%0.01` дает `x` ровно с двумя десятичными цифрами после запятой:

```
x = math.pi
print(x - x%0.01)                    --> 3.14
```

В качестве другого примера применения операции остатка от деления допустим, что вам требуется проверить, будет ли транспортное средство после поворота на заданный угол возвращаться в исходное положение. Если угол задан в градусах, то вы можете использовать следующую формулу:

```
local tolerance = 10
function isturnback (angle)
  angle = angle % 360
  return (math.abs(angle - 180) < tolerance)
end
```

Это определение работает даже для отрицательных углов:

```
print(isturnback(-180))          --> true
```

Если нам потребуется работать с радианами вместо градусов, то мы просто изменим константы в нашей функции:

```
local tolerance = 0.17
function isturnback (angle)
  angle = angle % (2*math.pi)
  return (math.abs(angle - math.pi) < tolerance)
end
```

Операция `angle%(2*math.pi)` — это все, что нам нужно для приведения любого угла к значению в интервале [0, 2π).

3.2. Операции сравнения

Lua предоставляет следующие операции сравнения:

```
<    >    <=    >=    ==    ~=
```

Все эти операции всегда производят булево значение.

Операция `==` проверяет на равенство; операция `~=` проверяет на неравенство. Мы можем применять обе эти операции к любым двум значениям. Если значения обладают разными типами, то Lua считает,

что они не равны. В противном случае Lua сравнивает их в соответствии с их типами. В частности, nil равно только самому себе.

Lua сравнивает таблицы и пользовательские данные по ссылке, то есть два таких значения считаются равными, только если они являются одним и тем же объектом. Например, после выполнения следующего кода:

```
a = {}; a.x = 1; a.y = 0
b = {}; b.x = 1; b.y = 0
c = a
```

мы получим `a == c`, но `a ~= b`.

Мы можем применять операции порядка лишь к двум числам или двум строкам. Lua сравнивает строки в алфавитном порядке, следуя установленной для Lua локали. Например, для португальской локали Latin-1 мы получим `"acai" < "açaí" < "acorde"`. Значения, отличные от чисел и строк, можно сравнивать только на равенство (и неравенство).

При сравнении значений разных типов нужно быть аккуратным: помните, что `"0"` отличается от `0`. Более того, `2<15` очевидно истинно, но `"2"<"15"` ложно (из-за алфавитного порядка). Чтобы избежать непостоянных результатов, Lua выбрасывает ошибку, когда вы смешиваете строки и числа при выявлении их порядка, таком как `2<"15"`.

3.3. Логические операции

Логические операции — это **and**, **or** и **not**. Как и управляющие структуры, все логические операции трактуют false и nil как ложные, а все остальные — как истинные значения. Операция **and** возвращает свой первый аргумент, если он ложный, иначе она возвращает свой второй аргумент. Операция **or** возвращает свой первый аргумент, если он не ложный; иначе она возвращает свой

второй аргумент:

```
print(4 and 5)          --> 5
print(nil and 13)       --> nil
print(false and 13)     --> false
print(4 or 5)           --> 4
print(false or 5)       --> 5
```

Обе операции, **and** и **or**, используют сокращенное вычисление, то есть они вычисляют свой второй операнд только при необходимости. Сокращенное вычисление обеспечивает отсутствие ошибок во время выполнения для выражений вроде `(type(v) == "table" and v.tag == "h1")`: Lua не будет пытаться вычислить `v.tag`, когда `v` не является таблицей.

Полезной идиомой Lua является `x=x or v`, которая эквивалентна

```
if not x then x = v end
```

То есть значение `x` по умолчанию устанавливается равным значению `v`, если `x` не определен (при условии, что `x` не равен false).

Другой полезной идиомой является `(a and b) or c` или просто `a and b or c`, поскольку у **and** более высокий приоритет, чем у **or**. Она эквивалентна выражению `a?b:c` в языке C, при условии что `b` не ложно. Например, мы можем выбрать максимум из двух чисел `x` и `y` при помощи такого оператора:

```
max = (x > y) and x or y
```

Когда `x > y`, первое выражение с **and** истинно, поэтому **and** возвращает свое второе выражение (`x`), которое всегда истинно (поскольку это число), и затем выражение с **or** возвращает значение своего первого выражения, `x`. Когда `x > y` ложно, выражение с **and** тоже ложно, поэтому **or** возвращает свое второе выражение, `y`.

Операция **not** всегда возвращает булево значение:

```
print(not nil)          --> true
```

```
print(not false)      --> true
print(not 0)          --> false
print(not not 1)      --> true
print(not not nil)    --> false
```

3.4. Конкатенация

Lua обозначает операцию конкатенации строк как `..` (две точки). Если один из операндов является числом, то Lua переведет его в строку. (Некоторые языки используют для конкатенации операцию '`+`', но в Lua `3+5` отличается от `3..5`.)

```
print("Hello " .. "World")    --> Hello World
print(0 .. 1)                 --> 01
print(000 .. 01)              --> 01
```

Помните, что строки в Lua являются неизменяемыми значениями. Операция конкатенации всегда создает новую строку, не изменяя свои операнды:

```
a = "Hello"
print(a .. " World")    --> Hello World
print(a)                --> Hello
```

3.5. Операция длины

Операция длины работает со строками и таблицами. Со строками она дает количество байт в строке. С таблицами она возвращает длину представленной ими *последовательности*.

С операцией длины связано несколько распространенных идиом для работы с последовательностями:

```
print(a[#a])     -- печатает последнее значение
последовательности 'a'
a[#a] = nil      -- удаляет это последнее значение
a[#a + 1] = v    -- добавляет 'v' к концу списка
```

Как мы видели в предыдущей главе, операция длины непредсказуема для списков с дырами (т.е. nil). Она работает только для последовательностей, которые определены как списки без дыр. Более точно, *последовательность* — это таблица, где числовые ключи образуют множество `1,...,`*n* для некоторого *n*. (Помните, что любой ключ со значением nil на самом деле в таблице отсутствует.) В частности, таблица без числовых ключей — это последовательность нулевой длины.

С годами было много предложений по расширению значения операции длины на списки с дырами, но это легче сказать, чем сделать. Проблема в том, что поскольку список на самом деле является таблицей, то понятие «длины» несколько расплывчато. Например, рассмотрим список, получаемый в результате следующего кода:

```
a = {}
a[1] = 1
a[2] = nil    -- ничего не делает, так как a[2] уже
nil
a[3] = 1
a[4] = 1
```

Легко сказать, что длина этого списка четыре, и у него есть дыра по индексу 2. Однако, что можно сказать о следующем похожем примере?

```
a = {}
a[1] = 1
a[10000] = 1
```

Должны ли мы рассматривать `a` как список с 10 000 элементами, где 9 998 из них равны nil? Теперь программа делает следующее:

```
a[10000] = nil
```

Какова теперь длина этого списка? Должна ли она быть равна 9 999, поскольку программа удалила последний элемент? Или может

быть она по-прежнему равна 10 000, так как программа всего лишь изменила значение последнего элемента на nil? Или же длина должна уменьшиться до 1?

Другое распространенное предложение — сделать так, чтобы операция `#` возвращала общее число элементов в таблице. Эта семантика ясна и хорошо определена, но не несет в себе никакой пользы. Рассмотрим все предыдущие примеры и представим, насколько полезной оказалось бы подобная операция для алгоритмов, работающих со списками или массивами.

Еще более проблемными являются значения nil в конце списка. Какой должна быть длина следующего списка?

```
a = {10, 20, 30, nil, nil}
```

Вспомним, что для Lua поле с nil не отличается от отсутствующего поля. Таким образом, предыдущая таблица эквивалентна `{10,20,30}`; ее длина равна 3, а не 5.

Вы можете считать, что nil в конце списка — это крайне особенный случай. Однако, многие списки строятся путем добавления элементов по одному за раз. Любой список с дырами, построенный таким образом, должен был иметь значения nil в своем конце во время построения.

Многие списки, которые мы используем в наших программах, являются последовательностями (например, строка файла не может быть nil), и поэтому большую часть времени применение операции длины безопасно. Если вам действительно нужно обрабатывать списки с дырами, то вы должны хранить их длину явным образом где-то еще.

3.6. Приоритеты операций

Приоритеты операций в Lua, от высшего к низшему, следуют этой таблице:

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

Все бинарные операции левоассоциативны, за исключением '^' (возведение в степень) и '..' (конкатенация), которые правоассоциативны. Поэтому следующие выражения слева эквивалентны выражениям справа:

```
a+i < b/2+1          <-->    (a+i) < ((b/2)+1)
5+x^2*8              <-->    5+((x^2)*8)
a < y and y <= z     <-->    (a < y) and (y <= z)
-x^2                 <-->    -(x^2)
x^y^z                <-->    x^(y^z)
```

Когда сомневаетесь, всегда используйте круглые скобки. Это легче, чем заглядывать в справочник, так как при перечитывании кода у вас вновь могут возникнуть те же сомнения.

Конструкторы таблиц

Конструкторы — это выражения, которые создают и инициализируют таблицы. Они являются отличительной чертой Lua и одним из его наиболее полезных и универсальных механизмов.

Простейший конструктор — это пустой конструктор, `{}`, который создает пустую таблицу; мы уже видели его раньше. Конструкторы также инициализируют списки. Например, оператор

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"}
```

проинициализирует `days[1]` строкой "`Sunday`" (первый элемент конструктора имеет индекс 1, а не 0), `days[2]` строкой "`Monday`" и т.

д.:

```
print(days[4])    --> Wednesday
```

Lua также предлагает специальный синтаксис для инициализации таблиц, похожий на записи, как в следующем примере;

```
a = {x=10, y=20}
```

Эта строка эквивалентна следующим командам:

```
a = {}; a.x=10; a.y=20
```

Исходное выражение, тем не менее, проще и быстрее, поскольку Lua сразу создает таблицу с правильным размером.

Вне зависимости от того, каким конструктором мы пользовались для создания таблицы, из нее всегда можно добавлять и удалять поля:

```
w = {x=0, y=0, label="console"}
x = {math.sin(0), math.sin(1), math.sin(2)}
w[1] = "another field"    -- добавляет ключ 1 в
таблицу 'w'
x.f = w                   -- добавляет ключ "f" в
таблицу 'x'
print(w["x"])             --> 0
print(w[1])               --> another field
print(x.f[1])             --> another field
w.x = nil                 -- удаляет поле "x"
```

Однако, как я только что заметил, создание таблицы при помощи правильного конструктора более эффективно и, кроме того, более наглядно.

Мы можем смешивать эти два стиля инициализации (записи и списки) в одном и том же конструкторе:

```
polyline = {color="blue",
            thickness=2,
            npoints=4,
            {x=0, y=0},       -- polyline[1]
```

```
    {x=-10, y=0},     -- polyline[2]
    {x=-10, y=1},     -- polyline[3]
    {x=0, y=1}        -- polyline[4]
}
```

Приведенный выше пример также показывает, как можно вкладывать конструкторы один в другой для представления более сложных структур данных. Каждый из элементов `polyline[i]` — это таблица, представляющая собой запись:

```
print(polyline[2].x)    --> -10
print(polyline[4].y)    --> 1
```

У этих двух форм конструктора есть свои ограничения. Например, вы не можете инициализировать поля отрицательными индексами или строковыми индексами, которые не являются правильными идентификаторами. Для таких целей есть другой, более общий формат. В этом формате мы явно пишем индекс, который должен быть инициализирован как выражение, между квадратными скобками:

```
opnames = {["+"] = "add", ["-"] = "sub",
           ["*"] = "mul", ["/"] = "div"}

i = 20; s = "-"
a = {[i+0] = s, [i+1] = s..s, [i+2] = s..s..s}

print(opnames[s])    --> sub
print(a[22])         --> ---
```

Этот синтаксис хоть и более громоздкий, но при этом более гибкий: обе формы (в стиле списков и в стиле записей) являются частными случаями этого более общего синтаксиса. Конструктор `{x = 0, y = 0}` эквивалентен `{["x"] = 0, ["y"] = 0}`, а конструктор `{"r", "g", "b"}` эквивалентен `{[1] = "r", [2] = "g", [3] = "b"}`.

Вы всегда можете поставить запятую после последней записи. Эти замыкающие запятые необязательны, но всегда допустимы:

```
a = {[1]="red", [2]="green", [3]="blue",}
```

Данная гибкость освобождает программы, генерирующие конструкторы Lua, от необходимости обрабатывать последний элемент особым образом.

Наконец, вы всегда можете использовать в конструкторе точку с запятой вместо запятой. Я обычно использую точки с запятой для отделения различных секций в конструкторе, например, для отделения части в виде списка от части в виде записей:

```
{x=10, y=45; "one", "two", "three"}
```

Упражнения

Упражнение 3.1. Что напечатает следующая программа?

```
for i = -10, 10 do
  print(i, i % 3)
end
```

Упражнение 3.2. Что является результатом выражения `2^3^4`? А что насчет `2^-3^4`?

Упражнение 3.3. Мы можем представить многочлен $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$ в Lua как список его коэффициентов $\{a0, a1, \ldots, an\}$.

Напишите функцию, которая получает многочлен (представленный в виде таблицы) и значение `x`, а затем возвращает значение многочлена.

Упражнение 3.4. Можете ли вы написать функцию из предыдущего упражнения так, чтобы использовать n сложений и n умножений (и не использовать возведение в степень)?

Упражнение 3.5. Как вы можете проверить, что значение является булевым, не прибегая к функции `type`?

Упражнение 3.6. Рассмотрим следующее выражение:

```
(x and y and (not z)) or ((not y) and x)
```

Нужны ли в этом выражении круглые скобки? Как бы вы посоветовали использовать их в данном выражении?

Упражнение 3.7. Что напечатает следующий скрипт? Объясните.

```
sunday = "monday"; monday = "sunday"
t = {sunday = "monday", [sunday] = monday}
print(t.sunday, t[sunday], t[t.sunday])
```

Упражнение 3.8. Предположим, вы хотите создать таблицу, которая связывает каждую экранированную последовательность для строк (см. раздел 2.4) с ее значением. Как бы вы написали конструктор для такой таблицы?

ГЛАВА 4

Операторы

Lua поддерживает почти традиционный набор операторов, похожих на операторы в C или Pascal. Традиционные операторы включают в себя присваивание, управляющие структуры и вызовы процедур. Lua также поддерживает менее распространенные операторы, такие как множественное присваивание и объявления локальных переменных.

4.1. Операторы присваивания

Присваивание — это базовое средство изменения значения переменной или поля таблицы:

```
a = "hello" .. "world"
t.n = t.n + 1
```

Lua позволяет осуществлять *множественное присваивание* (multiple assignment), которое присваивает список значений списку переменных за один шаг. Например, в присваивании

```
a, b = 10, 2*x
```

переменная `a` получает значение `10`, а переменная `b` — значение `2*x`.

Во множественном присваивании Lua сперва вычисляет все значения и только затем выполняет присваивания. Поэтому мы можем использовать множественное присваивание, чтобы поменять местами два значения, как в следующих примерах:

```
x, y = y, x                    -- меняет местами значения
'x' с 'y'
a[i], a[j] = a[j], a[i]        -- меняет местами значения
```

```
'a[i]' с 'a[j]'
```

Lua всегда приводит количество значений к количеству переменных: когда список значений короче списка переменных, дополнительные переменные получают в качестве своих значений nil; когда длиннее список значений, дополнительные значения просто отбрасываются:

```
a, b, c = 0, 1
print(a, b, c)             --> 0    1     nil
a, b = a+1, b+1, b+2       -- значение b+2 игнорируется
print(a, b)                --> 1    2
a, b, c = 0
print(a, b, c)             --> 0    nil   nil
```

Последнее присваивание в примере выше показывает распространенную ошибку. Для инициализации набора переменных вы должны предоставить значение для каждой из них:

```
a, b, c = 0, 0, 0
print(a, b, c)         --> 0   0   0
```

В действительности большинство предыдущих примеров в чем-то искусственно. Я редко использую множественное присваивание просто, чтобы записать несколько не связанных между собой присваиваний в одну строку. В частности, множественное присваивание не быстрее эквивалентных ему одиночных присваиваний. Тем не менее, зачастую нам действительно требуется множественное присваивание. Мы уже видели пример, меняющий местами значения у двух переменных. Более распространенное использование заключается в получении нескольких значений при вызове функции. Как мы подробно обсудим в разделе 5.1, вызов функции может возвращать несколько значений. В таких случаях единственное выражение может предоставить значения для нескольких переменных. Например, в присваивании `a,b=f()` вызов `f` возвращает два значения: `a` получает первое, а `b` получает второе.

4.2. Локальные переменные и блоки

Кроме глобальных, Lua поддерживает и локальные переменные. Мы создаем локальные переменные при помощи оператора `local`:

```
j = 10          -- глобальная переменная
local i = 1     -- локальная переменная
```

В отличие от глобальных переменных, область видимости локальной переменной ограничена блоком, где она была объявлена. *Блок* — это тело управляющей структуры, тело функции или кусок кода (файл или строка, где переменная была объявлена):

```
x = 10
local i = 1          -- локальная для куска

while i <= x do
  local x = i*2      -- локальная для тела while
  print(x)           --> 2, 4, 6, 8, ...
  i = i + 1
end

if i > 20 then
  local x            -- локальная для тела "then"
  x = 20
  print(x + 2)       -- (напечатает 22, если условие
выполнится)
else
  print(x)           --> 10 (глобальная переменная)
end

print(x)             --> 10 (глобальная переменная)
```

Обратите внимание, что этот пример не будет работать так, как вы ожидаете, если ввести его в интерактивном режиме. В интерактивном режиме каждая строка — это самостоятельный кусок (кроме случая, когда команда набрана не полностью). Как только вы введете вторую строку примера (`local i=1`), Lua выполнит ее и начнет новый кусок кода со следующей строки. К тому моменту

локальное объявление уже выйдет из своей области видимости. Для решения этой проблемы мы можем явно ограничить весь этот блок, заключив его между ключевыми словами **do** и **end**. Как только вы введете **do**, команда закончится только на соответствующем ему **end**, поэтому Lua не будет пытаться выполнить каждую строку по отдельности.

Подобные блоки с **do** удобны и тогда, когда вам нужен более точный контроль над областью видимости некоторых локальных переменных:

```
do
  local a2 = 2*a
  local d = (b^2 - 4*a*c)^(1/2)
  x1 = (-b + d)/a2
  x2 = (-b - d)/a2
end                 -- область видимости 'a2' и 'd'
заканчивается здесь
print(x1, x2)
```

Хороший стиль программирования заключается в применении локальных переменных везде, где это возможно. Локальные переменные помогают вам избежать засорения глобального окружения ненужными именами. Более того, доступ к локальной переменной быстрее, чем к глобальной. И наконец, локальная переменная перестает существовать, как только заканчивается ее область видимости, позволяя сборщику мусора освободить память, занимаемую ее значением.

Lua обрабатывает объявления локальных переменных как операторы. Таким образом, вы можете записывать локальные объявления везде, где вы можете записать оператор. Область видимости объявленных переменных начинается сразу после объявления и длится до конца блока. Каждое объявление может включать инициализирующее присваивание, которое действует так же, как и традиционное присваивание: лишние значения отбрасываются, а лишние переменные получают значение nil. Если в

объявлении нет инициализирующего присваивания, то оно инициализирует все свои переменные значением nil:

```
local a, b = 1, 10
if a < b then
  print(a)     --> 1
  local a      -- неявное '= nil'
  print(a)     --> nil
  end          -- завершает блок, начатый с 'then'
print(a, b)    --> 1    10
```

В Lua распространена следующая идиома:

```
local foo = foo
```

Этот код создает локальную переменную `foo` и инициализирует ее значением глобальной переменной `foo`. (Локальная `foo` становится видимой только *после* этого объявления.) Эта идиома удобна, когда куску необходимо предварительно сохранить первоначальное значение переменной, даже если позже какая-нибудь другая функция изменит значение глобальной `foo`; это также ускоряет доступ к `foo`.

Поскольку многие языки вынуждают вас объявлять все локальные переменные в начале блока (или процедуры), некоторые считают, что объявлять переменные в середине блока является плохой практикой. Все как раз наоборот: объявляя переменную, только когда она действительно нужна, вам редко понадобится объявлять ее без инициализирующего значения (и поэтому вы вряд ли забудете ее проинициализировать). Более того, вы уменьшаете область видимости переменной, что облегчает чтение кода.

4.3. Управляющие конструкции

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

until завершает **repeat**.

Условное выражение управляющей структуры может дать любое значение. Помните о том, что Lua считает все значения, отличные от false и nil, истинными. (В частности, Lua считает истинными ноль и пустую строку.)

if then else

Оператор **if** проверяет свое условие и в зависимости от результата выполняет одну из своих частей, **then** или **else**. Часть с **else** является необязательной.

```
if a < 0 then a = 0 end

if a < b then return a else return b end

if line > MAXLINES then
  showpage()
  line = 0
end
```

Для записи вложенных операторов **if** вы можете использовать **elseif**. Это аналогично **else**, за которым следует **if**, но при этом не возникает необходимости в нескольких **end**:

```
if op == "+" then
  r = a + b
elseif op == "-" then
  r = a - b
elseif op == "*" then
  r = a*b
elseif op == "/" then
  r = a/b
else
  error("invalid operation")
end
```

Поскольку в Lua нет оператора switch, такие конструкции довольно распространены.

while

Как следует из названия, **while** повторяет свое тело до тех пор, *пока* условие истинно. Как обычно, Lua сперва проверяет условие **while**; если оно ложно, то цикл завершается; в противном случае Lua выполняет тело цикла и повторяет данный процесс.

```
local i = 1
while a[i] do
  print(a[i])
  i = i + 1
end
```

repeat

Как следует из названия, оператор **repeat–until** *повторяет* свое тело до тех пор, *пока* условие *не* станет истинным. Данный оператор производит проверку условия после выполнения тела, поэтому тело цикла будет выполнено хотя бы один раз.

```
-- печатает первую непустую введенную строку
repeat
  line = io.read()
until line ~= ""
print(line)
```

В отличие от многих других языков, в Lua условие входит в область видимости локальной переменной, объявленной внутри цикла:

```
local sqr = x/2
repeat
  sqr = (sqr + x/sqr)/2
  local error = math.abs(sqr^2 - x)
until error < x/10000 -- local 'error' still visible
here
```

Числовой for

Оператор **for** существует в двух вариантах: *числовой* **for** и *общий* **for**.

Числовой **for** имеет следующий синтаксис:

```
for var = exp1, exp2, exp3 do
  <что-либо>
end
```

Этот цикл будет выполнять *что-либо* для каждого значения `var` от `exp1` до `exp2`, используя `exp3` как *шаг* для увеличения `var`. Это третье выражение необязательно; когда оно отсутствует, Lua считает значение шага равным 1. В качестве типичных примеров можно привести

```
for i = 1, f(x) do print(i) end

for i = 10, 1, -1 do print(i) end
```

Если вам нужен цикл без верхнего предела, то вы можете использовать константу `math.huge`:

```
for i = 1, math.huge do
  if (0.3*i^3 - 20*i^2 - 500 >= 0) then
    print(i)
    break
  end
end
```

У цикла **for** есть некоторые тонкости, которые вам желательно знать, чтобы применять его наиболее эффективно. Во-первых, все три выражения вычисляются только один раз, перед началом цикла. Скажем, в нашем первом примере Lua вызовет `f(x)` всего один раз. Во-вторых, управляющая переменная является локальной переменной, автоматически объявляемой оператором **for**, и она видна лишь внутри цикла. Типичная ошибка — полагать, что эта переменная все еще существует после окончания цикла:

```
for i = 1, 10 do print(i) end
```

```
    max = i       -- вероятно, неправильно! здесь 'i'
    глобальная
```

Если вам нужно значение управляющей переменной после цикла (обычно когда вы прерываете цикл), то вы должны сохранить ее значение в другой переменной:

```
-- находит значение в списке
local found = nil
for i = 1, #a do
  if a[i] < 0 then
    found = i          -- сохраняет значение 'i'
    break
  end
end
print(found)
```

В-третьих, вы никогда не должны изменять значение управляющей переменной: эффект подобных изменений не предсказуем. Если вы хотите закончить цикл **for** до его нормального завершения, используйте **break** (как в предыдущем примере).

Общий for

Общий **for** обходит все значения, возвращаемые итерирующей функцией:

```
-- печатает все значения таблицы 't'
for k, v in pairs(t) do print(k, v) end
```

Этот пример использует `pairs`, удобную итерирующую функцию для перебора таблицы, предоставляемую базовой библиотекой Lua. На каждом шаге этого цикла `k` получает ключ, а `v` получает значение, связанное с этим ключом.

Несмотря на свою кажущуюся простоту, общий **for** обладает широкими возможностями. С подходящими итераторами вы можете в легкочитаемой форме перебрать практически все, что угодно.

Стандартные библиотеки предоставляют несколько итераторов, позволяющих нам перебирать строки файла (`io.lines`), пары таблицы (`pairs`), элементы последовательности (`ipars`), слова внутри строки (`string.gmatch`) и т.д.

Конечно, мы можем написать и наши собственные итераторы. Хотя использовать общий **for** легко, у задачи написания итерирующих функций есть свои нюансы, поэтому мы рассмотрим эту тему позже, в главе 7.

У общего и числового циклов есть два одинаковых свойства: переменные цикла локальны для тела цикла, и вы никогда не должны присваивать им какие-либо значения.

Рассмотрим более конкретный пример применения общего **for**. Допустим, у вас есть таблица с названиями дней недели:

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"}
```

Теперь вы хотите перевести название дня в его номер по счету в неделе. Вы можете искать заданное имя в таблице. Однако, как вы скоро узнаете, в Lua поиск применяется редко. Более эффективным подходом является построение обратной таблицы, скажем, `revDays`, в которой названия дней используются как индексы, а номера используются как значения. Эта таблица выглядела бы следующим образом:

```
revDays = {["Sunday"]   = 1, ["Monday"]    = 2,
           ["Tuesday"]  = 3, ["Wednesday"] = 4,
           ["Thursday"] = 5, ["Friday"]    = 6,
           ["Saturday"] = 7}
```

Тогда все, что вам нужно, чтобы найти номер дня по названию, — это обратиться по индексу этой обратной таблицы:

```
x = "Tuesday"
print(revDays[x])    --> 3
```

Разумеется, нам не требуется объявлять эту обратную таблицу

вручную. Мы можем автоматически построить ее из первоначальной:

```
revDays = {}
for k,v in pairs(days) do
  revDays[v] = k
end
```

Этот цикл выполнит присваивание для каждого элемента `days`, где переменная `k` получит ключ (`1, 2, ...`), а `v` получит значение (`"Sunday", "Monday", ...`).

4.4. break, return и goto

Операторы **break** и **return** позволяют нам выпрыгнуть (осуществить безусловный переход) из блока. Оператор **goto** позволяет нам прыгнуть практически в любую точку функции.

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

Оператор **return** возвращает результаты из функции, если они есть, или просто завершает ее. В конце каждой функции присутствует неявный возврат из нее, поэтому вам необязательно его использовать, если ваша функция завершается естественным образом, не возвращая никакого значения.

Из синтаксических соображений оператор **return** может быть только последним оператором блока: другими словами, либо последним оператором в вашем куске, либо прямо перед **end**, **else** или **until**. В следующем примере **return** — это последний оператор блока **then**.

```
local i = 1
while a[i] do
```

```
    if a[i] == v then return i end
    i = i + 1
  end
```

Обычно это именно те места, где мы используем **return**, поскольку любые другие операторы, следующие за ним, были бы недоступны. Иногда, тем не менее, удобно писать **return** в середине блока; например, вы можете отлаживать функцию и хотите избежать ее выполнения. В подобных случаях вы можете использовать явный блок **do** вокруг оператора **return**:

```
function foo ()
  return                --<< СИНТАКСИЧЕСКАЯ ОШИБКА
  -- 'return' является последним оператором в
следующем куске
  do return end         -- OK
  <другие операторы>
end
```

Оператор **goto** передает выполнение программы соответствующей метке. Насчет **goto** были долгие обсуждения, и некоторые люди даже сейчас считают, что эти операторы вредны для языков программирования и должны быть из них исключены. Однако, ряд существующих языков предлагает **goto** совершенно обоснованно. Данные операторы являются мощным механизмом, который, будучи использованным с осторожностью, способен лишь улучшить качество вашего кода.

В Lua синтаксис для оператора **goto** вполне традиционный: это зарезервированное слово **goto**, за которым следует имя метки, которое может быть любым допустимым идентификатором. Синтаксис для метки немного более запутанный: два двоеточия, за которыми следует имя метки, после которого идут еще два двоеточия, например, как в `::name::`. Эта сложность намеренная, чтобы заставить программиста дважды подумать, прежде чем использовать **goto**.

Lua накладывает некоторые ограничения на то, куда вы можете

прыгнуть при помощи **goto**. Во-первых, метки следуют обычным правилам видимости, поэтому вы не можете прыгнуть внутрь блока (поскольку метка внутри блока невидима за его пределами). Во-вторых, вы не можете выпрыгнуть из функции. (Обратите внимание, что первое правило уже исключает возможность прыгнуть внутрь функции.) В-третьих, вы не можете прыгнуть внутрь области видимости локальной переменной.

Типичным и хорошо зарекомендовавшим себя применением **goto** является моделирование некоторой конструкции, которую вы узнали из другого языка, но которая отсутствует в Lua, например continue, многоуровневые break, redo и т. п. Оператор continue — это просто безусловный переход к метке в конце блока цикла; оператор redo осуществляет переход к началу блока:

```
while some_condition do
  ::redo::
  if some_other_condition then goto continue
  else if yet_another_condition then goto redo
  end
  <какой-нибудь код>
  ::continue::
end
```

Полезной деталью в спецификации Lua является то, что область видимости локальной переменной заканчивается на первом *непустом* операторе блока, в котором эта переменная определена; метки считаются пустыми операторами. Чтобы увидеть полезность данной детали, рассмотрим следующий фрагмент кода:

```
while some_condition do
  if some_other_condition then goto continue end
  local var = something
  <какой-нибудь код>
  ::continue::
end
```

Вы можете подумать, что этот **goto** перепрыгивает в область

видимости переменной `var`. Однако, метка `continue` находится после последнего непустого оператора блока, и поэтому не в области видимости переменной `var`.

Оператор **goto** также полезен при написании конечных автоматов (машин состояний). В качестве примера листинг 4.1 показывает программу, проверяющую, содержит ли ее ввод четное количество нулей. Существуют лучшие способы написания этой специфической программы, но данный подход удобен, если вы хотите автоматически перевести конечный автомат в код Lua (подумайте о динамической генерации кода).

Листинг 4.1. Пример конечного автомата с использованием **goto**

```
::s1:: do
  local c = io.read(1)
  if c == '0' then goto s2
  elseif c == nil then print'ok'; return
  else goto s1
  end
end

::s2:: do
  local c = io.read(1)
  if c == '0' then goto s1
  elseif c == nil then print'not ok'; return
  else goto s2
  end
end

goto s1
```

В качестве другого примера рассмотрим простую игру в лабиринт. Лабиринт содержит несколько комнат, в каждой до четырех дверей: север, юг, восток и запад. На каждом шаге пользователь вводит направление движения. Если в этом направлении есть дверь, то пользователь переходит в соответствующую комнату; иначе программа печатает

предупреждение. Целью является дойти от начальной комнаты до конечной.

Эта игра является типичной машиной состояний, где текущая комната является состоянием. Мы можем реализовать этот лабиринт, используя один блок для каждой комнаты и оператор **goto** для перехода из одной комнаты в другую. Листинг 4.2 показывает, как можно написать небольшой лабиринт с четыремя комнатами.

Для этой простой игры вы можете решить, что программа, управляемая данными, в которой вы описываете комнаты и перемещения при помощи таблиц, является более удачным решением. Однако, если в игре для каждой комнаты предусмотрены особые ситуации, то этот подход на основе конечного автомата является вполне уместным.

Листинг 4.2. Игра «лабиринт»

```
goto room1 -- начальная комната

::room1:: do
  local move = io.read()
  if move == "south" then goto room3
  elseif move == "east" then goto room2
  else
    print("invalid move")
    goto room1 -- остаемся в этой же комнате
  end
end

::room2:: do
  local move = io.read()
  if move == "south" then goto room4
  elseif move == "west" then goto room1
  else
    print("invalid move")
    goto room2
  end
end
```

```
::room3:: do
  local move = io.read()
  if move == "north" then goto room1
  elseif move == "east" then goto room4
  else
    print("invalid move")
    goto room3
  end
end

::room4:: do
  print("Congratulations, you won!")
end
```

Упражнения

Упражнение 4.1. Большинство языков с C-образным синтаксисом не предлагает конструкцию **elseif**. Почему эта конструкция больше нужна в Lua, чем в тех языках?

Упражнение 4.2. Опишите четыре различных способа написать безусловный цикл в Lua. Какой способ предпочитаете вы?

Упражнение 4.3. Многие считают, что **repeat—until** используется редко и потому не должен присутствовать в минималистических языках вроде Lua. Чты вы думаете об этом?

Упражнение 4.4. Перепишите конечный автомат из листинга 4.2 без использования **goto**.

Упражнение 4.5. Можете ли вы объяснить, почему в Lua присутствует ограничение на то, что **goto** не может выпрыгнуть из функции? (Подсказка: как бы вы реализовали данную возможность?)

Упражнение 4.6. Допустив, что **goto** может выпрыгнуть из функции, объясните, что делала бы программа в листинге 4.3. (Попытайтесь рассуждать о метке при помощи тех же правил области видимости, что используются для локальных переменных.)

Листинг 4.3. Странное (и недопустимое) использование **goto**

```
function getlabel ()
  return function () goto L1 end
  ::L1::
  return 0
end

function f (n)
  if n == 0 then return getlabel()
  else
    local res = f(n - 1)
    print(n)
    return res
  end
end

x = f(10)
x()
```

ГЛАВА 5

Функции

Функции являются основным механизмом абстракции операторов и выражений в Lua. Функции могут выполнять определенное задание (иногда называемое *процедурой* или *подпрограммой* в других языках) или вычислять и возвращать значения. В первом случае мы используем вызов функции как оператор; во втором случае мы используем его как выражение:

```
print(8*9, 9/8)
a = math.sin(3) + math.cos(10)
print(os.date())
```

В обоих случаях заключение списка аргументов в круглые скобки обозначает вызов; если у вызова функции нет аргументов, то мы все равно должны написать пустой список `()` для обозначения вызова. Существует особое исключение из этого правила: если у функции всего один аргумент и этот аргумент либо строковый литерал, либо конструктор таблицы, то круглые скобки необязательны:

```
print "Hello World"     <-->    print("Hello World")
dofile 'a.lua'          <-->    dofile ('a.lua')
print [[a multi-line    <-->    print([[a multi-line
 message]]              <-->     message]])
f{x=10, y=20}           <-->    f({x=10, y=20})
type{}                  <-->    type({})
```

Lua также предлагает специальный синтаксис для объектно-ориентированных вызовов — операцию двоеточия. Выражение вроде `o:foo(x)` — это всего лишь другой способ написать `o.foo(o,x)`, то есть вызвать `o.foo`, добавляя `o` в качестве первого дополнительного аргумента. В главе 16 мы обсудим подобные

вызовы (и объектно-ориентированное программирование) более подробно.

Программа Lua может использовать функции, написанные как на Lua, так и на C (или любом другом языке, который используется основным приложением). Например, все функции из стандартной библиотеки Lua написаны на C. Тем не менее, при вызове нет никакой разницы между функциями, определенными в Lua, и функциями, определенными в C.

Как мы видели в других примерах, определение функции следует традиционному синтаксису, например, как показано ниже:

```
-- складывает элементы последовательности 'a'
function add (a)
  local sum = 0
  for i = 1, #a do
    sum = sum + a[i]
  end
  return sum
end
```

В этом синтаксисе определение функции содержит *имя* (в примере это `add`), список *параметров* и *тело*, которое является списком операторов.

Параметры работают в точности как локальные переменные, проинициализированные значениями аргументов, которые были переданы вызову функции. Вы можете вызвать функцию с числом аргументов, отличным от ее числа параметров. Lua приведет число аргументов к числу параметров так же, как и при множественном присваивании: лишние аргументы отбрасываются, лишние параметры получают nil. Например, рассмотрим следующую функцию:

```
function f (a, b) print(a, b) end
```

Она обладает следующим поведением:

```
f(3)          --> 3    nil
```

```
f(3, 4)        --> 3    4
f(3, 4, 5)     --> 3    4    (5 отбрасывается)
```

Хотя подобное поведение может привести к ошибкам программирования (легко обнаруживаемым во время выполнения), оно довольно удобно, особенно для аргументов по умолчанию. Например, рассмотрим следующую функцию, инкрементирующую глобальный счетчик:

```
function incCount (n)
  n = n or 1
  count = count + n
end
```

У этой функции 1 служит в качестве аргумента по умолчанию; т.е. вызов `incCount()` без аргументов увеличит `count` на единицу. Когда вы вызываете `incCount()`, Lua сперва инициализирует `n` значением nil; выражение с `or` возвращает свой второй операнд, и в результате Lua присваивает переменной `n` стандартное значение 1.

5.1. Множественные результаты

Нетрадиционной, но довольно удобной особенностью Lua является то, что функции могут возвращать несколько значений. Так делают некоторые предопределенные функции в Lua. Примером является функция `string.find`, которая находит местоположение образца в строке. Эта функция возвращает два индекса, когда находит образец: индекс символа, с которого начинается образец, и индекс символа, на котором он заканчивается. Множественное присваивание позволяет программе получить оба результата:

```
s, e = string.find("hello Lua users", "Lua")
print(s, e)    --> 7    9
```

(Обратите внимание, что индекс первого символа строки равен 1.)

Функции, которые мы пишем в Lua, также могут возвращать

множественные результаты при помощи перечисления их после ключевого слова **return**. Например, функция для нахождения максимального элемента в последовательности может возвращать и максимальный элемент, и его местонахождение:

```
function maximum (a)
  local mi = 1             -- индекс максимального
значения
  local m = a[mi]          -- максимальное значение
  for i = 1, #a do
    if a[i] > m then
      mi = i; m = a[i]
    end
  end
  return m, mi
end

print(maximum({8,10,23,12,5}))    --> 23    3
```

Lua всегда приводит количество результатов функции к обстоятельствам ее вызова. Когда мы вызываем функцию как оператор, Lua отбрасывает все результаты функции. Когда мы используем вызов как выражение, Lua оставляет только первый результат. Мы получаем все результаты лишь тогда, когда вызов является последним (или единственным) выражением в списке выражений. В Lua эти списки встречаются в четырех конструкциях: множественные присваивания, аргументы в вызовах функций, конструкторы таблиц и операторы **return**. Для иллюстрации всех этих случаев допустим, что у нас есть такие определения для наших последующих примеров:

```
function foo0() end                    -- возвращает 0
значений
function foo1() return "a" end         -- возвращает 1
значение
function foo2() return "a", "b" end    -- возвращает 2
значения
```

При множественном присваивании вызов функции в качестве

последнего (или единственного) выражения произведет столько результатов, сколько у нас переменных:

```
x,y = foo2()          -- x="a", y="b"
x = foo2()            -- x="a", "b" отбрасывается
x,y,z = 10,foo2()     -- x=10, y="a", z="b"
```

Если функция не возвращает значения или возвращает их меньше, чем нужно, то Lua произведет в качестве недостающих значений nil:

```
x,y = foo0()      -- x=nil, y=nil
x,y = foo1()      -- x="a", y=nil
x,y,z = foo2()    -- x="a", y="b", z=nil
```

Вызов функции, который не является последним элементом в списке, всегда дает ровно один результат:

```
x,y = foo2(), 20         -- x="a", y=20
x,y = foo0(), 20, 30     -- x=nil, y=20, 30
отбрасывается
```

Когда вызов функции является последним (или единственным) аргументом другого вызова, то все результаты первого вызова передаются как аргументы для второго. Мы уже видели примеры этой конструкции с функцией `print`. Поскольку функция `print` может получать переменное число аргументов, оператор `print (g())` печатает все результаты, возвращенные функцией `g`.

```
print(foo0())           -->
print(foo1())           --> a
print(foo2())           --> a     b
print(foo2(), 1)        --> a     1
print(foo2() .. "x")    --> ax    (смотрите далее)
```

Когда вызов функции `foo2` происходит внутри выражения, Lua приводит число результатов к одному; поэтому в последней строке конкатенация использует только "a".

Если мы напишем `f(g(x))`, где у `f` фиксированное число аргументов, то Lua приведет число результатов `g` к числу

параметров `f`, как мы уже видели ранее.

Конструктор таблицы тоже собирает все результаты вызова, без каких-либо изменений:

```
t = {foo0()}           -- t = {} (пустая таблица)
t = {foo1()}           -- t = {"a"}
t = {foo2()}           -- t = {"a", "b"}
```

Как обычно, это поведение встречается лишь тогда, когда вызов является последним выражением в списке; вызовы в любых других местах дают ровно один результат:

```
t = {foo0(), foo2(), 4}    -- t[1] = nil, t[2] = "a",
t[3] = 4
```

Наконец, оператор наподобие `return f()` возвращает все значения, возвращаемые `f`:

```
function foo (i)
  if i == 0 then return foo0()
  elseif i == 1 then return foo1()
  elseif i == 2 then return foo2()
  end
end

print(foo(1))    --> a
print(foo(2))    --> a b
print(foo(0))    -- (без результатов)
print(foo(3))    -- (без результатов)
```

Вы можете заставить вызов вернуть ровно один результат, заключив его в дополнительную пару круглых скобок:

```
print((foo0()))    --> nil
print((foo1()))    --> a
print((foo2()))    --> a
```

Будьте внимательны, так как оператор **return** не требует скобок вокруг возвращаемого значения; любая пара круглых скобок, помещенная там, считается дополнительной. Поэтому оператор

вроде `return(f(x))` всегда возвращает ровно одно значение, вне зависимости от того, сколько значений возвращает функция `f`. Иногда это именно то, что вам нужно, иногда нет.

Специальной функцией, возвращающей несколько значений, является `table.unpack`. Она получает массив и возвращает в качестве результатов все элементы этого массива, начиная с индекса 1:

```
print(table.unpack{10,20,30})     --> 10 20 30
a,b = table.unpack{10,20,30}      -- a=10, b=20, 30
отбрасывается
```

Одна из важных областей применения `unpack` — механизм общего вызова. Механизм общего вызова позволяет вам динамически вызвать любую функцию с любыми аргументами. В ANSI C, например, нет способа написать код общего вызова. Вы можете объявить функцию, которая получает переменное число аргументов (при помощи `stdarg.h`), и вы можете вызвать функцию с несколькими переменными, используя указатели на функции. Однако, вы не можете вызвать функцию с переменным числом аргументов: у каждого вызова, который вы пишете на C, есть фиксированное число аргументов, а каждый аргумент обладает фиксированным типом. В Lua, если вы хотите вызвать функцию `f` с переменным числом аргументов, хранящимся в массиве `a`, вы просто пишите так:

```
f(table.unpack(a))
```

Вызов `unpack` возвращает все значения из `a`, которые становятся аргументами вызова `f`. Например, рассмотрим следующий вызов:

```
print(string.find("hello", "ll"))
```

Вы можете динамически построить эквивалентный вызов при помощи следующего кода:

```
f = string.find
```

```
a = {"hello", "ll"}

print(f(table.unpack(a)))
```

Обычно `unpack` использует операцию длины, чтобы узнать, сколько элементов следует вернуть, поэтому она работает только с правильными последовательностями. При необходимости вы можете задать для нее явные границы:

```
print(table.unpack({"Sun", "Mon", "Tue", "Wed"}, 2,
3))
  --> Mon    Tue
```

Хотя предопределенная функция `unpack` написана на C, мы могли бы написать ее на Lua, используя рекурсию:

```
function unpack (t, i, n)
  i = i or 1
  n = n or #t
  if i <= n then
    return t[i], unpack(t, i + 1, n)
  end
end
```

Первый раз, когда мы вызываем ее с единственным аргументом, `i` получает 1, а `n` получает длину последовательности. Затем функция возвращает `t[1]` вместе со всеми результатами `unpack(t,2,n)`, что, в свою очередь, возвращает `t[2]` со всеми результаты `unpack(t,3,n)` и т. д., останавливаясь после `n` элементов.

5.2. Вариадические функции

Функция в Lua может быть вариадической (variadic), т.е. иметь переменное число аргументов. Например, мы уже вызывали `print` с одним, с двумя и более чем с двумя аргументами. Хотя `print` определена в C, мы можем определять вариадические функции и в Lua.

В качестве простого примера следующая функция возвращает сумму всех своих аргументов:

```
function add (...)
  local s = 0
  for i, v in ipairs{...} do
    s = s + v
  end
  return s
end

print(add(3, 4, 10, 25, 12))    --> 54
```

Три точки (`...`) в списке параметров указывают на то, что эта функция является вариадической. При вызове этой функции Lua внутренним образом собирает все ее аргументы; мы называем эти собранные аргументы *дополнительными аргументами* функции. Функция может получать доступ к своим дополнительным аргументам опять же при помощи трех точек, теперь уже в качестве выражения. В нашем примере выражение `{...}` производит массив со всеми собранными аргументами. Затем функция обходит массив для сложения его элементов.

Мы называем выражение с `...` *выражением с переменным числом аргументов* (vararg expression). Оно ведет себя как функция с возвратом нескольких значений, возвращая все дополнительные аргументы текущей функции. Например, команда `print(...)` напечатает все дополнительные аргументы текущей функции. Аналогично, следующая команда создаст две локальные переменные со значениями первых двух необязательных аргументов (или nil, если таких аргументов нет).

```
local a, b = ...
```

На самом деле мы можем имитировать стандартный механизм передачи параметров в Lua, переводя

```
function foo (a, b, c)
```

в

```
function foo (...)
local a, b, c = ...
```

Тем, кому нравится изящный механизм передачи параметров в Perl, может понравиться эта вторая форма.

Функция, наподобие следующей, просто возвращает все свои аргументы при вызове:

```
function id (...) return ... end
```

Это многозначная функция тождества. Следующая функция ведет себя так же, как и другая функция `foo`, за исключением того, что перед вызовом она печатает сообщение со своими аргументами:

```
function foo1 (...)
  print("calling foo:", ...)
  return foo(...)
end
```

Это полезный трюк для отслеживания вызовов конкретной функции.

Давайте рассмотрим еще один полезный пример. Lua предоставляет отдельные функции для форматирования текста (`string.format`) и его записи (`io.write`). Несложно объединить обе функции в одну вариадическую:

```
function fwrite (fmt, ...)
  return io.write(string.format(fmt, ...))
end
```

Обратите внимание на присутствие фиксированного параметра `fmt` перед точками. Вариадические функции могут иметь любое количество фиксированных параметров перед своей вариадической частью. Lua присваивает этим параметрам первые аргументы; остальные (если есть) идут как дополнительные параметры. Ниже мы покажем несколько примеров вызовов и значения

соответствующих параметров:

```
ВЫЗОВ                    ПАРАМЕТРЫ

fwrite()                 fmt = nil, без дополнительных
аргументов
fwrite("a")              fmt = "a", без дополнительных
аргументов
fwrite("%d%d", 4, 5)     fmt = "%d%d", доп. аргументы =
4 и 5
```

(Обратите внимание, что вызов `fwrite()` приведет к ошибке, поскольку `string.format` требует строку в качестве своего первого аргумента.)

Для обхода своих дополнительных аргументов функция может использовать выражение `{...}`, чтобы собрать их всех в таблицу, как мы это сделали в нашем определении `add`.

Однако, в тех редких случаях, когда дополнительные аргументы могут быть допустимыми nil, таблица, созданная при помощи `{...}`, может не быть правильной последовательностью. Например, для такой таблицы не существует способа узнать, были ли конечные nil в исходных аргументах. Для этих случаев Lua предлагает функцию `table.pack` (Примечание: Эта функция появилась только в Lua 5.2). Эта функция получает произвольное число аргументов и возвращает новую таблицу со всеми своими аргументами, как и `{...}`, но в этой таблице будет дополнительное поле '`n`', содержащее полное число ее аргументов. Следующая функция использует `table.pack`, чтобы проверить, что среди ее аргументов нет nil:

```
function nonils (...)
  local arg = table.pack(...)
  for i = 1, arg.n do
    if arg[i] == nil then return false end
  end
  return true
end

print(nonils(2,3,nil))     --> false
```

```
print(nonils(2,3))          --> true
print(nonils())             --> true
print(nonils(nil))          --> false
```

Однако, не забывайте, что `{...}` быстрее и понятнее, чем `table.pack(...)`, когда среди дополнительных аргументов не может быть nil.

5.3. Именованные аргументы

Механизм передачи параметров в Lua является *позиционным*: при вызове функции аргументы сопоставляются с параметрами соответственно их позициям. Первый аргумент дает значение первому параметру и т. д. Тем не менее, иногда удобно задавать аргументы по имени. Чтобы проиллюстрировать данный момент, давайте рассмотрим функцию `os.rename` (из библиотеки `os`), которая переименовывает файл. Довольно часто мы забываем, какое имя идет первым, новое или старое; поэтому мы можем захотеть переопределить эту функцию так, чтобы она получала два именованных аргумента:

```
-- недопустимый код
rename(old="temp.lua", new="temp1.lua")
```

В Lua нет непосредственной поддержки этого синтаксиса, но мы можем добиться такого же итогового эффекта путем небольшого синтаксического изменения. Идея заключается в упаковке всех аргументов в таблицу и использовании этой таблицы в качестве единственного аргумента функции. Специальный синтаксис, который Lua предоставляет для вызовов функций, с единственным конструктором таблицы в качестве аргумента, поможет нам с этим приемом:

```
rename{old="temp.lua", new="temp1.lua"}
```

Соответственно, мы переопределяем функцию `rename` только с одним параметром и получаем действительные аргументы из этого параметра:

```
function rename (arg)
  return os.rename(arg.old, arg.new)
end
```

Этот способ передачи параметров особенно удобен, когда у функции много аргументов и большинство из них необязательные. Например, функция, которая создает новое окно в библиотеке GUI, может иметь десятки аргументов, по большей части необязательных, которые лучше задавать при помощи имен:

```
w = Window{ x=0, y=0, width=300, height=200,
            title = "Lua", background="blue",
            border = true
          }
```

Функция `Window` затем вольна проверять обязательные аргументы, добавлять значения по умолчанию и т.п. Предположив, что у нас есть примитивная функция `_Window`, которая на самом деле создает новое окно (и которой все аргументы требуются в определенном порядке), мы могли бы определить `Window` как показано в листинге 5.1.

Листинг 5.1. Функция с именованными необязательными параметрами

```
function Window (options)
  -- проверка обязательных опций
  if type(options.title) ~= "string" then
    error("no title")
  elseif type(options.width) ~= "number" then
    error("no width")
  elseif type(options.height) ~= "number" then
    error("no height")
  end
```

```
  -- everything else is optional
  _Window(options.title,
          options.x or 0,       -- значение по
умолчанию
          options.y or 0,       -- значение по
умолчанию
          options.width, options.height,
          options.background or "white",    -- по
умолчанию
          options.border        -- по умолчанию false
(nil)
        )
end
```

Упражнения

Упражнение 5.1. Напишите функцию, которая получает произвольное число строк и возвращает их соединенными вместе.

Упражнение 5.2. Напишите функцию, которая получает массив и печатает все элементы этого массива. Рассмотрите преимущества и недостатки использования `table.unpack` в этой функции.

Упражнение 5.3. Напишите функцию, которая получает произвольное число значений и возвращает их все, кроме первого.

Упражнение 5.4. Напишите функцию, которая получает массив и печатает все комбинации элементов этого массива. (Подсказка: вы можете использовать рекурсивную формулу для получения комбинаций: $C\ (n,\ m) = C\ (n-1,\ m-1) + C\ (n-1,\ m)$. Для получения всех $C(n,\ m)$ комбинаций из n элементов в группах размера m вы сперва добавляете первый элемент к результату и затем генерируете все комбинации $C(n-1,\ m-1)$ из оставшихся элементов на оставшихся позициях. Когда n меньше, чем m, комбинаций больше нет. Когда m равно нулю, существует только одна комбинация, и она не использует никаких элементов.)

ГЛАВА 6

Еще раз о функциях

Функции в Lua являются значениями первого класса с соответствующей лексической областью видимости.

Что для функций означает быть «значениями первого класса»? Это значит, что в Lua функция — это значение, обладающее теми же правами, что и традиционные значения вроде чисел и строк. Мы можем хранить функции в переменных (локальных и глобальных) и таблицах, мы можем передавать функции как аргументы и возвращать их из других функций.

Что для функций означает иметь «лексическую область видимости»? Это значит, что функции могут обращаться к переменным окружающих их функций (Примечание: Это также значит, что Lua содержит в себе полноценное лямбда-исчисление). Как мы увидим в этой главе, это на первый взгляд безобидное свойство дает огромную мощь языку, поскольку позволяет нам применять в Lua многие эффективные приемы из мира функционального программирования. Даже если вы совсем не интересуетесь функциональным программированием, все равно стоит немного узнать об этих приемах, так как они могут сделать ваши программы меньше и проще.

Несколько запутанным понятием в Lua является то, что функции, как и другие значения, являются анонимными; у них нет имен. Когда мы говорим об имени функции, такой как `print`, на самом деле мы имеем в виду переменную, которая хранит данную функцию. Как и с любой другой переменной, хранящей любое другое значение, мы можем манипулировать этими переменными множеством способов. Следующий пример, хоть и немного нелепый, показывает этот момент:

```
a = {p = print}
a.p("Hello World")     --> Hello World
print = math.sin       -- 'print' теперь ссылается на
функцию sin
a.p(print(1))          --> 0.841470
sin = a.p              -- 'sin' теперь ссылается на
функцию print
sin(10, 20)            --> 10 20
```

(Позже мы увидим полезные применения этой возможности.)

Если функции являются значениями, то существуют ли выражения, которые создают функции? Да. По сути, стандартный способ написать функцию в Lua, такой как

```
function foo (x) return 2*x end
```

является всего лишь образцом того, что мы называем *синтаксическим сахаром*; это просто более красивый способ написать следующий код:

```
foo = function (x) return 2*x end
```

Таким образом, определение функции — это по сути оператор (присваивание, если более точно), который создает значение типа "`function`" и присваивает его переменной. Мы можем рассматривать выражение `function (x)` *тело* `end` как конструктор функции, точно так же, как `{}` является конструктором таблицы. Мы называем результат подобных конструкторов функций *анонимной функцией*. Хотя мы часто присваиваем функции глобальным переменным, задавая им что-то вроде имени, бывают случаи, когда функции остаются анонимными. Давайте рассмотрим несколько примеров.

Табличная библиотека предоставляет функцию `table.sort`, которая получает таблицу и сортирует ее элементы. Подобная функция должна позволять бесконечные вариации порядка сортировки: по возрастанию или по убыванию, числовой или алфавитный, таблицы с сортировкой по ключу и т. д. Вместо

попытки предоставить все виды опций, `sort` предоставляет единственный необязательный параметр, который является *порядковой функцией*: функция, которая принимает два элемента и определяет, должен ли первый элемент идти перед вторым в отсортированном списке. Например, допустим, что у нас есть такая таблица записей:

```
network = {
  {name = "grauna",  IP = "210.26.30.34"},
  {name = "arraial", IP = "210.26.30.23"},
  {name = "lua",     IP = "210.26.23.12"},
  {name = "derain",  IP = "210.26.23.20"},
}
```

Если нам нужно отсортировать таблицу по полю `name` в обратном алфавитном порядке, то достаточно написать так:

```
table.sort(network, function (a,b) return (a.name >
b.name) end)
```

Посмотрите, насколько удобна анонимная функция в этом операторе.

Функция, которая получает другую функцию как аргумент, такая как `sort`, является тем, что мы называем *функцией высшего порядка*. Функции высшего порядка являются эффективным механизмом программирования, а использование анонимных функций в качестве их аргументов служит колоссальным источником гибкости. Однако, запомните, что функции высших порядков не являются чем-то особенным; они — прямое следствие способности Lua работать с функциями как со значениями первого класса.

Чтобы проиллюстрировать применение функций высших порядков, мы напишем упрощенное определение распространенной функции высшего порядка — производной. Следуя неформальному определению, производная функции f в точке x — это значение $(f(x + d) - f(x))/d$, где d стремится к бесконечно малой величине. В соответствии с этим определением мы можем вычислить

приближенное значение производной следующим образом:

```
function derivative (f, delta)
  delta = delta or 1e-4
  return function (x)
    return (f(x + delta) - f(x))/delta
  end
end
```

Если задать функцию `f`, вызов `derivative(f)` вернет ее производную (как приближенное значение), которая является еще одной функцией:

```
c = derivative(math.sin)
> print(math.cos(5.2), c(5.2))
  -->  0.46851667130038     0.46856084325086
print(math.cos(10), c(10))
  --> -0.83907152907645    -0.83904432662041
```

Поскольку функции в Lua являются значениями первого класса, мы можем хранить их не только в глобальных переменных, но и в локальных переменных и полях таблиц. Как мы увидим в дальнейшем, использование функций в полях таблицы является ключевым компонентом для некоторых продвинутых областей применения Lua, таких как модули и объектно-ориентированное программирование.

6.1. Замыкания

Когда мы определяем одну функцию внутри другой, она получает полный доступ к локальным переменным окружающей ее функции; мы называем это свойство *лексической областью видимости* (lexical scoping). Хотя это правило видимости может показаться очевидным, на самом деле это не так. Лексическая область видимости вместе с функциями первого класса является высокоэффективной концепцией в языках программирования, но

многие языки ее не поддерживают.

Давайте начнем с простого примера. Пусть у вас есть список имен студентов и таблица, которая ассоциирует имена с оценками; вам требуется отсортировать список имен по их оценкам, начиная с более высоких. Вы можете добиться этого следующим образом:

```
names = {"Peter", "Paul", "Mary"}
grades = {Mary = 10, Paul = 7, Peter = 8}
table.sort(names, function (n1, n2)
  return grades[n1] > grades[n2]           --
сравнивает оценки
end)
```

Теперь допустим, что вам нужно создать функцию для решения данной задачи:

```
function sortbygrade (names, grades)
  table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2]         --
сравнивает оценки
  end)
end
```

Интересным моментом в этом примере является то, что анонимная функция, переданная функции `sort`, обращается к параметру `grades`, который является локальным для окружающей его функции `sortbygrade`. Внутри этой анонимной функции `grades` не является ни глобальной переменной, ни локальной переменной, а является тем, что мы называем *нелокальной переменной*. (По историческим причинам нелокальные переменные в Lua также называются *верхними значениями* (upvalue).)

Почему данный момент так интересен? Потому что функции являются значениями первого класса, и вследствие этого они могут *выйти* из начальной области видимости своих переменных. Рассмотрим следующий пример:

```
function newCounter ()
  local i = 0
```

```
  return function ()    -- анонимная функция
    i = i + 1
    return i
  end
end

c1 = newCounter()
print(c1())    --> 1
print(c1())    --> 2
```

В этом коде анонимная функция ссылается на нелокальную переменную `i` для хранения своего счетчика. Однако, к тому времени, как мы вызовем анонимную функцию, переменная `i` уже выйдет из своей области видимости, поскольку функция, которая создала эту переменную (`newCounter`), уже будет возвращена. Тем не менее, Lua правильно обрабатывает эту ситуацию, используя концепцию *замыкания* (closure). Проще говоря, замыкание — это функция вместе со всем, что ей нужно для правильного доступа к нелокальным переменным. Если мы снова вызовем `newCounter`, то она создаст новую локальную переменную `i`, поэтому мы получим новое замыкание, действующее поверх новой переменной:

```
c2 = newCounter()
print(c2()) --> 1
print(c1()) --> 3
print(c2()) --> 2
```

Таким образом, `c1` и `c2` — это разные замыкания поверх одной и той же функции, и каждое замыкание воздействует на независимый экземпляр локальной переменной `i`.

С технической точки зрения, значением в Lua является замыкание, а не функция. Сама по себе функция — это лишь прототип для замыканий. Тем не менее, мы продолжим использовать термин «функция» для обозначения замыкания везде, где это не приведет к путанице.

Во многих случаях замыкания оказываются ценным инструментом. Как мы уже видели, они удобны в качестве

аргументов функций высших порядков, таких как `sort`. Замыкания также ценны для функций, которые строят другие функции, как в нашем примере с `newCounter` или с производной; этот механизм позволяет программам Lua внедрять современные методы из мира функционального программирования. Замыкания также удобны для функций *обратного вызова* (callback). Здесь подходит типичный пример про создание кнопок в традиционном инструментарии GUI. У каждой кнопки есть функция обратного вызова, которая должна быть вызвана, когда пользователь нажимает эту кнопку; вам требуется, чтобы разные кнопки совершали немного разные действия при нажатии. Например, цифровому калькулятору нужно десять похожих кнопок, по одной на каждую цифру. Вы можете создать каждую кнопку с помощью функции наподобие этой:

```
function digitButton (digit)
  return Button{ label = tostring(digit),
                 action = function ()
                   add_to_display(digit)
                 end
               }
end
```

В данном примере мы предполагаем, что `Button` — это функция из инструментария, которая создает новые кнопки; `label` — это метка кнопки; `action` — это замыкание обратного вызова, которое нужно вызвать при нажатии кнопки. Обратный вызов может быть произведен спустя длительное время после того, как `digitButton` выполнила свою задачу, и после того, как локальная переменная `digit` вышла из области видимости, так как он по-прежнему может обращаться к этой переменной.

Замыкания также ценны в совсем другом случае. Поскольку функции Lua хранятся в обычных переменных, мы можем их легко переопределять, даже если они встроенные. Эта возможность является одной из причин, почему Lua столь гибок. Часто, когда вы переопределяете функцию, в новой реализации вам все равно нужна

изначальная функция. Например, предположим, что вы хотите переопределить функцию `sin`, чтобы она работала с градусами вместо радиан. Эта новая функция конвертирует свой аргумент и затем вызывает исходную функцию `sin` для выполнения настоящей работы. Ваш код при этом может выглядеть следующим образом:

```
oldSin = math.sin
math.sin = function (x)
  return oldSin(x*math.pi/180)
end
```

Далее приведен немного более аккуратный способ выполнить это переопределение:

```
do
  local oldSin = math.sin
  local k = math.pi/180
  math.sin = function (x)
    return oldSin(x*k)
  end
end
```

Теперь мы сохраняем старую версию в локальной переменной; единственный способ обратиться к ней — через новую функцию.

Вы можете воспользоваться этим подходом и для создания безопасных окружений, также называемых *песочницами* (sandbox). Безопасные окружения крайне важны при выполнении ненадежного кода, например, полученного сервером через Интернет. Скажем, чтобы ограничить файлы, к которым программа может обратиться, мы можем переопределить функцию `io.open`, используя замыкания:

```
do
  local oldOpen = io.open
  local access_OK = function (filename, mode)
    <проверка доступа>
  end
  io.open = function (filename, mode)
    if access_OK(filename, mode) then
```

```
        return oldOpen(filename, mode)
      else
        return nil, "access denied"
      end
    end
  end
```

Что делает этот пример особенно удачным, так это то, что после данного переопределения для программы нет иного способа вызвать неограниченную функцию `open`, кроме как через новую версию с ограничениями. При этом небезопасная версия хранится внутри замыкания как закрытая переменная без внешнего доступа. С этим подходом вы можете строить песочницы Lua на самом Lua с его обычными преимуществами: простотой и гибкостью. Вместо какого-то универсального решения Lua предлагает мета-механизм, чтобы вы могли подогнать ваше окружение под ваши конкретные требования к безопасности.

6.2. Неглобальные функции

Очевидным следствием того, что функции являются значениями первого класса, является то, что мы можем хранить функции не только в глобальных переменных, но и в локальных переменных и полях таблицы.

Мы уже видели некоторые примеры функций внутри полей таблиц: большинство библиотек Lua использует этот механизм (например, `io.read`, `math.sin`). Для создания подобных функций в Lua нам нужно просто соединить стандартный синтаксис для функций с синтаксисом для таблиц:

```
Lib = {}
Lib.foo = function (x,y) return x + y end
Lib.goo = function (x,y) return x - y end

print(Lib.foo(2, 3), Lib.goo(2, 3)) --> 5 -1
```

Разумеется, мы также можем использовать конструкторы:

```
Lib = {
  foo = function (x,y) return x + y end,
  goo = function (x,y) return x - y end
}
```

Кроме того, Lua предлагает еще один синтаксис для определения подобных функций:

```
Lib = {}
function Lib.foo (x,y) return x + y end
function Lib.goo (x,y) return x - y end
```

Когда мы сохраняем функцию в локальной переменной, мы получаем *локальную функцию*, то есть функцию с ограниченной областью видимости. Подобные определения особенно удобны для пакетов: поскольку Lua рассматривает каждый кусок как функцию, кусок может объявлять локальные функции, которые видны только внутри него. Лексическая область видимости позволяет другим функциям из пакета использовать эти локальные функции:

```
local f = function (<параметры>)
  <тело>
end
local g = function (<параметры>)
  <какой-нибудь код>
  f()           -- 'f' здесь видима
  <какой-нибудь код>
end
```

Lua поощряет подобное применение локальных функций посредством синтаксического сахара:

```
local function f (<params>)
  <тело>
end
```

При определении рекурсивных локальных функций возникает

требующий уточнения момент. Дело в том, что обычный подход здесь не работает. Рассмотрим следующее определение:

```
local fact = function (n)
  if n == 0 then return 1
  else return n*fact(n-1)    -- глюк
  end
end
```

Когда Lua компилирует вызов `fact(n–1)` в теле функции, локальная функция `fact` еще не определена. Поэтому данное выражение попытается вызвать глобальную функцию `fact`, а не локальную. Мы можем решить эту проблему, сперва определив локальную переменную, а затем уже саму функцию:

```
local fact
fact = function (n)
  if n == 0 then return 1
  else return n*fact(n-1)
  end
end
```

Теперь `fact` внутри функции ссылается на локальную переменную. Ее значение при определении функции не важно; к моменту выполнения функции, `fact` уже получит правильное значение.

Когда Lua предлагает свой синтаксический сахар для локальных функций, он не использует простое определение. Вместо этого определение наподобие

```
local function foo (<параметры>) <тело> end
```

расширяется до

```
local foo; foo = function (<параметры>) <тело> end
```

Поэтому мы можем спокойно использовать этот синтаксис для рекурсивных функций.

Конечно, этот прием не сработает, если у вас косвенно рекурсивные функции (поочередно вызывающие друг друга). В таких случаях вы должны использовать эквивалент явного предварительного объявления:

```
local f, g    -- предварительные объявления

function g ()
  <какой-нибудь код> f() <какой-нибудь код>
end

function f ()
  <какой-нибудь код> g() <какой-нибудь код>
end
```

Остерегайтесь писать `local function f` в последнем определении. Иначе Lua создаст новую локальную переменную `f`, оставив изначальную `f` (к которой привязана `g`) неопределенной.

6.3. Корректные хвостовые вызовы

Другой интересной особенностью функций в Lua является то, что Lua выполняет устранение хвостовых вызовов. (Это значит, что Lua поддерживает *корректную хвостовую рекурсию*, хотя данное понятие не связано непосредственно с рекурсией; см. упражнение 6.3.)

Хвостовой вызов (tail call) — это фактически **goto**, выглядящий как вызов функции. Хвостовой вызов происходит, когда одна функция вызывает другую в качестве своего последнего действия, и потому ей больше нечего делать. Например, в следующем коде вызов функции `g` является хвостовым:

```
function f (x) return g(x) end
```

После того, как `f` вызовет `g`, ей больше нечего делать. В подобных ситуациях программе не требуется возвращаться в

вызывающую функцию по завершении вызванной функции. Поэтому после хвостового вызова программе не нужно хранить какую-либо информацию о вызывающей функции в стеке. Когда `g` возвращает управление, оно может непосредственно перейти к моменту вызова `f`. Некоторые реализации языков, например, интерпретатор Lua, извлекают из данного факта выгоду и на самом деле не используют какое-либо дополнительное место в стеке при совершении хвостового вызова. Мы говорим, что эти реализации поддерживают *устранение хвостовых вызовов* (tail-call elimination).

Поскольку хвостовые вызовы не используют место в стеке, количество вложенных хвостовых вызовов, которое программа может выполнить, не ограничено. Скажем, мы можем вызвать следующую функцию, передав любое число в качестве аргумента:

```
function foo (n)
  if n > 0 then return foo(n - 1) end
end
```

Этот вызов никогда не приведет к переполнению стека.

Когда мы прибегаем к устранению хвостовых вызовов, возникает тонкий вопрос: какой именно вызов считать хвостовым? Некоторые вполне очевидные кандидаты не соответствуют требованию о том, что вызывающей функции больше нечего делать после вызова. Например, в следующем коде вызов `g` не является хвостовым:

```
function f (x) g(x) end
```

Проблема в этом примере состоит в том, что после вызова `g` функция `f` должна отбросить результаты `g` перед возвратом. Все следующие вызовы также не удовлетворяют данному требованию:

```
return g(x) + 1      -- необходимо выполнить сложение
return x or g(x)     -- необходимо привести к одному
значению
return (g(x))        -- необходимо привести к одному
значению
```

В Lua хвостовым считается лишь вызов вида `return` *функция(аргументы)*. Однако, и *функция*, и ее *аргументы* могут быть сложными выражениями, поскольку Lua вычислит их перед вызовом. Например, следующий вызов является хвостовым:

```
return x[i].foo(x[j] + a*b, i + j)
```

Упражнения

Упражнение 6.1. Напишите функцию `integral`, которая принимает функцию `f` и возвращает приближенное значение ее интеграла.

Упражнение 6.2. В упражнении 3.3 вам надо было написать функцию, которая получает многочлен (представленный в виде таблицы) и значение его переменной, а затем возвращает значение этого многочлена. Напишите *каррированную* (принимающая свои аргументы по одному за раз) версию данной функции. Ваша функция должна принимать многочлен и возвращать функцию, которая, будучи вызвана для какого-либо значения `x`, вернет значение многочлена для этого `x`. Например:

```
f = newpoly({3, 0, 1})
print(f(0))     --> 1
print(f(5))     --> 76
print(f(10))    --> 301
```

Упражнение 6.3. Иногда язык с корректными хвостовыми вызовами называют языком с поддержкой корректной хвостовой рекурсии, аргументируя это тем, что данное качество имеет значение только тогда, когда у нас есть рекурсивные вызовы (Без рекурсивных вызовов максимальная глубина вызовов была бы статически фиксированной.)

Покажите, что это утверждение не верно для динамически типизированных языков вроде Lua: напишите программу, которая совершает неограниченную цепочку вызовов без рекурсии.

(Подсказка: см. раздел 8.1.)

Упражнение 6.4. Как мы видели, хвостовой вызов — это замаскированный **goto**. Придерживаясь этой идеи, перепишите код простой игры в лабиринт из раздела 4.4 с применением хвостовых вызовов. Каждый блок должен стать новой функцией, а каждый **goto** должен стать хвостовым вызовом.

ГЛАВА 7

Итераторы и общий for

В этой главе мы рассмотрим, как писать итераторы для общего **for**. Начав с простых итераторов, мы узнаем, как использовать всю силу общего **for** для написания более простых и эффективных итераторов.

7.1. Итераторы и замыкания

Итератор (iterator) — это любая конструкция, которая позволяет вам перебирать элементы коллекции. В Lua мы обычно представляем итераторы при помощи функций: каждый раз, когда мы вызываем функцию, она возвращает «следующий» элемент из коллекции.

Любой итератор должен где-то хранить свое состояние между последовательными вызовами, чтобы знать, где он находится и как себя вести с этого места. Замыкания предоставляют великолепный механизм для этой задачи. Вспомним, что замыкание — это функция, которая обращается к одной или нескольким локальным переменным из охватывающего ее окружения. Эти переменные хранят свои значения между последовательными вызовами замыкания, позволяя тем самым замыканию помнить, где оно находится при переборе элементов. Разумеется, для создания нового замыкания мы также должны создать его нелокальные переменные. Поэтому конструкция замыкания обычно включает в себя две функции: само замыкание и *фабрику* — функцию, которая создает замыкание вместе с окружающими ее переменными.

В качестве примера давайте напишем простой итератор для списка. В отличие от `ipairs`, этот итератор возвращает не индекс

каждого элемента, а лишь его значение:

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

В этом примере `values` — это фабрика. Каждый раз, когда мы вызываем эту фабрику, она создает новое замыкание (сам итератор). Это замыкание хранит свое состояние в своих внешних переменных `t` и `i`. Каждый раз, когда мы вызываем этот итератор, он возвращает следующее значение из списка `t`. После последнего элемента итератор вернет nil, что означает конец итерации.

Мы можем использовать этот итератор в цикле **while**:

```
t = {10, 20, 30}
iter = values(t)              -- создает итератор
while true do
  local element = iter()      -- вызывает итератор
  if element == nil then break end
  print(element)
end
```

Тем не менее, гораздо легче использовать общий **for**. В конце концов, он был разработан для данного вида итераций:

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

Общий **for** ведет для итерирующего цикла всю бухгалтерию: он хранит внутри итерирующую функцию, поэтому нам не нужна переменная `iter`; он вызывает итератор для каждой новой итерации; и он останавливает цикл, когда итератор возвращает nil. (В следующем разделе мы увидим, что это далеко не все, что делает общий **for**.)

Как более продвинутый пример, листинг 7.1 показывает

итератор для перебора всех слов из текущего входного файла. Для такого перебора нам нужно два значения: содержимое текущей строки (переменная `line`) и где мы находимся внутри этой строки (переменная `pos`). С этими данными мы можем всегда сгенерировать следующее слово. Основная часть итерирующей функции — это вызов `string.find`. Этот вызов ищет слово в текущей строке, начиная с текущей позиции. Он описывает «слово», используя образец `'%w+'`, которому удовлетворяет один или более буквенно-цифровых символов. Если этот вызов находит слово, то функция обновляет текущую позицию на первый символ после слова и возвращает это слово (Примечание: Функция `string.sub` извлекает подстроку из `line` между заданными позициями; мы подробно рассмотрим ее в разделе 21.2). Иначе итератор читает следующую строку и повторяет поиск. Если строк больше нет, он возвращает nil, чтобы сообщить об окончании итерации.

Несмотря на его сложность, использование `allwords` крайне просто:

```
for word in allwords() do
  print(word)
end
```

Это типичная ситуация с итераторами: их может быть не так легко написать, зато их легко использовать. Серьезной проблемы в этом нет; гораздо чаще конечные пользователи, программирующие на Lua, не определяют итераторы, а лишь используют те, что предоставлены приложением.

Листинг 7.1. Итератор для обхода всех слов из входного файла

```
function allwords ()
  local line = io.read()     -- текущая строка
  local pos = 1              -- текущая позиция в
строке
  return function ()         -- итерирующая функция
    while line do            -- повторяет, пока есть
строки
```

```
      local s, e = string.find(line, "%w+", pos)
      if s then                -- слово найдено?
        pos = e + 1            -- следующая позиция после
этого слова
        return string.sub(line, s, e)    -- возвращает
это слово
      else
        line = io.read()       -- слово не найдено;
пробует следующую строку
        pos = 1                -- перезапуск с первой
позиции
      end
    end
    return nil                 -- строк больше нет; конец
обхода
  end
end
```

7.2. Семантика общего for

Единственным недостатком ранее рассмотренных итераторов является то, что нам необходимо создавать новое замыкание для инициализации каждого нового цикла. Для большинства случаев это не является проблемой. Например, в случае итератора `allwords` цена создания одного единственного замыкания несравнима с ценой чтения целого файла. Однако, в некоторых ситуациях эта лишняя нагрузка может быть неприемлемой. В таких случаях мы можем использовать сам общий **for** для хранения состояния итерации. В данном разделе мы увидим, какие возможности по хранению состояния предлагает общий **for**.

Мы видели, что общий **for** во время выполнения цикла хранит итерирующую функцию внутри себя. На самом деле он хранит три значения: *итерирующую функцию*, *инвариантное состояние* (неизменяющееся) и *управляющую переменную*. Теперь давайте обратимся к деталям.

Синтаксис общего **for** следующий:

```
for <список_переменных> in <список_выражений> do
  <тело>
end
```

Здесь *список_переменных* — это список из одного или нескольких имен переменных, разделенных запятыми, а *список_выражений* — это список из одного или нескольких выражений, также разделенных запятыми. Часто список выражений состоит из единственного элемента — вызова фабрики итераторов. В следующем коде, например, список переменных — это `k`, `v`, а список выражений состоит из единственного элемента `pairs(t)`:

```
for k, v in pairs(t) do print(k, v) end
```

Часто список переменных тоже состоит всего из одной переменной, как в следующем цикле:

```
for line in io.lines() do
  io.write(line, "\n")
end
```

Мы называем первую переменную в списке *управляющей переменной*. В течение всего цикла ее значение никогда не равно nil, поскольку когда она становится равной nil, цикл завершается.

Первое, что делает цикл **for**, — вычисляет выражения после **in**. Эти выражения должны дать три значения, которые хранит **for**: итерирующая функция, инвариантное состояние и начальное значение управляющей переменной. Как и во множественном присваивании, только последний (или единственный) элемент списка может дать более одного значения; число этих значений приводится к трем, лишние значения отбрасываются, вместо недостающих добавляются nil. (Когда мы используем простые итераторы, фабрика возвращает только итерирующую функцию, поэтому инвариантное состояние и управляющая переменная получают значение nil.)

После этого шага инициализации **for** вызывает итерирующую функцию с двумя аргументами: инвариантным состоянием и

управляющей переменной. (С точки зрения конструкции **for**, это инвариантное состояние вообще не имеет никакого смысла. Оператор **for** лишь передает значение состояния из шага инициализации в вызовы итерирующей функции.) Затем **for** присваивает значения, возвращенные итерирующей функцией, переменным, объявленным в его списке переменных. Если первое возвращенное значение (присваиваемое управляющей переменной) равно nil, то цикл завершается. Иначе **for** выполняет свое тело и вновь вызывает итерирующую функцию, повторяя процесс.

Точнее говоря, конструкция вида

```
for var_1, ..., var_n in <список_выражений> do
<блок> end
```

эквивалента следующему коду:

```
do
  local _f, _s, _var = <список_выражений>
  while true do
    local var_1, ... , var_n = _f(_s, _var)
    _var = var_1
    if _var == nil then break end
    <блок>
  end
end
```

Поэтому если наша итерирующая функция — *f*, неизменяемое состояние — *s*, а начальное состояние для управляющей переменной — a_0, то управляющая переменная будет пробегать следующие значения $a_1 = f(s, a_0)$, $a_2 = f(s, a_1)$ и т. д., до тех пор, пока a_i не станет равной nil. Если у **for** есть другие переменные, то они просто получат дополнительные значения, возвращаемые при каждом вызове *f*.

7.3. Итераторы без состояния

Как следует из названия, итератор без состояния — это итератор, который не хранит в себе какое-либо состояние. Поэтому мы можем использовать один и тот же итератор без состояния во многих циклах, избегая тем самым платы за создание новых замыканий.

Как мы только что видели, оператор **for** вызывает свою итерирующую функцию с двумя аргументами: инвариантное остояние и управляющая переменная. Итератор без состояния генерирует следующий элемент цикла, используя только эти два значения. Типичным примером этого вида итератора является `ipairs`, который перебирает все элементы массива:

```
a = {"one", "two", "three"}
for i, v in ipairs(a) do
  print(i, v)
end
```

Состояние этой итерации — это таблица, которую мы перебираем (т.е. инвариантное состояние, которое не меняется на протяжении цикла), и текущий индекс (управляющая переменная). И `ipairs` (фабрика), и итератор очень просты; мы могли бы написать их на Lua следующим образом:

```
local function iter (a, i)
  i = i + 1
  local v = a[i]
  if v then
    return i, v
  end
end

function ipairs (a)
  return iter, a, 0
end
```

Когда Lua вызывает `ipairs(a)` в цикле **for**, она получает три значения: функцию **iter** как итератор, `a` как инвариантное состояние и ноль как начальное значение для управляющей переменной. Затем

Lua вызывает `iter(a,0)`, что дает `1,a[1]` (если только `a[1]` уже не nil). На второй итерации вызывается `iter(a,1)`, что дает `2,a[2]`, и т. д., до первого элемента, равного nil.

Функция `pairs`, которая перебирает все элементы таблицы, аналогична, за исключением того, что итерирующая функция — это функция `next`, которая в Lua является примитивной:

```
function pairs (t)
  return next, t, nil
end
```

Вызов `next(t,k)`, где `k` — это ключ таблицы `t`, возвращает следующий ключ в таблице в произвольном порядке, а также связанное с этим ключом значение как второе возвращаемое значение. Вызов `next(t,nil)` возвращает первую пару. Когда пар больше нет, `next` возвращает nil.

Некоторые предпочитают использовать `next` напрямую, избегая вызова `pairs`:

```
for k, v in next, t do
  <тело цикла>
end
```

Вспомним, что **for** приводит свой список выражений к трем результатам, чтобы получить `next`, `t` и nil; это именно то, что получается при вызове `pairs`.

Итератор для обхода связанного списка является еще одним интересным примером итератора без состояния. (Как мы уже упомянули, связанные списки нечасто встречаются в Lua, но иногда они нам нужны.)

```
local function getnext (list, node)
  if not node then
    return list
  else
    return node.next
  end
```

```
end

function traverse (list)
  return getnext, list, nil
end
```

Суть этого приема состоит в использовании основного узла списка в качестве инвариантного состояния (второе значение, возвращаемое `traverse`) и текущего узла в качестве управляющей переменной. При первом вызове итерирующей функции `getnext` переменная `node` будет равна nil, и поэтому функция вернет `list` как первый узел. В последующий вызовах `node` не будет равна nil, и поэтому итератор, как и ожидалось, вернет `node.next`. Как обычно, использование этого итератора крайне просто:

```
list = nil
for line in io.lines() do
  list = {val = line, next = list}
end

for node in traverse(list) do
  print(node.val)
end
```

7.4. Итераторы со сложным состоянием

Часто итератору требуется хранить больший объем состояния, чем помещается в единственное инвариантное состояние и управляющую переменную. Простейшим решением является использование замыканий. Альтернативным решением будет запаковать все, что нужно итератору, в таблицу и использовать эту таблицу как инвариантное состояние для итерации. С помощью таблицы итератор можем хранить столько данных, сколько ему потребуется на протяжении цикла. Кроме того, во время итерации он может менять эти данные. Хотя состояние — это все время одна и та же таблица (поэтому оно инвариантное), содержимое таблицы

может меняться на протяжении цикла. Поскольку такие итераторы хранят все свои данные в состоянии, обычно они игнорируют второй аргумент, предоставляемый общим циклом **for** (итерационная переменная).

В качестве примера такого подхода мы перепишем итератор `allwords`, который обходит все слова текущего входного файла. На этот раз мы будем хранить его состояние в таблице с двумя полями: `line` и `pos`.

Функция, которая начинает итерацию, довольно проста. Она должна вернуть итерирующую функцию и начальное состояние:

```
local iterator    -- будет определена позже

function allwords ()
  local state = {line = io.read(), pos = 1}
  return iterator, state
end
```

Основную работу выполняет функция `iterator`:

```
function iterator (state)
  while state.line do                -- повторяет, пока
есть строки
    -- ищет следующее слово
    local s, e = string.find(state.line, "%w+",
state.pos)
    if s then                        -- слово найдено?
      -- обновляет следующую позицию (после этого
слова)
      state.pos = e + 1
      return string.sub(state.line, s, e)
    else                             -- слово не
найдено
      state.line = io.read()         -- пробует
следующую строку...
      state.pos = 1                  -- ... начиная с
первой позиции
    end
  end
  return nil                         -- строк больше
```

```
    нет: конец цикла
    end
```

При любой возможности вам следует пытаться написать итераторы без состояния, такие, что хранят все свое состояние в переменных цикла **for**. С ними вы не создаете новых объектов, когда начинаете цикл. Если для вашей итерации эта модель не подходит, то вам следует попробовать замыкания. Помимо большей изящности, замыкание обычно более эффективно, чем итератор с применением таблиц: во-первых, дешевле создать замыкание, чем таблицу; во-вторых, доступ к нелокальным переменным быстрее, чем доступ к полям таблицы. Позже мы увидим еще один способ писать итераторы, основанный на сопрограммах. Это самое эффективное решение, хотя и более затратное.

7.5. Подлинные итераторы

Термин «итератор» несколько неточен, поскольку итерацию выполняют не наши итераторы, а цикл **for**. Итераторы лишь предоставляют последовательные значения для итерации. Пожалуй, более удачным термином был бы «генератор», но термин «итератор» уже получил широкое распространение в других языках, таких как Java.

Тем не менее, существует другой способ построения итераторов, при котором итераторы действительно выполняют итерацию. Когда мы используем такие итераторы, мы не пишем цикл; вместо этого мы просто вызываем итератор с аргументом, описывающим, что итератор должен делать на каждой итерации. Точнее итератор получает в качестве аргумента функцию, которую он вызывает внутри своего цикла.

В качестве конкретного примера давайте еще раз перепишем итератор `allwords`, придерживаясь этого стиля:

```
function allwords (f)
```

```
  for line in io.lines() do
    for word in string.gmatch(line, "%w+") do
      f(word)    -- вызывает функцию
    end
  end
end
```

Для использования этого итератора мы просто должны предоставить тело цикла как функцию. Если нам нужно всего лишь напечатать каждое слово, то мы просто используем `print`:

```
allwords(print)
```

Зачастую в качестве тела цикла используется анонимная функция. Например, следующий фрагмент кода считает, сколько раз слово «hello» встречается во входном файле:

```
local count = 0
allwords(function (w)
  if w == "hello" then count = count + 1 end
end)
print(count)
```

Та же самая задача, записанная в стиле предыдущего итератора, не сильно отличается:

```
local count = 0
for w in allwords() do
  if w == "hello" then count = count + 1 end
end
print(count)
```

Подобные итераторы были популярны в более старых версиях Lua, когда в языке не было оператора **for**. Какое отношение они имеют к итераторам в стиле генераторов? У обоих стилей примерно одинаковые затраты: один вызов функции на итерацию. С одной стороны, легче писать итератор с использованием подлинных итераторов (хотя мы можем получить эту же легкость при помощи сопрограмм). С другой стороны, стиль генераторов более гибкий.

Во-первых, он позволяет две и более параллельных итерации. (Например, рассмотрим случай перебора сразу двух файлов, сравнивая их пословно.) Во-вторых, он позволяет использование **break** и **return** внутри тела итератора. С подлинными итераторами **return** возвращает из анонимной функции, но не из функции, совершающей итерацию. В основном, я обычно предпочитаю генераторы.

Упражнения

Упражнение 7.1. Напишите итератор `fromto` так, чтобы следующие два цикла оказались эквивалентными:

```
for i in fromto(n, m)
  <тело>
end

for i = n, m
  <тело>
end
```

Можете ли вы реализовать его при помощи итератора без состояния?

Упражнение 7.2. Добавьте параметр, отвечающий за шаг, к итератору из предыдущего упражнения. Можете ли вы по-прежнему реализовать его как итератор без состояния?

Упражнение 7.3. Напишите итератор `uniquewords`, который возвращает все слова из заданного файла без повторений. (Подсказка: начните с кода `allwords` из листинга 7.1; используйте таблицу, чтобы хранить все слова, которые вы уже вернули.)

Упражнение 7.4. Напишите итератор, который возвращает все непустые подстроки заданной строки. (Вам понадобится функция `string.sub`.)

ГЛАВА 8

Компиляция, выполнение и ошибки

Хотя мы называем Lua интерпретируемым языком, Lua всегда предкомпилирует исходный код в промежуточную форму перед его выполнением. (В этом нет ничего страшного: многие интерпретируемые языки делают то же самое.) Наличие этапа компиляции может казаться неуместным в интерпретируемом языке вроде Lua. Однако, отличительным признаком интерпретируемых языков является не то, что они не компилируются, а то, что в них возможно (и легко) выполнять код, генерируемый на лету. Можно сказать, что наличие функции вроде `dofile` — это то, что позволяет Lua называться интерпретируемым языком.

8.1. Компиляция

Ранее мы ввели `dofile` как своего рода примитивную операцию для выполнения кусков кода Lua, но `dofile` на самом деле является вспомогательной функцией: всю тяжелую работу выполняет `loadfile`. Как и `dofile`, `loadfile` загружает кусок кода Lua из файла, но при этом не выполняет его. Вместо этого он только компилирует этот кусок и возвращает скомпилированный кусок как функцию. Более того, в отличие от `dofile`, `loadfile` не вызывает ошибки, а наоборот возвращает их коды, чтобы мы могли разобраться с ошибкой. Мы могли бы определить `dofile` следующим образом:

```
function dofile (filename)
  local f = assert(loadfile(filename))
  return f()
end
```

Обратите внимание на использование `assert` для вызова ошибки, если `loadfile` даст сбой.

Для простых задач `dofile` удобна, поскольку выполняет всю работу за один вызов. Однако `loadfile` более гибкая. В случае ошибки `loadfile` возвращает nil и сообщение об ошибке, что позволяет нам обработать ошибку более подходящими для нас способами. Более того, если нам нужно выполнить файл несколько раз, то мы можем один раз вызвать `loadfile` и несколько раз вызвать возвращенную им функцию. Этот подход гораздо дешевле, чем несколько раз вызывать `dofile`, поскольку файл компилируется лишь один раз.

Функция `load` похожа на `loadfile`, за исключением того, что она берет кусок кода не из файла, а из строки (Примечание: В Lua 5.1 функция `loadstring` выполняет роль `load`). Например, рассмотрим следующую строку:

```
f = load("i = i + 1")
```

После этого кода `f` станет функцией, которая при вызове выполняет `i = i + 1`:

```
i = 0
f(); print(i)    --> 1
f(); print(i)    --> 2
```

Функция `load` довольно мощная, и мы должны использовать ее с осторожностью. Это также затратная функция (по сравнению с некоторыми альтернативами), которая может привести к непонятному коду. Перед ее использованием, убедитесь, что в вашем распоряжении нет более простого способа решить задачу.

Если хотите по-быстрому воспользоваться `dostring` (т.е. загрузить и выполнить кусок), можете непосредственно использовать результат `load`:

```
load(s)()
```

Однако, если есть хотя бы одна синтаксическая ошибка, то `load` вернет nil и окончательным сообщением об ошибке будет что-то вроде *"attempt to call a nil value"*. Для более понятных сообщений об ошибках используйте `assert`:

```
assert(load(s))()
```

Обычно нет никакого смысла применять функцию `load` на строковом литерале. Например, следующие две строки примерно эквиваленты:

```
f = load("i = i + 1")

f = function () i = i + 1 end
```

Тем не менее, вторая строка кода гораздо быстрее, поскольку Lua скомпилирует функцию вместе с окружающим ее куском. В первой строке вызов `load` включает отдельную компиляцию.

Поскольку `load` не компилирует с учетом лексической области действия, то те две строки из предыдущего примера могут быть не совсем эквивалентны. Чтобы увидеть разницу, давайте слегка изменим пример:

```
i = 32
local i = 0
f = load("i = i + 1; print(i)")
g = function () i = i + 1; print(i) end
f()    --> 33
g()    --> 1
```

Функция `g` работает с локальной `i`, как и ожидалось, но `f` работает с глобальной `i`, поскольку `load` всегда компилирует свои куски в глобальном окружении.

Наиболее типичным использованием `load` является выполнение внешнего кода, то есть фрагментов кода, поступающих извне вашей программы. Например, вам может потребоваться построить график функции, заданной пользователем; пользователь вводит код

функции, а вы затем используете `load`, чтобы выполнить его. Обратите внимание, что `load` ожидает получить кусок, то есть операторы. Если вы хотите вычислить выражение, то вы можете поставить перед выражением **return**, что даст вам оператор, возвращающий значение заданного выражения. Посмотрите на пример:

```
print "enter your expression:"
local l = io.read()
local func = assert(load("return " .. l))
print("the value of your expression is " .. func())
```

Поскольку функция, возвращенная `load`, является обычной, вы можете вызвать ее несколько раз:

```
print "enter function to be plotted (with variable
'x'):"
local l = io.read()
local f = assert(load("return " .. l))
for i = 1, 20 do
  x = i -- глобальная 'x' (чтобы быть видимой из этого
куска кода)
  print(string.rep("*", f()))
end
```

(Функция `string.rep` повторяет строку заданное число раз.)

Мы также можем вызвать функцию `load`, передав ей в качестве аргумента *считывающую функцию* (reader function). Считывающая функция может возвращать кусок кода частями; `load` последовательно вызывает ее до тех, пока она не вернет nil, обозначающий конец куска. В качестве примера следующий вызов эквивалентен `loadfile`:

```
f = load(io.lines(filename, "*L"))
```

Как мы подробно рассмотрим в главе 22, вызов `io.lines(filename,"*L")` возвращает функцию, которая при каждом своем вызове возвращает следующую строку из заданного

файла (Примечание: Опции для `io.lines` появились только в Lua 5.2). Таким образом, `load` будет читать кусок из файла строка за строкой. Следующий вариант похож, но более эффективен:

```
f = load(io.lines(filename, 1024))
```

Здесь итератор, возвращенный `io.lines`, читает файл блоками по 1024 байта.

Lua рассматривает каждый независимый кусок как тело анонимной функции с переменным числом аргументов. Например, `load("a = 1")` возвращает аналог следующего выражения:

```
function (...) a = 1 end
```

Как и любые другие функции, куски могут объявлять локальные переменные:

```
f = load("local a = 10; print(a + 20)")
f() --> 30
```

Используя эти возможности, мы можем переписать наш пример с построением графика так, чтобы избежать применения глобальной переменной `x`:

```
print "enter function to be plotted (with variable
'x'):"
local l = io.read()
local f = assert(load("local x = ...; return " .. l))
for i = 1, 20 do
  print(string.rep("*", f(i)))
end
```

Мы ставим объявление "`local x = ...`" в начало куска, чтобы определить `x` как локальную переменную. Затем мы вызываем `f` с аргументом `i`, который становится значением выражения с переменным числом аргументов `(...)`.

Функции загрузки кусков никогда не вызывают ошибки. В случае ошибки любого рода они возвращают nil и сообщение об

ошибке:

```
print(load("i i"))
  --> nil    [string "i i"]:1: '=' expected near 'i'
```

Более того, у этих функций нет никакого побочного эффекта. Они только компилируют кусок во внутреннее представление и возвращают результат как анонимную функцию. Распространенная ошибка — полагать, что загрузка куска определяет функции. В Lua определения функций являются присваиваниями; поэтому они происходят во время выполнения, а не во время компиляции. Например, допустим, что у нас есть файл `foo.lua` наподобие следующего:

```
-- файл 'foo.lua'
function foo (x)
  print(x)
end
```

Затем мы выполняем команду

```
f = loadfile("foo.lua")
```

После этой команды `foo` скомпилирована, но еще не определена. Чтобы определить ее, мы должны выполнить этот кусок:

```
print(foo)    --> nil
f()           -- определяет 'foo'
foo("ok")     --> ok
```

В готовой к распространению программе, которой требуется выполнять внешний код, вы должны обрабатывать любые ошибки, возникающие при загрузке куска. Более того, вам может понадобится запустить новый кусок в защищенном окружении, чтобы избежать неприятных побочных эффектов. Мы подробно обсудим окружения в главе 14.

8.2. Предкомпилированный код

Как я отметил в начале этой главы, Lua предкомпилирует исходный код перед его выполнением. Lua также позволяет распространять код в предкомпилированной форме.

Простейшим способом получения предкомпилированного файла (*бинарного куска* на жаргоне Lua) является использование программы `luac`, которая входит в стандартную поставку. Например, следующий вызов создает новый файл `prog.1c` с предкомпилированной версией файла `prog.lua`:

```
$ luac -o prog.lc prog.lua
```

Интерпретатор может выполнить этот новый файл прямо как обычный код Lua, работая с ним точно так же, как и с исходным файлом:

```
$ lua prog.lc
```

Lua принимает предкомпилированный код практически везде, где он допускает исходный код. В частности, `loadfile` и `load` принимают предкомпилированный код.

Мы можем написать упрощенную версию `luac` непосредственно на Lua:

```
p = loadfile(arg[1])
f = io.open(arg[2], "wb")
f:write(string.dump(p))
f:close()
```

Основная функция здесь — это `string.dump`: она получает функцию Lua и возвращает ее предкомпилированный код как строку, правильно оформленную для ее обратной загрузки в Lua.

Программа `luac` предлагает некоторые другие интересные опции. В частности, опция `-l` печатает список кодов операций, которые компилятор генерирует для данного куска. В качестве примера листинг 8.1 содержит вывод программы `luac` с опцией `-l` для следующего однострочного файла:

```
a = x + y - z
```

(Мы не будем обсуждать внутреннее устройство Lua в этой книге; если вас интересуют подробности об этих кодах операций, то поиск в Интернете по фразе "`lua opcode`" должен дать вам подходящий материал.)

Листинг 8.1. Пример вывода **`luac -l`**

```
main stdin:0,0 (7 instructions, 28 bytes at 0x988cb30)
0+ params, 2 slots, 0 upvalues, 0 locals, 4 constants,
0 functions
 1 [1] GETGLOBAL 0 -2 ; x
 2 [1] GETGLOBAL 1 -3 ; y
 3 [1] ADD 0 0 1
 4 [1] GETGLOBAL 1 -4 ; z
 5 [1] SUB 0 0 1
 6 [1] SETGLOBAL 0 -1 ; a
 7 [1] RETURN 0 1
```

Код в предкомпилированной форме не всегда меньше исходного кода, но он загружается быстрее. Еще одним преимуществом является то, что это дает вам защиту от случайных изменений в исходниках. Однако, в отличие от исходного кода, поврежденный злоумышленником бинарный код может привести к падению интерпретатора Lua или даже выполнить предоставленный пользователем машинный код. При выполнении обычного кода вам не о чем беспокоиться. Однако, вам следует избегать выполнения ненадежного кода в предкомпилированной форме. У функции `load` есть опция как раз для этой задачи.

Кроме обязательного первого аргумента, у `load` есть еще три необязательных. Вторым аргументом является имя куска, которое используется только в сообщениях об ошибках. Четвертый аргумент — это окружение, которое мы обсудим в главе 14. Третий аргумент — это именно то, что нас здесь интересует; он управляет тем, какие виды кусков могут быть загружены. При наличии этот

аргумент должен быть строкой: строка "t" позволяет загружать лишь текстовые (обычные) куски, "b" позволяет загружать лишь бинарные (предкомпилированные) куски, а "bt", значение по умолчанию, разрешает оба формата.

8.3. Код C

В отличие от кода, написанного на Lua, код C должен быть скомпонован (связан) с приложением перед его использованием. В некоторых популярных операционных системах наиболее лекий способ создания этой связи заключается в применении средства динамической компоновки. Однако, данное средство не является частью спецификации ANSI C; поэтому переносимого способа его реализации не существует.

Обычно Lua не содержит средства, которые не могут быть реализованы в ANSI C. Однако, с динамической компоновкой иная ситуация. Мы можем рассматривать ее как основу всех других средств: имея ее, мы способны динамически подгружать любое другое средство, которого нет в Lua. Поэтому в данном особом случае Lua отказывается от правил переносимости и реализует средство динамической компоновки для ряда платформ. Стандартная реализация предлагает его поддержку для Windows, Mac OS X, Linux, FreeBSD, Solaris и большинства других реализаций UNIX. Перенос данного средства на другие платформы должен быть не сложным; проверьте ваш дистрибутив. (Для этого выполните `print(package.loadlib("a","b"))` из приглашения ввода Lua и посмотрите на результат. Если он жалуется на несуществующий файл, то у вас есть средство динамической компоновки. В противном случае сообщение об ошибке должно указать, что данное средство не поддерживается или не установлено.)

Lua предоставляет все возможности динамической компоновки посредством единственной функцию под названием

`package.lodlib`. У нее есть два строковых аргумента: полный путь к библиотеке и имя функции из этой библиотеки. Поэтому ее типичный вызов выглядит как следующий фрагмент кода:

```
local path = "/usr/local/lib/lua/5.1/socket.so"
local f = package.loadlib(path, "luaopen_socket")
```

Функция `loadlib` загружает заданную библиотеку и связывает ее с Lua. Однако, она не вызывает заданную функцию. Вместо этого она возвращает функцию C как функцию Lua. В случае ошибки при загрузке библиотеки или нахождении инициализирующей функции `loadlib` возвращает nil и сообщение об ошибке.

Функция `loadlib` является очень низкоуровневой. Мы должны предоставить полный путь к библиотеке и правильное имя функции (включая символы подчеркивания в начале, время от времени добавляемые компилятором). Чаще всего мы загружаем библиотеки C посредством `require`. Эта функция ищет библиотеку и использует `loadlib`, чтобы загрузить для нее инициализирующую функцию. При вызове эта инициализирующая функция строит и возвращает таблицу с функциями из этой библиотеки подобно тому, как это делает обычная библиотека Lua. Мы обсудим `require` в разделе 15.1 и подробнее рассмотрим библиотеки C в разделе 27.3.

8.4. Ошибки

Errare humanum est (Человеку свойственно ошибаться). Поэтому мы должны обрабатывать ошибки лучшим из доступных нам способов. Поскольку Lua является расширяющим языком, часто встраиваемым в приложение, он не может просто упасть или завершить работу в случае возникновения ошибки. Вместо этого, каждый раз, когда возникает ошибка, Lua завершает текущий кусок и возвращает управление в приложение.

Любая неожиданная ситуация, с которой сталкивается Lua,

вызывает ошибку. Ошибки возникают, когда вы (то есть ваша программа) пытаетесь сложить значения, которые не являются числами, индексировать не таблицу и т. п. (Вы можете изменить это поведение при помощи *метатаблиц*, как мы увидим позже.) Вы также можете напрямую вызвать ошибку при помощи вызова функции `error` с сообщением об ошибке в качестве аргумента. Обычно эта функция является подходящим способом для оповещения об ошибках в вашем коде:

```
print "enter a number:"
n = io.read("*n")
if not n then error("invalid input") end
```

Данная конструкция вызова `error` для некоторых условий настолько распространена, что для нее в Lua есть встроенная функция `assert`:

```
print "enter a number:"
n = assert(io.read("*n"), "invalid input")
```

Функция `assert` проверяет, действительно ли ее первый аргумент не ложен, и просто возвращает этот аргумент; если аргумент ложен, то `assert` вызывает ошибку. Ее второй аргумент (сообщение) не обязателен. Однако, имейте в виду, что `assert` — это обычная функция. В связи с этим Lua всегда вычисляет ее аргументы перед вызовом. Поэтому если вы напишите что-то вроде

```
n = io.read()
assert(tonumber(n), "invalid input: " .. n .. " is not
a number")
```

то Lua всегда выполнит конкатенацию, даже когда `n` является числом. В таких случаях разумнее бывает использовать явную проверку.

Когда функция обнаруживает непредвиденную ситуацию (*исключение*), ей доступны две основные линии поведения: вернуть код ошибки (обычно nil) или вызвать ошибку посредством вызова

функции `error`. Не существует жестких правил для выбора между этими двумя вариантами, но мы можем дать общую рекомендацию: исключение, которое легко обходится, должно вызывать ошибку; иначе оно должно вернуть код ошибки.

Например, давайте рассмотрим функцию `sin`. Как она должна себя вести при вызове на таблице? Предположим, она возвращает код ошибки. Если бы нам понадобилась проверка на ошибки, мы бы написали что-то вроде

```
local res = math.sin(x)
if not res then    -- ошибка?
  <код обработки ошибки>
```

Однако, мы могли бы легко проверить это исключение *перед* вызовом функции:

```
if not tonumber(x) then    -- x не является числом?
  <код обработки ошибки>
```

Часто мы не проверяем ни аргумент, ни результат вызова `sin`; если аргумент не является числом, значит, вероятно, с нашей программой творится нечто неладное. В подобной ситуации прекратить вычисления и вызвать ошибку — это простейший и наиболее практичный способ обработки данного исключения.

С другой стороны, давайте рассмотрим функцию `io.open`, которая открывает файл. Как она должна себя вести, если попросить ее прочитать несуществующий файл? В этом случае не существует простого способа проверки на исключение перед вызовом этой функции. Во многих системах единственным способом узнать, что файл существует, является попытка его открыть. Поэтому если `io.open` не может открыть файл по какой-то внешней причине (например, «файл не существует» или «недостаточно прав»), то она возвращает nil и строку с сообщением об ошибке. Таким образом, у вас есть шанс обработать ситуацию подходящим образом, например, попросив у пользователя другое имя файла:

```
local file, msg
repeat
  print "enter a file name:"
  local name = io.read()
  if not name then return end    -- ввода данных не
было
  file, msg = io.open(name, "r")
  if not file then print(msg) end
until file
```

Если вы не хотите обрабатывать подобные ситуации, но по-прежнему хотите перестраховаться, то для защиты данной операции вы можете просто использовать `assert`:

```
file = assert(io.open(name, "r"))
```

Это типичная для Lua идиома: если `io.open` даст сбой, то `assert` вызовет ошибку.

```
file = assert(io.open("no-file", "r"))
--> stdin:1: no-file: No such file or directory
```

Обратите внимание, как сообщение об ошибке, которое является вторым результатом `io.open`, оказывается вторым аргументом для `assert`.

8.5. Обработка ошибок и исключений

Для многих приложений вам не нужно выполнять никакой обработки ошибок в Lua; этим занимается прикладная программа. Вся работа Lua начинается с вызова от приложения, которое обычно просит выполнить кусок. При любой ошибке этот вызов возвращает код ошибки, чтобы приложение могло предпринять соответствующие действия. В случае автономного интерпретатора его главный цикл лишь печатает сообщение об ошибке, а затем продолжает показывать приглашение ввода и выполнять команды.

Однако, если вам надо обрабатывать ошибки в Lua, вы должны

использовать pcall (*«protected call» — защищенный вызов*) для инкапсуляции вашего кода.

Допустим, вы хотите выполнить фрагмент кода Lua и поймать любую ошибку, вызванную при его выполнении. Вашим первым шагом будет инкапсулировать этот фрагмент кода в функцию; в большинстве случаев для этого используются анонимные функции. Затем вы вызываете эту функцию посредством pcall:

```
local ok, msg = pcall(function ()
  <какой-нибудь код>
  if unexpected_condition then error() end
  <какой-нибудь код>
  print(a[i])    -- потенциальная ошибка: 'a' может не
быть таблицей
  <какой-нибудь код>
end)

if ok then    -- при выполнении защищенного кода
ошибок нет
<обычный код>
else    -- защищенный код вызвал ошибку: примите
соответственные меры
  <код обработки ошибки>
end
```

Функция pcall вызывает свой первый аргумент в *защищенном режиме*, чтобы перехватывать любые ошибки во время выполнения функции. Если ошибок нет, то pcall возвращает true и все значения, возвращенные тем вызовом. Иначе она возвращает false и сообщение об ошибке.

Несмотря на свое название, сообщение об ошибке не обязано быть строкой: pcall вернет любое значение Lua, которое вы передали error.

```
local status, err = pcall(function ()
error({code=121}) end)
print(err.code)    --> 121
```

Эти механизмы предоставляют все, что вам необходимо для обработки исключений в Lua. Мы выбрасываем исключение при помощи `error` и перехватываем его при помощи `pcall`. Сообщение об ошибке идентифицирует вид ошибки.

8.6. Сообщения об ошибках и обратные трассировки

Хотя в качестве сообщения об ошибке мы можем использовать значение любого типа, обычно сообщения об ошибках — это строки, описывающие, что пошло не так. В случае возникновения внутренней ошибки (например, при попытке индексировать нетабличное значение) Lua генерирует сообщение об ошибке; иначе сообщением об ошибке становится значение, переданное функции `error`. В тех случаях, когда сообщение об ошибке является строкой, Lua пытается добавить некоторую информацию о том месте, где произошла ошибка:

```
local status, err = pcall(function () a = "a"+1 end)
print(err)
  --> stdin:1: attempt to perform arithmetic on a
string value

local status, err = pcall(function () error("my
error") end)
print(err)
  --> stdin:1: my error
```

Эта информация о месте ошибки содержит имя файла (в примере это `stdin`) и номер его строки кода (в примере это 1).

У функции `error` есть второй дополнительный параметр, который задает *уровень*, на котором она должна докладывать об ошибке; вы используете этот параметр, чтобы обвинить кого-то другого в случае ошибки. Скажем, вы написали функцию, чьей первой задачей является проверка, что она была правильно

вызвана:

```
function foo (str)
  if type(str) ~= "string" then
    error("string expected")
  end
  <обычный код>
end
```

Затем кто-то вызывает вашу функцию с неправильным аргументом:

```
foo({x=1})
```

Если оставить все как есть, Lua покажет на вашу функцию — ведь это же `foo` вызвала `error`, — а не на настоящего виновника, который вызвал ее с неправильным аргументом. Для исправления данной проблемы вы информируете `error`, что ошибка, о которой вы докладываете, возникла на уровне 2 в иерархии вызовов (уровень 1 — это ваша собственная функция):

```
function foo (str)
  if type(str) ~= "string" then
    error("string expected", 2)
  end
  <обычный код>
end
```

Часто при возникновении ошибки мы хотим получить больше отладочной информации, а не только место возникновения. Как минимум нам нужна обратная трассировка, показывающая полный стек вызовов, приведших к ошибке. Когда `pcall` возвращает свое сообщение об ошибке, она уничтожает часть стека (часть от нее до момента возникновения ошибки). Следовательно, если нам требуется обратная трассировка, мы должны построить ее до возврата из `pcall`. Для этого Lua предоставляет функцию `xpcall`. Кроме функции, которую нужно вызвать, она получает второй аргумент — *функцию обработки сообщений*. В случае ошибки Lua

вызывает этот обработчик сообщений перед раскруткой стека, поэтому он может использовать отладочную библиотеку для получения любой дополнительной информации об ошибке, которая ему потребуется. Двумя наиболее распространенными обработчиками ошибок являются `debug.debug`, предоставляющий вам приглашение ввода Lua, чтобы вы могли сами посмотреть, что происходило в момент возникновения ошибки; и `debug.traceback`, который строит расширенное сообщение об ошибке с обратной трассировкой (Примечание: В главе 24 мы больше узнаем об этих функциях, когда обсудим отладочную библиотеку). Именно последнюю функцию использует автономный интерпретатор для построения своих сообщений об ошибках.

Упражнения

Упражнение 8.1. Часто бывает удобно добавить какой-нибудь префикс к куску кода при его загрузке. (Ранее мы рассматривали пример в данной главе, где мы ставили **return** перед загружаемым выражением.) Напишите функцию `loadwithprefix`, которая работает как `load`, за исключением того, что она добавляет свой первый дополнительный аргумент (строку) в качестве префикса для загружаемого куска.

Как и оригинальная `load`, функция `loadwithprefix` должна принимать куски, представленные как строками, так и считывающими функциями. Даже в случае, когда изначальный кусок является строкой, `loadwithprefix` в действительности не должна конкатенировать префикс с куском. Вместо этого она должна вызвать `load` с соответствующей считывающей функцией, которая сперва возвратит префикс, и лишь затем исходный кусок.

Упражнение 8.2. Напишите функцию `multiload`, которая обобщает `loadwithprefix`, получая список считывающих функций, как в следующем примере:

```
f = multiload("local x = 10;",
              io.lines("temp", "*L"),
              " print(x)")
```

Для приведенного выше примера `multiload` должна загрузить кусок, эквивалентный конкатенации строки `"local..."` с содержимым файла `temp` и строки `"print(x)"`. Как и функция `loadwithprefix` из предыдущего упражнения, данная функция в действительности не должна ничего конкатенировать.

Упражнение 8.3. Функция `string.rep` в листинге 8.2 использует алгоритм двоичного умножения для конкатенации `n` копий заданной строки `s`. Для любого фиксированного `n` мы можем создать специализированную версию `string.rep`, развертывая цикл в последовательность команд `r=r..s` и `s=s..s`. В качестве примера для `n=5` развертка даст нам следующую функцию:

```
function stringrep_5 (s)
  local r = ""
  r = r .. s
  s = s .. s
  s = s .. s
  r = r .. s
  return r
end
```

Напишите функцию, которая для заданного `n` возвращает функцию `stringrep_`*n*. Вместо использования замыкания ваша функция должна построить текст функции Lua с соответствующей последовательностью команд (`r=r..s` и `s=s..s`) и затем использовать `load` для получения итоговой функции. Сравните быстродействие общей функции `string.rep` (или замыкания с ее использованием) и вашими заказными функциями.

Листинг 8.2. Повторение строк

```
function stringrep (s, n)
  local r = ""
  if n > 0 then
```

```
    while n > 1 do
      if n % 2 ~= 0 then r = r .. s end
      s = s .. s
      n = math.floor(n / 2)
    end
    r = r .. s
  end
  return r
end
```

Упражнение 8.4. Можете ли вы найти такое значение для `f`, что выражение `pcall(pcall,f)` вернет false в качестве своего первого результата?

ГЛАВА 9

Сопрограммы

Сопрограмма (coroutine) похожа на нить (в смысле многонитевости): это поток выполнения со своим стеком, своими локальными переменными и своим указателем команд; но он разделяет глобальные переменные и почти все остальное с другими сопрограммами. Основное отличие между нитями и сопрограммами — это то, что концептуально (или буквально, в случае многопроцессорной машины) программа с нитями выполняет несколько нитей параллельно. Сопрограммы, с другой стороны, работают совместно: в любой момент времени программа с сопрограммами выполняет только одну из своих сопрограмм, и эта выполняемая сопрограмма приостанавливает свое выполнение, только когда ее явно попросят приостановиться.

Сопрограмма — это очень мощная концепция. И поэтому иногда ее трудно применять. Не волнуйтесь, если вы не поймете некоторые примеры из этой главы при первом чтении. Вы можете дочитать до конца книги и вернуться сюда позже. Но, пожалуйста, вернитесь; ваше время будет потрачено не напрасно.

9.1. Основы сопрограмм

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

```
co = coroutine.create(function () print("hi") end)

print(co)    --> thread: 0x8071d98
```

Сопрограмма может быть в одном из четырех состояний: приостановленное — **suspended**, выполняемое — **running**, завершенное — **dead**, обычное — **normal**. Мы можем проверить состояние сопрограммы при помощи функции `status`:

```
print(coroutine.status(co))    --> suspended
```

Когда мы создаем сопрограмму, она запускается в приостановленном состоянии; сопрограмма не начинает автоматически выполнять свое тело при создании. Функция `coroutine.resume` (пере)запускает выполнение сопрограммы, меняя ее состояние из приостановленного в выполняемое:

```
coroutine.resume(co)    --> hi
```

В этом первом примере тело сопрограммы просто печатает "`hi`" и прекращает выполнение, оставляя сопрограмму в завершенном состоянии, из которого ей уже не вернуться:

```
print(coroutine.status(co))    --> dead
```

До сих пор сопрограммы выглядели не более чем усложненным способ вызова функций. Настоящая сила сопрограмм идет от функции `yield`, которая позволяет выполняемой сопрограмме приостановить свое выполнение (иными словами, *уступить управление*), чтобы она могла быть возобновлена позже. Давайте рассмотрим простой пример:

```
co = coroutine.create(function ()
  for i = 1, 10 do
    print("co", i)
    coroutine.yield()
  end
end)
```

Теперь, когда мы возобновляем эту сопрограмму, она начинает свое выполнение и выполняется до первого `yield`:

```
coroutine.resume(co)    --> co    1
```

Если мы проверим ее состояние, то увидим, что данная сопрограмма приостановлена и, следовательно, может быть снова возобновлена:

```
print(coroutine.status(co))    --> suspended
```

С точки зрения сопрограммы, вся деятельность, которая происходит, пока сопрограмма приостановлена, происходит внутри вызова `yield`. Когда мы возобновляем эту сопрограмму, из вызова `yield` возвращается управление, и сопрограмма продолжает свое выполнение до следующего `yield` или до своего окончания:

```
coroutine.resume(co) --> co    2
coroutine.resume(co) --> co    3
  ...
coroutine.resume(co) --> co    10
coroutine.resume(co) -- ничего не печатает
```

Во время последнего вызова `resume` тело сопрограммы завершает цикл, а затем возвращает управление без печати чего-либо. Если мы попытаемся возобновить ее снова, то `resume` вернет false и сообщение об ошибке:

```
print(coroutine.resume(co))
--> false    cannot resume dead coroutine
```

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

Когда одна сопрограмма возобновляет другую, она не приостанавливается; в конце концов, мы не можем ее возобновить. Однако, она и не выполняется, так как выполняемой является другая сопрограмма. Поэтому ее собственное состояние мы

называем *обычным*.

Полезным средством в Lua является то, что пара `resume`—`yield` может обмениваться данными. Первая `resume`, у которой нет соответственной ожидающей ее `yield`, передает свои дополнительные аргументы главной функции сопрограммы:

```
co = coroutine.create(function (a, b, c)
    print("co", a, b, c + 2)
  end)
coroutine.resume(co, 1, 2, 3) --> co    1    2    5
```

Вызов `resume` возвращает после true, сообщающего, что нет ошибок, все аргументы, переданные соответственной `yield`:

```
co = coroutine.create(function (a,b)
    coroutine.yield(a + b, a - b)
  end)
print(coroutine.resume(co, 20, 10))    --> true    30
10
```

Аналогично `yield` возвращает все дополнительные аргументы, переданные в соответственной `resume`:

```
co = coroutine.create (function (x)
    print("co1", x)
    print("co2", coroutine.yield())
  end)
coroutine.resume(co, "hi")    --> co1    hi
coroutine.resume(co, 4, 5)    --> co2    4    5
```

Наконец, когда сопрограмма завершается, любые значения, возвращенные ее главной функцией, передаются соответственной `resume`:

```
co = coroutine.create(function ()
    return 6, 7
  end)
print(coroutine.resume(co))    --> true    6    7
```

Обычно мы редко используем все эти средства в одной и той же

сопрограмме, но у всех из них есть свое применение.

Для тех, кто уже знает что-то о сопрограммах, важно прояснить некоторые понятия перед тем, как продолжить. Lua предлагает то, что называется *асимметричными сопрограммами*. Это значит, что у нее есть одна функция для приостановки выполнения сопрограммы и другая функция для возобновления приостановленной сопрограммы. Некоторые другие языки предлагают *симметричные сопрограммы*, когда есть лишь одна функция для передачи управления от одной сопрограммы к другой.

Некоторые называют асимметричные сопрограммы *полусопрограммами* (не будучи симметричными, они не являются полноценными *со-*). Однако, другие используют этот же термин для обозначения ограниченной реализации сопрограмм, где сопрограмма может приостановить свое выполнение, только когда она не вызывает никакую другую функцию, то есть когда у нее нет ожидающих вызовов в ее управляющем стеке. Другими словами, уступить управление может только главное тело такой полусопрограммы. Примером полусопрограмм в этом понимании являются *генераторы* в Python.

В отличие от разницы между симметричными и асимметричными сопрограммами, разница между сопрограммами и генераторами (в представлении Python) очень глубока; генераторы просто не достаточно мощные, чтобы реализовать некоторые интересные конструкции, которые мы можем писать с полноценными сопрограммами. Lua предлагает полноценные асимметричные сопрограммы. Те, кто предпочитают симметричные сопрограммы, могут реализовать их на основе асимметричных средств Lua. Это не сложная задача. (Фактически каждая передача управления выполняет `yield`, за которым следует `resume`.)

9.2. Каналы и фильтры

Одним из наиболее хрестоматийных примеров сопрограмм является проблема потребителя (producer) и производителя (consumer). Давайте представим, что у нас есть функция, которая постоянно производит значения (например, читает их из файла), и другая функция, которая постоянно потребляет эти значения (например, пишет в другой файл). Обычно эти две функции выглядят следующим образом:

```
function producer ()
  while true do
    local x = io.read()    -- производит новое
значение
    send(x)                -- отправляет его
потребителю
  end
end

function consumer ()
  while true do
    local x = receive()    -- получает значение от
производителя
    io.write(x, "\n")      -- потребляет его
  end
end
```

(В этой реализации и производитель, и потребитель выполняются вечно. Но их легко изменить, чтобы они останавливались, когда больше нет данных для обработки.) Задача здесь заключается в том, чтобы сопоставить `send` с `receive`. Это типичный образец проблемы «у кого главный цикл». И производитель, и потребитель активны, у обоих есть свои главные циклы, и каждый из них полагает, что другой является вызываемым сервисом. Для этого конкретного примера можно легко изменить структуру одной из функций, развернув ее цикл и сделав ее пассивным агентом. Однако, в других реальных случаях подобное изменение структуры может быть далеко не таким легким.

Сопрограммы предоставляют идеальный механизм для

сопоставления производителя с потребителем, поскольку пара `resume`—`yield` переворачивает типичное отношение между вызывающим и вызываемым. Когда сопрограмма вызывает `yield`, она не входит в новую функцию; вместо этого она возвращает управление ожидающему вызову (`resume`). Аналогично вызов `resume` не начинает новую функцию, а возвращает вызов `yield`. Это именно то, что нам нужно для сопоставления `send` с `receive` таким образом, чтобы каждый из них действовал так, будто главным является именно он, а второй является подчиненным. Поэтому `receive` возобновляет производителя, чтобы он мог произвести новое значение, а `send` посредством `yield` возвращает это значение обратно потребителю:

```
function receive ()
  local status, value = coroutine.resume(producer)
  return value
end

function send (x)
coroutine.yield(x)
end
```

Разумеется, теперь производитель должен быть сопрограммой:

```
producer = coroutine.create(
  function ()
    while true do
    local x = io.read()    -- производит новое
значение
      send(x)
  end
end)
```

При такой схеме программа начинает с вызова потребителя. Когда потребителю нужен какой-то элемент, он возобновляет работу производителя, который выполняется до тех пор, пока у него не будет элемента для передачи потребителю, а затем останавливается, пока потребитель снова его не возобновит. Таким образом, мы

получаем то, что называется схемой, *ориентированной на потребителя* (consumer-driven). Другим вариантом было бы написать программу с применением схемы, *ориентированной на производителя* (producer-driven), где потребитель является сопрограммой.

Мы можем расширить эту схему при помощи фильтров, которые являются заданиями, находящимися между производителем и потребителем и выполняющими своего рода преобразование данных. *Фильтр* — это производитель и потребитель в одно и то же время, поэтому он возобновляет производителя для получения новых значений и посредством `yield` передает эти преобразованные значения потребителю, В качестве простого примера мы можем добавить к нашему предыдущему коду фильтр, который в начало каждой строки вставляет ее номер. Код приведен в листинге 9.1. Этот последний кусочек просто создает нужные ему компоненты, соединяет их и начинает выполнение итогового потребителя:

```
p = producer()
f = filter(p)
consumer(f)
```

Или еще лучше:

```
consumer(filter(producer()))
```

Листинг 9.1. Потребитель и производитель с фильтрами

```
function receive (prod)
  local status, value = coroutine.resume(prod)
  return value
end

function send (x)
  coroutine.yield(x)
end

function producer ()
```

```
  return coroutine.create(function ()
    while true do
      local x = io.read()       -- производит новое
значение
      send(x)
    end
  end)
end

function filter (prod)
  return coroutine.create(function ()
    for line = 1, math.huge do
      local x = receive(prod)    -- получает новое
значение
      x = string.format("%5d %s", line, x)
      send(x)                    -- отправляет его
потребителю
    end
  end)
end

function consumer (prod)
  while true do
    local x = receive(prod)      -- получает новое
значение
    io.write(x, "\n")            -- потребляет новое
значение
  end
end
```

Если после прочтения предыдущего примера вы подумали о каналах (pipe) в UNIX, то вы не одиноки. В конце концов, сопрограммы — это разновидность (невытесняющей) многонитевости. С каналами каждая задача выполняется в отдельном процессе; с сопрограммами каждая задача выполняется в отдельной сопрограмме. Каналы предоставляют буфер между пишущим (производителем) и читающим (потребителем), поэтому возможна некоторая свобода в их относительных скоростях. Применительно к каналам это важно, поскольку цена переключения между процессами высока. С сопрограммами цена переключения

между задачами намного меньше (примерно равна вызову функции), поэтому пишущий и читающий могут идти нога в ногу.

9.3. Сопрограммы как итераторы

Мы можем рассматривать итераторы на основе цикла в качестве конкретного примера схемы производитель-потребитель: итератор производит элементы, потребляемые телом цикла. Поэтому вполне естественным будет использовать сопрограммы для написания итераторов. Действительно, сопрограммы являются эффективным инструментом для решения этой задачи. Опять же, ключевой особенностью является их способность вывернуть наизнанку отношение между вызывающим и вызываемым. С этой особенностью мы можем писать итераторы, не волнуясь о хранении состояния между последовательными вызовами итератора.

Чтобы проиллюстрировать этот вариант использования, давайте напишем итератор для перебора всех перестановок заданного массива. Написание подобного итератора напрямую не так легко, но несложно будет написать рекурсивную функцию, которая генерирует все эти перестановки. Идея проста: по очереди помещать каждый элемент массива на последнюю позицию и рекурсивно генерировать все перестановки оставшихся элементов. Код приведен в листинге 9.2. Чтобы он сработал, мы должны определить соответствующую функцию `printResult` и вызвать `permgen` с подходящими аргументами:

```
function printResult (a)
  for i = 1, #a do
    io.write(a[i], " ")
  end
  io.write("\n")
end

permgen ({1,2,3,4})
--> 2 3 4 1
```

```
--> 3 2 4 1
--> 3 4 2 1
  ...
--> 2 1 3 4
--> 1 2 3 4
```

Листинг 9.2. Функция для получения всех перестановок из первых **n** элементов **a**

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

Как только генератор готов, преобразовать его в итератор — простая задача. Во-первых, заменим `printResult` на `yield`:

```
function permgen (a, n)
  n = n or #a
  if n <= 1 then
    coroutine.yield(a)
  else
  <как прежде>
```

Затем мы определяем фабрику, которая делает так, чтобы генератор выполнялся внутри сопрограммы, а затем создаем итерирующую функцию. Для получения следующей перестановки итератор просто

возобновляет сопрограмму:

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

Имея в распоряжении такой механизм, перебрать все перестановки массива при помощи оператора **for** не составит труда:

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

Функция `permutations` использует типичную для Lua схему, которая пакует вызов `resume` с соответственной сопрограммой внутри функции. Эта схема настолько распространена, что Lua предоставляет для нее особую функцию: `coroutine.wrap`. Как и `create`, `wrap` создает новую сопрограмму. В отличие от `create`, `wrap` не возвращает саму сопрограмму; вместо этого она возвращает функцию, которая при вызове возобновляет эту сопрограмму. В отличие от исходной `resume`, она не возвращает код ошибки как свой первый результат; вместо этого при необходимости она вызывает ошибку. Используя `wrap`, мы можем написать `permutations` следующим образом:

```
function permutations (a)
  return coroutine.wrap(function () permgen(a) end)
end
```

Обычно использовать `coroutine.wrap` проще, чем `coroutine.create`. Она дает нам именно то, что нам нужно от сопрограммы: функцию для ее возобновления. Однако, она менее гибкая. Не существует способа проверить состояние сопрограммы, созданной при помощи `wrap`. Более того, мы не можем проверять на ошибки во время выполнения.

9.4. Невытесняющая многонитевость

Как мы видели ранее, сопрограммы обеспечивают разновидность совместной многонитевости. Каждая сопрограмма эквивалентна нити. Пара `yield`—`resume` переключает управление с одной нити на другую. Однако, в отличие от обычной многонитевости, сопрограммы не являются вытесняющими. Пока сопрограмма выполняется, она не может быть остановлена извне. Она прерывает свое выполнение, только когда явно запрашивает это (через вызов `yield`). Для ряда приложений это не является проблемой, скорее наоборот. Программирование гораздо проще в отсутствие вытеснения. Вам не нужно беспокоиться об ошибках синхронизации, поскольку вся синхронизация среди нитей в вашей программе явная. Вам лишь нужно убедиться в том, что сопрограмма вызывает `yield` вне критической области кода.

Однако, при невытесняющей многонитевости, как только какая-то нить вызывает блокирующую операцию, вся программа блокируется до тех пор, пока эта операция не завершится. Для большинства приложений это недопустимое поведение, которое привело к тому, что многие программисты не рассматривают сопрограммы как альтернативу традиционной многонитевости. Как мы здесь увидим, у этой проблемы есть интересное (и очевидное, если заглянуть в прошлое) решение.

Давайте рассмотрим типичную многонитевую задачу: мы хотим скачать несколько удаленных файлов по HTTP. Для скачивания

нескольких удаленных файлов сначала мы должны разобраться, как скачать один удаленный файл. В этом примере мы используем разработанную Диего Нехабом библиотеку *LuaSocket*. Для скачивания файла сперва нужно установить соединение с его сайтом, отправить запрос на файл, получить этот файл (блоками) и закрыть соединение. На Lua мы можем написать это следующим образом. Для начала мы загружаем библиотеку LuaSocket:

```
local socket = require "socket"
```

Затем мы определяем хост и файл, который хотим скачать. В этом примере мы скачаем справочное руководство по HTML 3.2 с сайта консорциума World Wide Web:

```
host = "www.w3.org"
file = "/TR/REC-html32.html"
```

Затем мы открываем TCP-соединение с портом 80 (стандартный порт для HTTP-соединений) данного сайта:

```
c = assert(socket.connect(host, 80))
```

Эта операция возвращает объект соединения, который мы используем для отправки запроса на получение файла:

```
c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
```

Затем мы читаем файл блоками по 1 Кб, записывая каждый блок в стандартный вывод:

```
while true do
  local s, status, partial = c:receive(2^10)
  io.write(s or partial)
  if status == "closed" then break end
end
```

Функция `receive` возвращает или строку, которую прочла, или nil в случае ошибки; в последнем случае она также возвращает код ошибки (`status`) и что она прочла до ошибки (`partial`). Когда хост

закрывает соединение, мы печатаем оставшиеся входные данные и прерываем цикла приема данных.

После скачивания файла мы закрываем соединение:

```
c:close()
```

Теперь, когда мы знаем, как скачать один файл, давайте вернемся к проблеме скачивания нескольких файлов. Простейшим подходом будет скачивать по одному файлу за раз. Однако, этот последовательный подход, когда мы начинаем читать файл только после того, как закончим с предыдущим, слишком медленный. При чтении удаленного файла программа проводит основную часть времени, ожидая прибытия данных. Более точно, она проводит большую часть времени блокированной в вызове `receive`. Поэтому программа может выполняться гораздо быстрее, если будет скачивать все файлы параллельно. Тогда, когда у соединения нет доступных данных, программа может читать их из другого соединения. Понятно, что сопрограммы предлагают удобный способ для организации этих одновременных скачиваний. Мы создаем новую нить для каждой задачи приема данных. Когда у нити нет доступных данных, она уступает управление простому диспетчеру, который вызывает другую нить.

Чтобы переписать программу с применением сопрограмм, для начала нужно переписать предыдущий код для скачивания как функцию. Результат приведен в листинге 9.3. Поскольку нам не интересно содержимое удаленного файла, функция подсчитывает и печатает размер файла вместо записи файла в стандартный вывод. (С несколькими нитями, читающими сразу несколько файлов, на выходе получилась бы полная мешанина).

Листинг 9.3. Код для скачивания веб-страницы

```
function download (host, file)
  local c = assert(socket.connect(host, 80))
  local count = 0 -- counts number of bytes read
```

```
  c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
  while true do
    local s, status = receive(c)
    count = count + #s
    if status == "closed" then break end
  end
  c:close()
  print(file, count)
end
```

В этом новом коде мы используем вспомогательную функцию (`receive`) для получения данных из соединения. При последовательном подходе код выглядел бы следующим образом:

```
function receive (connection)
  local s, status, partial = connection:receive(2^10)
  return s or partial, status
end
```

Для параллельной реализации эта функция должна получать данные без блокирования. Вместо этого, если данных не достаточно, она уступает управление. Ее новый код выглядит следующим образом:

```
function receive (connection)
  connection:settimeout(0)    -- не блокирует данные
  local s, status, partial = connection:receive(2^10)
  if status == "timeout" then
    coroutine.yield(connection)
  end
  return s or partial, status
end
```

Вызов `settimeout(0)` делает любую операцию над соединением неблокирующей. Когда статус операции равен "`timeout`", это значит, что операция вернула управление, не выполнив свою задачу. В этом случае нить уступает управление. Отличный от false аргумент, переданный `yield`, сообщает диспетчеру, что данная нить все еще выполняет свою задачу. Обратите внимание, что даже в случае статуса "`timeout`" в переменной `partial` все равно содержатся

прочитанные ранее данные, которое возвратило соединение.

Листинг 9.4. содержит код диспетчера и некоторый дополнительный код. Таблица `threads` содержит список всех активных нитей для диспетчера. Функция `get` следит за тем, чтобы каждая загрузка выполнялась в отдельной нити. Сам диспетчер в основном является циклом, который перебирает все нити, возобновляя их одну за другой. Также он должен удалять из списка те нити, которые уже завершили свои задачи. Цикл останавливается, когда больше нет нитей для выполнения.

Листинг 9.4. Диспетчер

```
threads = {}    -- список всех живых нитей

function get (host, file)
  -- создает сопрограмму
  local co = coroutine.create(function ()
    download(host, file)
  end)
  -- вставляет ее в список
  table.insert(threads, co)
end

function dispatch ()
  local i = 1
  while true do
    if threads[i] == nil then    -- нитей больше нет?
      if threads[1] == nil then break end    -- список
пуст?
      i = 1                      -- перезапускает цикл
    end
    local status, res = coroutine.resume(threads[i])
    if not res then              -- нить выполнила
свою задачу?
      table.remove(threads, i)
    else
      i = i + 1                  -- перейти к
следующей нити
    end
  end
end
```

Наконец, главная программа создает требуемые нити и вызывает диспетчер. Например, чтобы загрузить четыре документа с сайта W3C, главная программа может выглядеть, как показано ниже:

```
host = "www.w3.org"

get(host, "/TR/html401/html40.txt")
get(host, "/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")
get(host, "/TR/REC-html32.html")
get(host, "/TR/2000/REC-DOM-Level-2-Core-
20001113/DOM2-Core.txt")

dispatch()    -- главный цикл
```

На моем компьютере скачивание этих четырех файлов с использованием сопрограмм занимает 6 секунд. С последовательным скачиванием это требует в два с половиной раза больше времени (15 секунд).

Несмотря на такой прирост скорости, эта последняя реализация далека от оптимальной. Все работает хорошо до тех пор, пока хотя бы у одной нити есть что читать. Однако, когда ни у одной нити нет данных для чтения, диспетчер находится в активном ожидании, постоянно переключаясь с нити на нить лишь, чтобы убедиться в том, данных у них по-прежнему нет. В результате эта реализация сопрограммы потребляет почти в 30 раз больше процессорного времени, чем последовательная версия.

Чтобы избежать подобного поведения, мы можем использовать функцию `select` из библиотеки LuaSocket: она позволяет заблокировать программу, находящуюся в ожидании, пока изменится статус в группе сокетов. Изменения в реализации незначительны: нам нужно изменить только диспетчер, как показано в листинге 9.5. В цикле новый диспетчер собирает в таблице `timedout` соединения, у которых превышено время ожидания. (Помните, что `receive` передает подобные соединения функции `yield`, поэтому вызов `resume` их возвращает). Если у всех

соединений превышено время ожидания, то диспетчер вызывает `select`, чтобы ждать изменения статуса у одного из этих. Эта окончательная реализация работает так же быстро, как и предыдущая с сопрограммами. Более того, без активного ожидания она потребляет лишь немногим больше процессорного времени, чем последовательная реализация.

Листинг 9.5. Диспетчер, использующий **`select`**

```
function dispatch ()
  local i = 1
  local timedout = {}
  while true do
    if threads[i] == nil then          -- больше нет
нитей?
      if threads[1] == nil then break end
      i = 1                            --
перезапускает цикл
      timedout = {}
    end
    local status, res = coroutine.resume(threads[i])
    if not res then                    -- нить
выполнила свою задачу?
      table.remove(threads, i)
    else                               -- превышено
время ожидания
      i = i + 1
      timedout[#timedout + 1] = res
      if #timedout == #threads then    -- все нити
заблокированы?
        socket.select(timedout)
      end
    end
  end
end
```

Упражнения

Упражнение 9.1. Используйте сопрограммы для

преобразования функции из упражнения 5.4 в генератор для комбинаций, который может быть использован следующим образом:

```
for c in combinations({"a", "b", "c"}, 2) do
  printResult(c)
end
```

Упражнение 9.2. Реализуйте и запустите код из предыдущего раздела (невытесняющая многонитевость).

Упражнение 9.3. Реализуйте функцию `transfer` на Lua. Если подумать о том, что вызовы `resume`—`yeild` аналогичны вызову функции и возврату из нее, то эта функция будет как **goto**: она приостанавливает выполняемую сопрограмму и возобновляет любую другую сопрограмму, заданную в качестве аргумента. (Подсказка: используйте что-то вроде `dispatch` для управления вашими сопрограммами. Тогда `transfer` уступит управление `dispatch`, сообщая о том, какую следующую сопрограмму нужно выполнять, и `dispatch` возобновит ее.)

ГЛАВА 10

Законченные примеры

Чтобы завершить введение в язык, мы покажем три хоть и простых, но законченных примера программ. Первый пример о проблеме восьми королев. Второй является программой частотности слов, которая печатает самые часто встречающиеся слова в тексте. Последний пример — это реализация алгоритма цепи Маркова, описанная Керниганом и Пайком (Kernighan & Pike) в их книге *«The Practice of Programming»* (Addison-Wesley, 1999).

10.1. Задача о восьми королевах

Наш первый пример — это очень простая программа, которая решает *задачу о восьми королевах*: цель состоит в размещении восьми королев на шахматной доске таким образом, чтобы ни одна из королев не могла напасть на другую.

Первый шаг решения данной проблемы состоит в том, чтобы отметить, что у любого правильного решения должно быть ровно по одной королеве в каждой строке. Таким образом, мы можем представить решения при помощи простого массива из восьми чисел, по одному для каждой строки; каждое число сообщает нам, в каком столбце расположена королева из каждой строки. Например, массив `{3,7,2,1,8,6,5,4}` обозначает, что одна королева находится в строке 1 и столбце 3, другая — в строке 2 и столбце 7 и т. д. (Кстати, это не правильное решение; например, королева в строке 3 и столбце 2 нападает на королеву в строке 4 и столбце 1). Обратите внимание, что любое правильное решение должно быть перестановкой целых чисел от 1 до 8, так как правильное решение также должно содержать по одной королеве в каждом столбце.

Полная программа приведена в листинге 10.1. Первой функцией является `isplaceok`, которая проверяет, что заданная позиция на доске не может быть атакована ранее размещенными королевами. Зная о том, что по формулировке две королевы не могут находиться на одной строке, функция `isplaceok` проверяет, что нет других королев в том же столбце или той же диагонали, что у новой позиции.

Листинг 10.1. Программа «восемь королев»

```
local N = 8 -- board size

-- проверяет, что позиция (n,c) находится вне атак
local function isplaceok (a, n, c)
  for i = 1, n - 1 do    -- для каждой уже размещенной
королевы
    if (a[i] == c) or              -- тот же столбец?
       (a[i] - i == c - n) or      -- та же диагональ?
       (a[i] + i == c + n) then    -- та же диагональ?
      return false       -- позиция может быть
атакована
    end
  end
  return true            -- никаких атак; позиция в
порядке
end

-- печатает доску
local function printsolution (a)
  for i = 1, N do
    for j = 1, N do
      io.write(a[i] == j and "X" or "-", " ")
    end
    io.write("\n")
  end
  io.write("\n")
end

-- добавляет на доску 'a' всех королев от 'n' до 'N'
local function addqueen (a, n)
  if n > N then     -- были размещены все королевы?
    printsolution(a)
```

```
  else                -- пытается разместить n-ую
королеву
    for c = 1, N do
      if isplaceok(a, n, c) then
        a[n] = c      -- помещает n-ую королеву в
столбце 'c'
        addqueen(a, n + 1)
      end
    end
  end
end

-- запускает программу
addqueen({}, 1)
```

Далее у нас идет функция `printsolution`, которая печатает доску. Она просто обходит всю доску, печатая '`X`' на позициях с королевой и '`-`' на других позициях. Каждый результат выглядит примерно так:

```
X - - - - - - -
- - - - X - - -
- - - - - - - X
- - - - - X - -
- - X - - - - -
- - - - - - X -
- X - - - - - -
- - - X - - - -
```

Последняя функция `addqueen` является ядром программы. Она применяет отслеживание в обратном порядке для поиска правильных решений. Сначала она проверяет, является ли решение законченным, и если да, то печатает это решение. В противном случае она перебирает все столбцы; для каждого столбца, свободного от атак, программа помещает туда королеву и рекурсивно пытается разместить оставшихся королев.

Наконец, главное тело программы просто вызывает `addqueen` для начала решения задачи.

10.2. Самые часто встречающиеся слова

Наш следующий пример — это простая программа, которая читает текст и печатает самые часто встречающиеся в этом тексте слова.

Главная структура данных этой программы является простой таблицей, которая связывает каждое слово в тексте с его счетчиком частотности. С этой структурой данных у программы есть три основные задачи:

- Прочесть текст, посчитав число вхождений каждого слова.
- Отсортировать список слов в нисходящем порядке по их частотности.
- Напечатать первые *n* элементов отсортированного списка.

Для чтения текста мы можем использовать итератор `allwords`, который мы разработали в разделе 7.1. Для каждого прочитанного слова мы увеличиваем его соответствующий счетчик:

```
local counter = {}
for w in allwords do
  counter[w] = (counter[w] or 0) + 1
end
```

Обратите внимание на трюк с **or** для обработки неинициализированных счетчиков.

Следующим шагом будет сортировка списка слов. Однако, как внимательный читатель мог заметить, у нас нет списка слов для сортировки! Несмотря на это, его легко создать, используя слова, которые представлены ключами в таблице `counter`:

```
local words = {}
for w in pairs(counter) do
  words[#words + 1] = w
end
```

Как только мы получим этот список, мы можем отсортировать его при помощи предопределенной функции `table.sort`, которую мы кратко обсуждали в главе 6:

```
table.sort(words, function (w1, w2)
  return counter[w1] > counter[w2] or
         counter[w1] == counter[w2] and w1 < w2
end)
```

Слова с наибольшими значениями счетчиков идут первыми; слова с равными значениями счетчиков идут в алфавитном порядке.

Законченная программа приведена в листинге 10.2. Обратите внимание на применение сопрограммы для выворачивания наизнанку итератора `auxwords`, который использован в следующем цикле. В последнем цикле, печатающем результат, программа считает, что ее первый аргумент — это число слов, которое нужно напечатать, и по умолчанию использует значение 10, если аргументов передано не было.

Листинг 10.2. Программа «частотность слов»

```
local function allwords ()
  local auxwords = function ()
    for line in io.lines() do
      for word in string.gmatch(line, "%w+") do
        coroutine.yield(word)
      end
    end
  end
  return coroutine.wrap(auxwords)
end

local counter = {}
for w in allwords() do
  counter[w] = (counter[w] or 0) + 1
end

local words = {}
for w in pairs(counter) do
  words[#words + 1] = w
```

```
end

table.sort(words, function (w1, w2)
  return counter[w1] > counter[w2] or
         counter[w1] == counter[w2] and w1 < w2
end)

for i = 1, (tonumber(arg[1]) or 10) do
  print(words[i], counter[words[i]])
end
```

10.3. Алгоритм цепи Маркова

Наш последний пример — это реализация *алгоритма цепи Маркова*. Программа генерирует псевдослучайный текст на основании того, какие слова могут следовать за последовательностью из *n* предыдущих слов в базовом тексте. Для этой реализации мы будем считать, что *n* равно 2.

Первая часть читает базовый текст и строит таблицу, которая для каждого префикса из двух слов дает список всех слов, которые в тексте следуют за этим префиксом. После построения таблицы программа использует ее для генерации случайного текста, где каждое слово следует за двумя предыдущими с той же вероятностью, что и в базовом тексте. В результате мы получаем случайный на вид текст. Например, применив эту программу к английскому тексту данной книги, мы получим фрагменты вроде *«Constructors can also traverse a table constructor, then the parentheses in the following line does the whole file in a field n to store the contents of each function, but to show its only argument. If you want to find the maximum element in an array can return both the maximum value and continues showing the prompt and running the code. The following words are reserved and cannot be used to convert between degrees and radians»*.

Мы будем кодировать каждый префикс при помощи его двух

слов, соединенных посредством пробела:

```
function prefix (w1, w2)
  return w1 .. " " .. w2
end
```

Мы воспользуемся строкой `NOWORD` (перевод строки) для инициализации префиксных слов и обозначения конца текста. Например, для текста "`the more we try the more we do`" таблица последующих слов выглядела бы таким образом:

```
{ ["\n \n"]    = {"the"},
  ["\n the"]   = {"more"},
  ["the more"] = {"we", "we"},
  ["more we"]  = {"try", "do"},
  ["we try"]   = {"the"},
  ["try the"]  = {"more"},
  ["we do"]    = {"\n"},
}
```

Программа хранит свою таблицу в переменной `statetab`. Для вставки нового слова в список префиксов данной таблицы, мы используем следующую функцию:

```
function insert (index, value)
  local list = statetab[index]
  if list == nil then
    statetab[index] = {value}
  else
    list[#list + 1] = value
  end
end
```

Сначала она проверяет, есть ли уже у данного префикса список; если нет, то создает новый список с новым значением. В противном случае она вставляет новое значение в конец существующего списка.

Для построения таблицы `statetab` мы будем использовать две переменные `w1` и `w2`, содержащие два последних прочитанных слова. Для каждого нового прочитанного слова мы добавляем его к

списку, связанному с `w1` и `w2`, а затем обновляем `w1` и `w2`.

После построения таблицы программа начинает генерировать текст, состоящий из `MAXGEN` слов. Для начала она заново инициализирует переменные `w1` и `w2`. Затем для каждого префикса она наугад выбирает следующее слово из списка допустимых последующих слов, печатает это слово и обновляет `w1` и `w2`. Листинги 10.3 и 10.4 содержат законченную программу. В отличие от нашего предыдущего примера с наиболее часто встречающимися словами, здесь мы используем реализацию `allwords`, основанную на замыканиях.

Листинг 10.3. Вспомогательные определения для программы «цепь Маркова»

```
function allwords ()
  local line = io.read()    -- текущая строка
  local pos = 1             -- текущая позиция в
строке
  return function ()        -- итерирующая функция
    while line do           -- повторяет, пока есть
строки
      local s, e = string.find(line, "%w+", pos)
      if s then                          -- словой
найдено?
        pos = e + 1                      -- обновляет
следующую позицию
        return string.sub(line, s, e)    -- возвращает
слово
      else
        line = io.read()    -- слово не найдено;
пробует следующую строку
        pos = 1             -- перезапуск с первой
позиции
      end
    end
    return nil              -- строк больше нет; конец
обхода
  end
end
```

```
function prefix (w1, w2)
  return w1 .. " " .. w2
end

local statetab = {}

function insert (index, value)
  local list = statetab[index]
  if list == nil then
    statetab[index] = {value}
  else
    list[#list + 1] = value
  end
end
```

Листинг 10.4. Программа «цепь Маркова»

```
local N = 2
local MAXGEN = 10000
local NOWORD = "\n"

-- строит таблицу
local w1, w2 = NOWORD, NOWORD
for w in allwords() do
  insert(prefix(w1, w2), w)
  w1 = w2; w2 = w;
end
insert(prefix(w1, w2), NOWORD)

-- генирирует текст
w1 = NOWORD; w2 = NOWORD     -- новая инициализация
for i = 1, MAXGEN do
  local list = statetab[prefix(w1, w2)]
  -- наугад выбирает элемент из списка
  local r = math.random(#list)
  local nextword = list[r]
  if nextword == NOWORD then return end
  io.write(nextword, " ")
  w1 = w2; w2 = nextword
end
```

Упражнения

Упражнение 10.1. Измените программу с восьмью королевами, чтобы она останавливалась после печати первого решения.

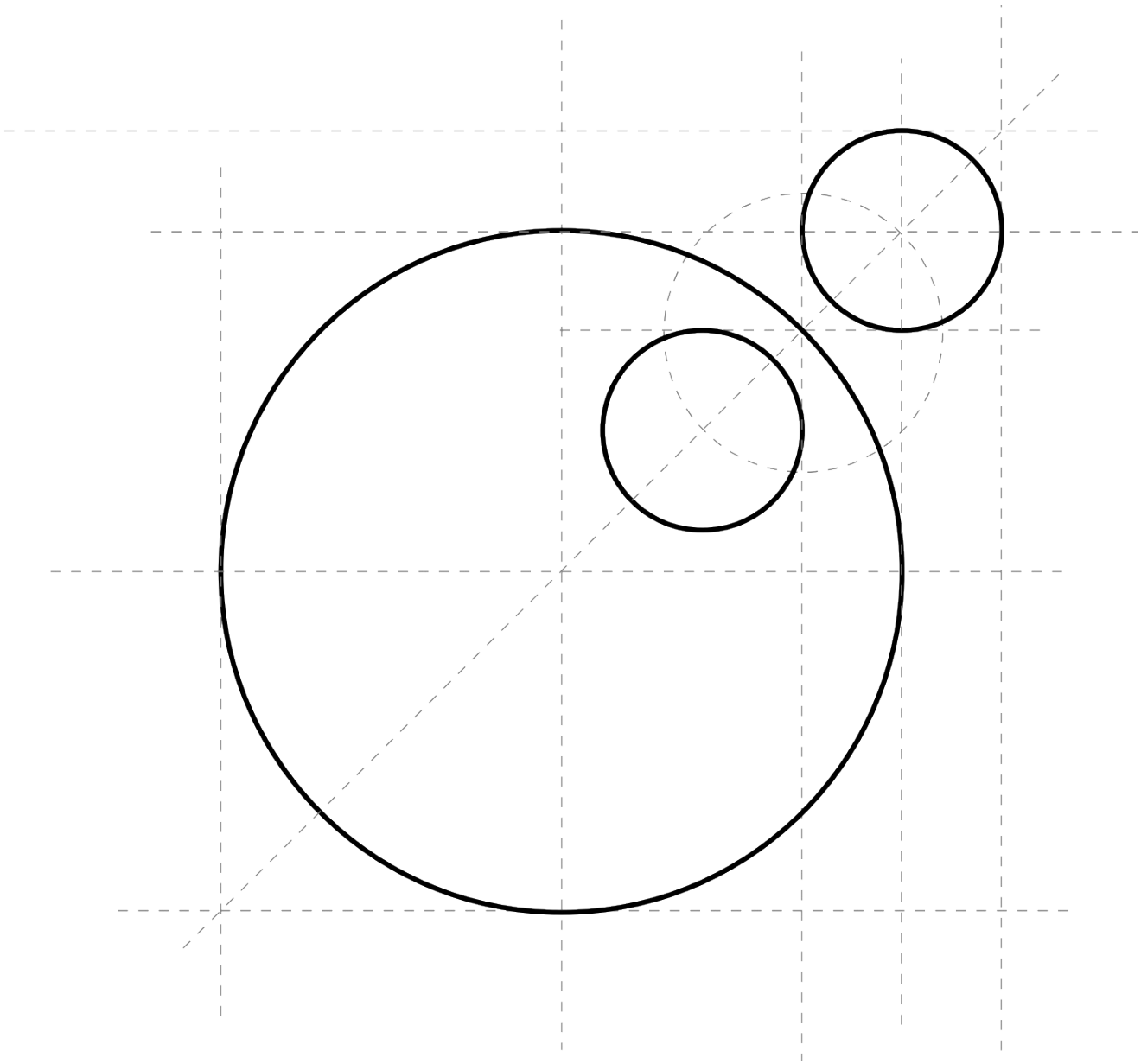
Упражнение 10.2. Альтернативной реализацией задачи о восьми королевах было бы построение всех возможных перестановок чисел от 1 до 8 и проверка на правильность для каждой такой перестановки. Измените программу с применением этого подхода. Как отличается быстродействие новой программы по сравнению со старой? (Подсказка: сравните полное число перестановок с числом раз, когда исходная программа вызывает функцию `isplaceok`.)

Упражнение 10.3. Когда мы применяем программу частотности слов к тексту, то обычно самыми часто встречаемыми словами оказываются короткие неинтересные слова вроде артиклей и предлогов. Измените программу так, чтобы она пропускала слова, состоящие менее чем из четырех букв.

Упражнение 10.4. Обобщите алгоритм цепи Маркова так, чтобы он мог использовать любой размер для последовательности предыдущих слов, используемой при выборе следующего слова.

Часть II

Таблицы и объекты



ГЛАВА 11

Структуры данных

Таблицы в Lua — это не одна из структур данных, — это *единственная* структура данных. Все структуры, которые предлагают другие языки, — массивы, записи, списки, очереди, множества — могут быть представлены в Lua при помощи таблиц. Главное, таблицы Lua эффективно реализуют все эти структуры.

В традиционных языках, таких как C и Pascal, мы реализуем большинство структур данных при помощи массивов и списков (где списки = записи + указатели). Хотя мы можем реализовать массивы и списки при помощи таблиц Lua (и иногда мы это делаем), таблицы гораздо мощнее массивов и списков; многие алгоритмы с использованием таблиц становятся до банальности простыми. Например, мы редко пишем алгоритм поиска в Lua, поскольку таблицы предоставляют прямой доступ к любому типу.

Требуется некоторое время, чтобы понять, как эффективно использовать таблицы. В этой главе я покажу, как реализовать типичные структуры данных при помощи таблиц, и приведу некоторые примеры их использования. Мы начнем с массивов и списков не потому, что они понадобятся нам для других структур, а потому, что большинство программистов уже знакомы с ними. Мы уже знакомы с азами данного материала в главах о языке, но я также повторю их здесь для полноты.

11.1. Массивы

Мы реализуем массивы в Lua, просто индексируя таблицы целыми числами. Таким образом, массивы не имеют фиксированного размера и растут по мере необходимости. Обычно

при инициализации массива мы неявно определяем его размер. Например, после выполнения следующего кода любая попытка обратиться к полю вне диапазона 1—1000 вернет nil вместо 0:

```
a = {}      -- новый массив
for i = 1, 1000 do
  a[i] = 0
end
```

Операция длины ('#') использует данный факт для нахождения размера массива:

```
print(#a)    --> 1000
```

Вы можете начать массив с нуля, единицы или любого другого значения:

```
-- создает массив с индексами от -5 до 5
a = {}
for i = -5, 5 do
  a[i] = 0
end
```

Однако, в Lua принято начинать массивы с индекса 1. Библиотеки Lua придерживаются этой традиции, как и операция длины. Если ваши массивы не начинаются с 1, то вы не сможете использовать данные средства.

Мы можем использовать конструктор для создания и инициализации массивов в едином выражении:

```
squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Подобные конструкторы могут быть настолько большими, насколько это нужно (по крайней мере, до нескольких миллионов элементов).

11.2. Матрицы и многомерные массивы

Есть два основных способа представления матриц в Lua. Первый — это использовать массив массивов, то есть таблицу, каждый элемент которой является другой.таблицей. Например, вы можете создать матрицу из нулей размером `N` на `M` при помощи следующего кода:

```
mt = {}          -- создает матрицу
for i = 1, N do
  mt[i] = {}     -- создает новую строку
  for j = 1, M do
    mt[i][j] = 0
  end
end
```

Поскольку таблицы являются объектами в Lua, для создания матрицы вы должны явно создавать каждую строку. С одной стороны, это определенно более громоздко, чем просто объявить матрицу, как это делается в языках C и Pascal. С другой стороны, это дает больше гибкости. Например, вы можете создать треугольную матрицу, изменив цикл `for j=1,M do ... end` в предыдущем примере на `for j=1,i do ... end`. С этим кодом треугольная матрица использует лишь половину памяти по сравнению с исходной.

Второй способ представления матриц в Lua заключается в объединении двух индексов в один. Если оба индекса являются целыми числами, то вы можете просто умножить первый на соответствующую константу и затем добавить второй индекс. С этим подходом следующий код создаст нашу матрицу из нулей размером `N` на `M`:

```
mt = {}     -- создает матрицу
for i = 1, N do
  for j = 1, M do
    mt[(i - 1)*M + j] = 0
  end
end
```

Если индексы являются строками, то вы можете создать один индекс, соединив эти строки с некоторым символом между ними. Например, вы можете индексировать матрицу `m` со строковыми индексами `s` и `t` при помощи кода `m[s..":"..t]`, при условии, что и `s`, и `t` не содержат двоеточия; в противном случае пары вроде `("a:","b")` и `("a",":b")` сольются в один индекс "`a::b`". Когда сомневаетесь, можете использовать управляющий символ вроде '`\0`' для разделения индексов.

Довольно часто приложения используют *разреженную матрицу*, то есть матрицу, где большинство элементов — ноль или nil. Например, вы можете представить граф при помощи его матрицы смежности, в которой значение на позиции `m,n` равно `x`, если между узлами `m` и `n` есть соединение ценой `x`. Когда эти узлы не соединены, то значение на позиции `m,n` равно nil. Чтобы представить граф с десятью тысячами узлов, где каждый узел имеет около пяти соседей, вам нужна матрица со ста миллионами элементов (квадратная матрица из 10 000 столбцов и 10 000 строк), но только примерно пятьдесят тысяч из них будут не равны nil (пять ненулевых столбцов для каждой строки, соответствующих пяти соседям каждого узла). Многие книги по структурам данных пространно обсуждают, как можно реализовать подобные разреженные матрицы, не тратя на них 400 Мб памяти, но вам редко понадобятся эти приемы при программировании на Lua. Так как массивы представлены таблицами, они могут быть разрежены естественным образом. С нашим первым представлением (таблица таблиц) вам понадобятся десять тысяч таблиц, каждая из которых содержит около пяти элементов, то есть всего около пятидесяти тысяч элементов. Со вторым представлением у вас будет одна таблица с пятьюдесятью тысячами элементов. Независимо от представления, вам понадобится память только для элементов, отличных от nil.

При работе с разреженными матрицами мы не можем использовать операцию длины из-за дырок (значений nil) между

активными элементами. Однако, это не большая потеря; даже если бы мы могли его использовать, то делать этого не стоило бы. Для большинства операций было бы крайне неэффективно перебирать все эти пустые элементы. Вместо этого мы можем использовать `pairs` для обхода только элементов, отличных от nil. Например, чтобы умножить строку на константу, мы можем использовать следующий код:

```
function mult (a, rowindex, k)
  local row = a[rowindex]
  for i, v in pairs(row) do
    row[i] = v * k
  end
end
```

Однако, примите к сведению, что у ключей нет внутреннего порядка в таблице, поэтому итерация при помощи `pairs` не гарантирует, что мы посетим все столбцы по возрастанию. Для некоторых задач (вроде нашего предыдущего примера) это не проблема. Для других задач вы можете использовать альтернативный подход, например, связанные списки.

11.3. Связанные списки

Поскольку таблицы являются динамическими сущностями, то реализовать связанные списки в Lua довольно легко. Каждый узел представлен таблицей, а ссылки являются просто полями таблицы, которые содержат ссылки на другие таблицы. Например, давайте реализуем базовый список, где каждый узел содержит два поля, `next` и `value`. Корнем списка является простая переменная:

```
list = nil
```

Для вставки элемента со значением `v` в начало списка, мы делаем:

```
list = {next = list, value = v}
```

Для перебора списка мы пишем:

```
local l = list
while l do
  <проверить l.value>
  l = l.next
end
```

Другие разновидности списков, например двунаправленные или кольцевые списки, также легко реализуются. Однако, подобные структуры вам редко понадобятся в Lua, поскольку обычно есть более простой способ представления ваших данных без использования связанных списков. Например, мы можем представить стек в виде (неограниченного) массива.

11.4. Очереди и двойные очереди

Простейшим способом реализации очередей в Lua является использование функций `insert` и `remove` из библиотеки `table`. Эти функции вставляют и удаляют элементы из произвольной позиции массива, сдвигая остальные элементы для согласования действий. Однако, подобные перемещения могут быть дорогими для больших структур. Более эффективная реализация использует две индекса, один для первого элемента и один для последнего:

```
function ListNew ()
  return {first = 0, last = -1}
end
```

Во избежание загрязнения глобального пространства имен мы определим все операции со списком внутри таблицы, которую мы соответственно назовем `List` (таким образом, мы создадим *модуль*). Тогда мы перепишем наш последний пример следующим образом:

```
List = {}
function List.new ()
```

```
  return {first = 0, last = -1}
end
```

Теперь мы можем вставлять и удалять элементы с обоих концов за постоянное время:

```
function List.pushfirst (list, value)
  local first = list.first - 1
  list.first = first
  list[first] = value
end

function List.pushlast (list, value)
  local last = list.last + 1
  list.last = last
  list[last] = value
end

function List.popfirst (list)
  local first = list.first
  if first > list.last then error("list is empty") end
  local value = list[first]
  list[first] = nil     -- чтобы разрешить сборку
мусора
  list.first = first + 1
  return value
end

function List.poplast (list)
  local last = list.last
  if list.first > last then error("list is empty") end
  local value = list[last]
  list[last] = nil     -- чтобы разрешить сборку
мусора
  list.last = last - 1
  return value
end
```

Если вы будете использовать эту структуру для обслуживания в порядке поступления, вызывая только `pushlast` и `popfirst`, то и `first`, и `last` будут постоянно расти. Однако, так как мы представляем массивы в Lua при помощи таблиц, вы можете

индексировать их как с 1 до 20, так и с 16 777 216 до 16 777 236. Поскольку Lua использует для представления чисел двойную точность, ваша программа можем выполняться на протяжении двухсот лет, делая по миллиону вставок в секунду, прежде чем возникнет проблема с переполнением.

11.5. Множества и мультимножества

Предположим, вы хотите составить список всех идентификаторов, используемых в программе; каким-то образом вам нужно отфильтровывать зарезервированные слова при составлении вашего списка. Некоторые программисты на C могут поддаться искушению представить множество зарезервированных слов в виде массива строк и затем для проверки обыскивать этот массив, чтобы узнать, есть ли заданное слово в этом множестве. Чтобы ускорить поиск, для представления множества они даже могут воспользоваться бинарным деревом.

В Lua эффективным и простым способом представить такие множества будет поместить их элементы в таблицу в качестве *индексов*. Тогда вместо поиска заданного элемента в таблице вы всего лишь индексируете эту таблицу и проверяете, равен ли полученный результат nil. В нашем примере мы могли бы написать следующий код:

```
reserved = {
  ["while"] = true,     ["end"] = true,
  ["function"] = true,  ["local"] = true,
}

for w in allwords() do
  if not reserved[w] then
    <сделать что-нибудь с 'w'>     -- 'w' — не
зарезервированное слово
  end
end
```

(Поскольку эти слова зарезервированы в Lua, мы не можем использовать их в качестве идентификаторов; например, мы не можем написать `while=true`. Вместо этого мы пишем `["while"]=true`.)

Вы можете получить более понятную инициализацию при помощи дополнительной функции для построения множества:

```
function Set (list)
  local set = {}
  for _, l in ipairs(list) do set[l] = true end
  return set
end

reserved = Set{"while", "end", "function", "local", }
```

Мультимножество (bag) отличается от обычных множеств тем, что каждый элемент может встречаться несколько раз. Простое представление мультимножеств в Lua похоже на предыдущее представление для множеств, но для каждого ключа есть связанный с ним счетчик. Чтобы вставить элемент, мы увеличиваем его счетчик на единицу:

```
function insert (bag, element)
  bag[element] = (bag[element] or 0) + 1
end
```

Для удаления элемента мы уменьшаем его счетчик на единицу:

```
function remove (bag, element)
  local count = bag[element]
  bag[element] = (count and count > 1) and count - 1
or nil
end
```

Мы храним счетчик, только если он уже существует и по-прежнему больше нуля.

11.6. Строковые буферы

Допустим, вы строите строку по частям, например, построчно читая файл. Ваш код выглядел бы примерно следующим образом:

```
local buff = ""
for line in io.lines() do
  buff = buff .. line .. "\n"
end
```

Несмотря на его безобидный вид, этот код может сильно ударить по быстродействию для больших файлов: например, чтение файла размером 1 Мб занимает 1,5 минуты на моем старом Пентиуме (Примечание: «Мой старый Пентиум» — это компьютер с одноядерным 32-битовым Pentium частотой 3 ГГц. Все данные о быстродействии в этой книге получены на нем).

Почему так? Чтобы понять, что происходит, представим, что мы находимся в середине цикла чтения; каждая строка из файла состоит из 20 байтов, и мы уже прочли 2 500 этих строк, поэтому `buff` — это строка размером 50 Кб. Когда Lua соединяет `buff..line.."\n"`; она выделяет новую строку размером 50 020 байт и копирует 50 000 байт из `buff` в эту новую строку. Таким образом, для каждой новой строки из файла Lua перемещает в памяти примерно 50 Кб, и этот размер только растет. Более точно, этот алгоритм имеет квадратичную сложность. После прочтения 100 новых строк (всего 2 Кб) из файла Lua уже переместил более 2 Мб памяти. Когда Lua завершит чтение 350 Кб, он переместит более 50 Гб. (Эта проблема свойственна не только Lua: другие языки, где строки неизменяемы, также сталкиваются с подобной проблемой. Наиболее известным примером такого языка является Java).

Прежде чем мы продолжим, необходимо заметить, что, несмотря на все сказанное, данная ситуация не является распространенной проблемой. Для маленьких строк вышеприведенный цикл отлично работает. Для чтения всего файла Lua предоставляет опцию `io.read("*a")`, с которой файл читается за один раз. Однако, иногда от этой проблемы никуда не деться. Для

борьбы с ней Java использует структуру `stringBuffer`. В Lua в качестве строкового буфера мы можем использовать таблицу. В основе этого подхода лежит функция `table.concat`, которая возвращает результат конкатенации всех строк из заданного списка. При помощи `concat` мы можем переписать наш предыдущий цикл следующим образом:

```
local t = {}
for line in io.lines() do
  t[#t + 1] = line .. "\n"
end
local s = table.concat(t)
```

Этот алгоритм требует менее 0,5 секунды для чтения того же самого файла, которому требовалась почти минута с первоначальным кодом. (Несмотря на это, для чтения всего файла по-прежнему лучше использовать `io.read` с опцией "`*a`".)

Можно сделать еще лучше. Функция `concat` принимает второй необязательный аргумент, который является разделителем для вставки между строками. С этим разделителем нам не нужно вставлять перевод строки после каждой из них:

```
local t = {}
for line in io.lines() do
  t[#t + 1] = line
end
s = table.concat(t, "\n") .. "\n"
```

Функция `concat` вставляет разделитель между строками, но нам все равно нужно добавить один последний перевод строки. Эта последняя конкатенация копирует итоговую строку, что может быть довольно долго. Не существует опции, чтобы добиться от `concat` вставки дополнительного разделителя, но мы можем обмануть ее, добавив в `t` лишнюю пустую строку:

```
t[#t + 1] = ""
s = table.concat(t, "\n")
```

Дополнительный перевод строки, который `concat` добавит перед пустой строкой, будет находиться в конце итоговой строки, как мы и хотели.

11.7. Графы

Как и любой практичный язык, Lua предлагает ряд реализаций для графов, каждый из которых лучше приспособлен под какие-то специфические алгоритмы. Здесь мы рассмотрим простую объектно-ориентированную реализацию, в которой мы будем представлять узлы как объекты (ну разумеется, таблицы), а ребра как ссылки между узлами.

Мы будем представлять каждый узел как таблицу с двумя полями: `name` — имя узла, и `adj` — множество соседних с ним узлов. Поскольку мы будем читать граф из текстового файла, нам будет нужен способ найти узел по его имени. Для этого мы будем использовать дополнительную таблицу, отображающую имена на узлы. Функция `name2node` возвращает узел, получив его имя:

```
local function name2node (graph, name)
  local node = graph[name]
  if not node then
    -- узел не существует; создадим новый узел
    node = {name = name, adj = {}}
    graph[name] = node
  end
  return node
end
```

Листинг 11.1 содержит функцию, которая строит граф. Она читает файл, где каждая строка содержит имена двух узлов, тем самым обозначая ребро от первого узла ко второму. Для каждой строки она использует `string.match`, чтобы разбить строку на два имени, а затем находит соответствующие этим именам узлы (создавая эти узлы при необходимости) и соединяет их.

Листинг 11.1. Чтение графа из файла

```
function readgraph ()
  local graph = {}
  for line in io.lines() do
    -- делит строку на два имени
    local namefrom, nameto = string.match(line, "
(%S+)%s+(%S+)")
    -- находит соответственные узлы
    local from = name2node(graph, namefrom)
    local to = name2node(graph, nameto)
    -- добавляет 'to' к смежному множеству 'from'
    from.adj[to] = true
  end
  return graph
end
```

Листинг 11.2 иллюстрирует алгоритм с применением подобных графов. Функция `findpath` ищет путь между двумя узлами при помощи обхода графов в глубину. Ее первый параметр — это текущий узел; второй задает целевой узел; третий параметр хранит путь от начала к текущему узлу; последний параметр — это множество всех уже посещенных узлов (чтобы избежать циклов). Обратите внимание, как алгоритм напрямую работает с узлами, избегая использования их имен. Например, `visited` — это множество узлов, а не имен узлов. Аналогично `path` — это список узлов.

Листинг 11.2. Нахождение пути между двумя узлами

```
function findpath (curr, to, path, visited)
  path = path or {}
  visited = visited or {}
  if visited[curr] then      -- узел уже посетили?
    return nil               -- пути здесь нет
  end
  visited[curr] = true       -- помечает узел как
посещенный
  path[#path + 1] = curr     -- добавляет его к пути
  if curr == to then         -- конечный узел?
```

```
    return path
  end
  -- пробует все соседние узлы
  for node in pairs(curr.adj) do
    local p = findpath(node, to, path, visited)
    if p then return p end
  end
  path[#path] = nil         -- удаляет узел из пути
end
```

Для проверки этого кода мы добавим функцию, которая печатает путь, и дополнительный код, чтобы это все заработало:

```
function printpath (path)
  for i = 1, #path do
    print(path[i].name)
  end
end

g = readgraph()
a = name2node(g, "a")
b = name2node(g, "b")
p = findpath(a, b)
if p then printpath(p) end
```

Упражнения

Упражнение 11.1. Измените реализацию очереди так, чтобы оба индекса возвращались на ноль, когда очередь пуста.

Упражнение 11.2. Повторите упражнение 10.3, но вместо использования длины в качестве критерия для отбрасывания слов данная программа должна прочесть из текстового файла список отбрасываемых слов.

Упражнение 11.3. Измените структуру графа так, чтобы она хранила метку для каждого ребра. Данная структура также должна представлять каждое ребро при помощи объекта с двумя полями: его меткой и узлом, на который он указывает. Вместо множества

соседних узлов каждый узел должен хранить множество соседних ребер, берущих начало из этого узла.

Адаптируйте функцию `readgraph` так, чтобы она из каждой строки входного файла читала два имени узлов и метку. (Допустим, что метка является числом).

Упражнение 11.4. Предположим, представление графа из предыдущего упражнения устроено так, что метка каждого ребра представляет собой расстояние между его конечными узлами. Напишите функцию, которая находит кратчайший путь между двумя заданными узлами. (Подсказка: используйте алгоритм Дейкстры.)

ГЛАВА 12

Файлы с данными и сохраняемость

При обработке файлов с данными обычно гораздо проще писать эти данные, чем их читать. Когда мы пишем в файл, мы полностью контролируем все, что происходит. С другой стороны, когда мы читаем из файла, то мы не знаем, чего ждать. Помимо обработки всех видов данных, которые могут содержаться в правильном файле, устойчивая программа также должна достойно справляться с дефектными файлами. Поэтому написание устойчивых к ошибкам программ для чтения данных всегда сложно.

В этой главе мы увидим, как можно использовать Lua, чтобы избавиться от лишнего кода для чтения данных в наших программах, просто записывая данные в подходящем формате.

12.1. Файлы с данными

Конструкторы таблиц представляют интересную альтернативу файловым форматам. С небольшой доработкой записи данных чтение становится пустяком. Подход заключается в том, чтобы писать наш файл в виде кода Lua, который при выполнении создает необходимые данные для нашей программы. С конструкторами таблиц эти куски кода могут выглядеть удивительно похожими на простые файлы с данными.

Как обычно, чтобы стало понятнее, давайте рассмотрим пример. Если у нашего файла с данными предопределенный формат, такой как CSV (Comma-Separated Values — значения, разделенные запятыми) или XML, то наш выбор крайне мал. Однако, если мы хотим создать файл для нашего собственного использования, то мы в качестве нашего формата можем использовать конструкторы Lua.

В этом формате мы представляем каждый элемент данных в виде конструктора Lua. Вместо записи в наш файл чего-то вроде

```
Donald E. Knuth,Literate Programming,CSLI,1992
Jon Bentley,More Programming Pearls,Addison-
Wesley,1990
```

мы пишем:

```
Entry{"Donald E. Knuth",
      "Literate Programming",
      "CSLI",
      1992}

Entry{"Jon Bentley",
      "More Programming Pearls",
      "Addison-Wesley",
      1990}
```

Вспомним, что `Entry{`*код*`}` — это то же самое, что и `Entry({`*код*`})`, то есть вызов функции `Entry` с таблицей в качестве единственного аргумента. Поэтому вышеприведенный фрагмент данных является программой Lua. Для чтения этого файла нам лишь нужно выполнить его, имея подходящее определение для `Entry`. Например, следующая программа считает число записей в файле с данными:

```
local count = 0
function Entry () count = count + 1 end
dofile("data")
print("number of entries: " .. count)
```

Следующая программа создает множество из всех имен авторов, найденных в файле, и печатает их (не обязательно в том же порядке, что и в файле):

```
local authors = {}    -- множество для хранения имен
авторов
function Entry (b) authors[b[1]] = true end
dofile("data")
```

```
for name in pairs(authors) do print(name) end
```

Обратите внимание на событийно-ориентированный подход в этих фрагментах кода: `Entry` выступает в роли функции обратного вызова, которая вызывается во время работы `dofile` для каждой записи в файле с данными.

Когда нас не волнует размер файла, мы можем в качестве нашего представления использовать пары имя-значение (Примечание: Если этот формат напоминает вам BibTeX, то это не случайность. Формат BibTeX послужил одним из прототипов для синтаксиса конструкторов в Lua):

```
Entry{
  author = "Donald E. Knuth",
  title = "Literate Programming",
  publisher = "CSLI",
  year = 1992
}

Entry{
  author = "Jon Bentley",
  title = "More Programming Pearls",
  year = 1990,
  publisher = "Addison-Wesley",
}
```

Этот формат мы называем *форматом самоописываемых данных*, поскольку каждый фрагмент данных содержит краткое описание своего значения. Самоописываемые данные более читаемы (по крайней мере, людьми), чем CSV или другие компактные форматы; при необходимости их легко отредактировать вручную; и они позволяют нам вносить небольшие изменения в базовый формат, не требуя изменять файлы с данными. Например, если мы добавим новое поле, то нам потребуется внести лишь небольшое изменение в читающую программу, чтобы она предоставляла значение по умолчанию, когда поле не указано.

При помощи формата имя-значение наша программа для

составления списка авторов становится следующей:

```
local authors = {}    -- множество для хранения имен
авторов
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

Теперь порядок полей не важен. Даже если у некоторых записей нет автора, то нам понадобится изменить лишь функцию `Entry`:

```
function Entry (b)
  if b.author then authors[b.author] = true end
end
```

Lua не только быстро выполняется, но и быстро компилируется. Например, вышеприведенная программа для составления списка авторов обрабатывает 1 Мб данных за одну десятую секунды. (Примечание: На моем старом Пентиуме.) И это не случайно. Описание данных было одной из главных областей применения Lua с момента его создания, и мы уделяем огромное внимание тому, чтобы его компилятор работал быстро для больших программ.

12.2. Сериализация

Часто нам нужно сериализовать какие-то данные, то есть перевести эти данные в поток байтов или символов, чтобы мы могли сохранить их в файл или послать по сети. Мы можем представлять сериализованные данные в виде кода Lua так, чтобы при выполнении этого кода он воссаздавал сохраненные значения для считывающей программы.

Обычно если мы хотим восстановить значение глобальной переменной, то наш кусок кода будет чем-то вроде `varname = `*выражение* , где *выражение* — это код на Lua для получения значения. С `varname` все просто, поэтому давайте посмотрим, как написать код, который создает значение. Для числового значения

задача проста:

```
function serialize (o)
  if type(o) == "number" then
    io.write(o)
  else <прочие случаи>
  end
end
```

Тем не менее, при записи числа в десятичном виде есть риск потерять точность. В Lua 5.2 можно использовать шестнадцатеричный формат, чтобы избежать подобной проблемы:

```
if type(o) == "number" then
  io.write(string.format("%a", o))
```

С этим форматом ("%a") прочитанное число будет состоять ровно из тех же битов, что и исходное.

Для строкового значения наивный подход выглядел бы примерно так:

```
if type(o) == "string" then
  io.write("'", o, "'")
```

Однако, если строка содержит специальные символы (такие как кавычки или переводы строк), то итоговый код уже не будет допустимой программой Lua.

Может показаться заманчивым решить эту проблему путем изменения вида кавычек:

```
if type(o) == "string" then
  io.write("[[", o, "]]")
```

Но так делать не стоит. Если какой-то пользователь со злым умыслом сумеет заставить вашу программу сохранить что-то вроде "]]..os.execute('rm *')..[[" (например, передав данную строку в качестве своего адреса), то в результате вашим куском кода будет

```
varname = [[ ]]..os.execute('rm *')..[[ ]]
```

При попытке загрузить эти «данные» вас ждет неприятный сюрприз.

Простым и безопасным способом заключить строку в кавычки будет использование опции "%q" из функции string.format. Она окружает строку двойными кавычками и корректно экранирует двойные кавычки, переводы строк и некоторые другие символы внутри этой строки:

```
a = 'a "problematic" \\string'
print(string.format("%q", a))    --> "a
\"problematic\" \\string"
```

С данной возможностью наша функция serialize теперь выглядит следующим образом:

```
function serialize (o)
  if type(o) == "number" then
    io.write(o)
  elseif type(o) == "string" then
    io.write(string.format("%q", o))
  else <прочие случаи>
  end
end
```

Начиная с версии 5.1 Lua предлагает другой вариант для безопасного заключения произвольных строк в кавычки — при помощи записи [=[...]=] для длинных строк. Однако, эта новая запись в основном предназначена для собственноручно написанного кода, когда мы в любом случае не хотим изменять строковые литерала. В автоматически генерируемом коде легче экранировать проблематичные символы, как это делает опция "%q" из string.format.

Если же вы все равно хотите использовать длинностроковую запись для автоматически генерируемого кода, то вам нужно позаботиться о некоторых деталях. Первая деталь состоит в том,

что вам нужно подобрать правильное число знаков равенства. Подходящее число — то, которое больше максимальной длины их последовательности в исходной строке. Поскольку строки, содержащие длинные цепочки из знаков равенства, не являются редкостью (например, комментарии, разделяющие фрагменты исходного кода), мы можем ограничиться рассмотрением последовательностей знаков равенства, заключенных между квадратными скобками; другие последовательности не могут привести к ошибочному маркеру конца строки. Второй деталью является то, что Lua всегда отбрасывает перевод строки в начале длинной строки; простым методом борьбы с этим является добавление перевода строки, который будет отброшен.

Листинг 12.1. Заключение в кавычки произвольных строковых литералов

```
function quote (s)
  -- находит максимальную длину последовательностей из
знаков равенства
  local n = -1
  for w in string.gmatch(s, "]=*]") do
    n = math.max(n, #w - 2)    -- -2 для удаления всех
']'
  end

  -- производит строку с 'n' плюс один знаков
равенства
  local eq = string.rep("=", n + 1)

  -- строит строку в кавычках
  return string.format(" [%s[\n%s]%s] ", eq, s, eq)
end
```

Функция `quote` из листинга 12.1 является результатом наших предыдущих замечаний. Она принимает произвольную строку и возвращает ее отформатированной как длинную строку. Вызов `string.gmatch` создает итератор для перебора всех

последовательностей образца `']=*]'` (то есть закрывающей квадратной скобки, после которой стоит последовательность из ноля или больше знаков равенства, за которыми следует еще одна закрывающая квадратная скобка) в строке `s`. (Примечание: Мы обсудим сопоставление с образцом в главе 21.) Для каждого вхождения цикл обновляет `n` до максимального на данный момент числа знаков равенства. После этого цикла мы используем функцию `string.rep`, чтобы повторить знак равенства `n+1` раз, то есть на один знак больше, чем у максимальной последовательности, встреченной в этой строке. Наконец, `string.format` заключает `s` между парами квадратных скобок с надлежащим числом знаков равенства внутри и добавляет дополнительные пробелы вокруг помещаемой в кавычки строки и перевод строки в начале содержащей ее строки.

Сохранение таблиц без циклов

Нашей следующей (и более сложной) задачей является сохранение таблиц. Существует несколько способов сохранения их в соответствии с тем, какие ограничения мы накладываем на структуру таблицы. Не существует одного алгоритма, который бы подходил для всех случаев. Дело не только в том, что простым таблицам нужны более простые алгоритмы, но и в том, что получающиеся при этом файлы могут быть более наглядны.

Наша следующая попытка представлена в листинге 12.2. Несмотря на свою простоту, эта функция довольно практична. Она даже обрабатывает вложенные таблицы (то есть таблицы внутри других таблиц) до тех пор, пока структура таблицы является деревом (то есть не содержит общих подтаблиц и циклов). Небольшим визуальным улучшением будет добавление отступов к иногда встречающимся вложенным таблицам (см. упражнение 12.1).

Листинг 12.2. Сериализация таблиц без циклов

```
function serialize (o)
  if type(o) == "number" then
    io.write(o)
  elseif type(o) == "string" then
    io.write(string.format("%q", o))
  elseif type(o) == "table" then
    io.write("{\n")
    for k,v in pairs(o) do
      io.write(" ", k, " = ")
      serialize(v)
      io.write(",\n")
    end
    io.write("}\n")
  else
    error("cannot serialize a " .. type(o))
  end
end
```

Предыдущая функция предполагает, что все ключи в таблице являются допустимыми идентификаторами. Если в таблице есть числовые ключи или строковые ключи, которые не являются синтаксически допустимыми идентификаторами в Lua, то у нас проблема. Простым путем ее разрешения является использование следующего кода для записи каждого ключа:

```
io.write(" ["); serialize(k); io.write("] = ")
```

С этим изменением мы увеличили надежность нашей функции за счет наглядности получающегося файла. Рассмотрим следующий вызов:

```
serialize{a=12, b='Lua', key='another "one"'}
```

Результат этого вызова при использовании первой версии `serialize` будет следующим:

```
{
  a = 12,
  b = "Lua",
  key = "another \"one\"",
```

```
}
```

Сравните с результатом при использовании второй версии:

```
{
  ["a"] = 12,
  ["b"] = "Lua",
  ["key"] = "another \"one\"",
}
```

Мы можем получить и надежность, и наглядность, проверяя в каждом конкретном случае необходимость квадратных скобок; и вновь мы оставим это улучшение в качестве упражнения.

Сохранение таблиц с циклами

Для обработки таблиц с общей топологией (то есть с циклами и общими подтаблицами) нам потребуется другой подход. Конструкторы не могут представлять подобные таблицы, поэтому мы их и не будем использовать. Для представления циклов нам нужны имена, поэтому наша следующая функция в качестве аргументов получит значение, которое следует сохранить, и имя. Более того, мы должны отслеживать имена уже сохраненных таблиц, чтобы переиспользовать их при обнаружении цикла. Для этого отслеживания мы воспользуемся дополнительной таблицей. Эта таблица будет содержать таблицы в качестве индексов и их имена в качестве связанных с ними значений.

Листинг 12.3. Сохранение таблиц с циклами

```
function basicSerialize (o)
  if type(o) == "number" then
    return tostring(o)
  else    -- предположим, что это строка
    return string.format("%q", o)
  end
end
```

```
function save (name, value, saved)
  saved = saved or {}                       -- начальное
значение
  io.write(name, " = ")
  if type(value) == "number" or type(value) ==
"string" then
    io.write(basicSerialize(value), "\n")
  elseif type(value) == "table" then
    if saved[value] then                    -- значение
уже сохранено?
      io.write(saved[value], "\n")          --
использует его прежнее имя
    else
      saved[value] = name                   -- сохраняет
имя для другого раза
      io.write("{}\n")                      -- создает
новую таблицу
      for k,v in pairs(value) do            -- сохраняет
ее поля
        k = basicSerialize(k)
        local fname = string.format("%s[%s]", name, k)
        save(fname, v, saved)
      end
    end
  else
    error("cannot save a " .. type(value))
  end
end
```

Итоговый код показан в листинге 12.3 . Мы придерживаемся ограничения, что таблицы, которые мы хотим сохранить, содержат лишь числа и строки в качестве ключей. Функция `basicSerialize` сериализует эти базовые типы, возвращая результат. Следующая функция, `save`, выполняет всю тяжелую работу. Параметр `saved` — это таблица, которая отслеживает уже сохраненные таблицы. Например, если мы построим таблицу так

```
a = {x=1, y=2; {3,4,5}}
a[2] = a      -- цикл
a.z = a[1]    -- общая подтаблица
```

то вызов `save("a", a)` сохранит ее следующим образом:

```
a = {}
a[1] = {}
a[1][1] = 3
a[1][2] = 4
a[1][3] = 5

a[2] = a
a["y"] = 2
a["x"] = 1
a["z"] = a[1]
```

Порядок этих присваиваний может меняться, так как он зависит от обхода таблицы. Тем не менее, алгоритм следит за тем, чтобы любой предыдущий узел, необходимый для нового определения, был определен заранее.

Если мы хотим сохранить несколько значений с общими частями, то мы можем сделать несколько вызовов `save` при помощи той же таблицы `saved`. Например, предположим, у нас есть следующие две таблицы:

```
a = {{"one", "two"}, 3}
b = {k = a[1]}
```

Если мы сохраним их независимо друг от друга, то у результата не будет общих частей:

```
save("a", a)
save("b", b)

  --> a = {}
  --> a[1] = {}
  --> a[1][1] = "one"
  --> a[1][2] = "two"
  --> a[2] = 3
  --> b = {}
  --> b["k"] = {}
  --> b["k"][1] = "one"
  --> b["k"][2] = "two"
```

Однако, если мы используем ту же самую таблицу `saved` для обоих вызовов `save`, то у результата будут общие части:

```
local t = {}
save("a", a, t)
save("b", b, t)

  --> a = {}
  --> a[1] = {}
  --> a[1][1] = "one"
  --> a[1][2] = "two"
  --> a[2] = 3
  --> b = {}
  --> b["k"] = a[1]
```

Как принято в Lua, есть несколько других вариантов. Они позволяют сохранять значение без выдачи ему глобального имени (например, кусок строит локальное значение и возвращает его), обрабатывать функции (путем построения вспомогательной таблицы, связывающей каждую функцию с ее именем) и т. д. Lua дает вам силу; механизмы строите вы.

Упражнения

Упражнение 12.1. Измените код из листинга 12.2, чтобы он добавлял отступы ко вложенным таблицам.

(Подсказка: добавьте дополнительный параметр к `serialize` в виде строки с отступом.)

Упражнение 12.2. Измените код из листинга 12.2, чтобы он использовал синтаксис `["key"]` =*value* так, как предложено в разделе 12.1.

Упражнение 12.3. Измените код предыдущего упражнения так, чтобы он использовал синтаксис `["key"]`=*value*, только при необходимости (то есть когда ключ в виде строки не является допустимым идентификатором).

Упражнение 12.4. Измените код предыдущего упражнения,

чтобы он использовал синтаксис конструкторов для списков везде, где это возможно. Например, он должен сериализовать таблицу `{14,15,19}` как `{14,15,19}`, а не как `{[1]=14,[2]=15,[3]=19}`. (Подсказка: начните с сохранения значений ключей 1, 2,..., если только они не равны nil. Позаботьтесь о том, чтобы не сохранить их снова при переборе остальной части таблицы.)

Упражнение 12.5. Подход с отказом от конструкторов при сохранении таблиц с циклами слишком радикальный. Можно сохранить таблицу в более приятном виде, применяя конструкторы в общем случае, а затем используя присваивания только для исправления общего доступа и циклов.

Заново реализуйте функцию `save` с использованием данного подхода. Добавьте к ней все вкусности, что вы уже реализовали в предыдущих упражнениях (оступы, синтаксис записей и синтаксис списков).

ГЛАВА 13

Метатаблицы и метаметоды

Обычно для каждого значения в Lua есть вполне предсказуемый набор операций. Мы можем складывать числа, соединять строки, вставлять пары ключ-значение в таблицы и т. д. Однако, мы не можем складывать таблицы, сравнивать функции и вызывать строку. Если только мы не используем метатаблицы.

Метатаблицы позволяют изменить поведение значения при его встрече с неожиданной операцией. Например, при помощи метатаблиц мы можем определить то, как Lua вычисляет выражение `a+b`, где `a` и `b` — это таблицы. Когда Lua пытается сложить две таблицы, он проверяет, есть ли хотя бы в одной из них *метатаблица* и содержит ли эта метатаблица поле `__add`. Если Lua находит это поле, он вызывает соответствующее значение — так называемый *метаметод*, который должен быть функцией, — для вычисления суммы.

Каждое значение в Lua может иметь связанную с ним метатаблицу. У таблиц и пользовательских данных отдельные метатаблицы; значения остальных типов совместно используют одну единственную метатаблицу для всех значений своего типа. Lua всегда создает новые таблицы без метатаблиц:

```
t = {}
print(getmetatable(t))    --> nil
```

Мы можем использовать функцию `setmetatable`, чтобы задать или изменить метатаблицу для любой таблицы:

```
t1 = {}
setmetatable(t, t1)
print(getmetatable(t) == t1)    --> true
```

Из Lua мы можем устанавливать метатаблицы только для таблиц; для работы с метатаблицами значений других типов мы должны использовать код C (Примечание: Главная причина этого ограничения состоит в том, чтобы помешать злоупотреблению метатаблиц для распространенных типов. Опыт предыдущих версий Lua показал, что подобные глобальные изменения часто ведут к одноразовому коду). Как мы позже увидим в главе 21, строковая библиотека устанавливает метатаблицу для строк. У всех прочих типов по умолчанию нет метатаблиц:

```
print(getmetatable("hi"))      --> table: 0x80772e0
print(getmetatable("xuxu"))    --> table: 0x80772e0
print(getmetatable(10))        --> nil
print(getmetatable(print))     --> nil
```

Любая таблица может быть метатаблицей для любого значения; группа связанных между собой таблиц может совместно использовать общую метатаблицу, которая описывает их общее поведение; таблица может быть метатаблицей для самой себя, таким образом описывая свое собственное индивидуальное поведение. Допустимо использовать любую схему.

13.1. Арифметические метаметоды

В этом разделе мы приведем простой пример, чтобы объяснить, как использовать метатаблицы. Пусть мы используем таблицы для представления множеств с функциями для вычисления их объединения, пересечения и т. п., как показано в листинге 13.1. Чтобы не засорять наше пространство имен, мы будем хранить эти функции в таблице `Set`.

Листинг 13.1. Простая реализация множеств

```
Set = {}

-- создает новое множество со значениями из заданного
```

```
списка
function Set.new (l)
  local set = {}
  for _, v in ipairs(l) do set[v] = true end
  return set
end

function Set.union (a, b)
  local res = Set.new{}
  for k in pairs(a) do res[k] = true end
  for k in pairs(b) do res[k] = true end
  return res
end

function Set.intersection (a, b)
  local res = Set.new{}
  for k in pairs(a) do
  res[k] = b[k]
  end
  return res
end

-- представляет множество как строку
function Set.tostring (set)
  local l = {} -- list to put all elements from the
set
  for e in pairs(set) do
    l[#l + 1] = e
  end
  return "{" .. table.concat(l, ", ") .. "}"
end

-- печатает множество
function Set.print (s)
  print(Set.tostring(s))
end
```

Теперь нам нужно использовать операцию сложения ('+') для вычисления объединения двух множеств. Для этого мы дадим возможнось всем таблицам, представляющим множества, использовать одну общую метатаблицу. Эта метатаблица определит, как таблицы должны реагировать на операцию сложения. Нашим

первым шагом будет создание обычной таблицы, которую мы будем использовать как метатаблицу для множеств:

```
local mt = {}    -- метатаблица для множеств
```

Следующим шагом будет изменение функции `Set.new`, создающей множества. В новой версии этой функции будет лишь одна дополнительная строка, которая для создаваемых таблиц устанавливает `mt` как метатаблицу:

```
function Set.new (l)    -- вторая версия
  local set = {}
  setmetatable(set, mt)
  for _, v in ipairs(l) do set[v] = true end
  return set
end
```

После этого каждое множество, которое мы создадим при помощи `Set.new`, будет иметь одну и ту же таблицу в качестве метатаблицы:

```
s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
print(getmetatable(s1)) --> table: 00672B60
print(getmetatable(s2)) --> table: 00672B60
```

Наконец, мы добавим к метатаблице метаметод в виде поля `__add`, которое определяет, как нужно выполнять сложение:

```
mt.__add = Set.union
```

После этого каждый раз, когда Lua попытается сложить два множества, он будет вызывать функцию `Set.union`, передавая оба операнда в качестве аргументов.

С метаметодом мы можем использовать операцию сложения для выполнения объединения множеств:

```
s3 = s1 + s2
Set.print(s3)    --> {1, 10, 20, 30, 50}
```

Аналогично мы можем определить операцию умножения для

выполнения пересечения множеств:

```
mt.__mul = Set.intersection
Set.print((s1 + s2)*s1)    --> {10, 20, 30, 50}
```

Для каждой арифметической операции существует соответствующее имя поля в метатаблице. Кроме `__add` и `__mul` есть `__sub` (для вычитания), `__div` (для деления), `__unm` (для отрицания), `__mod` (для взятия остатка от деления) и `__pow` (для возведения в степень). Мы также можем определить поле `__concat` для описания поведения операции конкатенации.

Когда мы складываем два множества, то вопрос о том, какую метатаблицу использовать, не возникает. Однако, мы можем записать выражение, в котором участвуют два значения с разными метатаблицами, например такое, как ниже:

```
s = Set.new{1,2,3}
s = s + 8
```

При поиске метаметода Lua выполняет следующие шаги: если у первого значения есть метатаблица с полем `__add`, то Lua использует это поле в качестве метаметода независимо от второго значения; иначе, если у второго значения есть метатаблица с полем `__add`, Lua использует его поле в качестве метаметода; в противном случае Lua вызовет ошибку. Таким образом, в последнем примере будет вызвана `Set.union`, как и для выражений `10+s` и `"hello"+s`.

Lua не беспокоится об этих смешанных типах, но они важны для нашей реализации. Например, если мы выполним `s=s+8`, то получим ошибку внутри `Set.union`:

```
bad argument #1 to 'pairs' (table expected, got
number)
```

Если нам нужны более понятные сообщения об ошибках, то мы должны явно проверять типы операндов перед попыткой выполнения операции:

```
function Set.union (a, b)
  if getmetatable(a) ~= mt or getmetatable(b) ~= mt
then
    error("attempt to 'add' a set with a non-set
value", 2)
  end
  <как прежде>
```

Помните, что второй аргумент функции `error` (2 в нашем случае) направляет сообщение об ошибке туда, где данная операция была вызвана.

13.2. Метаметоды сравнения

Метатаблицы также позволяют придать смысл операциям сравнения посредством метаметодов `__eq` (*равно*), `__lt` (*меньше, чем*) и `__le` (*меньше или равно, чем*). Для трех оставшихся операций сравнения нет отдельных метаметодов: Lua переводит `a~=b` в `not(a==b)`, `a>b` в `b<a` и `a>=b` в `b<=a`.

До версии 4.0 Lua переводил все операции упорядочивания в одну, переводя `a<=b` в `not(b<a)`. Однако, такой перевод некорректен, когда мы имеем дело с *частичным порядком*, то есть когда не все элементы нашего типа надлежащим образом упорядочены. Например, числа с плавающей точкой не являются полностью упорядоченными на большинстве компьютеров из-за значения *NaN* (*Not a Number* — не число). В соответствии со стандартом IEEE 754 NaN представляет неопределенные значения, такие как результат `0/0`. Согласно стандарту, любое сравнение, включающее в себя NaN, должно быть ложным. Это значит, что `NaN<=x` всегда ложно, но и `x<=NaN` также ложно. Из этого также следует, что перевод `a<=b` в `not(b<a)` в данном случае неверен.

В нашем примере с множествами мы имеем дело с похожей проблемой. Очевидным (и полезным) значением `<=` для множеств является вхождение множества: `a<=b` означает, что `a` — это

подмножество `b`. С этим значением все равно возможно, что `a<=b` и `b<a` ложны; таким образом, нам нужны отдельные реализации для `__le` (*меньше или равно*) и `__lt` (*меньше, чем*):

```
mt.__le = function (a, b)    -- вхождение множеств
  for k in pairs(a) do
    if not b[k] then return false end
  end
  return true
end

mt.__lt = function (a, b)
return a <= b and not (b <= a)
end
```

Наконец, мы можем определить равенство множеств через их вхождение:

```
mt.__eq = function (a, b)
  return a <= b and b <= a
end
```

После этих определений мы готовы сравнивать множества:

```
s1 = Set.new{2, 4}
s2 = Set.new{4, 10, 2}
print(s1 <= s2)          --> true
print(s1 < s2)           --> true
print(s1 >= s1)          --> true
print(s1 > s1)           --> false
print(s1 == s2 * s1)     --> true
```

Для типов с полным порядком мы можем не определять метаметод `__le`. При его отсутствии Lua использует поле `__lt`.

У сравнения на равенство есть некоторые ограничения. Если у двух объектов разные базовые типы или метаметоды, то операция сравнения на равенство вернет false, даже не вызывая метаметоды. Таким образом множество всегда будет отличаться от числа, независимо от того, что говорит метаметод.

13.3. Библиотечные метаметоды

До сих пор все метаметоды, что мы видели, находились в ядре Lua. Виртуальная машина сама проверяет, содержат ли значения, участвующие в операции, метатаблицы с метаметодами для этой операции. Однако, поскольку метатаблицы являются обычными таблицами, их может использовать любой. Поэтому для библиотек в порядке вещей определять свои собственные поля в метатаблицах.

Функция `tostring` является типичным примером. Как мы видели ранее, `tostring` представляет таблицы довольно простым образом:

```
print({}) --> table: 0x8062ac0
```

Функция `print` всегда вызывает `tostring` для форматирования своего вывода. Однако, при форматировании какого-либо значения `tostring` сначала проверяет, есть ли у значения метаметод `__tostring`. Если такой метаметод есть, то `tostring` вызывает его, передавая ему объект в качестве аргумента. То, что вернет этот метаметод, и будет результатом `tostring`.

В нашем примере с множествами мы уже определили функцию для представления множества в виде строки. Поэтому нам нужно лишь выставить поле `__tostring` в этой метатаблице:

```
mt.__tostring = Set.tostring
```

После этого, когда бы мы не вызвали `print` с множеством в качестве аргумента, `print` вызовет `tostring`, которая, в свою очередь, вызовет `Set.tostring`:

```
s1 = Set.new{10, 4, 5}
print(s1)    --> {4, 5, 10}
```

Функции `setmetatable` и `getmetatable` также используют метаполе, в данном случае для защиты метатаблиц. Предположим,

вы хотите защитить ваши множества так, что пользователи не смогут ни увидеть, ни изменить их метатаблицы. Если задать в метатаблице поле `__metatable`, то `getmetatable` вернет значение этого поля, в то время как `setmetatable` вызовет ошибку:

```
mt.__metatable = "not your business"

s1 = Set.new{}
print(getmetatable(s1))     --> not your business
setmetatable(s1, {})
  stdin:1: cannot change protected metatable
```

В Lua 5.2 `pairs` и `ipairs` также обладают метатаблицами, поэтому таблица может изменить способ своего обхода (или можно добавить обход для объектов, не являющихся таблицами).

13.4. Метаметоды доступа к таблице

Метаметоды для арифметических операций и операций сравнения лишь определяют поведение для ситуаций, которые иначе приводили бы к возникновению ошибок. Они не влияют на обычное поведение языка. Но Lua также предлагает способ для изменения поведения таблиц в двух обычных случаях: запросе и изменении несуществующего поля в таблице.

Метаметод `__index`

Ранее я уже говорил, что когда мы обращаемся к отсутствующему полю в таблице, результатом является nil. Это так, но это не вся правда. На самом деле при подобном обращении интерпретатор начинает искать метаметод `__index`: если такого метода нет, что обычно и бывает, то возвращается nil; иначе результат будет предоставлен этим метаметодом.

Типичным примером здесь является наследование. Пусть мы

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

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

Теперь мы определим метаметод `__index`:

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

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

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

Lua обнаружит, что у `w` нет требуемого поля, но есть метатаблица с полем `__index`. Поэтому Lua вызовет этот метаметод с аргументами `w` (таблица) и "`width`" (отсутствующий ключ). Затем метаметод индексирует этот прототип заданным ключом и возвращает полученное значение.

Использование метаметода `__index` для наследования в Lua настолько распространено, что Lua предоставляет сокращенный вариант. Хотя его и зовут *методом*, метаметод `__index` не обязан быть функцией: например, он может быть таблицей. Когда он является функцией, Lua вызывает его, передавая таблицу и отсутствующий ключ в качестве аргументов, как мы только что видели. Когда он является таблицей, Lua перенаправляет к ней обращение. Поэтому в нашем предыдущем примере мы могли просто определить `__index` следующим образом:

```
mt.__index = prototype
```

Теперь, когда Lua будет искать метаметод `__index`, он найдет значение `prototype`, которое является таблицей. Соответственно, Lua повторит обращение к этой таблице, то есть выполнит аналог `prototype["width"]`. Это обращение и приведет к нужному результату.

Использование таблицы в качестве метаметода `__index` дает простой и быстрый способ реализации одиночного наследования. Функция, хотя и более затратна, предоставляет больше гибкости: мы можем реализовать множественное наследование, кэширование и ряд других вариантов. Мы обсудим эти формы наследования в главе 16.

Когда мы хотим обратиться к таблице без вызова ее метаметода `__index`, мы используем функцию `rawget`. Вызов `rawget(t,i)` осуществляет *непосредственный* доступ к таблице `t`, то есть примитивное обращение без использования метатаблиц. Применение непосредственного доступа не ускорит ваш код (затраты на вызов функции уничтожат любую прибавку), но иногда, как мы увидим позже, он необходим.

Метаметод `__newindex`

Метаметод `__newindex` делает то же, что и `__index`, но работает при обновлениях таблиц, а не при доступе к ним. Когда вы присваиваете значение отсутствующему индексу в таблице, интерпретатор ищет метаметод `__newindex`: если он есть, то интерпретатор вызывает его вместо выполнения присваивания. Подобно `__index`, если метаметод является таблицей, то интерпретатор выполняет присваивание для этой таблицы вместо исходной. Более того, есть функция с прямым доступом, которая позволяет миновать метаметод: `rawset (t, k, v)` записывает значение `v` по ключу `k` в таблицу `t`, не вызывая никаких метаметодов.

Совместное использование метаметодов `__index` и `__newindex` позволяет реализовать в Lua ряд довольно мощных конструкций, таких как таблицы, доступные только для чтения, таблицы со значениями по умолчанию и наследование для объектно-ориентированного программирования. В этой главе мы увидим некоторые области их применения. Объектно-ориентированному программированию посвящена его собственная глава.

Таблицы со значениями по умолчанию

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

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

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

После вызова `setDefault` любой доступ к отсутствующему полю в

`tab` вызовет его метаметод `__index`, который вернет ноль (значение `d` для этого метаметода).

Функция `setDeafult` создает новое замыкание и новую метатаблицу для каждой таблицы, которой нужно значение по умолчанию., Это может оказаться затратным, если у нас много таблиц, которым нужны значения по умолчанию. У метатаблицы значение по умолчанию `d` «зашито» в ее метаметод, поэтому мы не можем использовать одну и ту же метатаблицу для всех таблиц. Чтобы можно было использовать одну и ту же метатаблицу для таблиц с разными значениями по умолчанию, мы можем запоминать значение по умолчанию в самой таблице, используя для этого специальное поле. Если мы не беспокоимся о конфликтах имен, мы можем использовать для нашего особого поля ключ вроде "___":

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

Обратите внимание, что теперь мы создаем таблицу `mt` лишь один раз, вне функции `setDefault`.

Если мы беспокоимся о конфликтах имен, то можно легко убедиться в уникальности особого ключа. Все, что нам нужно, — это создать новую таблицу и использовать ее в качестве ключа:

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

Другим способом связывания значения по умолчанию с каждой таблицей является использование отдельной таблицы, где индексы являются этими таблицами, а значения этих индексов являются значениями этих таблиц по умолчанию. Однако, для корректной

реализации данного подхода нам нужна особая разновидность таблиц — так называемые *слабые таблицы* (weak tables), поэтому здесь мы не будем его использовать; мы вернемся к данной теме в главе 17.

Иной подход состоит в том, чтобы *запоминать* (memorize) метатаблицы, чтобы затем многократно использовать одну и ту же метатаблицу для таблиц с одинаковыми значениями по умолчанию. Однако, для этого тоже нужны слабые таблицы, поэтому нам вновь придется подождать до главы 17.

Отслеживание доступа к таблице

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

```
t = {} -- исходная таблица (созданная где-то еще)

-- хранит закрытый доступ к первоначальной таблице
local _t = t

-- создает посредника
t = {}

-- создает метатаблицу
local mt = {
  __index = function (t, k)
    print("*access to element " .. tostring(k))
    return _t[k]     -- доступ к исходной таблице
```

```
  end,
  __newindex = function (t, k, v)
    print("*update of element " .. tostring(k) ..
          " to " .. tostring(v))
    _t[k] = v          -- обновляет исходную таблицу
  end
}
setmetatable(t, mt)
```

Этот код отслеживает каждое обращение к `t`:

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

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

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

Нам может понадобиться нечто подобное для `__ipairs`.

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

```
local index = {}            -- создает закрытый индекс
local mt = {                -- создает метатаблицу
  __index = function (t, k)
```

```
    print("*access to element " .. tostring(k))
    return t[index][k]     -- доступ к исходной
таблице
  end,
  __newindex = function (t, k, v)
  print("*update of element " .. tostring(k) ..
        " to " .. tostring(v))
  t[index][k] = v          -- обновляет исходную
таблицу
end,
__pairs = function (t)
  return function (t, k)
    return next(t[index], k)
  end, t
end
}

function track (t)
  local proxy = {}
  proxy[index] = t
  setmetatable(proxy, mt)
  return proxy
end
```

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

Таблицы, доступные только для чтения

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

```
function readOnly (t)
  local proxy = {}
  local mt = {     -- создает метатаблицу
    __index = t,
    __newindex = function (t, k, v)
      error("attempt to update a read-only table", 2)
    end
  }
  setmetatable(proxy, mt)
  return proxy
end
```

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

```
days = readOnly{"Sunday", "Monday", "Tuesday",
"Wednesday",
                "Thursday", "Friday", "Saturday"}

print(days[1])    --> Sunday
days[2] = "Noday"
stdin:1: attempt to update a read-only table
```

Упражнения

Упражнение 13.1. Определите метаметод `__sub`, который возвращает разницу двух множеств. (Множество a — b является множеством всех элементов из a, которые не содержатся в b.)

Упражнение 13.2. Определите для множеств метаметод `__len` так, чтобы `#s` возвращал число элементов в множестве `s`.

Упражнение 13.3. Дополните реализацию посредников в разделе 13.4 метаметодом `__ipairs`.

Упражнение 13.4. Другим способом реализации таблиц, доступных только для чтения, является использование функции в качестве метаметода `__index`. Этот подход делает обращения к таблице более затратными, но создание таких таблиц обходится дешевле, так как все таблицы, доступные только для чтения, могут

совместно использовать одну метатаблицу. Перепишите функцию `readOnly` с использованием данного подхода.

ГЛАВА 14

Окружение

Lua хранит все свои глобальные переменные в обычной таблице, называемой *глобальным окружением* (global environment). (Точнее, Lua хранит свои «глобальные» переменные в нескольких окружениях, но мы некоторое время не будем обращать внимание на данные сложности.) Одним из преимуществ этой структуры является то, что она упрощает внутреннюю реализацию Lua, поскольку нет необходимости в специальной структуре данных для глобальных переменных. Другое преимущество состоит в том, что мы можем работать с этой таблицей так же, как и с любой другой. Для упрощения такой работы Lua хранит само окружение в глобальной переменной `_G`. (Да, `_G._G` равно `_G`.) Например, следующий код печатает имена всех глобальных переменных, определенных в глобальном окружении:

```
for n in pairs(_G) do print(n) end
```

В этой главе мы увидим несколько полезных методов для работы с окружением.

14.1. Глобальные переменные с динамическими именами

Обычно для обращения к глобальным переменным и установки их значений достаточно присваивания. Однако, довольно часто нам требуется какая-либо форма метапрограммирования, например, когда мы хотим работать с глобальной переменной, чье имя содержится в другой переменной или каким-то образом вычисляется во время

выполнения. Чтобы получить значение такой переменной, многие программисты пытаются писать нечто подобное:

```
value = loadstring("return " .. varname)()
```

Если, скажем, `varname` равно `x`, то результатом конкатенации будет "`return x`", что при выполнении даст нам желаемый результат. Однако, этот код включает в себя создание и компиляцию нового куска кода, что является довольно затратным. Вы можете добиться того же эффекта при помощи следующего кода, который в десятки раз эффективнее предыдущего:

```
value = _G[varname]
```

Поскольку окружение — это обычная таблица, то вы можете просто индексировать ее нужным ключом (именем переменной).

Похожим образом можно присвоить значение глобальной переменной, чье имя вычисляется динамически, если написать `_G[varname] = value`. Однако, будьте внимательны: некоторые программисты так радуются подобной возможности, что заканчивают написанием кода вроде `_G["a"] = _G["var1"]`, что является всего лишь усложненным вариантом написания `a=var1`.

Обобщением предыдущей задачи является разрешение использования полей в динамических именах, таких как "`io.read`" или "`a.b.c.d`". Если написать `_G["io.read"]`, то очевидно, что мы не получим поле `read` из таблицы `io`. Но мы можем написать функцию `getfield`, такую что `getfield ("io.read")` вернет ожидаемое значение. Эта функция преимущественно является циклом, который начинает с `_G` и развивается поле за полем:

```
function getfield (f)
  local v = _G    -- начинает с таблицы глобальных
переменных
  for w in string.gmatch(f, "[%w_]+") do
    v = v[w]
  end
  return v
```

```
end
```

Мы полагаемся на `gmatch` из библиотеки `string` для перебора всех слов в `f` (где «слово» — это последовательность из одного или более буквенно-цифровых символов и знаков подчеркивания).

Соответствующая функция для задания полей немного сложнее. Присваивание вроде `a.b.c.d=v` эквивалентно следующему коду:

```
local temp = a.b.c
temp.d = v
```

То есть мы должны извлечь последнее имя и затем отдельно его обработать. Следующая функция `setfield` решает данную задачу и при этом создает промежуточные таблицы в пути, когда они не существуют:

```
function setfield (f, v)
  local t = _G              -- начинает с таблицы
глобальных переменных
  for w, d in string.gmatch(f, "([%w_]+)(%.?)") do
    if d == "." then        -- имя не последнее?
      t[w] = t[w] or {}     -- создает таблицу при ее
отсутствии
      t = t[w]              -- получает таблицу
    else                    -- последнее имя
      t[w] = v              -- производит присваивание
    end
  end
end
```

Этот новый образец захватывает имя поля в переменную `w` и следующую за ним необязательную точку в переменную `d`. (Примечание: Мы подробно обсудим сопоставление с образцом в главе 21). Если за именем не следует точка, то это последнее имя.

При наличии вышеприведенных функций следующий вызов создает глобальную таблицу `t` и таблицу `t.x` , после чего присваивает `t.x.y` значение 10:

```
setfield("t.x.y", 10)
```

```
print(t.x.y)                --> 10
print(getfield("t.x.y"))    --> 10
```

14.2. Объявления глобальных переменных

В Lua глобальным переменным не нужны объявления. Хотя это и удобно для небольших программ, в больших программах простая опечатка может привести к трудно обнаруживаемым ошибкам. Однако, при желании мы можем изменить это поведение. Поскольку Lua хранит свои глобальные переменные в обычной таблице, мы можем использовать метатаблицы для изменения его поведения при обращении к глобальным переменным.

Первый подход состоит в простом отслеживании любых обращений к отсутствующим ключам в глобальной таблице:

```
setmetatable(_G, {
  __newindex = function (_, n)
    error("attempt to write to undeclared variable "
.. n, 2)
  end,
  __index = function (_, n)
    error("attempt to read undeclared variable " .. n,
2)
  end,
})
```

После выполнения этого кода любая попытка обратиться к несуществующей глобальной переменной приведет к возникновению ошибки:

```
> print(a)
stdin:1: attempt to read undeclared variable a
```

Но как же нам объявлять глобальные переменные? Одним из вариантов является использование `rawset`, которая минует метаметод:

```
function declare (name, initval)
  rawset(_G, name, initval or false)
end
```

(Конструкция **or false** следит за тем, чтобы глобальная переменная всегда получала значение, отличное от nil.)

Вариант попроще — разрешить присваивания новым глобальным переменным только внутри функций, при этом не ограничивая присваивания за пределами куска.

Для проверки того, что присваивание происходит в главном куске, нам нужно использовать отладочную библиотеку. Вызов `debug.getinfo(2, "S")` возвращает таблицу, чье поле `what` сообщит о том, является ли функция, вызвавшая метаметод, главным куском, обычной функцией Lua или функцией C. (Мы обсудим `debug.getinfo` более подробно в главе 24.) Посредством этой функции мы можем переписать метаметод `__newindex` следующим образом:

```
__newindex = function (t, n, v)
  local w = debug.getinfo(2, "S").what
  if w ~= "main" and w ~= "C" then
    error("attempt to write to undeclared variable "
.. n, 2)
  end
  rawset(t, n, v)
end
```

Эта новая версия также допускает присваивания из кода C, так как обычно его авторы знают, что они делают.

Для проверки существования переменной мы не можем просто сравнить ее с nil, поскольку если она nil, то обращение приведет к ошибке. Вместо этого мы используем `rawget`, которая избегает метаметод:

```
if rawget(_G, var) == nil then
  -- 'var' не объявлена
  ...
end
```

Пока что наша схема не допускает использование глобальных переменных со значением nil, поскольку они автоматически будут считаться необъявленными. Но это не сложно исправить. Все, что нам нужно, — это вспомогательная таблица, которая хранит имена объявленных переменных. При вызове метаметода он по этой таблице проверяет — объявлена эта переменная, или нет. Ее код может быть похож на приведенный в листинге 14.1. Теперь даже присваивания вроде `x=nil` достаточно, чтобы объявить глобальную переменную.

Листинг 14.1. Проверка описаний глобальных переменных

```
local declaredNames = {}

setmetatable(_G, {
  __newindex = function (t, n, v)
    if not declaredNames[n] then
      local w = debug.getinfo(2, "S").what
      if w ~= "main" and w ~= "C" then
        error("attempt to write to undeclared variable
"..n, 2)
      end
      declaredNames[n] = true
    end
    rawset(t, n, v)    -- производит настоящую
установку значений
  end,

  __index = function (_, n)
    if not declaredNames[n] then
      error("attempt to read undeclared variable "..n,
2)
    else
      return nil
    end
  end,
})
```

Затраты на оба решения крайне малы. При первом решении в нормальном режиме работы метаметод вообще не вызывается. При

втором решении метаметоды могут быть вызваны, но только когда программа обращается к переменной со значением nil.

В стандартную поставку Lua входит модуль `strict.lua`, который реализует проверку глобальных переменных, по сути состоящую из только что рассмотренного нами кода. Использовать его при разработке кода на Lua — хорошая привычка.

14.3. Неглобальные окружения

Одной из проблем окружения является то, что оно глобальное. Любое его изменение влияет на все части вашей программы. Например, когда вы устанавливаете метатаблицу для управления глобальным доступом, вся ваша программа должна ей руководствоваться. Если вы хотите воспользоваться библиотекой, которая использует глобальные переменные без их объявления, то вам не повезло.

В Lua глобальные переменные не обязаны быть действительно глобальными. Мы даже можем сказать, что в Lua нет глобальных переменных. Поначалу это может звучать странно, поскольку мы пользовались глобальными переменными все это время. Очевидно, Lua очень старается создать иллюзию наличия глобальных переменных. Давайте посмотрим, как Lua создает эту иллюзию. (Примечание: Обратите внимание, что этот механизм был одной из тех частей Lua, которые претерпели наибольшие изменения при переходе с версии 5.1 на 5.2. Следующее обсуждение относится только к Lua 5.2 и очень мало применимо к предыдущим версиям.)

Начнем с понятия свободных имен. *Свободное имя* (free name) — это имя, которое не привязано к явному объявлению, то есть не встречается внутри области видимости локальной переменной (или переменной цикла **for**, или параметра) с этим именем. Например, и `var1`, и `var2` — это свободные имена в следующем куске:

```
var1 = var2 + 3
```

В отличие от сказанного ранее, свободное имя не относится к глобальной переменной (по крайней мере, не прямым образом). Вместо этого компилятор Lua переводит любое свободное имя `var` в `_ENV.var`. Поэтому предыдущий кусок эквивалентен следующему:

```
_ENV.var1 = _ENV.var2 + 3
```

Но что такое эта новая переменная `_ENV`? Она не может быть глобальной переменной, иначе мы снова возвращаемся к исходной проблеме. И вновь это проделки компилятора. Я уже отметил, что Lua рассматривает каждый кусок как анонимную функцию. На самом деле Lua компилирует наш исходный кусок в следующий код:

```
local _ENV = <какое-нибудь значение>
return function (...)
  _ENV.var1 = _ENV.var2 + 3
end
```

То есть Lua компилирует любой кусок кода с участием предопределенного верхнего значения с именем `_ENV`.

Обычно когда мы загружаем кусок кода, функция `load` инициализирует это предопределенное верхнее значение посредством глобального окружения. Поэтому наш исходный кусок становится эквивалентным следующему:

```
local _ENV = <глобальное окружение>
return function (...)
  _ENV.var1 = _ENV.var2 + 3
end
```

Результатом всех этих присваиваний является то, что поле `var1` из глобального окружения получает значение поля `var2` плюс 3.

На первый взгляд это может показаться довольно запутанным способом работы с глобальными переменными. Я не буду утверждать, что это простейший способ, но он дает гибкость, которую трудно получить с более простой реализацией.

Прежде чем мы продолжим, давайте кратко сформулируем обработку глобальных переменных в Lua 5.2:

- Lua компилирует любой кусок внутри области видимости верхнего значения с именем `_ENV`.
- Компилятор переводит любое свободное имя `var` в `_ENV.var`.
- Функция `load` (или `loadfile`) инициализирует первое верхнее значение куска при помощи глобального окружения.

В конце концов, все не так уж и сложно.

Некоторых это смущает, поскольку они пытаются выявить особые механизмы, лежащие в основе этих правил. Но никаких особых механизмов нет. В частности, за первые два правила полностью отвечает компилятор. За исключением того, что `_ENV` предопределена компилятором, она является обычной переменной. Вне компиляции у `_ENV` нет никакого особого назначения в Lua. (Примечание: Если быть честными до конца, то Lua использует это имя для сообщений об ошибках, чтобы в докладе об ошибке с участием переменной `_ENV.x` указывать ее как `global x`.) Точно так же перевод из `var` в `_ENV.var` — это простая синтаксическая замена без скрытого смысла. В частности, после этого перевода `_ENV` будет относиться любой переменной `_ENV`, которая видна на этом этапе кода, исходя из стандартных правил видимости.

14.4. Использование `_ENV`

В этом разделе мы увидим некоторые способы для освоения той гибкости, которую дает нам `_ENV`. Имейте в виду, что большинство примеров из данного раздела должно выполняться отдельным куском. При построчном вводе кода в интерактивном режиме каждая строка становится отдельным куском и таким образом

получает свою переменную _ENV. Для выполнения фрагмента кода как отдельного куска вам нужно либо запустить его из файла, либо в интерактивном режиме поместить внутрь пары **do—end**.

Поскольку _ENV — это обычная переменная, мы можем обращаться к ней и присваивать значения как и любой другой переменной. Присваивание _ENV=nil сведет на нет любой прямой доступ к глобальным переменным в оставшейся части куска. Это может пригодиться для контроля над тем, какие переменные использует ваш код:

```
local print, sin = print, math.sin
_ENV = nil
print(13)              --> 13
print(sin(13))         --> 0.42016703682664
print(math.cos(13))    -- ошибка!
```

Любое присваивание свободному имени вызовет аналогичную ошибку.

Мы можем написать _ENV явным образом, чтобы миновать локальное объявление:

```
a = 13            -- глобальная
local a = 12
print(a)          --> 12    (локальная)
print(_ENV.a)     --> 13    (глобальная)
```

Конечно, главной областью применения _ENV является изменение окружения, используемого фрагментом кода. Как только вы измените ваше окружение, все обращения к глобальным переменным будут пользоваться новой таблицей:

```
-- изменяет текущее окрущение на новую пустую таблицу
_ENV = {}
a = 1    -- создает поле в _ENV
print(a)
  --> stdin:4: attempt to call global 'print' (a nil
value)
```

Если новое окружение пусто, то вы теряете доступ ко всем вашим глобальным переменным, включая `print`. Поэтому сперва вы должны заполнить его какими-нибудь полезными значениями, например старым окружением:

```
a = 15                  -- создает глобальную переменную
_ENV = {g = _G}         -- изменяет текущее окружение
a = 1                   -- создает поле в _ENV
g.print(a)              --> 1
g.print(g.a)            --> 15
```

Теперь, когда вы обращаетесь к «глобальной» `g`, вы получаете старое окружение, в котором вы найдете функцию `print`.

Мы можем переписать предыдущий пример, используя имя `_G` вместо `g`:

```
a = 15                   -- создает глобальную переменную
_ENV = {_G = _G}         -- изменяет текущее окружение
a = 1                    -- создает поле в _ENV
_G.print(a)              --> 1
_G.print(_G.a)           --> 15
```

Для Lua `_G` — такое же имя, как и все остальные. Его особый статус проявляется только тогда, когда Lua создает исходную глобальную таблицу и присваивает эту таблицу глобальной переменной `_G`. Для Lua не важно текущее значение этой переменной. Но обычно принято использовать одно и то же имя всякий раз, когда у нас есть ссылка на глобальное окружение, как в переписанном примере.

Другой способ заполнить ваше новое окружение — применить наследование:

```
a = 1
local newgt = {}      -- создает новое окружение
setmetatable(newgt, {__index = _G})
_ENV = newgt          -- устанавливает его
print(a)              --> 1
```

В этом коде новое окружение наследует из старого и `print`, и `a`. Несмотря на это, любое присваивание поступает в новую таблицу. Теперь изменение переменной в глобальном окружении по ошибке не опасно, хотя вы по-прежнему можете изменять их через `_G`:

```
-- продолжаем предыдущий код
a = 10
print(a)       --> 10
print(_G.a)    --> 1
_G.a = 20
print(_G.a)    --> 20
```

Поскольку `_ENV` является обычной переменной, она подчиняется обычным правилам видимости. В частности, функции, определенные внутри куска, обращаются к `_ENV` так же, как и к любой другой внешней переменной:

```
_ENV = {_G = _G}
local function foo ()
  _G.print(a)                  -- скомпилировано как
'_ENV._G.print(_ENV.a)'
end
a = 10                         -- _ENV.a
foo()                          --> 10
_ENV = {_G = _G, a = 20}
foo()                          --> 20
```

Если мы определим новую локальную переменную с именем `_ENV`, то ссылки на свободные имена будут привязаны к ней:

```
a = 2
do
  local _ENV = {print = print, a = 14}
  print(a)    --> 14
end
print(a)      --> 2    (возвращение к изначальной
_ENV)
```

Поэтому несложно построить функцию с закрытым окружением:

```
function factory (_ENV)
```

```
  return function ()
    return a    -- "глобальная" a
  end
end

f1 = factory{a = 6}
f2 = factory{a = 7}
print(f1())    --> 6
print(f2())    --> 7
```

Функция `factory` создает простые замыкания, которые возвращают значение из их глобальных `a`. При созданном замыкании видимая переменная `_ENV` является параметром `_ENV` из охватывающей его функции `factory`; поэтому замыкание использует эту внешнюю переменную (в качестве верхнего значения) для доступа к своим свободным именам.

Используя обычные правила видимости, мы можем работать с окружениями различными способами. Например, у нас может быть несколько функций с общим для них окружением или функция, которая изменяет окружение, общее с другими функциями.

14.5. `_ENV` и `load`

Как я ранее упоминал, `load` обычно инициализирует верхнее значение `_ENV` из загруженного куска посредством глобального окружения. Однако, у `load` есть необязательный четвертый параметр, который задает значение для `_ENV`. (У функции `loadfile` есть аналогичный параметр.)

В качестве первого примера допустим, что у нас есть типичный конфигурационный файл, определяющий различные константы и функции, используемые программой; это может быть что-то вроде:

```
-- файл 'config.lua'
width = 200
height = 300
...
```

Мы можем загрузить его при помощи следующего кода:

```
env = {}
f = loadfile("config.lua", "t", env)
f()
```

Весь код из конфигурационного файла будет выполнен в пустом окружении `env`. Важнее то, что все его определения перейдут именно в это окружение. Этот конфигурационный файл никоим образом не может повлиять на что-либо еще, даже по ошибке. Даже вредоносный код не сможет причинить много вреда. Все, что он может, — выполнить DoS-атаку (приводящую к отказу от обслуживания), тратя процессорное время и память.

Иногда вам может понадобиться выполнить кусок несколько раз, каждый раз с другой таблицей окружения. В этом случае дополнительный аргумент для `load` бесполезен. Вместо этого у нас есть два других варианта.

Первый вариант — это использовать функцию `debug.setupvalue` из отладочной библиотеки. Как следует из имени, `setupvalue` позволяет нам изменить любое верхнее значение заданной функции. Следующий фрагмент иллюстрирует его использование:

```
f = loadfile(filename)
...
env = {}
debug.setupvalue(f, 1, env)
```

Первый аргумент при вызове `setupvalue` — это функция, второй — индекс верхнего значения, а третий — новое значение для верхнего значения. При типичном применении второй аргумент всегда равен единице: когда функция является результатом `load` или `loadfile`, Lua следит за тем, чтобы у нее было лишь одно верхнее значение, равное `_ENV`.

Небольшим минусом данного решения является зависимость от отладочной библиотеки. Эта библиотека нарушает некоторые

стандартные соглашения о программах. Например, `debug.setupvalue` нарушает правила видимости Lua, которые следят за тем, чтобы к локальной переменной нельзя было обратиться вне ее лексической области видимости.

Другой способ выполнения куска с различными окружениями состоит в небольшом изменении куска при его загрузке. Представьте, что мы добавляем следующую строку прямо в начало загружаемого куска:

```
_ENV = ...;
```

Вспомним из раздела 8.1, что Lua компилирует любой кусок в виде вариадической функции. Поэтому эта дополнительная строка присвоит переменной `_ENV` первый аргумент куска, устанавливая его как окружение. После загрузки этого куска мы вызываем полученную функцию, передавая нужное нам окружение как первый аргумент. Следующий фрагмент кода иллюстрирует эту идею при помощи функции `loadwithprefix` из упражнения 8.1:

```
f = loadwithprefix("local _ENV = ...;",
io.lines(filename, "*L"))
...
env = {}
f(env)
```

Упражнения

Упражнение 14.1. Функция `getfield`, которую мы определили в начале этой главы, слишком неприхотлива, так как она допускает «поля» вроде `math?sin` или `string!!!gsub`. Перепишите ее так, чтобы она принимала в качестве разделителя имен только одиночную точку. (Для этого упражнения вам может понадобиться информация из главы 21.)

Упражнение 14.2. Объясните в деталях, что происходит в следующей программе и каким будет ее вывод.

```
local foo
do
  local _ENV = _ENV
  function foo () print(X) end
end
X = 13
_ENV = nil
foo()
X = 0
```

Упражнение 14.3. Объясните в деталях, что происходит в следующей программе и каким будет ее вывод.

```
local print = print
function foo (_ENV, a)
  print(a + b)
end

foo({b = 14}, 12)
foo({b = 10}, 1)
```

ГЛАВА 15

Модули и пакеты

Обычно Lua не обязывает к каким-либо соглашениям. Вместо этого Lua предоставляет механизмы, которые достаточно эффективны для групп разработчиков, чтобы реализовать наиболее подходящие для них соглашения. Однако, этот подход не годится для модулей. Одна из основных целей модульной системы состоит в том, чтобы позволить разным группам совместно использовать код. Отсутствие общих соглашений мешает такому использованию.

Начиная с версии 5.1, Lua определил набор соглашений для модулей и пакетов (пакет является набором модулей). Эти соглашения не требуют от языка никаких дополнительных средств; программисты могут реализовать их при помощи того, что мы уже видели: таблиц, функций, метатаблиц и окружений. Программисты могут использовать любые другие соглашения. Разумеется, альтернативные реализации могут привести к тому, что программы не смогут использовать чужие модули, а модули не смогут быть использованы чужими программами.

С точки зрения пользователя, *модуль* — это некоторый код (на Lua или C), который может быть загружен посредством `require` и который создает и возвращает таблицу. Все, что модуль экспортирует, будь то функции или таблицы, он определяет внутри этой таблицы, которая выступает в качестве пространства имен.

Например, все стандартные библиотеки — это модули. Вы можете использовать математическую библиотеку следующим образом:

```
local m = require "math"
print(m.sin(3.14))
```

Однако, автономный интерпретатор заранее загружает все стандартные библиотеки при помощи кода, эквивалентного следующему:

```
math = require "math"
string = require "string"
...
```

Эта предварительная загрузка позволяет нам записывать `math.sin` привычным образом.

Очевидное преимущество от использования таблиц для реализации модулей состоит в том, что мы можем работать с модулями как с любыми другими таблицами, и применять всю силу Lua для создания дополнительных средств. В большинстве языков модули не являются значениями первого класса (то есть не могут храниться в переменных, передаваться как аргументы функциям и т. п.), поэтому таким языкам нужны специальные механизмы для каждого дополнительного средства, которое они хотят предложить для модулей. В Lua вы получаете дополнительные средства бесплатно.

Например, для пользователя существует несколько способов вызвать функцию из модуля. Обычным способом является следующий:

```
local mod = require "mod"
mod.foo()
```

Пользователь может задать для модуля любое локальное имя:

```
local m = require "mod"
m.foo()
```

Также можно предоставить альтернативные имена для отдельных функций:

```
local m = require "mod"
local f = mod.foo
f()
```

Эти средства удобны тем, что они не требуют специальной поддержки от языка. Они используют только то, что уже предлагает язык.

Распространенная жалоба на `require` состоит в том, что эта функция не может передавать аргументы загружаемому модулю. Например, у математического модуля могла бы быть опция для выбора между градусами и радианами:

```
-- bad code
local math = require("math", "degree")
```

Проблема в том, что одна из основных задач `require` — избегать многократной загрузки модуля. Как только модуль загружен, он может быть многократно использован любой частью программы, которой он снова потребуется. Если бы один и тот же модуль был затребован с разными параметрами, это привело бы к конфликту:

```
-- плохой код
local math = require("math", "degree")

-- где-то еще в той же программе
local math = require("math", "radians")
```

В случае, когда вы действительно хотите, чтобы у вашего модуля были параметры, лучше создать явную функцию для их задания, например так:

```
local mod = require"mod"
mod.init(0, 0)
```

Если инициализирующая функция возвращает сам модуль, то мы можем написать код вроде следующего:

```
local mod = require"mod".init(0, 0)
```

Другой вариант — сделать так, чтобы модуль возвращал свою функцию для инициализации, и лишь эта функция возвращала бы

таблицу модуля:

```
local mod = require"mod"(0, 0)
```

В любом случае помните, что сам модуль загружается всего один раз; разрешение конфликтных инициализаций остается на его усмотрение.

15.1. Функция `require`

Функция `require` пытается свести к минимуму свои предположения о том, что является модулем. Для `require` модуль — это всего лишь какой-то код, который определяет некоторые значения (например, функции или таблицы, содержащие функции). Обычно этот код возвращает таблицу, состоящую из функций этого модуля. Однако, поскольку это делается кодом самого модуля, а не `require`, некоторые модули могут выбрать возвращать другие значения или даже иметь побочные эффекты.

Для загрузки модуля мы просто вызываем `require "`*имя_модуля*`"`. Первым шагом `require` является проверка по таблице `package.loaded`, не был ли загружен данный модуль ранее. Если да, `require` возвращает его соответствующее значение. Поэтому, как только модуль загружен, другие вызовы, для которых он требуется, просто вернут то же значение без повтороного выполнения какого-либо кода.

Если модуль еще не загружен, то `require` ищет файл Lua с именем модуля. Если он находит этот файл, то загружает его при помощи `loadfile`. Результатом этого является функция, которую мы называем *загрузчиком* (loader). (Загрузчик — это функция, которая при вызове загружает модуль.)

Если `require` не может найти файл Lua с именем модуля, то она ищет библиотеку C с этим именем модуля. Если она находит библиотеку C, то она загружает ее посредством `package.loadlib`

(которую мы обсудили в разделе 8.3) и ищет функцию с именем `luaopen_`*имя_модуля* (Примечание: В разделе 27.3 мы обсудим, как писать библиотеки C). В этом случае загрузчик является результатом `loadlib`, то есть функцией `luaopen_`*имя_модуля*, представленной в качестве функции Lua.

Независимо от того, где был найден модуль, — в файле Lua или в библиотеке C, у `require` теперь есть для него загрузчик. Для окончательной загрузки модуля `require` вызывает загрузчик с двумя аргументами: именем модуля и именем файла, из которого был взят этот загрузчик. (Большинство модулей просто игнорируют эти аргументы.) Если загрузчик возвращает какое-либо значение, `require` возвращает это значение и хранит в таблице `package.loaded`, чтобы всегда возвращать одинаковые значения при будущих вызовах этого же модуля. Если загрузчик не возвращает никакие значения, `require` ведет себя так, как если бы модуль вернул true. Без этого уточнения, последующие вызовы `require` снова бы выполняли этот модуль.

Чтобы заставить `require` загрузить один и тот же модуль дважды, мы просто стираем запись о нем из `package.loaded`:

```
package.loaded.<имя_модуля> = nil
```

В следующий раз, когда понадобится этот модуль, `require` проделает всю необходимую работу еще раз.

Переименование модуля

Обычно мы используем модули с их изначальными именами, но иногда мы должны переименовать модуль, чтобы избежать конфликта имен. Типичной ситуацией является загрузка разных версий одного и того же модуля, например для тестирования. У модулей Lua нет внутренней привязки к именам, поэтому обычно достаточно переименовать соответствующий файл с расширением

`.lua`. Однако, мы не можем отредактировать бинарную библиотеку для корректировки имени ее функции `luaopen_*`. Чтобы поддерживать подобные переименования у `require` есть небольшая хитрость: если имя модуля содержит дефис, то `require` отбрасывает этот дефис и все, что стоит перед ним, когда создает имя функции `luaopen_*`. Например, если модуль называется `a-b`, то `require` ожидает, что функция для его открытия будет называться `luaopen_b`, а не `luaopen_a-b` (что по-любому не является допустимым именем в языке C). Поэтому если нам нужно использовать два модуля с именем `mod`, то мы можем переименовать один из них в `v1-mod`, например. Когда мы вызовем `m1=require "v1-mod"`, функция `require` найдет переименованный файл `v1-mod` и внутри этого файла найдет функцию с изначальным именем `luaopen_mod`.

Поиск пути

При поиске файла Lua `require` использует путь, который несколько отличается от типичных путей. Типичный путь — это список директорий, в которых нужно искать заданный файл. Однако, в ANSI C (абстрактная платформа, на которой выполняется Lua) нет понятия директории. Поэтому путь, используемый `require`, — это список *шаблонов* (template) каждый из которых задает свой способ преобразования имени модуля (аргумента `require`) в имя файла. Более точно, каждый шаблон в пути — это имя файла, содержащее необязательные знаки вопроса. Для каждого шаблона `require` заменяет каждый '`?`' на имя модуля и проверяет, есть ли файл с получившимся именем; если нет, она переходит к следующему шаблону. Шаблоны в пути разделены точками с запятой (символ, редко используемый в именах файлов в большинстве операционных систем). Например, если путем является

```
?;?.lua;c:\windows\?;/usr/local/lua/?/?.lua
```

то вызов `require "sql"` попробует открыть следующие файлы Lua:

```
sql
sql.lua
c:\windows\sql
/usr/local/lua/sql/sql.lua
```

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

Путь, который `require` использует для поиска файлов Lua, — это всегда текущее значение переменной `package.path`. Во время запуска Lua инициализирует эту переменную значением переменной окружения `LUA_PATH_5_2`. Если эта переменная окружения не определена, Lua пытается использовать переменную окружения `LUA_PATH`. Если они обе не определены, Lua использует путь по умолчанию, заданный при компиляции (Примечание: В Lua 5.2 опция командной строки `-E` предотвращает использование этих переменных окружения и заставляет использовать значение по умолчанию). При использовании значения переменной окружения Lua подставляет путь по умолчанию вместо любой подстроки "`;;`". Например, если вы установите `LUA_PATH_5_2` на "`mydir/?.lua;;`", то окончательный путь будет шаблоном "`mydir/?.lua`", за которым следует путь по умолчанию.

Путь для поиска библиотек C работает точно так же, но его значение берется из переменной `package.cpath` (вместо `package.path`). Аналогично эта переменная получает свое начальное значение из переменной окружения `LUA_CPATH_5_2` или `LUA_CPATH`. Типичным значением этого пути в UNIX является следующее:

```
./?.so;/usr/local/lib/lua/5.2/?.so
```

Обратите внимание, что путь определяет расширение файла. Предыдущий пример использует `.so` для всех шаблонов; в

Windows типичный путь будет примерно таким:

```
.\?.dll;C:\Program Files\Lua502\dll\?.dll
```

Функция `package.searchpath` запрограммирована с учетом всех вышеприведенных правил для поиска библиотек. Она получает имя модуля и путь, а затем ищет файл, следуя этим правилам. Она возвращает либо имя первого найденного файла, либо nil и сообщение об ошибке, описывающее все файлы, которые она безуспешно пыталась открыть, как в следующем примере:

```
> path = ".\\?.dll;C:\\Program
Files\\Lua502\\dll\\?.dll"
> print(package.searchpath("X", path))
nil
 no file '.\X.dll'
 no file 'C:\Program Files\Lua502\dll\X.dll'
```

Искатели

В действительности `require` несколько сложнее, чем мы описали. Поиск файла Lua и библиотеки C — это лишь два частных случая более общего понятия *искателя* (searcher). Искатель — это просто функция, которая получает имя модуля и возвращает либо загрузчик для него, либо nil, если не может найти ни одного.

Массив `package.searchers` содержит перечень искателей, которыми пользуется `require`. При поиске модуля `require` поочередно вызывает каждого искателя из этого перечня, передавая ему имя модуля, до тех пор, пока не найдет загрузчик для этого модуля. Если поиск не дает результата, `require` вызывает ошибку.

В Lua 5.2 параметр командной строки `-E` предотвращает использование переменных окружения и приводит к использованию пути, заданного при компиляции.

Использование списка для управления поиском модуля придает огромную гибкость функции `require`. Например, если вы хотите

хранить модули сжатыми в zip-файлы, то вам лишь нужно предоставить соответствующую функцию-искатель и добавить ее к списку. Однако, чаще всего программам все же не нужно изменять содержимое `package.searchers`. В конфигурации по умолчанию искатель файлов Lua искатель библиотек C, которые мы описали выше, занимают в списке вторую и третью позиции, соответственно. Перед ними стоит искатель предзагрузки.

Искатель *предзагрузки* (preload) позволяет определить для загрузки модуля произвольную функцию. Он использует таблицу `package.preload` для отображения имен модулей в загрузочные функции. При поиске имени модуля данный искатель просто ищет заданное имя в этой таблице. Если он находит в ней функцию, он возвращает ее в качестве загрузчика модуля. Иначе он возвращает nil. Этот искатель предоставляет общий метод для обработки некоторых нетрадиционных ситуаций. Например, библиотека C, статически прилинкованная к Lua, может зарегистрировать свою функцию `luaopen_` в таблице `preload` так, что она будет вызвана, только когда (и если) пользователю понадобится этот модуль. Таким образом, программа не тратит время на открытие модуля, если он не используется.

По умолчанию `package.searchers` включает в себя четвертую функцию, которая нужна лишь для подмодулей. Мы обсудим ее в разделе 15.4.

15.2. Основной подход к написанию модулей на Lua

Простейший способ создать модуль на Lua поистине прост: мы создаем таблицу, помещаем все функции, которые мы хотим экспортировать, внутрь нее и возвращаем эту таблицу. Листинг 15.1 демонстрирует этот подход. Обратите внимание на то, как мы определяем функцию `inv` в качестве закрытой, просто объявляя ее

локальной для этого куска.

Листинг 15.1. Простой модуль для комплексных чисел

```
local M = {}

function M.new (r, i) return {r=r, i=i} end

-- определяет константу 'i'
M.i = M.new(0, 1)

function M.add (c1, c2)
  return M.new(c1.r + c2.r, c1.i + c2.i)
end

function M.sub (c1, c2)
  return M.new(c1.r - c2.r, c1.i - c2.i)
end

function M.mul (c1, c2)
  return M.new(c1.r*c2.r - c1.i*c2.i, c1.r*c2.i +
c1.i*c2.r)
end

local function inv (c)
  local n = c.r^2 + c.i^2
  return M.new(c.r/n, -c.i/n)
end

function M.div (c1, c2)
  return M.mul(c1, inv(c2))
end

function M.tostring (c)
  return "(" .. c.r .. "," .. c.i .. ")"
end

return M
```

Некоторым не нравится оператор **return** в конце. Одним из способов его устранения является присваивание таблицы модуля непосредственно `package.loaded`:

```
local M = {}
package.loaded[...] = M
  <как прежде>
```

Вспомним, что `require` вызывает загрузчик, передавая имя модуля как первый аргумент. Поэтому выражение с переменным числом аргументов `...` в индексе приводит к возврату этого имени. После этого присваивания нам больше не нужно возвращать `M` в конце модуля: если модуль не возвращает значение, то `require` вернет текущее значение `package.loaded[modname]` (если оно не nil). В любом случае, я предпочитаю писать **return** в конце модуля, поскольку это выглядит аккуратнее.

Другой способ записи модуля состоит в определении всех функций как локальных и построении возвращаемой таблицы в конце модуля, как в листинге 15.2. В чем преимущества этого подхода? Вам не нужно начинать каждое имя с `M.` или чего-то похожего; здесь явный список экспортируемых функций; вы определяете и используете экспортируемые и внутренние функции внутри модуля одним и тем же образом. В чем недостатки этого подхода? Список экспортируемых функций находится в конце модуля, а не в начале, где он был бы более удобен в качестве быстрой справки; и этот список в некоторой степени избыточен, так как каждое имя нужно писать дважды. (Этот последний недостаток может стать преимуществом, поскольку позволяет функциям иметь разные имена снаружи модуля и внутри него, но я думаю, что программисты редко этим пользуются.) Лично мне нравится данный стиль, но вкусы у всех разные.

Листинг 15.2. Модуль со списком экспортируемых функций

```
local function new (r, i) return {r=r, i=i} end

-- определяет константу 'i'
local i = complex.new(0, 1)

  <другие функции следуют этому же образцу>
```

```
return {
  new      = new,
  i        = i,
  add      = add,
  sub      = sub,
  mul      = mul,
  div      = div,
  tostring = tostring,
}
```

В любом случае, помните, что вне зависимости от того, как определен модуль, пользователи должны иметь возможность использовать его стандартным образом:

```
local cpx = require "complex"
print(cpx.tostring(cpx.add(cpx.new(3,4), cpx.i)))
  --> (3,5)
```

15.3. Использование окружений

Одним из недостатков тех базовых методов для создания модулей является то, что с ними слишком легко засорить глобальное пространство имен, например просто забыв **local** в закрытом объявлении.

Окружения предоставляют интересный подход к созданию модулей, который решает эту проблему. Когда у главного куска модуля есть свое окружение, то в эту таблицу переходят не только все его функции, но и все глобальные переменные. Поэтому мы можем объявить все открытые функции в качестве глобальных переменных, и они автоматически попадут в отдельную таблицу. Все, что нужно сделать модулю, — присвоить эту таблицу переменной `_ENV`. После этого, когда мы объявим функцию `add`, она станет `M.add`:

```
local M = {}
_ENV = M
```

```
function add (c1, c2)
  return new(c1.r + c2.r, c1.i + c2.i)
end
```

Более того, мы можем вызывать другие функции из этого же модуля без какого-либо префикса. В предыдущем коде `add` обращается к `new` из своего окружения, то есть вызывает `M.new`.

Этот метод обеспечивает хорошую поддержку модулей, требуя очень небольшой работы от программиста. Префиксы с ним вообще не нужны. Нет никакой разницы между вызовом экспортированной и закрытой функций. Если программист забывает вставить **local**, то он не засоряет глобальное пространство имен; вместо этого закрытая функция просто становится открытой.

Тем не менее, в настоящее время я по-прежнему предпочитаю один из двух методов, рассмотренных в предыдущем разделе. Хотя они могут потребовать чуть больше работы, результат выполнения такого кода более понятен. Чтобы не создать по ошибке глобальную переменную, я пользуюсь простым методом, который состоит в присваивании `_ENV` значения nil. После этого любое присваивание глобальной переменной вызовет ошибку.

Чего при этом не хватает, так это, разумеется, доступа к другим модулям. Как только мы изменим значение `_ENV`, мы потеряем доступ ко всем предыдущим глобальным переменным. Есть несколько способов восстановить этот доступ, каждый со своими плюсами и минусами.

Первый способ состоит в применении наследования:

```
local M = {}
setmetatable(M, {__index = _G})
_ENV = M
```

(Вам нужно вызвать `setmetatable` перед присваиванием `_ENV`; зачем?) С данной конструкцией у модуля есть прямой доступ к любому глобальному идентификатору, с небольшими затратами при каждом обращении. Любопытным последствием этого решения

является то, что ваш модуль, по идее, теперь содержит все глобальные переменные. Например, любой, кто использует ваш модуль, теперь может вызывать стандартную функцию для вычисления синуса, написав `complex.math.sin(x)`. (Данная особенность также есть в системе пакетов Perl.)

Еще одним быстрым методом доступа к другим модулям является объявление локальной переменной, которая хранит исходное окружение:

```
local M = {}
local _G = _G
_ENV = M        -- или _ENV = nil
```

Теперь вы должны начинать каждое глобальное имя с `_G.`, но доступ происходит немного быстрее, поскольку никакие метаметоды не задействованы.

Более взвешенный подход заключается в том, чтобы объявить локальными только те функции, которые вам нужны, или, по крайней мере, лишь нужные вам модули:

```
-- настройка модуля
local M = {}

-- Импортируемая секция:
-- объявляет все, что этому модулю потребуется извне
local sqrt = math.sqrt
local io = io

-- с данного момента внешнего доступа больше нет
_ENV = nil -- or _ENV = M
```

Этот подход требует больше работы, но он лучше документирует зависимости вашего модуля. При этом код, который получается при его применении, выполняется немного быстрее, чем в прежних схемах, из-за использования локальных переменных.

15.4. Подмодули и пакеты

Lua разрешает именам модулей быть иерархическими, используя точку для разделения уровней имен. Например, модуль с именем `mod.sub` является *подмодулем* модуля `mod`. *Пакет* — это полное дерево модулей; он является единицей распространения кода в Lua.

Когда вам нужен модуль с именем `mod.sub`, функция `require` сперва обращается к таблице `package.loaded`, а затем к таблице `package.preload`, используя полное имя "`mod.sub`" в качестве ключа; в этом случае точка является таким же символом в имене модуля, как и любой другой.

Однако, при поиске файла, определяющего этот подмодуль, `require` переводит точку в другой символ, обычно системный разделитель директорий (то есть '`/`' для UNIX и '`\`' для Windows). После этого преобразования `require` ищет получившееся имя, как и любое другое. Предположим, что у нас есть разделитель директорий '`/`' и следующий путь:

```
./?.lua;/usr/local/lua/?.lua;/usr/local/lua/?/init.lua
```

Вызов `require "a.b"` попробует открыть следующие файлы:

```
./a/b.lua
/usr/local/lua/a/b.lua
/usr/local/lua/a/b/init.lua
```

Это поведение позволяет всем модулям пакета находиться в одной директории. Например, если в пакете содержатся модули `p`, `p.a` и `p.b`, то соответствующими файлами могут быть `p/init.lua`, `p/a.lua` и `p/b.lua`, директория `p` которых содержится в другой подходящей директории.

Разделитель директорий, используемый Lua, настраивается во время компиляции и может быть любой строкой (вспомните, что Lua ничего не знает про директории). Например, системы без иерархических директорий могут использовать в качестве такого разделителя '`_`', так что `require "a.b"` будет искать файл `a_b.lua`.

Имена в C не могут содержать точки, поэтому библиотека C для

подмодуля `a.b` не может экспортировать функцию `luaopen_a.b`. В этом случае `require` переводит точку в другой символ — подчеркивание. Таким образом, библиотека C с именем `a.b` должна назвать свою инициализирующую функцию `luaopen_a_b`. Мы также можем воспользоваться здесь уловкой с дефисом, что может пригодиться в некоторых случаях. Например, если у нас есть библиотека C с именем `a` и мы хотим сделать ее подмодулем `mod`, то мы можем переименовать ее файл в `mod/v-a`. Когда мы напишем `require "mod.v-a"`, функция `require` правильно найдет новый файл `mod/v-a`, так же как и функцию `luaopen_a` внутри него.

В качестве дополнительного средства у `require` есть еще один искатель для загрузки подмодулей C. Когда он не может найти ни файл Lua, ни файл C для подмодуля, этот искатель опять ищет в пути для C, но на этот раз он ищет имя пакета. Например, если программе требуется подмодуль `a.b.c`, этот искатель будет искать `a`. Если он найдет библиотеку C с этим именем, то `require` будет искать в этой библиотеке соответствующую открывающую функцию, в нашем случае `luaopen_a_b_c`. Данное средство позволяет размещать несколько подмодулей в одной библиотеке C, каждый со своей открывающей функцией.

С точки зрения Lua, подмодули в одном пакете не имеют явной связи. Загрузка модуля `a` не приводит к автоматической загрузке любого из ее подмодулей; аналогично, при загрузке `a.b` не произойдет автоматической загрузки `a`. Конечно, разработчик пакета при желании может задать эти связи. Например, модуль `a` может при загрузке явно потребовать один или все свои подмодули.

Упражнения

Упражнение 15.1. Перепишите код в листинге 13.1 в виде соответственного модуля.

Упражнение 15.2. Что случится при поиске библиотеки, если

какая-то часть пути зафиксирована (то есть не содержит знак вопроса)? Может ли пригодиться такое поведение?

Упражнение 15.3. Напишите искатель, который одновременно ищет файлы Lua и библиотеки C. Например, путь для этого искателя может быть чем-то вроде:

```
./?.lua;./?.so;/usr/lib/lua5.2/?.so;/usr/share/lua5.2/
```

(Подсказка: используйте `package.searchpath` для поиска соответствующего файла и потом попытайтесь загрузить его, сначала при помощи `loadfile`, а затем при помощи `package.loadlib`.)

Упражнение 15.4. Что случится, если вы установите метатаблицу для `package.preload` при помощи метаметода `__index`? Может ли пригодиться такое поведение?

ГЛАВА 16

Объектно-ориентированное программирование

Таблица в Lua — это объект во многих отношениях. Подобно объектам, у таблиц есть состояние. Подобно объектам, у таблиц есть идентичность (собственное «я» — *self*), которая не зависит от ее значений; в частности, две таблицы с одинаковыми значениями являются разными объектами, и при этом каждый объект может иметь разные значения в разные моменты времени. Подобно объектам, у таблиц есть жизненный цикл, который не зависит от того, кто их создал или где они были созданы.

У объектов есть свои собственные действия. У таблиц также могут быть действия, как показано ниже:

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
end
```

Это определение создает новую функцию и хранит ее в поле `withdraw` объекта `Account`. Затем мы можем вызвать ее, как показано ниже:

```
Account.withdraw(100.00)
```

Данная разновидность функции — это почти то, что мы называем *методом*. Однако, использование глобального имени `Account` внутри функции является плохой практикой программирования. Во-первых, эта функция будет работать только для данного конкретного объекта. Во-вторых, даже для этого объекта функция будет работать ровно до тех пор, пока этот объект

хранится в той конкретной глобальной переменной. Если мы изменим имя объекта, то `withdraw` больше не будет работать:

```
a, Account = Account, nil
a.withdraw(100.00)    -- ОШИБКА!
```

Подобное поведение нарушает принцип, что у каждого объекта должен быть свой, независимый цикл жизни.

Более гибкий подход состоит в работе с *получателем* (receiver) операции. Для этого нашему методу понадобится дополнительный параметр со значением получателя. Этот параметр обычно называется *self* или *this*:

```
function Account.withdraw (self, v)
  self.balance = self.balance - v
end
```

Теперь при вызове метода мы должны задать объект, над которым он должен работать:

```
a1 = Account; Account = nil
...
a1.withdraw(a1, 100.00)    -- OK
```

При использовании параметра *self* мы можем использовать один и тот же метод для многих объектов:

```
a2 = {balance=0, withdraw = Account.withdraw}
...
a2.withdraw(a2, 260.00)
```

Данное применение параметра *self* является ключевым моментом в любом объектно-ориентированном языке. В большинстве объектно-ориентированных языков данный механизм частично скрыт от программиста, поэтому этот параметр не нужно объявлять (хотя он можно по-прежнему пользоваться именем *self* или *this* внутри метода). Lua также может скрывать этот параметр при помощи *операции двоеточия*. Мы можем переписать предыдущее

определение метода в виде

```
function Account:withdraw (v)
  self.balance = self.balance - v
end
```

а вызов метода как

```
a:withdraw(100.00)
```

Двоеточие добавляет дополнительный скрытый параметр в определение метода и добавляет дополнительный аргумент в вызов метода. Двоеточие является всего лишь синтаксическим сахаром, хотя и довольно удобным; ничего принципиально нового здесь нет. Мы можем определить функцию, воспользовавшись точкой, и вызвать ее, применив двоеточие, или наоборот, до тех пор, пока мы правильно обрабатываем дополнительный параметр:

```
Account = { balance=0,
            withdraw = function (self, v)
                         self.balance = self.balance -
v
                       end
          }

function Account:deposit (v)
  self.balance = self.balance + v
end

Account.deposit(Account, 200.00)
Account:withdraw(100.00)
```

К данному моменту у наших объектов есть идентичность, состояние и действия над этим состоянием. Им по-прежнему не хватает системы классов, наследования и скрытия членов. Давайте займемся первой задачей: как нам создать различные объекты с одинаковым поведением? В частности, как нам создать несколько счетов?

16.1. Классы

Класс работает как шаблон для создания объектов. Большинство объектно-ориентированных языков предлагает понятие класса. В таких языках каждый объект является экземпляром какого-то конкретного класса. В Lua нет понятия класса; каждый объект определяет свое собственное поведение и состояние. Однако, смоделировать в Lua классы не трудно, если следовать примеру языков на основе прототипов, вроде Self или NewtonScript. В этих языках у объектов нет классов. Вместо этого у каждого объекта может быть прототип, который является обычным объектом, в котором первый объект ищет неизвестные ему действия. Для представления классов в таких языках мы просто создаем объект, который будет использован только в качестве прототипа для других объектов (его экземпляров). И классы, и прототипы работают в качестве мест хранения поведения, общего для различных объектов.

В Lua мы можем реализовать прототипы, используя идею наследования из раздела 13.4. Точнее, если у нас есть два объекта `a` и `b`, то все, что нам нужно сделать, чтобы `b` стал прототипом для `a`, — это следующее:

```
setmetatable(a, {__index = b})
```

После этого `a` будет искать в `b` любое действие, которого у нет. Называть объект `b` классом объекта `a` — это не что иное, как замена терминов.

Давайте вернемся к нашему примеру с банковским счетом. Для создания других счетов с поведением, аналогичным `Account`, мы сделаем так, что эти новые объекты унаследуют свои действия от `Account` при помощи метаметода `__index`. Небольшая оптимизация будет состоять в том, что нам не нужно создавать дополнительную таблицу в качестве метатаблицы для объектов `Account`; для этой

цели мы будем использовать саму таблицу `Account`:

```
function Account:new (o)
  o = o or {}    -- создает таблицу, если пользователь
ее не предоставил
  setmetatable(o, self)
  self.__index = self
  return o
end
```

(Когда мы вызываем `Account:new`, параметр `self` равен `Account`; поэтому мы могли бы явно использовать `Account` вместо `self`. Однако, использование `self` отлично пригодится в следующем разделе, когда мы введем наследование.) Что произойдет после выполнения данного кода, когда мы создадим новый счет и вызовем его метод, как показано ниже?

```
a = Account:new{balance = 0}
a:deposit(100.00)
```

Когда мы создаем новый счет, у `a` в качестве метатаблицы будет `Account` (из-за параметра `self` в вызове `Account:new`). Затем, когда мы вызываем `a:deposit(100.00)`, на самом деле происходит вызов `a.deposit(a,100.00)`; двоеточие — это всего лишь синтаксический сахар. Однако, Lua не может найти запись "`deposit`" в таблице `a`; поэтому Lua ищет запись `__index` в метатаблице. Теперь ситуация выглядит примерно следующим образом:

```
getmetatable(a).__index.deposit(a, 100.00)
```

Метатаблицей `a` является `Account`, а `Account.__index` — это тоже `Account` (поскольку метод `new` выполнил `self.__index=self`). Поэтому предыдущее выражение сокращается до

```
Account.deposit(a, 100.00)
```

То есть Lua вызывает исходную функцию `deposit`, но передает ей `a` в качестве параметра `self`. Таким образом, новый счет `a`

унаследовал функцию `deposit` от `Account`. По этой же схеме он наследует все поля от `Account`.

Это наследование работает не только для методов, но также и для других полей, отсутствующих в новом счете. Поэтому класс может предоставлять не только методы, но и значения по умолчанию для полей экземпляра. Напомним, что в нашем первом определении `Account` мы предоставили поле `balance` со значением 0. Поэтому если мы создадим счет без начального значения баланса, то он унаследует это значение по умолчанию:

```
b = Account:new()
print(b.balance)    --> 0
```

При вызове метода `deposit` у `b` он выполнит код, эквивалентный следующему (поскольку `self` равен `b`):

```
b.balance = b.balance + v
```

Выражение `b.balance` дает 0, и метод присваивает начальный вклад `b.balance`. Последующие обращения к `b.balance` уже не приведут к вызову метаметода `__index`, так как у `b` теперь есть свое собственное поле `balance`.

16.2. Наследование

Поскольку классы являются объектами, они также могут получать методы от других классов. Это поведение позволяет легко реализовать наследование (в обычном объектно-ориентированном смысле) в Lua.

Пусть у нас есть базовый класс, такой как `Account`:

```
Account = {balance = 0}

function Account:new (o)
  o = o or {}
  setmetatable(o, self)
```

```
  self.__index = self
  return o
end

function Account:deposit (v)
  self.balance = self.balance + v
end

function Account:withdraw (v)
  if v > self.balance then error"insufficient funds"
end
  self.balance = self.balance - v
end
```

От этого класса мы можем унаследовать подкласс `SpecialAccount`, позволяющий покупателю снять больше, чем есть на его балансе. Мы начинаем с пустого класса, который просто наследует все свои операции от своего базового класса:

```
SpecialAccount = Account:new()
```

До сих пор `SpecialAccount` является лишь одним из экземпляров `Account`. Теперь происходит замечательная вещь:

```
s = SpecialAccount:new{limit=1000.00}
```

`SpecialAccount` наследует `new` от `Account`, как и любые другие методы. Однако, на этот раз при выполнении `new` его параметр `self` уже будет ссылаться на `SpecialAccount`. Поэтому метатаблицей `s` будет `SpecialAccount`, чье значение в поле `__index` тоже равно `SpecialAccount`. Поэтому `s` наследует от `SpecialAccount`, который, в свою очередь, наследует от `Account`. При вычислении

```
s:deposit(100.00)
```

Lua не сможет найти поле `deposit` в `s`, поэтому он будет искать его в `SpecialAccount`; там его он тоже не найдет и потому поищет в `Account`, где и обнаружит исходную реализацию этого метода.

Что делает `SpecialAccount` особенным, так это то, что мы

можем переопределить любой метод, унаследованный от его суперкласса. Все, что нам нужно, — это просто записать новый метод:

```
function SpecialAccount:withdraw (v)
  if v - self.balance >= self:getLimit() then
    error"insufficient funds"
  end
  self.balance = self.balance - v
end

function SpecialAccount:getLimit ()
  return self.limit or 0
end
```

Теперь, когда мы вызовем `s:withdraw(200.00)`, Lua не обратится к `Account`, поскольку первым он найдет новый метод `withdraw` в классе `SpecialAccount`. Так как `s.limit` равно 1000.00 (как вы помните, мы задали это поле при создании `s`), то программа осуществит снятие со счета, оставляя в результате `s` с отрицательным балансом.

У объектов в Lua есть интересный аспект: вам не нужно создавать новый класс для задания нового поведения. Если требуется изменить поведение лишь одного объекта, то мы можем реализовать данное поведение непосредственно в этом объекте. Например, если счет `s` представляет особого клиента, чей лимит всегда равен 10% от текущего баланса, то мы можем изменить лишь этот один счет:

```
function s:getLimit ()
  return self.balance * 0.10
end
```

После данного объявления вызов `s:withdraw(200.0)` выполнит метод `withdraw` из класса `SpecialAccount`, но когда `withdraw` вызовет `self:getLimit`, произойдет вызов только что определенной функции.

16.3. Множественное наследование

Поскольку в Lua объекты не являются примитивами, в нем есть несколько способов реализации объектно-ориентированного программирования. Подход с применением метаметода `__index`, который мы только что видели, является, наверное, лучшей комбинацией простоты, скорости и гибкости. Однако, есть и другие реализации, которые могут оказаться более подходящими в некоторых частных случаях. Ниже мы увидим альтернативную реализацию, которая обеспечивает множественное наследование в Lua.

Основой данной реализации является использование функции для метаполя `__index`. Вспомним о том, что когда у метатаблицы некоторой таблицы есть функция в поле `__index`, Lua будет вызывать эту функцию всякий раз, когда не сможет найти ключ в этой исходной таблице. При этом `__index` может искать отсутствующий ключ в любом количестве родительских классов.

Множественное наследование означает, что у класса может быть более одного суперкласса. Таким образом, мы не можем использовать метод класса для создания подклассов. Взамен с этой целью мы определим особую функцию `createClass`, у которой в качестве аргументов суперклассы нового класса (см. листинг 16.1). Эта функция создает таблицу для представления нового класса и устанавливает его метатаблицу с метаметодом `__index`, который и реализует множественное наследование. Несмотря на множественное наследование, каждый созданный объект по-прежнему принадлежит одному классу, в котором он ищет все свои методы. Поэтому отношения между классами и суперклассами отличается от отношений между классами и его экземплярами. В частности, класс не может одновременно быть метатаблицей для своих экземпляров и своих подклассов. В листинге 6.1 мы используем класс как метатаблицу для его экземпляров и создаем другую таблицу в

качестве метатаблицы класса.

Листинг 16.1. Реализация множественного наследования

```
-- ищет 'k' в списке таблиц 'plist'
local function search (k, plist)
  for i = 1, #plist do
    local v = plist[i][k]    -- пробует'i'-ый
суперкласс
    if v then return v end
  end
end

function createClass (...)
  local c = {}    -- новый класс
  local parents = {...}

  -- класс будет искать каждый метод в списке своих
родительских классов
  setmetatable(c, {__index = function (t, k)
    return search(k, parents)
  end})

  -- подготавливает 'c' стать метатаблицей для своих
экземпляров
  c.__index = c

  -- определяет новый конструктор для этого нового
класса
  function c:new (o)
    o = o or {}
    setmetatable(o, c)
    return o
  end

  return c    -- возвращает новый класс
end
```

Давайте проиллюстрируем использование `createClass` при помощи небольшого примера. Пусть у нас есть наш предыдущий класс `Account` и новый класс `Named` лишь с двумя методами: `setname` и `getname`.

```
Named = {}
function Named:getname ()
  return self.name
end

function Named:setname (n)
  self.name = n
end
```

Для создания нового класса `NamedAccount`, который является подклассом и `Account`, и `Named`, мы просто вызовем `createClass`:

```
NamedAccount = createClass(Account, Named)
```

Мы создаем и используем экземпляры этого класса как обычно:

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())    --> Paul
```

Теперь давайте проследим, как работает этот последний оператор. Lua не может найти поле "`getname`" в `account`; поэтому он ищет поле `__index` в метатаблице `account`, то есть в `NamedAccount`. Но в `NamedAccount` также нет поля "`getname`", поэтому Lua ищет поле `__index` в метатаблице `NamedAccount`. Поскольку это поле содержит функцию, Lua вызывает ее. Далее эта функция сперва ищет "`getname`" в `Account` и, не обнаружив его, проверяет `Named`, где она и находит отличное от nil значение, которое и становится окончательным результатом поиска.

Конечно, из-за сложной структуры такого поиска быстродействие множественного наследования не такое же, как у одиночного. Простым способом улучшить это быстродействие является копирование унаследованных методов в подклассы. С использованием этого подхода метаметод `__index` для классов будет выглядеть следующим образом:

```
setmetatable(c, {__index = function (t, k)
  local v = search(k, parents)
  t[k] = v     -- сохраняет для следующего обращения
```

```
    return v
  end})
```

При помощи данного приема доступ к унаследованным методам становится столь же быстрым, как и доступ к локальным методам (за исключением первого обращения). Недостаток состоит в том, что во время выполнения сложно изменить определения методов, поскольку эти изменения не передаются по цепочке наследования.

16.4. Конфиденциальность

Многие считают конфиденциальность (возможность скрытия элементов) неотъемлемой частью объектно-ориентированного языка; состояние каждого объекта является его личным делом. В некоторых объектно-ориентированных языках, таких как C++ и Java, вы можете управлять тем, будет ли поле объекта (также называемое *экземплярной переменной*) или его метод видны вне этого объекта. Язык Smalltalk, который популяризовал объектно-ориентированные языки, делает все переменные закрытыми (private), а все методы открытыми (public). Simula, самый первый объектно-ориентированный язык, вообще не обеспечивал защиту данных.

Основная схема работы с объектами в Lua, которую мы рассмотрели до этого, не предоставляет механизмы скрытия. Частично это является следствием нашего применения общих структур (таблиц) для представления объектов. Кроме того, Lua избегает чрезмерной избыточности и искусственных ограничений. Если вы не хотите обращаться к чему-либо внутри объекта, то *просто не делайте этого*.

Тем не менее, Lua стремится оставаться гибким, предлагая метамеханизмы, позволяющие моделировать множество прочих механизмов. Хотя базовая схема работы с объектами в Lua не предусматривает механизмы скрытия, мы можем реализовать

объекты иначе, и таким образом получить управление доступом. Хотя программисты нечасто применяют данную реализацию, узнать о ней будет полезным, поскольку она приоткрывает некоторые интересные аспекты Lua и может пригодиться для решения других задач.

Основная идея этой альтернативной схемы — представлять каждый объект при помощи двух таблиц: одна — для его состояния, а другая — для его действий (его интерфейс). Обращение к самому объекту происходит через вторую таблицу, то есть посредством действий, образующих его интерфейс. Во избежание несанкционированного доступа, таблица, представляющая состояние объекта, не хранится в поле другой таблицы; вместо этого она хранится лишь в замыкании методов. Например, применив данную схему для представления банковского счета, мы могли бы создавать новые объекты при помощи следующей функции-фабрики:

```
function newAccount (initialBalance)
  local self = {balance = initialBalance}

  local withdraw = function (v)
                     self.balance = self.balance - v
                   end

  local deposit = function (v)
                    self.balance = self.balance + v
                  end

  local getBalance = function () return self.balance
end

  return {
    withdraw = withdraw,
    deposit = deposit,
    getBalance = getBalance
  }
end
```

Сначала функция создает таблицу для содержания внутреннего

состояния объекта и хранит ее в локальной переменной `self`. Затем функция создает методы этого объекта. Наконец, функция создает и возвращает внешний объект, который отображает имена методов на их настоящие реализации. Ключевой момент здесь в том, что эти методы не получают `self` как дополнительный параметр; вместо этого они обращаются к `self` напрямую. Поскольку дополнительного аргумента нет, мы не используем синтаксис с двоеточием для работы с такими объектами. Мы вызываем их методы просто как обычные функции:

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance())    --> 60
```

Эта схема обеспечивает полную конфиденциальность всего, что хранится в таблице `self`. После возврата управления из функции `newAccount` нет никакого способа получить прямой доступ к этой таблице. Хотя наш пример помещает в таблицу скрытия лишь одну экземплярную переменную, мы можем хранить в этой таблице все закрытые части объекта. всего одну переменную в закрытой таблице, мы можем хранить все закрытые части объекта в этой таблице. Таким же образом мы можем определить и закрытые методы: они похожи на открытые, но мы не помещаем их в интерфейс. Например, наши счета могли бы предоставлять дополнительный 10%-ый кредит тем пользователям, чей баланс уже превысил определенный лимит, но нам не нужно, чтобы у пользователей был доступ к деталям таких расчетов. Мы можем реализовать эту функциональность следующим образом:

```
function newAccount (initialBalance)
  local self = {
    balance = initialBalance,
    LIM = 10000.00,
  }

  local extra = function ()
```

```
    if self.balance > self.LIM then
      return self.balance*0.10
    else
      return 0
    end
  end

  local getBalance = function ()
  return self.balance + extra()
  end

  <как прежде>
```

Опять же, ни один пользователь никоим образом не может обратиться к `extra` напрямую.

16.5. Подход с единственным методом

Частным случаем предыдущего подхода для объектно-ориентированного программирования является случай, когда у объекта есть лишь один метод. В подобном случае нам не нужно создавать интерфейсную таблицу; мы можем просто вернуть этот единственный метод в качестве представления объекта. Если это звучит немного странно, то стоит перечитать раздел 7.1, где мы конструировали итерирующие функции, хранящие свое состояние как замыкания. Итератор, хранящий свое состояние, — это ничто иное как объект с одним методом.

Другим интересным случаем объектов с единственным методом является случай, когда этот метод на самом деле диспетчером, который выполняет различные задачи в зависимости от выделенного с этой целью аргумента. Возможная реализация такого объекта приведена ниже:

```
function newObject (value)
  return function (action, v)
    if action == "get" then return value
    elseif action == "set" then value = v
```

```
      else error("invalid action")
      end
    end
  end
```

Его использование не вызывает затруднений:

```
d = newObject(0)
print(d("get"))    --> 0
d("set", 10)
print(d("get"))    --> 10
```

Такая необычная реализация объектов весьма эффективна. Синтаксис `d("set",10)` хотя и выглядит странно, всего на два символа длиннее, чем более традиционный `d:set(10)`. Каждый объект использует одно единственное замыкание, что дешевле одной таблицы. Здесь нет наследования, но зато у нас есть полная конфиденциальность: обратиться к состоянию объекта можно лишь одним способом — через его единственный метод.

Tcl/Tk использует схожий подход для своих виджетов. Имя виджета в Tk обозначает функцию (*команду виджета*), которая может выполнять все виды действий над виджетом.

Упражнения

Упражнение 16.1. Реализуйте класс `Stack` с методами `push`, `pop`, `top` и `isempty`.

Упражнение 16.2. Реализуйте класс `StackQueue` как подкласс `Stack`. Кроме унаследованных методов, добавьте к этому классу метод `insertbottom`, который вставляет элемент в конец стека. (Этот метод позволяет использовать объекты данного класса как очереди.)

Упражнение 16.3. Другой способ обеспечить конфиденциальность объектов — это реализовать их через посредников (см. раздел 13.4). Каждый объект представлен пустой

таблицей-посредником. Внутренняя таблица отображает посредников на таблицы, хранящие состояние объекта. Эта внутренняя таблица не доступна снаружи, но методы используют ее для перевода своего параметра `self` в реальную таблицу, с которой они работают. Реализуйте пример с классом `Account` при помощи этого подхода и рассмотрите его плюсы и минусы.

(С этим подходом есть одна маленькая проблема. Постарайтесь найти ее сами или обратитесь к разделу 17.3, где предлагается ее решение.)

ГЛАВА 17

Слабые таблицы и финализаторы

Lua осуществляет автоматическое управление памятью. Программы создают объекты (таблицы, нити и т. п.), но функции уничтожения объектов не существует. Lua автоматически уничтожает объекты, которые становятся мусором, при помощи *сборки мусора*. Это освобождает вас от бремени управления памятью и, что более важно, освобождает от большинства ошибок, связанных с этой деятельностью, таких как повисшие указатели и утечки памяти.

Применение сборщика мусора означает, что у Lua нет проблем с циклами. Вам не нужно предпринимать никаких специальных действий при использовании циклических структур данных; они освобождаются автоматически, как и любые другие данные. Тем не менее, иногда даже умному сборщику мусора нужна ваша помощь. Ни один сборщик мусора не позволит вам забыть обо всех проблемах управления ресурсами, таких как переполнение памяти и внешние ресурсы.

Слабые таблицы и финализаторы — это механизмы, которые вы можете использовать в Lua, чтобы помочь сборщику мусора. Слабые таблицы позволяют сбор объектов Lua, которые все еще доступны программе, в то время как финализаторы позволяют сборку внешних объектов, не находящихся под непосредственным контролем сборщика мусора. В этой главе мы обсудим оба этих механизма.

17.1. Слабые таблицы

Сборщик мусора может собрать только то, что гарантированно

является мусором; он не может догадаться, что считаете мусором именно вы. Типичным примером является стек, реализованный как массив с индексом для вершины стека. Вы знаете, что допустима лишь та часть массива, которая идет до вершины, но этого не знает Lua. Если вы выталкиваете элемент, просто уменьшая индекс вершины, то оставшийся в массиве объект не является мусором для Lua. Точно так же любой объект, который хранится в глобальной переменной, не будет мусором для Lua, даже если ваша программа никогда не воспользуется им снова. В обоих случаях вам (т.е. вашей программе) придется позаботиться о присваивании nil этим позициям, чтобы они в дальнейшем не помешали уничтожению данного объекта.

Тем не менее, простой чистки ваших ссылок не всегда достаточно. Для некоторых конструкций нужно организовать дополнительное взаимодействие между программой и сборщиком мусора. Типичным примером является хранение в вашей программе коллекции всех живых объектов определенного вида (например, файлов). Задача кажется простой: вам лишь требуется вставлять каждый новый объект в коллекцию. Однако, как только объект становится частью этой коллекции, его уничтожение становится невозможным! Даже если на него не ссылаются другие объекты, на него ссылается сама коллекция. Lua не может знать о том, что эта ссылка не должна препятствовать утилизации данного объекта, если только вы не сообщили Lua этот факт.

Слабые таблицы — это тот механизм, который вы используете, чтобы указать Lua на то, что ссылка не должна препятствовать уничтожению объекта. *Слабая ссылка* (weak reference) — это такая ссылка на объект, которая не учитывается сборщиком мусора. Если все ссылки, указывающие на объект, являются слабыми, то данный объект утилизируется, а эти слабые ссылки каким-либо образом удаляются. Lua реализует слабые ссылки как слабые таблицы: *слабая таблица* (weak table) — это такая таблица, все ссылки которой являются слабыми. Это значит, что если объект хранится

только внутри слабых таблиц, то Lua со временем его утилизирует.

У таблиц есть ключи и значения, которые при этом могут содержать любые виды объектов. В обычных обстоятельствах сборщик мусора не утилизирует объекты, которые являются ключами и ссылками в открытой для доступа таблице. То есть и ключи, и значения являются *сильными ссылками* (strong reference), то есть они предотвращают утилизацию тех объектов, на которые они указывают. В слабой таблице и ключи, и значения могут быть слабыми. Это значит, что существуют три вида слабых таблиц: таблицы со слабыми ключами, таблицы со слабыми значениями и полностью слабые таблицы, где и ключи, и значения являются слабыми. Независимо от вида таблицы, при уничтожении ключа или значения из нее удаляется вся запись.

Слабость таблицы задается полем `__mode` ее метатаблицы. Значение этого поля, когда оно присутствует, должно быть строкой; если эта строка равна "`k`", то ключи в этой таблице являются слабыми; если эта строка равна "`v`", то слабыми являются значения в этой таблице; если эта строка равна "`kv`", то и ключи, и значения в данной таблице являются слабыми. Следующий пример, хотя и искусственный, показывает основное поведение слабых таблиц:

```
a = {}
b = {__mode = "k"}
setmetatable(a, b)    -- теперь у 'a' есть слабые
ключи
key = {}              -- создает первый ключ
a[key] = 1
key = {}              -- создает второй ключ
a[key] = 2
collectgarbage()      -- принудительно задействует
цикл сборки мусора
for k, v in pairs(a) do print(v) end
  --> 2
```

В этом примере второе присваивание `key={}` перезаписывает ссылку на первый ключ. Вызов `collectgarbage` заставляет сборщик

мусора провести полную утилизацию. Поскольку на первый ключ больше не осталось ссылок, этот ключ утилизируется, а соответствующая запись в таблице удаляется. Однако второй ключ по-прежнему хранится в переменной `key` и поэтому не утилизируется.

Обратите внимание, что из слабой таблицы могут быть утилизированы лишь объекты. Значения, например числа и логические значения, сборщиком не собираются. Например, если мы вставим числовой ключ в таблицу `a` (из нашего предыдущего примера), то сборщик мусора никогда его не удалит. Разумеется, если значение, соответствующее числовому ключу, хранится в таблице со слабыми значениями, то из нее удаляется вся соответствующая запись.

Со строками есть одна тонкость: хотя строки и утилизируются сборщиком мусора, с точки зрения реализации они отличаются от остальных утилизируемых объектов. Другие объекты, такие как таблицы и нити, создаются явно. Например, когда Lua вычисляет выражение `{}`, то он создает новую таблицу. Однако, создает ли Lua новую строку при вычислении `"a".."b"`? Что, если в системе уже есть строка `"ab"`? Создаст ли Lua новую строку? Может ли компилятор создать эту строку перед выполнением программы? Это не имеет никакого значения: это все детали реализации. С точки зрения программиста, строки являются значениями, а не объектами. Поэтому, так же как и число или логическое значение, строка не удаляется из слабой таблицы (кроме случая, когда удаляется связанное с ней значение).

17.2. Функции с запоминанием

Распространенным программистским приемом является получение выигрыша во времени за счет проигрыша по памяти. Вы можете ускорить функцию посредством *запоминания* (memorizing)

ее результатов, чтобы в дальнейшем, когда вы вызовете эту же функцию с теми же аргументами, функция смогла воспользоваться тем же результатом.

Представьте себе обычный сервер, получающий запросы в виде строк, содержащих код Lua. Каждый раз при получении запроса сервер выполняет `load` для полученной строки и затем вызывает полученную функцию. Однако `load` — затратная функция, и некоторые команды для сервера могут довольно часто повторяться. Вместо повторного вызова `load` каждый раз, когда сервер получает распространенную команду вроде "`closeconnection()`", сервер может *запомнить* результат `load` при помощи вспомогательной таблицы. Перед вызовом `load` сервер проверяет по этой таблице — не была ли уже преобразована данная строка. Если сервер не может найти эту строку, тогда (и только тогда) он вызывает `load` и сохраняет результат в этой таблице. Мы можем заложить это поведение в новую функцию:

```
local results = {}
function mem_loadstring (s)
  local res = results[s]
  if res == nil then          -- результат не доступен?
    res = assert(load(s))     -- вычисляет новый
результат
    results[s] = res          -- сохраняет для
последующего переиспользования
  end
  return res
end
```

Выигрыш от этой схемы может быть огромным. Однако, она также может привести к непредвиденным затратам ресурсов. Хотя некоторые команды повторяются снова и снова, многие другие команды выполняются лишь однажды. Таблица `results` постепенно собирает все команды, которые сервер когда-либо получал, вместе с их кодами; спустя некоторое время это может привести к исчерпанию памяти на сервере. Слабые таблицы

предоставляют простое решение данной проблемы. Если таблица `results` хранит слабые значения, то каждый цикл сборки мусора удалит все неиспользуемые на данный момент преобразования (то есть практически все):

```
local results = {}
setmetatable(results, {__mode = "v"})    -- делает
значения слабыми
function mem_loadstring (s)
  <как прежде>
```

На самом деле, поскольку индексы всегда являются строками, при желании мы можем сделать эту таблицу полностью слабой:

```
setmetatable(results, {__mode = "kv"})
```

Окончательный результат такой же.

Техника запоминания также полезна, когда нужно быть уверенными в уникальности объекта определенного вида. Например, представим систему, в которой цвета представлены таблицами с полями `red`, `green` и `blue`. Примитивная фабрика цветов генерирует новый цвет при каждом новом запросе:

```
function createRGB (r, g, b)
  return {red = r, green = g, blue = b}
end
```

Используя технику запоминания, мы можем многократно использовать одну и ту же таблицу для одного и того же цвета. При создании уникального ключа для каждого цвета мы просто конкатенируем индексы цвета посредством какого-либо разделителя:

```
local results = {}
setmetatable(results, {__mode = "v"})    -- делает
значения слабыми
function createRGB (r, g, b)
  local key = r .. "-" .. g .. "-" .. b
  local color = results[key]
  if color == nil then
```

```
        color = {red = r, green = g, blue = b}
        results[key] = color
      end
      return color
    end
```

Интересным последствием этой реализации является то, что пользователь может сравнивать цвета при помощи стандартной операции сравнения, поскольку два сосуществующих одинаковых цвета всегда представлены одной и той же таблицей. Обратите внимание, что данный цвет может быть представлен разными таблицами в разные моменты времени, поскольку время от времени цикл сборки мусора очищает таблицу `results`. Однако, пока данный цвет используется, он не может быть удален из `results`. Поэтому, когда цвет существует так долго, что его сравнивают с новым цветом, его представление также будет существовать достаточно долго, чтобы пригодиться для нового цвета.

17.3. Атрибуты объекта

Другой важной областью применения слабых таблиц является связывание атрибутов с объектами. Существует множество ситуаций, в которых нам нужно прикрепить некоторый атрибут к объекту: имена к функциям, значения по умолчанию к таблицам, размеры к массивам и т. д.

Когда объект является таблицей, мы можем хранить атрибут в самой таблице, подобрав подходящий уникальный ключ. Как мы уже видели, простой и безошибочный способ создать уникальный ключ — это создать новый объект (обычно таблицу) и применять его в качестве ключа. Однако, если объект не является таблицей, то он не может хранить свои собственные атрибуты. Даже в случае таблиц нам не всегда нужно хранить атрибут в исходном объекте. Например, нам может понадобится хранить подобный атрибут закрытым или нам не нужно, чтобы атрибут мешал обходу

таблицы. Во всех этих случаях нам нужен иной способ связывания атрибутов с объектами.

Конечно, внешняя таблица предоставляет идеальный способ присоединения атрибутов к объектам (не случайно, что таблицы иногда называют *ассоциативными массивами*). Мы используем объекты как ключи, а их атрибуты — как значения. Внешняя таблица может хранить атрибуты объектов любого типа, так как Lua позволяет использовать объекты любого типа в качестве ключей таблицы. Более того, атрибуты, хранящиеся во внешней таблице, не влияют на другие объекты и могут быть закрытыми, так же как и сама таблица.

Однако, это на первый взгляд идеальное решение обладает огромным недостатком: как только мы использовали объект в качестве ключа в таблице, мы обрекли его на вечное существование. Lua не может утилизировать объект, который используется в качестве ключа. Если мы используем обычную таблицу , чтобы привязать к функциям их имена, то ни одна из этих функций никогда не будет удалена. Как вы вероятно догадались, мы можем избежать этого недостатка при помощи слабых таблиц. Однако, на этот раз нам понадобятся слабые ключи. Применение слабых ключей не предотвращает их утилизацию, когда на них не остается больше ссылок. С другой стороны, у таблицы не могут быть слабые значения; иначе атрибуты существующих объектов могли бы быть утилизированы.

17.4. Вновь таблицы со значениями по умолчанию

В разделе 13.4 мы обсуждали, как реализовать таблицы со значениями по умолчанию, отличными от nil. Мы рассмотрели один частный подход и заметили, что в двух других подходах применяются слабые таблицы, поэтому их мы отложили на потом. Теперь пора вернуться к этой теме. Как вы увидите, эти два

решения задачи реализации значений по умолчанию на самом деле являются частными случаями уже рассмотренных подходов: атрибутов объектов и запоминания.

В первом решении мы используем слабую таблицу, чтобы связать с каждой таблицей ее значения по умолчанию:

```
local defaults = {}
setmetatable(defaults, {__mode = "k"})
local mt = {__index = function (t) return defaults[t]
end}
function setDefault (t, d)
  defaults[t] = d
  setmetatable(t, mt)
end
```

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

Во втором решении мы используем разные метатаблицы для разных значений по умолчанию, но при этом мы многократно используем одну и ту же метатаблицу при каждом повторном использовании значения по умолчанию. Это типичное применение запоминания:

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

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

Какая из этих двух реализаций является лучшей? Как обычно, это зависит от обстоятельств. У обеих схожая сложность и схожее

быстродействие. Первая реализация требует нескольких слов (1 слово = 2 байта) памяти для каждой таблицы со значением по умолчанию (для записей в `defaults`). Вторая реализация требует нескольких десятков слов памяти для каждого отдельного значения по умолчанию (новая таблица, новое замыкание и запись в `metas`). Поэтому если в вашем приложении тысячи таблиц с всего несколькими различными значениями по умолчанию, то второе решение явно будет лучше. С другой стороны, если несколько таблиц обладают общими значениями по умолчанию, то вам лучше предпочесть первую реализацию.

17.5. Эфемерные таблицы

Сложная ситуация возникает, когда в таблице со слабыми ключами значение ссылается на свой собственный ключ.

Этот случай гораздо более распространен, чем может показаться. В качестве типичного примера возьмем фабрику функций-констант. Подобная фабрика получает объект и возвращает функцию, которая при вызове возвращает этот объект:

```
function factory (o)
  return function () return o end
end
```

Эта фабрика является хорошим кандидатом для запоминания, чтобы не создавать новое замыкание, когда уже есть готовое:

```
do
  local mem = {}
  setmetatable(mem, {__mode = "k"})
  function factory (o)
    local res = mem[o]
    if not res then
      res = function () return o end
      mem[o] = res
    end
    return res
```

```
    end
  end
```

Однако, здесь есть один подвох. Обратите внимание, что значение (функция-константа), связанное с объектом внутри `mem`, ссылается на свой собственный ключ (сам объект). Хотя ключи в этой таблице слабые, значения слабыми не являются. При стандартной интерпретации слабых таблиц ничто не будет удалено из таблицы с запоминанием. Поскольку значения не являются слабыми, всегда есть сильная ссылка на каждую функцию. Каждая функция ссылается на свой соответственный объект, поэтому на каждый объект всегда есть сильная ссылка. Таким образом, несмотря на слабые ключи, эти объекты не будут утилизированы.

Однако, эта ограниченная интерпретация не очень удобна. Большинство ожидает, что значение в таблице доступно только через соответствующий ключ. Поэтому мы можем рассматривать вышеописанный случай в качестве разновидности цикла, где замыкание ссылается на объект, который (через таблицу с запоминанием) в свою очередь ссылается на это замыкание.

Lua 5.2 решает данную проблему при помощи концепции эфемерных таблиц. В Lua 5.2 таблица со слабыми ключами и сильными значениями является *эфемерной таблицей* (ephemeron table). В эфемерной таблице доступность ключа управляет доступностью соответствующего значения. В частности, рассмотрим запись (*k*, *v*) в эфемерной таблице. Ссылка на *v* является сильной, только если существует сильная ссылка на *k*. В противном случае запись со временем удаляется из таблицы, даже если *v* ссылается (прямо или косвенно) на *k*.

17.6. Финализаторы

Хотя задачей сборщика мусора является утилизация объектов Lua, он также может помочь программам с освобождением внешних

ресурсов. С этой целью некоторые языки программирования предлагают механизм финализаторов. *Финализатор* (finalizer) — это функция, связанная с объектом, которая вызывается перед тем, как объект будет удален сборщиком мусора.

Lua реализует финализаторы при помощи метаметода `__gc`. Посмотрите на следующий пример:

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

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

У финализаторов Lua есть один нюанс, связанный с пометкой объекта для финализации. Мы помечаем объект для финализации, когда задаем для него метатаблицу с ненулевым метаметодом `__gc`. Если мы не пометим объект, то он не будет финализирован. Большая часть кода, который мы пишем, работает ожидаемым образом, но иногда возникают странные случаи вроде следующего:

```
o = {x = "hi"}
mt = {}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage()    --> (ничего не печатает)
```

В этом примере метатаблица, которую мы устанавливаем для `o`, не содержит метаметода `__gc`, поэтому объект и не помечается для финализации. Даже если потом добавить поле `__gc` метатаблице, Lua не посчитает это присваивание каким-то особенным, так что объект

не будет помечен. Как мы и сказали, это редко бывает проблемой; метаметоды редко изменяются после начала использования метатаблицы.

Если вы действительно хотите задать метаметод позже, то вы можете использовать любое значение для поля `__gc` в качестве временного:

```
o = {x = "hi"}
mt = {__gc = true}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage()    --> hi
```

Теперь, поскольку метатаблица содержит поле `__gc`, объект `o` надлежащим образом помечается для финализации. Нет никакой проблемы в том, чтобы задать метаметод позже; Lua вызывает финализатор, только если он является соответственной функцией.

Когда сборщик мусора утилизирует несколько объектов в одном и том же цикле, он вызывает их финализаторы в порядке, обратном тому, в котором объекты были помечены для финализации. Рассмотрим следующий пример, который создает связанный список объектов с финализаторами:

```
mt = {__gc = function (o) print(o[1]) end}
list = nil
for i = 1, 3 do
  list = setmetatable({i, link = list}, mt)
end
list = nil
collectgarbage()
  --> 3
  --> 2
  --> 1
```

Первым финализируемым объектом будет объект 3, который был последним помеченным объектом.

Считать, что ссылки между утилизируемыми объектами могут

повлиять на порядок их финализации — распространенное заблуждение. Например, можно подумать, что объект 2 в предыдущем примере должен быть финализирован перед объектом 1, поскольку существует ссылка от 2 к 1. Однако, ссылки могут формировать циклы. Поэтому они никак не влияют на порядок финализации.

Другим неочевидным моментом, связанным с финализаторами, является *восстановление* (resurrection). При своем вызове финализатор получает в качестве параметра финализируемый объект. Таким образом, объект снова становится живым, по крайней мере на время финализации. Я называю это *временным восстановлением* (transient resurrection). Во время выполнения финализатора ничто не мешает ему сохранить объект, скажем, в глобальной переменной, чтобы объект остался доступным после возврата из финализатора. Я называю это *перманентным восстановлением* (permanent resurrection).

Восстановление должно быть временным. Рассмотрим следующий фрагмент кода:

```
A = {x = "this is A"}
B = {f = A}
setmetatable(B, {__gc = function (o) print(o.f.x)
end})
A, B = nil
collectgarbage()    --> this is A
```

Финализатор для `B` обращается к `A`, поэтому `A` не может быть удален до финализации `B`. Lua должен восстановить и `A`, и `B` перед вызовом финализатора.

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

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

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

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

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

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

  -- создает первый объект
  setmetatable({}, mt)
end
```

```
collectgarbage()    --> новый цикл
collectgarbage()    --> новый цикл
collectgarbage()    --> новый цикл
```

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

```
-- таблица со слабыми ключами
wk = setmetatable({}, {__mode = "k"})

-- таблица со слабыми значениями
wv = setmetatable({}, {__mode = "v"})

o = {}    -- объект
wv[1] = o; wk[o] = 10    -- добавляет их к обеим
таблицам

setmetatable(o, {__gc = function (o)
print(wk[o], wv[1])
end})

o = nil; collectgarbage()    -->    10    nil
```

Во время выполнения финализатора он находит объект в таблице `wk`, но не в таблице `wv`. Обоснованием такого поведения является то, что мы часто храним свойства объекта в таблицах со слабыми ключами (как мы обсудили в разделе 17.3), и финализаторам может понадобиться доступ к этим атрибутам. Однако, мы используем таблицы со слабыми значениями для многократного использования живых объектов; в этом случае финализируемые объекты становятся бесполезны.

Упражнения

Упражнение 17.1. Напишите проверочный код, чтобы определить, действительно ли Lua использует эфемерные таблицы. (Не забудьте вызвать `collectgarbage` для принудительного цикла сборки мусора.) По возможности проверьте ваш код как в Lua 5.1, так и в Lua 5.2, чтобы увидеть разницу.

Упражнение 17.2. Рассмотрим первый пример из раздела 17.6, который создает таблицу с финализатором, печатающим сообщение лишь при вызове. Что произойдет, если программа завершится без цикла сборки мусора? Что случится, если программа вызовет `os.exit`? Что произойдет, если программа завершит свое выполнение с ошибкой?

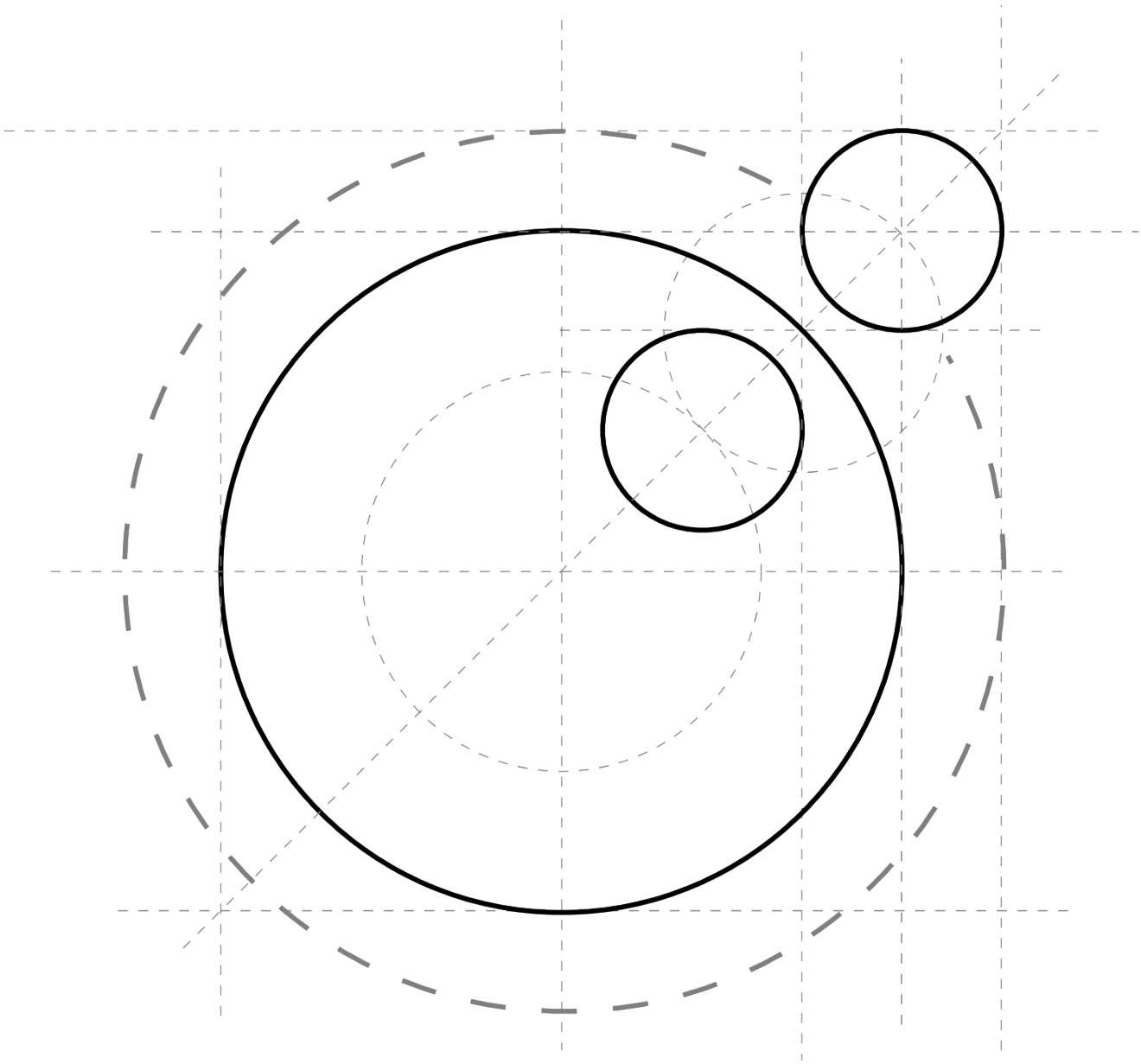
Упражнение 17.3. Допустим, вам нужно реализовать таблицу с запоминанием для функции, получающей строку и возвращающей строку. Применение слабой таблицы не позволит удалять записи, поскольку слабые таблицы не рассматривают строки как удаляемые объекты. Как вы можете реализовать запоминание в этом случае?

Упражнение 17.4. Объясните вывод следующей программы:

```
local count = 0

local mt = {__gc = function () count = count - 1 end}
local a = {}

for i = 1, 10000 do
  count = count + 1
  a[i] = setmetatable({}, mt)
end

collectgarbage()
print(collectgarbage"count" * 1024, count)
a = nil
collectgarbage()
print(collectgarbage"count" * 1024, count)
collectgarbage()
print(collectgarbage"count" * 1024, count)
```

Часть III

Стандартные библиотеки



ГЛАВА 18

Математическая библиотека

В этой и следующих главах, посвященных стандартным библиотекам, моей целью является не дать полную спецификацию каждой функции, а показать, какую функциональность предоставляет каждая библиотека. Для ясности изложения я могу опускать некоторые специфические опции или режимы работы. Главной целью является зажечь в вас любопытство, которое затем может быть удовлетворено чтением справочника Lua.

Библиотека `math` содержит стандартный набор математических функций, таких как тригонометрические функции (`sin`, `cos`, `tan`, `asin`, `acos` и т. п.), возведение в степень и логарифмирование (`exp`, `log`, `log10`), функции округления (`floor`, `ceil`), `min`, `max`, функции для генерации псевдослучайных чисел (`random`, `randomseed`), переменная `pi` и переменная `huge`, которая является наибольшим представимым в Lua числом (на некоторых платформах может принимать специальное значение *inf*).

Все тригонометрические функции работают с радианами. Вы можете использовать функции `deg` и `rad` для перевода между градусами и радианами. Если вы хотите работать с градусами, вы можете переопределить тригонометрические функции:

```
do
  local sin, asin, ... = math.sin, math.asin, ...
  local deg, rad = math.deg, math.rad
  math.sin = function (x) return sin(rad(x)) end
  math.asin = function (x) return deg(asin(x)) end
  ...
end
```

Функция `math.random` генерирует псевдослучайные числа. Вы

можете вызывать ее тремя способами. Когда мы вызываем ее без аргументов, она возвращает вещественное псевдослучайное число с равномерным распределением в диапазоне [0,1). Когда мы вызываем ее с единственным аргументом, целочисленным *n*, то она возвращает псевдослучайное целое число *x*, такое, что $1 \leq x \leq n$. Например, вы можете смоделировать бросок кубика при помощи `random(6)`. Наконец, мы можем вызвать `random` с двумя целочисленными аргументами *l* и *u*, чтобы получить псевдослучайное целое число *x*, такое что $l \leq x \leq u$.

Вы можете задать затравку (seed) для генератора псевдослучайных чисел при помощи функции `randomseed`; затравка является ее единственным числовым аргументом. Обычно, при запуске программы она инициализирует генератор некоторым фиксированным значением. Это значит, что каждый раз, когда вы запускаете вашу программу, она генерирует одну и ту же последовательность псевдослучайных чисел. Для отладки это качество весьма удобно, но в игре все будет происходить по одному и тому же сценарию снова и снова. Распространенным приемом для решения данной проблемы является использование в качестве затравки текущего времени путем вызова `math.randomseed(os.time())`. Функция `os.time` возвращает число, представляющее текущее время, обычно в виде числа секунд, прошедших с определенного момента времени.

Функция `math.random` использует функцию `rand` из стандартной библиотеки C. В некоторых реализациях эта функция возвращает числа с не очень хорошими статистическими свойствами. Вы можете попробовать отыскать более удачный генератор псевдослучайных чисел в независимых дистрибутивах. (Стандартная поставка Lua не включает в себя подобного генератора во избежание проблем с авторским правом. Она содержит только код, написанный авторами Lua.)

Упражнения

Упражнение 18.1. Напишите функцию для проверки того, является ли заданное число степенью двойки.

Упражнение 18.2. Напишите функцию для расчета объема прямого кругового конуса по его высоте и углу между его образующей и осью.

Упражнение 18.3. Реализуйте другой генератор псевдослучайных чисел в Lua. Поищите хороший алгоритм в Интернете. (Вам может понадобиться побитовая библиотека; см. главу 19.)

Упражнение 18.4. Используя `math.random`, напишите функцию для получения псевдослучайных чисел со стандартным нормальным распределением (по Гауссу).

Упражнение 18.5. Напишите функцию для перемешивания заданного списка. Убедитесь, что все варианты равновероятны.

ГЛАВА 19

Побитовая библиотека

Источником постоянных жалоб насчет Lua является отсутствие в нем побитовых операций. Это отсутствие вовсе не случайно. Не так легко помирить побитовые операции с числами с плавающей точкой.

Мы можем выразить некоторые побитовые операции как арифметические операции. Например, сдвиги влево соответствуют умножению на степени двух, сдвиги направо соответствуют делению, Однако, у побитовых AND и OR нет таких арифметических аналогов. Они определены для двоичных представлений целых чисел. Практически невозможно расширить их на операции с плавающей точкой. Даже некоторые простые операции теряют смысл. Что должно быть дополнением 0.0? Должно ли это быть равно -1? Или 0xFFFFFFFF (что в Lua равно 4 294 967 295, что явно неравно -1)? Или может быть 2^{64}-1 (число, которое нельзя точно представить при помощи значения типа double)?

Во избежание подобных проблем, Lua 5.2 вводит побитовые операции при помощи библиотеки, а не как встроенные в язык операции. Это делает ясным, что данные операции не являются «родными» для чисел в Lua, но они используют определенную интерпретацию для работы с этими числами. Более того, другие библиотеки могут предложить иные интерпретации побитовых операций (например, используя более 32 битов).

Для большинства примеров в этой главе я буду использовать шестнадцатеричную запись. Я буду использовать слово max для обозначения 0xFFFFFFFF (то есть 2^{32}-1). В примерах я буду использовать следующую дополнительную функцию:

```
function printx (x)
  print(string.format("0x%X", x))
```

```
end
```

Побитовая библиотека в Lua 5.2 называется `bit32`. Как следует из имени, она работает с 32-битовыми числами. Поскольку **and**, **or** и **not** являются зарезервированными в Lua словами, то соответствующие функции названы `band`, `bor` и `bnot`. Для последовательности в названиях функция побитового исключающего ИЛИ названа `bxor`:

```
printx(bit32.band(0xDF, 0xFD))    --> 0xDD
printx(bit32.bor(0xD0, 0x0D))     --> 0xDD
printx(bit32.bxor(0xD0, 0xFF))    --> 0x2F
printx(bit32.bnot(0))             --> 0xFFFFFFFF
```

Функции `band`, `bor` и `bxor` принимают любое количество аргументов:

```
printx(bit32.bor(0xA, 0xA0, 0xA00))        --> 0xAAA
printx(bit32.band(0xFFA, 0xFAF, 0xAFF))    --> 0xAAA
printx(bit32.bxor(0, 0xAAA, 0))            --> 0xAAA
printx(bit32.bor())                        --> 0x0
printx(bit32.band())                       -->
0xFFFFFFFF
printx(bit32.bxor())                       --> 0x0
```

(Они все коммутативны и ассоциативны.)

Побитовая библиотека работает с беззнаковыми целыми числами. В ходе работы любое число, переданное как аргумент, приводится к целому числу в диапазоне 0—`MAX`. Во-первых, неуказанные числа округляются неуказанным способом. Во-вторых, числа вне диапазона 0—`MAX` приводятся к нему при помощи операции остатка от деления: целое *n* становится $n\%2^{32}$. Эта операция эквивалентна получению двоичного представления числа и затем взятию его младших 32 бит. Как и ожидается, -1 становится `MAX`. Вы можете использовать следующие операции для нормализации числа (то есть отображения его в диапазон 0-`MAX`):

```
printx(bit32.bor(2^32))    --> 0x0
```

```
printx(bit32.band(-1))     --> 0xFFFFFFFF
```

Конечно, в стандартном Lua легче просто выполнить `n%(2^32)`.

Если явно не указано, все функции в библиотеке возвращают результат, который также лежит в 0—`MAX`. Однако, вам следует быть осторожными при использовании результатов побитовых операций в качестве обычных чисел. Иногда Lua компилируется, используя другой тип для чисел. В частности, некоторые системы с ограниченными возможностями используют 32-битовые числа в качестве чисел в Lua. В этих системах `MAX` равен -1. Более того, некоторые побитовые библиотеки используют различные соглашения для своих результатов. Поэтому всякий раз, когда вам нужно использовать результат побитовой операции в качестве числа, будьте осторожны. Избегайте сравнений: вместо `x<0` напишите `bit32.btest(x,0x80000000)`. (Мы скоро увидим функцию `btest`.) Используйте саму побитовую библиотеку для нормализации констант:

```
if bit32.or(a, b) == bit32.or(-1) then
  <какой-нибудь код>
```

Побитовая библиотека также определяет операции для сдвига и вращения бит: `lshift` для сдвига налево; `rshift` и `arshift` (арифметический сдвиг) для сдвига направо; `lrotate` для вращения налево и `rrotate` для вращения направо. За исключением арифметического сдвига (`arshift`), все сдвиги заполняют новые биты нулями. Арифметический сдвиг заполняет биты слева копиями своего последнего бита («сигнальным битом»):

```
printx(bit32.rshift(0xDF, 4))           --> 0xD
printx(bit32.lshift(0xDF, 4))           --> 0xDF0
printx(bit32.rshift(-1, 28))            --> 0xF
printx(bit32.arshift(-1, 28))           --> 0xFFFFFFFF
printx(bit32.lrotate(0xABCDEF01, 4))    --> 0xBCDEF01A
printx(bit32.rrotate(0xABCDEF01, 4))    --> 0x1ABCDEF0
```

Сдвиг или вращение на отрицательное число бит сдвигает

(вращает) в противоположную сторону. Например, сдвиг на -1 бит направо эквивалентен сдвигу на 1 бит влево. Результат сдвига на более чем 31 бит равен 0 или `MAX`, поскольку все исходные биты пропали:

```
printx(bit32.lrotate(0xABCDEF01, -4))     -->
0x1ABCDEF0
printx(bit32.lrotate(0xABCDEF01, -36))    -->
0x1ABCDEF0
printx(bit32.lshift(0xABCDEF01, -36))     --> 0x0
printx(bit32.rshift(-1, 34))              --> 0x0
printx(bit32.arshift(-1, 34))             -->
0xFFFFFFFF
```

Кроме этих, более или менее стандартных операций, побитовая библиотека также предоставляет три дополнительные функции. Функция `btest` осуществляет ту же операцию, что и `band`, но возвращает результат сравнения побитовой операции с нулем:

```
print(bit32.btest(12, 1)) --> false
print(bit32.btest(13, 1)) --> true
```

Другой распространенной операцией является извлечение заданных битов из числа. Обычно эта операция включает в себя сдвиг и побитовое AND; побитовая библиотека упаковывает все это в одну функцию. Вызов `bit32.extract(x, f,w)` возвращает `w` бит из `x`, начиная с бита `f`:

```
printx(bit32.extract(0xABCDEF01, 4, 8))      --> 0xF0
printx(bit32.extract(0xABCDEF01, 20, 12))    --> 0xABC
printx(bit32.extract(0xABCDEF01, 0, 12))     --> 0xF01
```

Эта операция считает биты от 0 до 31. Если третий аргумент `(w)` не задан, то он считается равным единице:

```
printx(bit32.extract(0x0000000F, 0))     --> 0x1
printx(bit32.extract(0xF0000000, 31))    --> 0x1
```

Обратной к операции `extract` является операция `replace`,

которая заменяет заданные биты. Первым параметром является исходное число. Второй параметр задает значение, которое надо вставить. Последние два параметра, `f` и `w`, имеют тот же смысл, что и в `bit32.extract`:

```
printx(bit32.replace(0xABCDEF01, 0x55, 4, 8))    -->
0xABCDE551
printx(bit32.replace(0xABCDEF01, 0x0, 4, 8))     -->
0xABCDE001
```

Обратите внимание, что для любых допустимых значений `x`, `f` и `w` выполняется следующее равенство:

```
assert(bit32.replace(x, bit32.extract(x, f, w), f, w)
== x)
```

Упражнения

Упражнение 19.1. Напишите функцию для проверки того, что заданное число является степенью двух.

Упражнение 19.2. Напишите функцию для вычисления веса Хэмминга заданного целого числа. (*Вес Хэмминга* для числа — это количество единичных бит в двоичном представлении этого числа).

Упражнение 19.3. Напишите функцию для проверки того, является ли двоичное представление числа палиндромом (перевертышем).

Упражнение 19.4. Определите операции сдвига и побитовый AND при помощи арифметических операций Lua.

Упражнение 19.5. Напишите функцию, которая получает строку, закодированную в UTF-8, и возвращает ее первый символ как число. Функция должна вернуть nil, если строка не начинается с допустимой в UTF-8 последовательности.

ГЛАВА 20

Табличная библиотека

Библиотека `table` состоит из вспомогательных функций для работы с таблицами как с массивами. Она предоставляет функции для вставки и удаления элементов из списков, для сортировки элементов массива и для конкатенации всех строк в массиве.

20.1. Функции `insert` и `remove`

Функция `table.insert` вставляет элемент в заданную позицию массива, отодвигая остальные элементы, чтобы освободить место. Например, если `t` — это массив `{10, 20, 30)`, то после вызова `table.insert(t,1,15)` массив `t` станет равен `{15, 10, 20, 30}`. В качестве особого (и частого) случая, если мы вызовем `insert` без указания позиции, то она вставит элемент в последнюю позицию массива (и потому не станет сдвигать элементы). В качестве примера следующий код построчно читает ввод программы, храня все строки в массиве:

```
t = {}
for line in io.lines() do
  table.insert(t, line)
end
print(#t)    --> (количество считанных строк)
```

В Lua 5.0 данная идиома была широко распространена. В более поздних версиях я предпочитаю идиому `t[#t + 1] = line`, чтобы добавить элементы к списку.

Функция `table.remove` удаляет (и возвращает) элемент с заданной позиции массива, при этом сдвигая остальные элементы,

чтобы занять его место. При вызове без указания позиции она удаляет последний элемент массива.

При помощи этих двух функций довольно легко реализовать стеки, очереди и двойные очереди. Мы можем инициализировать подобные структуры как `t={}`. Операция заталкивания эквивалентна `table.insert(t,x)`; операция выталкивания эквивалентна `table.remove(t)`. Вызов `table.insert(t,1,x)` вставляет элемент в другой конец этой структуры (по сути, в ее начало), а вызов `table.remove(t,1)` удаляет элемент с этого конца. Две последние операции не особенно эффективны, поскольку они должны сдвигать и отодвигать элементы. Однако, учитывая, что библиотека `table` реализована на C, эти циклы не слишком затратны и потому данная реализация довольно хороша для небольших массивов (скажем, до нескольких сот элементов).

20.2. Сортировка

Еще одной полезной функцией для массивов является `table.sort`; мы уже видели ее прежде. Она принимает массив и необязательную функцию упорядочения. Эта функция принимает два аргумента и должна вернуть true, когда ее первый аргумент должен идти перед вторым в отсортированном массиве. Если эта функция не предоставлена, `sort` по умолчанию использует сравнение «меньше, чем» (соответствующее операции '<').

Обычно путаница возникает, когда программист пытается упорядочить индексы в таблице. В таблице индексы образуют множество, в котором нет никакого порядка. Если вы хотите их упорядочить, то вам надо скопировать их в массив и затем этот массив отсортировать. Давайте рассмотрим пример. Допустим, вы прочли исходный файл и построили таблицу, которая для каждого имени функции хранит номер строки, в которой эта функция была определена; получится нечто вроде:

```
lines = {
  luaH_set = 10,
  luaH_get = 24,
  luaH_present = 48,
}
```

А теперь вам нужно напечатать имена этих функций в алфавитном порядке. Если вы обойдете эту таблицу при помощи `pairs`, то имена окажутся в произвольном порядке. Вы не можете отсортировать их непосредственно, поскольку эти имена являются ключами таблицы. Однако, если поместить их в массив, то вы сможете их отсортировать. Сначала вы должны создать массив с этими именами, затем отсортировать его и в итоге напечатать результат:

```
a = {}
for n in pairs(lines) do a[#a + 1] = n end
table.sort(a)
for _, n in ipairs(a) do print(n) end
```

Некоторых это смущает. В конце концов, в массивах Lua тоже нет никакого порядка (они же все-таки таблицы). Но раз мы умеем считать, мы навяжем порядок при обращении к массиву с помощью упорядоченных индексов. Именно поэтому вы всегда должны обходить массивы при помощи `ipairs`, а не `pairs`. Первая функция задает порядок ключей 1, 2, …, в то время как вторая работает с естественным произвольным порядком таблицы.

В качестве более продвинутого решения мы можем написать итератор, который перебирает таблицу, следуя порядку ее ключей. Необязательный параметр `f` позволяет задать другой порядок. Этот итератор сначала сортирует ключи в отдельный массив, а затем уже обходит этот массив. На каждом шаге он возвращает ключ и его значение из исходного таблицы:

```
function pairsByKeys (t, f)
  local a = {}
  for n in pairs(t) do a[#a + 1] = n end
```

```
  table.sort(a, f)
  local i = 0             -- итерационная переменная
  return function ()      -- итерирующая функция
    i = i + 1
    return a[i], t[a[i]]
  end
end
```

При помощи этой функции можно легко напечатать имена тех функций в алфавитном порядке:

```
for name, line in pairsByKeys(lines) do
  print(name, line)
end
```

20.3. Конкатенация

Мы уже видели `table.concat` в разделе 11.6. Она берет список строк и возвращает результат конкатенации всех этих строк. Необязательный второй аргумент задает разделитель строк, который вставляется между строками из списка. Данная функция также принимает два других необязательных аргумента, которые задают индексы первой и последней конкатенируемых строк.

Следующая функция является интересным обобщением `table.concat`. Она способна принимать вложенные списки строк:

```
function rconcat (l)
  if type(l) ~= "table" then return l end
  local res = {}
  for i = 1, #l do
    res[i] = rconcat(l[i])
  end
  return table.concat(res)
end
```

Для каждого элемента списка функция `rconcat` рекурсивно вызывает себя для конкатенации возможного вложенного списка. Затем она вызывает исходную `table.concat` для объединения всех

промежуточных результатов.

```
print(rconcat{{"a", {" nice"}}, " and", {{" long"}, {"
list"}}})
  --> аккуратный и длинный список
```

Упражнения

Упражнение 20.1. Перепишите функцию `rconcat` так, чтобы она могла принимать разделитель, как это делает `table.concat`:

```
print(rconcat({{{"a", "b"}, {"c"}}, "d", {}, {"e"}},
";")
  --> a;b;c;d;e
```

Упражнение 20.2. Проблема с `table.sort` в том, что эта сортировка не устойчива, то есть элементы, которые функция упорядочения считает равными, могут и не сохранить свой первоначальный порядок в массиве после сортировки. Как можно реализовать устойчивую сортировку в Lua?

Упражнение 20.3. Напишите функцию для проверки того, является ли заданная таблица допустимой последовательностью.

ГЛАВА 21

Строковая библиотека

Возможности самого интерпретатора Lua для работы со строками довольно ограничены. Программа может создавать строковые литералы, соединять их и получать длину строки. Но она не может извлекать подстроки или исследовать их содержимое. Полноценные возможности обработки строк в Lua заключены в его строковой библиотеке.

Строковая библиотека экспортирует свои функции в виде модуля с именем `string`. Начиная с Lua 5.1, она также экспортирует свои функции как методы, принадлежащие типу string (при помощи метатаблицы данного типа). Поэтому, например, перевод строки в заглавные буквы можно записать либо как `string.upper(s)`, либо как `s:upper()`. Выбирайте сами.

21.1. Основные строковые функции

Некоторые функции в строковой библиотеке крайне просты: вызов `string.len(s)` возвращает длину строки `s`. Это эквивалентно `#s`. Вызов `string.rep(s,n)` (или `s:rep(n)`) возвращает строку `s`, повторенную `n` раз. Вы можете создать строку в 1 Мб (например, для тестов) при помощи `string.rep("a",2^20)`. Вызов `string.lower(s)` возвращает копию `s`, у которой заглавные буквы преобразованы в строчные; все прочие символы остаются прежними. (Функция `string.upper` преобразует строчные буквы в заглавные.) Обычно, когда требуется отсортировать строки независимо от регистра их букв, можно написать что-то вроде этого:

```
table.sort(a, function (a, b)
```

```
    return a:lower() < b:lower()
  end)
```

Вызов `string.sub(s,i,j)` извлекает часть строки `s` от символа с индексом `i` по символ с индексом `j` включительно. В Lua индекс первого символа строки равен 1. При этом вы можете применять отрицательные индексы, которые отсчитываются с конца строки: индекс -1 ссылается на последний символ строки, -2 на предпоследний символ и т. д. Таким образом, вызов `string.sub(s,1,j)` (или `s:sub(1,j)`) возвращает префикс строки `s` с длиной, равной `j`; вызов `string.sub(s,j,-1)` (или просто `s:sub(j)`, поскольку значением по умолчанию для последнего аргумента является -1) возвращает окончание строки, начиная с символа с индексом `j`; а вызов `string.sub(s,2,-2)` возвращает копию строки `s`, в которой удалены первый и последний символы:

```
s = "[in brackets]"
print(s:sub(2, -2))    --> in brackets
```

Помните, что строки в Lua неизменяемы. Функция `string.sub`, как и любая другая функция в Lua, не изменяет значение строки, а возвращает новую строку. Написать нечто вроде `s:sub(2,-2)` и ждать, что это изменит значение строки `s`, — распространенная ошибка. Если вы хотите изменить значение переменной, то вы должны присвоить ей новое значение:

```
s = s:sub(2, -2)
```

Функции `string.char` и `string.byte` выполняют преобразование между символами и их внутренними числовыми представлениями. Функция `string.char` принимает ноль или более целых чисел, преобразует каждое из них в символ и возвращает строку, в которой все эти символы соединены. Вызов `string.byte(s,i)` возвращает внутреннее числовое представление символа с индексом `i` в строке `s`; второй аргумент необязателен, поэтому вызов `string.byte(s)` возвращает внутреннее числовое

представление первого (или единственного) символа строки `s`. В следующих примерах мы считаем, что символы представлены кодировкой ASCII:

```
print(string.char(97))                    --> a
i = 99; print(string.char(i, i+1, i+2))   --> cde
print(string.byte("abc"))                 --> 97
print(string.byte("abc", 2))              --> 98
print(string.byte("abc", -1))             --> 99
```

В последней строке кода мы использовали отрицательный индекс для обращения к последнему символу строки.

Начиная с Lua 5.1, функция `string.byte` поддерживает третий необязательный аргумент. Вызов наподобие `string.byte(s,i,j)` ряд значений в форме числовых представлений всех символов между индексами `i` и `j` включительно:

```
print(string.byte("abc", 1, 2))
```

Значением по умолчанию для `j` является `i`, поэтому вызов без третьего аргумента возвращает лишь символ по индексу `i`. Есть прекрасная идиома `{s:byte(1,-1)}`, которая создает таблицу с кодами всех символов строки `s`. По этой таблице мы можем воссоздать исходную строку путем вызова `string.char(table.unpack(t))`. Этот прием не работает для очень длинных строк (более 1 Мб), поскольку в Lua есть ограничение на число возвращаемых функцией значений.

Функция `string.format` — это мощный инструмент для форматирования строк, обычно для вывода. Она возвращает отформатированную версию всех своих аргументов (так как является варидической), следуя описанию, заданному своим первым аргументом, так называемой *форматирующей строкой* (format string). Форматирующая строка следует правилам, похожим на те, что используются в функции `printf` стандартного C: она состоит из обычного текста и указаний, которые руководят тем, где и как будет помещен каждый аргумент в отформатированной строке. *Указание*

(directive) состоит из символа '%' и буквы, которая поясняет, как отформатировать аргумент: 'd' для десятичных чисел, 'x' для шестнадцатеричных чисел, 'o' для восьмеричных, 'f' для чисел с плавающей точкой, 's' для строк; есть и другие варианты. Между '%' и буквой могут быть включены другие опции, управляющие деталями форматирования, например, количеством десятичных цифр для числа с плавающей точкой:

```
print(string.format("pi = %.4f", math.pi))          -->
pi = 3.1416
d = 5; m = 11; y = 1990
print(string.format("%02d/%02d/%04d", d, m, y))     -->
05/11/1990
tag, title = "h1", "a title"
print(string.format("<%s>%s</%s>", tag, title, tag))
  --> <h1>a title</h1>
```

В первом примере `%.4f` означает число с плавающей точкой с четырьмя цифрами после десятичной точки. Во втором примере `%02d` обозначает десятичное число минимум из двух цифр с дополнением нулями при необходимости; указание `%2d` без нуля дополняло бы число пробелами. За полным описанием этих указаний обратитесь к справочнику по Lua или, еще лучше, обратитесь к справочнику по C, так как Lua вызывает функции из стандартной библиотеки C для выполнения данной тяжелой работы.

21.2. Функции сопоставления с образцом

Наиболее эффективными функциями в строковой библиотеке являются функци `find` (поиск), `match` (сопоставление), `gsub` (глобальная замена) и `gmatch` (глобальное сопоставление). Все они основаны на *образцах* (pattern).

В отличие от ряда других скриптовых языков, Lua не использует для сопоставления с образцом (pattern matching) регулярные выражения из POSIX или Perl. Главной причиной этого

решения является размер: типичная реализация регулярных выражений POSIX занимает более 4000 строк кода. Это больше размера всех вместе взятых стандартных библиотек Lua. Для сравнения реализация сопоставления с образцом в Lua занимает менее 600 строк. Конечно, сопоставление с образцом в Lua уступает полноценной реализации POSIX. Тем не менее, сопоставление с образцом в Lua является мощным инструментом и включает в себя некоторые возможности, для которых нелегко подобрать аналог в стандартных реализациях POSIX.

Функция `string.find`

Функция `string.find` ищет образец внутри заданной обрабатываемой строки. Простейшей формой образца является слово, которое соответствует лишь копии самого себя. Например, образец '`hello`' задаст поиск подстроки "`hello`" внутри обрабатываемой строки. При нахождении образца `find` возвращает два значения: индекс, с которого начинается совпадение, и индекс, на котором совпадение заканчивается. Если совпадение не найдено, возвращается nil:

```
s = "hello world"
i, j = string.find(s, "hello")
print(i, j)                           --> 1 5
print(string.sub(s, i, j))            --> hello
print(string.find(s, "world"))        --> 7 11
i, j = string.find(s, "l")
print(i, j)                           --> 3 3
print(string.find(s, "lll"))          --> nil
```

Когда совпадение найдено, мы можем вызвать `string.sub` со значениями, возвращенными `string.find`, чтобы получить часть обрабатываемой строки, удовлетворяющей образцу. Для простых образцов этой частью будет сам образец.

У функции `string.find` есть необязательный третий параметр:

индекс, указывающий, с какого места обрабатываемой строки следует начать поиск. Этот параметр удобен, когда мы хотим обработать все индексы, где был обнаружен заданный образец: мы систематически ищем новый образец, каждый раз начиная с той позиции, где мы нашли предыдущий. В качестве примера следующий код строит таблицу с позициями всех переводов строки внутри заданной строки:

```
local t = {}                          -- таблица для
хранения индексов
local i = 0
while true do
i = string.find(s, "\n", i+1)     -- ищет следующий
перевод строки
if i == nil then break end
t[#t + 1] = i
end
```

Позже мы увидим более простой способ записи подобных циклов — с применением итератора `string.gmatch`.

Функция `string.match`

Функция `string.match` похожа на `string.find` в том смысле, что она тоже ищет образец в строке. Однако, вместо возвращения тех позиций, где был найден образец, она возвращает часть обрабатываемой строки, удовлетворяющую образцу:

```
print(string.match("hello world", "hello"))    -->
hello
```

Для фиксированных образцов вроде '`hello`' эта функция не имеет смысла. Она показывает свою эффективность, когда используется с переменными образцами, как в следующем примере:

```
date = "Today is 17/7/1990"
d = string.match(date, "%d+/%d+/%d+")
print(d)   --> 17/7/1990
```

Вскоре мы обсудим значение образца `'%d+/%d+/%d+'` и более продвинутое применение `string.match`.

Функция `string.gsub`

У функции `string.gsub` три обязательных параметра: обрабатываемая строка, образец и замещающая строка. Ее основное применение состоит в замене всех вхождений образца на замещающую строку внутри обрабатываемой:

```
s = string.gsub("Lua is cute", "cute", "great")
print(s)            --> Lua is great
s = string.gsub("all lii", "l", "x")
print(s)            --> axx xii
s = string.gsub("Lua is great", "Sol", "Sun")
print(s)            --> Lua is great
```

Необязательный четвертый параметр ограничивает число выполняемых замен:

```
s = string.gsub("all lii", "l", "x", 1)
print(s)            --> axl lii
s = string.gsub("all lii", "l", "x", 2)
print(s)            --> axx lii
```

Функция `string.gsub` также возвращает в качестве второго значения число выполненных замен. Например, простой способ посчитать число пробелов в строке:

```
count = select(2, string.gsub(str, " ", " "))
```

Функция `string.gmatch`

Функция `string.gmatch` возвращает функцию, которая перебирает все вхождения образца в строку. Например, следующий пример собирает все слова в заданной строке `s`:

```
words = {}
for w in string.gmatch(s, "%a+") do
  words[#words + 1] = w
end
```

Как мы вскоре обсудим, образец '%a+' соответствует последовательностям из одной или более букв (то есть словам). Поэтому цикл **for** обойдет все слова внутри обрабатываемой строки, сохраняя их в список words.

Следующий пример реализует функцию, аналогичную package.searchpath, при помощи gmatch и gsub:

```
function search (modname, path)
  modname = string.gsub(modname, "%.", "/")
  for c in string.gmatch(path, "[^;]+") do
    local fname = string.gsub(c, "?", modname)
    local f = io.open(fname)
    if f then
      f:close()
      return fname
    end
  end
  return nil    -- не найден
end
```

Первым шагом будет замена всех точек на разделитель путей директорий, которым в данном примере является '/' (Как мы в дальнейшем увидим, у точки в образцах особое назначение. Для сопоставления с символом точки мы должны написать '%.'.) Далее функция перебирает все составляющие пути, где каждая составляющая — это непрерывный ряд любых символов, отличных от точки с запятой. Для каждой составляющей функция меняет все вопросительные знаки на имя модуля и проверяет, существует ли такой файл. Если да, то функция закрывает этот файл и возвращает его имя.

21.3. Образцы

Вы можете сделать образцы более полезными при помощи классов символов. *Класс символов* (character class) — это элемент в образце, который может соответствовать любому символу из заданного множества. Например, класс `%d` соответствует любой цифре. Таким образом, вы можете искать дату в формате `dd/mm/yyyy` при помощи образца '`%d%d/%d%d/%d%d%d%d`':

```
s = "Deadline is 30/05/1999, firm"
date = "%d%d/%d%d/%d%d%d%d"
print(string.sub(s, string.find(s, date)))    -->
30/05/1999
```

Следующая таблица содержит список всех классов символов:

`.`	все символы
`%a`	буквы
`%c`	управляющие символы
`%d`	цифры
`%g`	печатные символы кроме пробельных
`%l`	строчные буквы
`%p`	символы пунктуации
`%s`	пробельные символы
`%u`	заглавные буквы
`%w`	буквенно-цифровые символы
`%x`	шестнадцатеричные цифры

Любая из этих букв в верхнем регистре представляет собой дополнение класса, т.е. множество не входящих в него символов. Например, '`%A`' соответствует всем небуквенным символам:

```
print(string.gsub("hello, up-down!", "%A", "."))
  --> hello..up.down. 4
```

(Число 4 не являются частью итоговой строки. Это второе значение, возвращаемое `gsub`, — полное число выполненных замен. В дальнейших примерах с выводом результата `gsub` я буду опускать это число.)

У некоторых символов, называемых *магическими символами* (magic character), есть особое значение при использовании в образце. Магическими символами являются

```
( ) . % + - * ? [ ] ^ $
```

Символ '%' работает для этих магических символов как экран. Таким образом, '%.' соответствует точке, а '%%' соответствует самому символу '%'. Вы можете применять экранирующий '%' не только с магическими символами, но и с любыми символами, отличными от буквенно-цифровых. Когда сомневаетесь, не рискуйте и используйте экран.

Для парсера Lua образцы являются обычными строками. К ним нет особого отношения — они подчиняются тем же правилам, что и остальные строки. Только функции сопоставления с образцом рассматривают их как образцы, и только эти функции считают символ '%' экраном. Для помещения кавычек внутрь образца используются те же самые приемы, что и для других строк; например, вы можете экранировать кавычки при помощи '\', который является экранирующим символом в Lua.

Множество символов (char-set) позволяет вам создавать ваши собственные классы символов, группируя различные классы и одиночные символы внутри квадратных скобок. Например, множество '[%w_]' соответствует как буквенно-цифровым символам, так и символу подчеркивания; множество '[01]' соответствует двоичным цифрам; а множество '[%[%]]' соответствует квадратным скобкам. Для подсчета гласных в тексте вы можете написать

```
nvow = select(2, string.gsub(text, "[AEIOUaeiou]",
""))
```

При этом вы также можете включать во множества символов диапазоны, записывая первый и последний символы с дефисом между ними. Я редко пользуюсь данным средством, поскольку все наиболее употребительные диапазоны уже предопределены;

например, '[0-9]' — это то же самое, что и '%d', а '[0-9a-fA-F]' — это то же самое, что и '%x' Однако, если вам потребуется найти восьмеричную цифру, то вы можете предпочесть '[0-7]' вместо явного перечисления '[01234567]'. Вы также можете получить дополнение любого множества символов, поставив перед ним '^': так, образец '[^0-7]' находит любой символ, который не является восьмеричной цифрой, а '[^\n]' соответствует любому символу, отличному от перевода строки. Но не забывайте, что вы можете получить дополнение для простых классов посредством их версии в верхнем регистре: '%S' проще, чем '[^%s]'.

Вы можете сделать образцы еще удобнее с помощью модификаторов для повторений и необязательных частей. Образцы в Lua предлагают четыре таких модификатора:

+	1 или более повторений
*	0 или более повторений
-	0 или более коротких повторений
?	0 или 1 вхождение (необязательный образец)

Модификатор '+' соответствует одному или более символам первоначального класса. Он всегда возвращает самую длинную последовательность символов, которая соответствует образцу. Например, образец '%a+' означает одну или несколько букв, то есть слово:

```
print(string.gsub("one, and two; and three", "%a+",
"word"))
  --> word, word word; word word
```

Образец '%d+' соответствует одной или нескольким цифрам (целочисленному нумералу):

```
print(string.match("the number 1298 is even", "%d+"))
--> 1298
```

Модификатор '*' похож на '+' , но при этом он допускает нулевое число вхождений символов заданного класса. Обычно используется

для обозначения необязательных пробелов между частями образца. Например, для сопоставления с парой пустых круглых скобок, например `()` или `( )`, можно использовать образец `'%(%s*%)'`: образец `'%s*'` соответствует нулю и более пробелов. (У круглых скобок также есть особое значение в образцах, поэтому мы должны их экранировать.) В качестве другого примера образец `'[_%a][_%w]*'` соответствует идентификаторам внутри программы на Lua: начинается с пробела или символа подчеркивания, за которым идет ноль или большее количество подчеркиваний и буквенно-цифровых символов.

Подобно `'*'`, модификатор `'-'` также соответствует нулю или большему количеству символов заданного класса. Однако, вместо соответствия самой длинной последовательности он соответствует самой короткой. Иногда между `'*'` и `'-'` нет никакой разницы, но обычно они дают совершенно разные результаты. Например, если вы попытаетесь найти идентификатор при помощи образца `'[_%a][_%a]-'`, то вы получите лишь первую букву, поскольку `'[_%w]-'` всегда соответствует пустой последовательности. С другой стороны, предположим, что вы хотите найти комментарии в программе C. Многие сперва пробуют `'/%*.*%*/'` (то есть `'/*'`, за которыми следует последовательность любых символов, за которой следует `"*/"`, и все это с соответственными экранами). Однако, поскольку `'.*'`. длится столько, сколько сможет, первые комбинация `"/*"` в вашей программе закроется лишь самой последней `"*/"`:

```
test = "int x; /* x */ int y; /* y */"
print(string.match(test, "/%*.*%*/"))
  --> /* x */ int y; /* y */
```

Образец `'.-'` наоборот захватит наименьшее количество символов, необходимое для нахождения первого сочетания `'*/'`, и даст, таким образом, желаемый результат:

```
test = "int x; /* x */ int y; /* y */"
print(string.gsub(test, "/%*.-%*/", ""))
```

```
--> int x; int y;
```

Последний модификатор '?' соответствует необязательному символу. Например, предположим, что мы хотим найти целое число в тексте, которое при этом может содержать необязательный знак. Образец '[+-]?%d+' справляется с работой, находя такие последовательности, как "-12", "23" и "+1009". Класс '[+-]' соответствует либо знаку '+', либо знаку '-'; следующий за ним '?' делает этот знак необязательным.

В отличие от других систем, в Lua модификатор может быть применен только к классу символов; группировать образцы под одним модификатором нельзя. Например, в Lua нет образца, соответствующего необязательному слову (если только вы не ищете однобуквенные слова). Обычно это ограничение можно обойти при помощи продвинутых приемов, которые мы рассмотрим в конце данной главы.

Если образец начинается с символа '^', то он будет сопоставляться только с началом обрабатываемой строки. И точно так же, если образец заканчивается символом '$', то он будет сопоставляться только с концом обрабатываемой строки. Вы можете использовать эти две метки для создания образцов. Например, следующий тест проверяет, начинается ли строка с цифры:

```
if string.find(s, "^%d") then ...
```

А этот тест проверяет, что строка является целым числом, без других символов в начале или конце:

```
if string.find(s, "^[+-]?%d+$") then ...
```

Символы '^' и '$' являются магическими лишь тогда, когда применяются в начале или конце образца. Иначе они служат обычными символами, которые соответствуют самим себе.

Еще одним элементом в образце является '%b', который соответствует сбалансированным строкам (у которых парные

символы по краям). Мы записываем его как '%b*xy*', где *x* и *y* — это два разных символа; символ *x* выступает как открывающий символ, а *y* — как закрывающий. Например, образец '%b()' соответствует частям строки, которые начинаются с '(' и заканчиваются на соответствующей ')':

```
s = "a (enclosed (in) parentheses) line"
print(string.gsub(s, "%b()", ""))    --> a line
```

Обычно мы используем этот образец в виде '%b()', '%b[]', '%b{}' или '%b<>', но вы можете использовать в качестве разделителей отличные друг от друга символы.

Наконец, элемент '%f[*множество_символов*]' является *пограничным образцом* (frontier pattern). Он соответствует пустой строке, только если следующий символ входит во *множество_символов*, а предыдущий — нет:

```
s = "the anthem is the theme"
print(s:gsub("%f[%w]the%f[%W]", "one"))
  --> one anthem is one theme
```

Образец '%f[%w]' соответствует границе между небуквенно-цифровым и буквенно-цифровым символами, а образец '%f[%w]' соответствует границе между буквенно-цифровым символом и небуквенно-цифровым символом. Поэтому данный образец сопоставляет строке "the" только целое слово. Обратите внимание, что мы должны писать множество символов внутри квадратных скобок даже тогда, когда оно является одним единственным классом.

Позиции перед первым и после последнего символов в обрабатываемой строке трактуются как содержащие нуль-символ (символ с ASCII кодом 0). В предыдущем примере первое "the" начинается с границы между нуль-символом (не во множестве '[%w]') и 't' (во множестве '[%w]').

Пограничный образец был реализован еще в Lua 5.1, но не был документирован. Официальным он стал только в Lua 5.2.

21.4. Захваты

Механизм *захвата* (capture) позволяет образцу выдергивать из обрабатываемой строки те части, которые удовлетворяют частям образца, в целях дальнейшего использования. Вы можете указать захват посредством записи захватываемых частей образца внутри круглых скобок.

Когда в образце есть захваты, то функция `string.match` возвращает каждое захваченное значение как отдельный результат; другими словами, она разбивает строку на ее захваченные части.

```
pair = "name = Anna"
key, value = string.match(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)    --> name Anna
```

Образец '`%a+`' задает непустую последовательность букв; образец '`%s*`' задает возможно пустую последовательность пробелов. Поэтому в примере выше весь образец задает последовательности и знак равенства в следующем порядке: буквы, пробелы, знак равенства, пробелы, буквы. У обеих последовательностей букв их образцы заключены в круглые скобки, поэтому при соответствии они будут захвачены. Ниже похожий пример:

```
date = "Today is 17/7/1990"
d, m, y = string.match(date, "(%d+)/(%d+)/(%d+)")
print(d, m, y) --> 17 7 1990
```

Внутри образца элемент вида '*%d*', где *d* — это одиночная цифра, соответствует лишь копии *d*-ого захвата. Чтобы продемонстрировать типичное применение, рассмотрим случай, когда вы хотите внутри строки найти подстроку, заключенную в одинарные или двойные кавычки. Вы можете попробовать образец наподобие '`["'].-["']`', то есть кавычка, за которой следует что угодно, за которым следует другая кавычка; но при этом у вас будут проблемы со строками вроде "`it's all right`". Для

решения данной проблемы можно захватить первую кавычку и использовать ее для задания второй кавычки:

```
s = [[then he said: "it's all right"!]]
q, quotedPart = string.match(s, "([\"'])(.-)%1")
print(quotedPart)    --> it's all right
print(q)             --> "
```

Первый захват — это сам символ кавычки, а второй — это подстрока между кавычками (подстрока, удовлетворяющая '`.-`')

Похожим примером является образец, соответствующий длинным строкам в Lua:

```
%[(=*)%[(.-)%]%1%]
```

Он соответствует символам в следующем порядке: открывающая квадратная скобка, ноль или большее число знаков равенства, еще одна открывающая квадратная скобка, что угодно (содержимое строки), закрывающая квадратная скобка, то же самое количество знаков равенства, еще одна закрывающая квадратная скобка. Пример:

```
p = "%[(=*)%[(.-)%]%1%]"
s = "a = [=[[[ something ]] ]==] ]=]; print(a)"
print(string.match(s, p))    --> = [[ something ]]
]==]
```

Первый захват — это последовательность знаков равенства (в данном примере лишь один знак); второй захват — это содержимое строки.

Третья область применения захваченных значений — замещающая строка для `gsub`. Как и образец, замещающая строка может содержать элементы наподобие '`%`*d*', которые заменяются на соответствующие захваты при выполнении подстановки. В частности, элемент '`%0`' меняется на полное соответствие. (Кстати, '`%`' в замещающей строке должен быть экранирован как '`%%`'.) В качестве примера, следующая команда дублирует каждую букву в

строке, добавляя дефис между копиями:

```
print(string.gsub("hello Lua!", "%a", "%0-%0"))
  --> h-he-el-ll-lo-o L-Lu-ua-a!
```

Этот пример меняет местами соседние символы:

```
print(string.gsub("hello Lua", "(.)(.)", "%2%1"))    -
-> ehll ouLa
```

В качестве более полезного примера давайте напишем простой преобразователь формата, который получает строку с командами в стиле LaTeX и переводит их в формат XML:

```
\command{some text}    --> <command>some
text</command>
```

Если нам не нужны вложенные команды, то поможет следующий вызов `string.gsub`:

```
s = [[the \quote{task} is to \em{change} that.]]
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
print(s)
  --> the <quote>task</quote> is to <em>change</em>
that.
```

(В следующем разделе мы увидим, как обрабатывать вложенные команды.)

Еще один полезный пример удаляет пробелы по краям строки:

```
function trim (s)
  return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end
```

Обратите внимание на разумный выбор формата образцов. Два якоря ('`^`' и '`$`') обеспечивают получение всей строки. Поскольку '`.-`' старается выбрать наименьшее число символов, два образца '`%s*`' соответствуют всем пробелам по краям. Также обратите внимание, что поскольку `gsub` возвращает два значения, то мы используем круглые скобки вокруг его вызова для отбрасывания лишнего

результата (числа замен).

21.5. Замены

Вместо строки в качестве третьего аргумента `string.gsub` мы можем использовать функцию или таблицу. При вызове с функцией `string.gsub` вызывает эту функцию каждый раз, когда находит соответствие; аргументами каждого вызова являются захваты, а возвращенное функцией значение используется в качестве замещающей строки. При вызове с таблицей `string.gsub` обращается к ней, используя первый захват как ключ, а связанное с ним значение как замещающую строку. Если результатом вызова или обращения к таблице является nil, то `gsub` не производит замену совпавшей части.

В качестве первого примера следующая функция расширяет переменные путем замены внутри строки значения переменной `$varname` на значение глобальной переменной `varname`:

```
function expand (s)
  return (string.gsub(s, "$(%w+)", _G))
end

name = "Lua"; status = "great"
print(expand("$name is $status, isn't it?"))
  --> Lua is great, isn't it?
```

При каждом совпадении с '`$(%w+)`' (знак доллара, за которым следует имя переменной) функция `gsub` ищет захваченное имя в глобальной таблице `_G`; результат заменяет совпавшую часть строки. Когда в таблице нет соответствующего ключа, замена не производится:

```
print(expand("$othername is $status, isn't it?"))
  --> $othername is great, isn't it?
```

Если вы не уверены в том, есть ли у данных переменных

строковые значения, то вы можете попробовать применить к их значениям `tostring`. В таком случае в качестве замещающего значения вы можете использовать функцию:

```
function expand (s)
  return (string.gsub(s, "$(%w+)", function (n)
                                     return
tostring(_G[n])
                                   end))
end

print(expand("print = $print; a = $a"))
  --> print = function: 0x8050ce0; a = nil
```

Теперь при каждом совпадении с образцом '`$(%w+)`' функция `gsub` вызывает заданную функцию с захваченным именем в качестве аргумента; возвращенное значение используется для замены.

В последнем примере мы возвращаемся к нашему преобразователю формата из предыдущего раздела. И вновь мы хотим преобразовывать команды в стиле LaTeX (`\example{text}`) в команды в стиле XML (`text`), но на этот раз мы разрешим вложенные команды. Следующая функция использует для этого рекурсию:

```
function toxml (s)
  s = string.gsub(s, "\\(%a+)(%b{})", function (tag,
body)
        body = string.sub(body, 2, -2)    -- убирает
квадратные скобки
        body = toxml(body)                --
обрабатывает вложенные команды
        return string.format("<%s>%s</%s>", tag, body,
tag)
      end)
  return s
end

print(toxml("\\title{The \\bold{big} example}"))
  --> <title>The <bold>big</bold> example</title>
```

Кодировка URL

Для нашего следующего примера мы воспользуемся *кодировкой URL* (URL encoding), которая применяется HTTP для передачи параметров в URL. Кодировка преобразует специальные символы (наподобие '=', '&' и '+') в '%*xx*', где *xx* — это шестнадцатеричный код этих символов. После этого она меняет пробелы на '+'. Например, данная кодировка зашифрует "a+b = c" как "%2Bb+%3D+c". И наконец, она записывает имя каждого параметра и его значение со знаком равенства между ними и соединяет получившиеся пары name=value с помощью амперсанда ('&') между ними. Например, значения

```
name = "al"; query = "a+b = c"; q="yes or no"
```

будут закодированы как "name=al&query=a%2Bb+%3D+c&q=yes+or+no".

Теперь допустим, что нам требуется декодировать этот URL и сохранить каждое значение в таблицу, проиндексировав его именем соответственного параметра. Следующая функция выполняет основное декодирование:

```
function unescape (s)
  s = string.gsub(s, "+", " ")
  s = string.gsub(s, "%%(%x%x)", function (h)
      return string.char(tonumber(h, 16))
    end)
  return s
end
```

Первый оператор заменяет каждый '+' в строке на пробел. Второй gsub сопоставляет все шестнадцатеричные нумералы из двух цифр, перед которыми стоит '%' и для каждого совпадения вызывает анонимную функцию. Эта функция преобразует шестнадцатеричный нумерал в число (tonumber по основанию 16) и возвращает соответствующий символ (string.char). Например:

```
print(unescape("a%2Bb+%3D+c"))    --> a+b = c
```

Для декодирования пар `name=value` мы воспользуемся `gmatch`. Поскольку и имя, и значение не могут содержать '`&`' или '`=`', то мы можем сопоставить их при помощи образца '`[^&=]+`':

```
cgi = {}
function decode (s)
  for name, value in string.gmatch(s, "([^&=]+)=
([^&=]+)") do
    name = unescape(name)
    value = unescape(value)
    cgi[name] = value
  end
end
```

Вызов `gmatch` сопоставляет все пары вида `name=value`. Для каждой пары итератор возвращает соответствующие захваты (отмеченные круглыми скобками в образце) как значения для `name` и `value`. Тело цикла просто вызывает `unsecape` для обеих строк и записывает эту пару в таблицу `cgi`.

Также легко записать и соответствующее кодирование. Для начала мы напишем функцию `escape`; эта функция кодирует все специальные символы как '`%`', за которым следует шестнадцатеричный код символа (функция `format` с опцией "`%02X`" создает шестнадцатеричное число из двух цифр, используя 0 как заполнитель), а затем меняет пробелы на '`+`':

```
function escape (s)
  s = string.gsub(s, "[&=+%%%c]", function (c)
      return string.format("%%%02X", string.byte(c))
    end)
  s = string.gsub(s, " ", "+")
  return s
end
```

Функция `encode` обходит всю таблицу, которую нужно закодировать, строя при этом итоговую строку:

```
function encode (t)
  local b = {}
  for k,v in pairs(t) do
    b[#b + 1] = (escape(k) .. "=" .. escape(v))
  end
  return table.concat(b, "&")
end

t = {name = "al", query = "a+b = c", q = "yes or no"}
print(encode(t))    -->
q=yes+or+no&query=a%2Bb+%3D+c&name=al
```

Разложение символов табуляции на пробелы

У пустого захвата наподобие '`()`' в Lua есть особое значение. Вместо того, чтобы не захватывать ничего (довольно бесполезное занятие), этот образец захватывает свою позицию в обрабатываемой строке как число:

```
print(string.match("hello", "()ll()"))    --> 3    5
```

(Обратите внимание, что результат этого примера отличается от результата вызова `string.find`, поскольку позиция второго пустого захвата следует *после* совпадения с образцом.)

Прекрасным примером использования позиционных захватов является разложение символов табуляции в строке на пробелы:

```
function expandTabs (s, tab)
  tab = tab or 8 -- tab "size" (default is 8)
  local corr = 0
  s = string.gsub(s, "()\t", function (p)
      local sp = tab - (p - 1 + corr)%tab
      corr = corr - 1 + sp
      return string.rep(" ", sp)
    end)
  return s
end
```

Образец `gsub` находит все символы табуляции внутри строки,

захватывая их позиции. Для каждого символа табуляции внутренняя функция использует эту позицию, чтобы вычислить количество пробелов, которое нужно, чтобы получить столбец, кратный значению `tab`: сначала она вычитает единицу из этой позиции, чтобы выставить начало в ноль, а затем добавляет `corr` для компенсации предыдущих символов табуляции (замена каждого символа табуляции влияет на позицию последующих символов). Затем функция вычисляет поправку для следующего символа табуляции: вычитает единицу для удаляемого таба и прибавляет `sp` для добавляемых пробелов. Наконец, она возвращает строку с соответствующим числом пробелов.

Для полноты давайте рассмотрим, как можно обратить эту операцию, преобразуя пробелы в символы табуляции. Можно было бы тоже начать с использования пустых захватов для обработки позиций, но есть более простое решение: на каждом восьмом символе мы вставим пометку внутрь строки. Затем всякий раз, когда перед этой пометкой будут стоять пробелы, мы заменим эту последовательность из пробелов и метки символом табуляции:

```
function unexpandTabs (s, tab)
  tab = tab or 8
  s = expandTabs(s)
  local pat = string.rep(".", tab)
  s = string.gsub(s, pat, "%0\1")
  s = string.gsub(s, " +\1", "\t")
  s = string.gsub(s, "\1", "")
  return s
end
```

Эта функция начинает с разложения строки для удаления всех предыдущих символов табуляции. Затем она вычисляет вспомогательный образец для сопоставления с ним всех последовательностей длиной `tab` символов, и использует этот образец для добавления метки (управляющего символа `\1`) после каждой последовательности длиной `tab` символов. Далее функция заменяет все последовательности пробелов, за которыми следует

метка, символами табуляции. И наконец, она удаляет все оставшиеся метки (те, что без пробелов перед ними).

21.6. Специфические приемы

Сопоставление с образцом — это мощный инструмент для работы со строками. Вы можете выполнить множество сложных действий всего несколькими вызовами `string.gsub`. Однако, как и любой другой мощный инструмент, ее следует применять аккуратно.

Сопоставление с образцом не заменит полноценный парсер. Для программ на скорую руку, вы можете воспользоваться им для удобной обработки исходного кода, но получить качественный продукт таким образом будет тяжело. В качестве хорошего примера рассмотрим образец, который мы использовали для сопоставления с комментариями в программе C: `'/%*.-%*/'`. Если в вашей программе есть строковый литерал, содержащий `"/*"`, то вы можете получить неверный результат:

```
test = [[char s[] = "a /* here"; /* a tricky string
*/]]
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
  --> char s[] = "a <COMMENT>
```

Строки с подобным содержимым довольно редки, и для ваших личных целей подобный образец, скорее всего, будет работать. Но вы не должны распространять программу с таким недостатком.

Обычно сопоставление с образцом в Lua довольно эффективно: моему старому Pentium нужно менее 0.3 секунды, чтобы провести сопоставление со всеми словами в тексте размером 4.4 Мб (850К слов). Но вы можете предпринять меры предосторожности. Вы всегда должны делать образец как можно более точным, так как неточные образцы медленнее точных. Крайне простой пример — образец `'(.-)%$'` для получения всего текста до первого знака доллара. Если в обрабатываемой строке есть знак доллара, то все

пройдет гладко; но предположим, что в строке вообще нет ни одного знака доллара. Алгоритм попытается найти соответствие образцу, начав с первой позиции строки. В поисках доллара он пробежит всю строку. Когда строка закончится, алгоритм даст сбой *для первой позиции* строки. Затем алгоритм выполнит полностью новый поиск, начав со второй позиции строки, и обнаружит, что здесь тоже нет соответствия образцу, и т. д. Это выльется в квадратичную зависимость от времени, занимая более 4 минут на моем старом Pentium для строки из 100K символов. Вы можете легко исправить данную проблему, привязав образец к первой позиции строки при помощи '`^(.-)%$`'. С этой привязкой образец будет сопоставлен за сотую долю секунды.

Также обращайте внимание на *пустые* образцы, то есть образцы, которым соответствует пустая строка. Например, если вы попробуете найти имена при помощи образца наподобие '`%a*`', то вы везде будете находить их повсюду:

```
i, j = string.find(";$% **#$hello13", "%a*")
print(i,j)    --> 1    0
```

В этом примере вызов `string.find` правильно находит пустую последовательность букв в начале строки.

Всегда бессмысленно писать образец, который начинается или заканчивается модификатором '`-`', поскольку ему будет удовлетворять лишь пустая строка. Этому модификатору обычно требуется что-то вокруг него, чтобы уменьшить область его соответствия. Образцы, включающие в себя '`.*`', тоже довольно коварны, поскольку эта конструкция может вобрать больше символов, чем вы планировали.

Иногда удобнее использовать сам Lua для построения образцов. Мы уже пользовались данным приемом в нашей функции для преобразования пробелов в символы табуляции. В качестве другого примера давайте рассмотрим, как мы можем найти строки текста длиной, скажем, от 70 символов. Точнее длинной строкой будет

последовательность из 70 или более символов, отличных от перевода строки. Мы можем найти соответствие одиночному символу, не являющемуся переводом строки, при помощи класса символов '[^\n]'. Таким образом, мы можем найти соответствие длинной строке посредством образца, который 70 раз повторяет образец для одиночного символа, и после которого может следовать 0 или более этих символов. Вместо написания данного образца вручную, мы можем создать его при помощи string.rep:

```
pattern = string.rep("[^\n]", 70) .. "[^\n]*"
```

В качестве другого примера предположим, что вы хотите сделать поиск без учета регистра. Как вариант можно заменить каждую букву *x* в образце на класс '[*xX*]', то есть класс, включающий в себя и строчную, и прописную версии буквы. Мы можем автоматизировать это преобразование при помощи функции:

```
function nocase (s)
  s = string.gsub(s, "%a", function (c)
      return "[" .. string.lower(c) .. string.upper(c)
.. "]"
    end)
  return s
end

print(nocase("Hi there!"))    --> [hH][iI] [tT][hH]
[eE][rR][eE]!
```

Иногда вам требуется заменить каждое вхождение s1 на s2, без учета всяких магических символов. Если обе строки являются литералами, то при их написании вы можете сами добавить все необходимые экраны для магических символов. Но если эти строки являются значениями переменных, то вы можете использовать еще один gsub для вставки экранов за вас:

```
s1 = string.gsub(s1, "(%W)", "%%%1")
s2 = string.gsub(s2, "%%", "%%%%")
```

В строке поиска мы экранируем все небуквенно-цифровые символы (то есть заглавную "W"). В замещающей строке мы экранируем лишь '%'.

Еще одним полезным приемом при сопоставлении с образцом является обработка заданной строки перед основной работой. Предположим, мы хотим перевести в прописные буквы из всех строк внутри кавычек в каком-либо тексте, где строка в кавычках начинается и заканчивается двойными кавычками ('"'), но при этом может содержать экранированные кавычки '\"':

```
follows a typical string: "This is \"great\"!".
```

Одним из решений в подобных случаях является предварительная обработка текста таким образом, чтобы закодировать проблематичную последоватльность во что-то еще. Например, мы могли бы закодировать "\"" как "\1". Тем не менее, если в исходном тексте уже содержался символ "\1", то мы в беде. Простым способом выполнить кодирование и избежать данной проблемы является преобразование всех последовательностей " *x*" в " *ddd*", где *ddd* — это десятичное представление символа *x*:

```
function code (s)
  return (string.gsub(s, "\\(.)", function (x)
      return string.format("\\%03d", string.byte(x))
    end))
end
```

Теперь любая последовательность "*ddd*" всегда будет результатом кодирования, поскольку любая "*ddd*" в исходной строке тоже кодируется. Поэтому декодирование является простой задачей:

```
function decode (s)
  return (string.gsub(s, "\\(%d%d%d)", function (d)
      return "\\" .. string.char(tonumber(d))
    end))
end
```

Теперь мы можем завершить нашу задачу. Так как закодированная строка не содержит никаких экранированных кавычек ("\""), то мы запросто можем найти строки в кавычках при помощи '".-"':

```
s = [[follows a typical string: "This is
\"great\"!".]]
s = code(s)
s = string.gsub(s, '".-"', string.upper)
s = decode(s)
print(s)    --> follows a typical string: "THIS IS
\"GREAT\"!".
```

Или в более компактной записи:

```
print(decode(string.gsub(code(s), '".-"',
string.upper)))
```

21.7. Юникод

На данный момент строковая библиотека не предлагает явной поддержки Юникода. Однако, выполнять некоторые полезные и простые преобразования строк Юникода в кодировке UTF-8 можно и без дополнительных библиотек.

UTF-8 — это основная кодировка для Юникода в Интернете. Из-за ее совместимости с ASCII эта кодировка идеально подходит для Lua. Этой совместимости достаточно, чтобы обеспечить работу с UTF-8 ряда технологий для обработки строк ASCII без внесения изменений.

UTF-8 представляет каждый символ Юникода различным числом байт. Например, символ 'A' он представляет одним байтом равным 65; буква алеф из иврита, у которой в Юникоде код 1488, представлена двухбайтовой последовательностью 215-144. UTF-8 представляет все символы из ASCII как ASCII, то есть одним байтом со значением, меньшим 128. Все остальные символы представлены последовательностями байт, где первый байт лежит в

диапазоне [194, 244], а последующие байты лежат в диапазоне [128, 191]. Точнее, диапазон первого байта для двухбайтовых последовательностей — [194, 223], для трехбайтовых последовательностей — [224, 239] а для четырехбайтовых последовательностей — [240, 244]. Эта конструкция следит за тем, чтобы кодовая последовательность одного символа никогда не встретилась внутри кодовой последовательности другого. Например, байт меньше 128 никогда не встретится в многобайтовой последовательности; он всегда представлен своим ASCII-символом.

Так как Lua не использует нуль-символы, он может читать, записывать и хранить строки UTF-8 как любые обычные строки. Строковые литералы тоже могут содержать внутри себя данные в UTF-8. (Разумеется, вы можете редактировать исходный код как файл в UTF-8, если захотите.) Операция конкатенации корректно работает для всех строк в UTF-8. Операции сравнения строк (меньше, чем; меньше или равно и т. п.) сравнивают строки в UTF-8, следуя порядку символов в Юникоде.

Библиотека операционной системы и библиотека ввода-вывода по сути являются интерфейсами к системе, на которой они выполняются, поэтому их поддержка строк в UTF-8 зависит от этой системы. В Linux, например, мы можем использовать UTF-8 для имен файлов, но Windows использует UTF-16. Поэтому для работы с именами файлов в Юникоде в Windows понадобятся либо дополнительные библиотеки, либо внесение изменений в стандартные библиотеки Lua.

Теперь давайте посмотрим, как функции из строковой библиотеки работают со строками в UTF-8.

Функции `string.reverse`, `string.byte`, `string.char`, `string.upper` и `string.lower` не работают со строками в UTF-8, поскольку все они полагают, что размер одного символа равен одному байту.

Функции `string.format` и `string.rep` работают со строками в UTF-8 без проблем, за исключением опции форматирования `'%c'`,

которая считает, что один символ — это один байт. Функции `string.len` и `string.sub` корректно работают со строками в UTF-8, но при этом индексы указывают на количество байт (а не символов). Чаще всего это именно то, что вам нужно. Но, как мы скоро увидим, мы также можем посчитать и количество символов.

Для функций сопоставления с образцами их применимость к строкам в UTF-8 зависит от образца. Простые образцы работают без каких-либо проблем в связи с ключевым свойством UTF-8 — код одного символа никогда не может находиться внутри кода другого. Классы и множества символов работают лишь для ASCII-символов. Например, образец '`%s`' работает для строк в UTF-8, но он будет соответствовать только пробелам ASCII и не будет соответствовать дополнительным пробельным символам в Юникоде, таким как неразрывный пробел (U+00A0), разделитель параграфов (U+2029) или монгольский разделитель гласных (U+180E).

Некоторые образцы могут удачно использовать особенности UTF-8. Например, если вы хотите посчитать число символов в строке, то вы можете использовать следующее выражение:

```
#(string.gsub(s, "[\128-\191]", ""))
```

Этот `gsub` убирает продолжающие байты из строки, чтобы остались лишь однобайтовые последовательности и начальные байты многострочных последовательностей: один байт на каждый символ.

Взяв за основу похожие идеи, следующий пример показывает, как можно перебрать все символы в строке в UTF-8:

```
for c in string.gmatch(s, ".[\128-\191]*") do
  print(c)
end
```

Листинг 21.1 иллюстрирует некоторые приемы для работы с UTF-8 строками в Lua. Разумеется, для выполнения этих примеров вам нужна платформа, в которой `print` поддерживает UTF-8.

К сожалению, больше Lua предложить нечего. Полноценная

поддержка Юникода требует огромных таблиц, которые плохо соотносятся с маленьким размером Lua. У Юникода слишком много особенностей. Практически невозможно абстрагировать какое-либо понятие из конкретных языков. Размыто даже понятие того, что является символом, поскольку нет взаимно-однозначного соответствия между закодированными в Юникоде символами и графемами (то есть символами с диакритическими знаками и «полностью игнорируемыми» символами). Другие, казалось бы, фундаментальные понятия, например, того, что является буквой, также разнятся среди языков.

Чего, на мой взгляд, не хватает в Lua, так это функций преобразования последовательностей UTF-8 и комбинаций Юникода между собой и функций для проверки правильности строк в UTF-8. Возможно, они войдут в следующую версию Lua. Для тех, кому нужна поддержка Юникода, похоже лучшим вариантом будет использование внешней библиотеки вроде `slnunicode`.

Листинг 21.1. Примеры базовых операций над UTF-8 в Lua

```
local a = {}
a[#a + 1] = "Nähdään"
a[#a + 1] = "ação"
a[#a + 1] = "ÃøÆËÐ"

local l = table.concat(a, ";")

print(l, #(string.gsub(l, "[\128-\191]", "")))
  --> Nähdään;ação;ÃøÆËÐ 18

for w in string.gmatch(l, "[^;]+") do
  print(w)
end
  --> Nähdään
  --> ação
  --> ÃøÆËÐ

for c in string.gmatch(a[3], ".[\128-\191]*") do
  print(c)
end
```

```
--> Ã
--> ø
--> Æ
--> Ë
--> Ð
```

Упражнения

Упражнение 21.1. Напишите функцию `split`, которая получает строку и образец разделителя и возвращает последовательность из частей исходной строки между разделителями:

```
t = split("a whole new world", " ")
-- t = {"a", "whole", "new", "world"}
```

Как ваша функция обрабатывает пустые строки? (В частности, является ли пустая строка пустой последовательностью или последовательностью с одной пустой строкой?)

Упражнение 21.2. Образцы `'%D'` и `'[A%d]'` эквивалентны. А что насчет образцов `'[^%d%u]'` и `'[%D%U]'`?

Упражнение 21.3. Напишите функцию транслитерации. Эта функция получает строку и заменяет каждый символ в этой строке другим символом в соответствии с таблицей, заданной в качестве второго аргумента. Если таблица отображает `'a'` в `'b'`, то функция должна заменить каждое вхождение `'a'` на `'b'`. Если таблица отображает `'a'` в `false`, то функция должна удалить все вхождения символа `'a'` из строки.

Упражнение 21.4. Напишите функцию, которая переворачивает строку в UTF-8.

Упражнение 21.5. Напишите функцию транслитерации для символов UTF-8.

ГЛАВА 22

Библиотека ввода-вывода

Библиотека ввода-вывода предлагает две различные модели для работы с файлами. В простой модели используется *текущий входной файл* (current input file) и *текущий выходной файл* (current output file), и она производит над ними операции ввода-вывода. Полная модель использует явные дескрипторы файлов; она опирается на объектно-ориентированный подход, который определяет все операции как методы над дескрипторами файлов.

Простая модель удобна для простых вещей; мы применяли ее на протяжении всей книги. Но ее недостаточно для более гибкой работы с файлами, например для одновременного чтения или записи в несколько файлов. Для такой обработки нам нужна полная модель.

22.1. Простая модель ввода-вывода

Простая модель выполняет все свои операции над двумя текущими файлами. Библиотека инициализирует текущий входной файл как обрабатываемый стандартный ввод (`stdin`), а текущий выходной файл как обрабатываемый стандартный вывод (`stdout`). Таким образом, когда мы выполняем что-то вроде `io.read()`, мы читаем строку из стандартного ввода.

Мы можем изменить эти текущие файлы при помощи функций `io.input` и `io.output`. Вызов наподобие `io.input(filename)` открывает заданный файл в режиме чтения и устанавливает его как текущий входной файл. Начиная с этого момента, весь ввод будет поступать из этого файла, пока не произойдет другой вызов `io.input`; функция `io.output` делает то же самое, но для вывода.

В случае ошибки обе функции ее вызывают. Если вы хотите обрабатывать ошибки напрямую, то вы должны использовать полную модель.

Функция `write` проще, чем `read`, поэтому мы сперва рассмотрим ее. Функция `io.write` получает произвольное число строковых аргументов и записывает их в текущий выходной файл. Она преобразует числа в строки, следуя стандартным правилам преобразования; для полного контроля над этим преобразованием используйте функцию `string.format`:

```
> io.write("sin (3) = ", math.sin(3), "\n")
  --> sin (3) = 0.14112000805987
> io.write(string.format("sin (3) = %.4f\n",
math.sin(3)))
  --> sin (3) = 0.1411
```

Избегайте кода вроде `io.write(a..b..c)`; вызов `io.write(a,b,c)` дает тот же эффект с меньшими затратами, поскольку при этом конкатенация не нужна.

Как правило, вы должны использовать `print` для программ на скорую руку или для отладки, а `write` применять тогда, когда вам нужен полный контроль над выводом:

```
> print("hello", "Lua"); print("Hi")
  --> hello Lua
  --> Hi

> io.write("hello", "Lua"); io.write("Hi", "\n")
  --> helloLuaHi
```

В отличие от `print`, функция `write` не добавляет к выводу никаких дополнительных символов вроде символов табуляции или переводов строки. Кроме того, `write` позволяет вам перенаправить ваш вывод, тогда как `print` всегда использует стандартный вывод. Наконец, `print` автоматически применяет `tostring` к своим аргументам; это удобно для отладки, но может скрывать ошибки, если вы не внимательны к выводу.

Функция `io.read` читает строки из текущего входного файла. Ее аргументы управляют тем, что читать:

`*a` читает весь файл
`*l` читает следующую строку (без символа перевода строки)
`*L` читает следующую строку (с символом перевода строки)
`*n` читает число
число ограничивает количество читаемых символов

Вызов `io.read("*a")` читает весь текущий входной файл, начиная с текущей позиции. Если мы находимся в конце файла или файл пуст, то вызов возвращает пустую строку.

Поскольку Lua эффективно работает с длинными строками, простой прием написания фильтров на Lua состоит в том, чтобы прочесть весь файл в строку, выполнить обработку этой строки (обычно при помощи `gsub`) и затем записать строку в вывод:

```
t = io.read("*a")           -- читает из файла
t = string.gsub(t, ...)     -- делает свою работу
io.write(t)                 -- пишет в файл
```

В качестве примера следующий кусок является законченной программой для кодирования содержимого файла в MIME-кодировке *quoted-printable*. Каждый не ASCII-байт кодируется как *=xx*, где *xx* — это шестнадцатеричное значение байта. Для целостности кодирования символ '=' также должен быть закодирован:

```
t = io.read("*a")
t = string.gsub(t, "([\128-\255=])", function (c)
      return string.format("=%02X", string.byte(c))
    end)
io.write(t)
```

Этот образец, использованный в `gsub`, захватывает все байты от 128 до 255, а также знак равенства.

Вызов `io.read ("*l")` возвращает следующую строку из

текущего входного файла без символа перевода строки; вызов `io.read ("*L")` аналогичен, но он оставляет символ перевода строки (если он был в файле). Когда мы достигаем конца файла, вызов возвращает nil (так как нет следующей строки для возврата). По умолчанию в `read` используется "`*l`". Обычно я использую этот образец, только когда алгоритм естественным образом обрабатывает файл строка за строкой; в противном случае я предпочитаю прочесть весь файл за раз при помощи "`*a`" или читать его блоками, как мы увидим позже.

В качестве простого примера использования этого образца следующая программа копирует текущий ввод в текущий вывод, нумеруя при этом каждую строку:

```
for count = 1, math.huge do
  local line = io.read()
  if line == nil then break end
  io.write(string.format("%6d ", count), line, "\n")
end
```

Однако, чтобы построчно перебрать весь файл, лучше использовать итератор `io.lines`. Например, мы можем написать законченную программу для сортировки строк файла следующим образом:

```
local lines = {}
-- считывает строки в таблицу 'lines'
for line in io.lines() do lines[#lines + 1] = line end
-- сортирует
table.sort(lines)
-- записывает все строки
for _, l in ipairs(lines) do io.write(l, "\n") end
```

Вызов `io.read("*n")` читает число из текущего входного файла. Это единственный случай, когда функция `read` возвращает число, а не строку. Когда программе нужно прочесть слишком много чисел из файла, отсутствие промежуточных строк улучшает ее быстродействие. Опция `*n` пропускает все пробелы перед числом и поддерживает такие числовые форматы, как -3, +5.2,1000 и

-3.4e-23. Если функция не может найти число в текущей позиции (из-за неверного формата или конца файла), то она возвращает nil.

Вы можете вызвать `read` с несколькими опциями; для каждого аргумента функция вернет соответствующее значение. Допустим, у вас есть файл, содержащий по три числа на каждую строку:

```
6.0 -3.23 15e12
4.3 234 1000001
...
```

Теперь вы хотите напечатать максимальное число каждой строки. Вы можете прочесть все три числа за один вызов `read`:

```
while true do
  local n1, n2, n3 = io.read("*n", "*n", "*n")
  if not n1 then break end
  print(math.max(n1, n2, n3))
end
```

Кроме основных образцов для чтения, вы можете вызвать `read` с аргументом в виде числа *n*: в этом случае `read` пытается прочесть *n* символов из входного файла. Если она не может прочесть ни одного символа (конец файла), то она возвращает nil; в противном случае возвращается строка с не более чем *n* символами. В качестве примера данного образца для чтения следующая программа показывает эффективный способ (для Lua, конечно) скопировать файл из `stdin` в `stdout`:

```
while true do
  local block = io.read(2^13)    -- размер буфера
равен 8K
  if not block then break end
  io.write(block)
end
```

Как особый случай, `io.read(0)` работает как проверка конца файла: она возвращает либо пустую строку, если еще есть, что читать, либо nil, если читать нечего.

22.2. Полная модель ввода-вывода

Для большего контроля над вводом-выводом вы можете использовать полную модель. Данная модель основана на понятии *дескриптора файла* (file handle), который аналогичен потокам (`FILE*`) в C: он представляет собой открытый файл с текущей позицией.

Чтобы открыть файл, вы используете функцию `io.open`, которая подобна функции `fopen` в C. В качестве аргументов она принимает имя файла, который нужно открыть, и *строку режима* (mode string). Эта строка может содержать `'r'` для чтения, `'w'` для записи (которая при этом стирает любое предыдущее содержимое файла) или `'a'` для добавления к файлу; еще она может содержать необязательный `'b'` для открытия двоичных файлов. Функция `open` возвращает новый дескриптор файла. В случае ошибки `open` возвращает nil, а также сообщение об ошибке и ее код:

```
print(io.open("non-existent-file", "r"))
  --> nil non-existent-file: No such file or directory
2

print(io.open("/etc/passwd", "w"))
  --> nil /etc/passwd: Permission denied 13
```

Интерпретация кодов ошибок зависит от системы.

Типичная идиома для проверки на ошибки:

```
local f = assert(io.open(filename, mode))
```

Если `open` даст сбой, то сообщение об ошибке переходит вторым аргументом в `assert`, которая затем показывает это сообщение.

После открытия файла вы может читать из него и писать в него при помощи методов `read` и `write`. Они аналогичны функциям `read` и `write`, но вы вызываете их как методы дескриптора файла,

используя двоеточие. Например, чтобы открыть файл и прочесть все из него, вы можете использовать кусок вроде этого:

```
local f = assert(io.open(filename, "r"))
local t = f:read("*a")
f:close()
```

Библиотека ввода-вывода предлагает дескрипторы для трех предопределенных потоков C: `io.stdin`, `io.stdout` и `io.stderr`. Поэтому вы можете послать сообщение прямо в поток с ошибкой при помощи примерно такого кода:

```
io.stderr:write(message)
```

Мы можем смешивать полную модель с простой. Мы получаем дескриптор текущего входного файла посредством вызова `io.input()` без аргументов. Мы устанавливаем этот дескриптор путем вызова `io.input(hanle)` (аналогичные вызовы работают и для `io.output`). Например, если вы хотите временно изменить текущий входной файл, то вы можете написать что-то вроде этого:

```
local temp = io.input()     -- сохраняет текущий файл
io.input("newinput")        -- открывает новый текущий
файл
<делает что-нибудь с новым вводом>
io.input():close()          -- закрывает текущий файл
io.input(temp)              -- восстанавливает
предыдущий текущий файл
```

Вместо `io.read` для чтения из файла мы также можем использовать `io.lines`. Как мы уже видели в предыдущих примерах, `io.lines` возвращает итератор, последовательно читающий из файла.

Первым аргументом `io.lines` может быть имя файла или дескриптор файла. Если передано имя файла, то `io.lines` откроет файл в режиме для чтения и закроет файл после достижения конца файла. Если передан дескриптор файла, то `io.lines` будет

использовать данный файл для чтения; в этом случае `io.lines` не будет закрывать файл по достижении его конца. В случае вызова вообще без аргументов `io.lines` будет читать данные из текущего входного файла.

Начиная с Lua 5.2, `io.lines` принимает и те опции, которые принимает `io.read` после файлового аргумента. В качестве примера следующий код копирует файл в текущий вывод, используя `io.lines`:

```
for block in io.lines(filename, 2^13) do
  io.write(block)
end
```

Небольшой прием для увеличения быстродействия

Обычно в Lua быстрее прочесть файл целиком, чем читать его строка за строкой. Однако, иногда мы сталкиваемся с большим файлом (например, десятки или даже сотни мегабайт), читать который целиком было бы нецелесообразно. Если вы хотите получить максимальное быстродействие при работе с такими большими файлами, то быстрее всего будет читать его достаточно большими кусками (например, по 8К). Во избежание возможного разрыва строки, можно просто попросить прочесть еще одну строку:

```
local lines, rest = f:read(BUFSIZE, "*l")
```

Переменная `rest` получит остаток любой строки, разбитой при чтении куска. Затем мы объединяем кусок и полученный остаток. Таким образом кусок всегда будет завершаться на границе строк.

Пример из листинга 22.1 использует этот прием для реализации `wc`, программы, которая считает число символов, слов и строк в файле. Обратите внимание на использование `io.lines` для осуществления итераций и опции `"*b"` для чтения строки — это

доступно, начиная с Lua 5.2.

Листинг 22.1. Программа `wc`

```
local BUFSIZE = 2^13          -- 8K
local f = io.input(arg[1])    -- открывает входной
файл
local cc, lc, wc = 0, 0, 0    -- счетчики символов,
строк и слов
for lines, rest in io.lines(arg[1], BUFSIZE, "*L") do
  if rest then lines = lines .. rest end
  cc = cc + #lines
  -- подсчитывает слова в куске
  local _, t = string.gsub(lines, "%S+", "")
  wc = wc + t
  -- подсчитывает переводы строки в куске
  _,t = string.gsub(lines, "\n", "\n")
  lc = lc + t
end
print(lc, wc, cc)
```

Бинарные файлы

Функции `io.input` и `io.output` из простой модели всегда по умолчанию открывают файл в текстовом режиме. В UNIX нет никакой разницы между бинарными и текстовыми файлами. Но в некоторых системах, в частности в Windows, бинарные файлы нужно открывать со специальным флагом. Для обработки таких бинарных файлов вы должны использовать `io.open` с символом '`b`' в строке режима.

Lua работает с бинарными данными так же, как и с текстом. Строка в Lua может содержать любые байты, и почти все функции в библиотеках могут обрабатывать любые байты. Вы даже можете применять сопоставление с образцом к бинарным данным до тех пор, пока образец не содержит нулевого байта. Если вам нужно сопоставление с этим байтом, то вы можете воспользоваться для

этого классом `%z`.

Обычно бинарные данные читают либо при помощи образца `*a`, который читает весь файл, либо при помощи образца *n*, который читает *n* байт. В качестве простого примера следующая программа переводит текст из формата Windows в формат UNIX (то есть заменяет последовательность символов перевода каретки и перевода строки на символ перевода строки). Она не пользуется стандартными файлами ввода-вывода (`stdin-stdout`), поскольку они открыты в текстовом режиме. Вместо этого она полагает, что имена входного и выходного файлов переданы программе как аргументы:

```
local inp = assert(io.open(arg[1], "rb"))
local out = assert(io.open(arg[2], "wb"))

local data = inp:read("*a")
data = string.gsub(data, "\r\n", "\n")
out:write(data)

assert(out:close())
```

Вы можете вызвать эту программу при помощи следующей командной строки:

```
> lua prog.lua file.dos file.unix
```

В качестве еще одного примера следующая программа печатает все строки, найденные в бинарном файле:

```
local f = assert(io.open(arg[1], "rb"))
local data = f:read("*a")
local validchars = "[%g%s]"
local pattern = "(" .. string.rep(validchars, 6) ..
"+)\0"
for w in string.gmatch(data, pattern) do
  print(w)
end
```

Программа считает, что строка — это завершенная нулем

последовательность не менее чем из шести допустимых символов, где допустимым является любой символ, который соответствует образцу `validchars`. В нашем примере этот образец состоит из печатаемых символов. Мы используем `string.rep` и конкатенацию для создания образца, которому удовлетворяют последовательности из шести и более допустимых символов, за которыми следует нулевой байт. Круглые скобки в образце захватывают эту строку (без нулевого байта).

В качестве последнего примера следующая программа делает дамп бинарного файла:

```
local f = assert(io.open(arg[1], "rb"))
local block = 16
for bytes in f:lines(block) do
  for c in string.gmatch(bytes, ".") do
    io.write(string.format("%02X ", string.byte(c)))
  end
  io.write(string.rep(" ", block - string.len(bytes)))
  io.write(" ", string.gsub(bytes, "%c", "."), "\n")
end
```

Листинг 22.2. Получение дампа посредством программы **dump**

```
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  local
f = assert
28 69 6F 2E 6F 70 65 6E 28 61 72 67 5B 31 5D 2C
(io.open(arg[1],
20 22 72 62 22 29 29 0A 6C 6F 63 61 6C 20 62 6C
"rb")).local bl
6F 63 6B 20 3D 20 31 36 0A 66 6F 72 20 62 79 74  ock =
16.for byt
65 73 20 69 6E 20 66 3A 6C 69 6E 65 73 28 62 6C  es in
f:lines(bl
...
20 22 2C 20 73 74 72 69 6E 67 2E 67 73 75 62 28  ",
string.gsub(
62 79 74 65 73 2C 20 22 25 63 22 2C 20 22 2E 22
bytes, "%c", "."
29 2C 20 22 5C 6E 22 29 0A 65 6E 64 0A 0A        ),
"\n").end..
```

Как и раньше, первым аргументом программы является имя входного файла; выходной файл идет на стандартный вывод. Программа читает файл кусками по 16 байт. Для каждого куска выводится шестнадцатеричное представление каждого байта, а затем кусок записывается как текст, заменяя управляющие символы точками.

Листинг 22.2 показывает результат применения этой программы самой к себе (на UNIX-машине).

22.3. Прочие операции над файлами

Функция `tmpfile` возвращает дескриптор временного файла, открытого в режиме чтения-записи. Этот файл будет автоматически удален по завершении программы.

Функция `flush` применяет к файлу все отложенные операции записи. Подобно `write`, вы можете вызвать ее либо как функцию `io.flush()` для сброса текущего выходного файла на диск, либо как метод `f:flush()` для сброса конкретного файла `f`.

Метод `setvbuf` устанавливает режим буферизации потока. Его первым аргументом является строка: `"no"` означает отсутствие буферизации; `"full"` означает, что поток записывается лишь тогда, когда буфер заполнен или вы явно сбрасываете файл; `"line"` означает, что вывод буферизуется до перевода строки или при вводе из некоторых особых файлов (например, из терминала). Для последних двух опций `setvbuf` допускает необязательный второй аргумент, задающий размер буфера.

В большинстве систем стандартный поток ошибок (`io.stderr`) не буферизуется, в то время как стандартный выходной поток (`io.stdout`) буферизуется в построчном режиме. Поэтому если вы записываете незавершенные строки в стандартный вывод (например, индикатор состояния операции), то, чтобы увидеть вывод, вам может понадобиться сброс содержимого буфера.

Метод `seek` может как возвращать, так и устанавливать текущую позицию внутри файла. Его общей формой является `f:seek(`*место*`,`*смещение*`)`, где *место* — это строка, задающая, как надо интерпретировать *смещение*. Ее допустимыми значениями являются "`set`", когда смещение трактуется от начала файла, "`cur`", когда смещение трактуется от текущей позиции внутри файла, и "`end`", когда смещение трактуется с конца файла. Независимо от значения места вызов возвращает новую текущую позицию, измеренную в байтах от начала этого файла.

Значениями по умолчанию являются "`cur`" для места и `0` для смещения. Поэтому вызов `file:seek()` возвращает текущее положение внутри файла, не меняя его; вызов `file:seek("set")` возвращает позицию в начало файла (и возвращает ноль); а вызов `file:seek("end")` устанавливает позицию на конец файла и возвращает его размер. Следующая функция получает размер файла, не меняя текущую позицию внутри него:

```
function fsize (file)
  local current = file:seek()      -- получает текущую
позицию
  local size = file:seek("end")    -- получает размер
файла
  file:seek("set", current)        -- восстанавливает
позицию
  return size
end
```

В случае ошибки все эти функции возвращают nil и сообщение об ошибке.

Упражнения

Упражнение 22.1. Напишите программу, которая читает текстовый файл и перезаписывает его, используя его же строки, отсортированные в алфавитном порядке. При вызове без

аргументов она должна читать из стандартного входного файла и записывать в стандартный выходной файл. При вызове с одним аргументом в виде имени файла, она должна читать из этого файла и записывать в стандартный выходной файл. При вызове с двумя аргументами в виде имен файлов она должна читать из первого файла и писать во второй.

Упражнение 22.2. Измените предыдущую программу так, чтобы она запрашивала подтверждение, если пользователь задает имя существующего файла для ее вывода.

Упражнение 22.3. Сравните быстродействие программы на Lua, которая копирует стандартный входной файл в стандартный выходной файл следующими способами:

- побайтно;
- построчно;
- кусками по 8К;
- весь файл за раз.

Насколько большим может быть входной файл для последнего варианта?

Упражнение 22.4. Напишите программу, которая печатает последнюю строку текстового файла. Постарайтесь избежать чтения всего файла, когда файл большой и к нему можно применить `seek`.

Упражнение 22.5. Обобщите предыдущую программу так, чтобы она печатала последние *n* строк текстового файла. Опять же постарайтесь избежать чтения всего файла, когда он большой и к нему можно применить `seek`.

ГЛАВА 23

Библиотека операционной системы

Библиотека операционной системы включает в себя функции для работы с файлами (не с потоками), получения текущих даты и времени и другие средства, касающиеся операционной системы. Она определена в таблице `os`. Переносимость Lua сказалась на этой библиотеке: поскольку Lua написана на чистом ANSI C, то эта библиотека включает в себя только функциональность предоставляемую стандартом ANSI. Многие средства ОС, такие как работа с директориями и сокетами, не входят в этот стандарт, и поэтому данная библиотека их не предоставляет. Существуют другие библиотеки Lua, не включенные в основную поставку, которые обеспечивают расширенный доступ к ОС. Примеры таких библиотек: `posix`, предоставляющая для Lua всю функциональность стандарта POSIX.1, `luasocket` для работы с сетью и `LuaFileSystem` для работы с директориями и атрибутами файлов.

Все, что предлагает данная библиотека для работы с файлами, — это функции `os.rename` для изменения имени файла и `os.remove` для удаления файла.

23.1. Дата и время

Вся функциональность для работы с датами и временем в Lua обеспечивается двумя функциями — `time` и `date`.

Функция `time`, когда она вызвана без аргументов, возвращает текущую дату и время, представленные как число. (В большинстве систем это число секунд, прошедших с определенного начала отсчета.) При вызове с аргументом в виде таблицы, она возвращает

число, которое представляет дату и время, описанное этой таблицей. У таких *дата-таблиц* (date tables) есть следующие значимые поля:

`year` полный год
`month` 01—12 (месяц)
`day` 01—31 (день)
`hour` 00—23 (час)
`min` 00—59 (минута)
`sec` 00—59 (секунда)
`isdst` логическое значение, true при переходе на летнее время

Первые три поля обязательны; для остальных значением по умолчанию, когда оно не предоставлено, является полдень (12:00:00). В UNIX-системе, где началом отсчета является 00:00:00 1 января 1970 г. по универсальному координированному времени, для Рио-де-Жанейро (который на три часа западнее Гринвича) у нас будут следующие результаты:

```
print(os.time{year=1970, month=1, day=1, hour=0})
  --> 10800
print(os.time{year=1970, month=1, day=1, hour=0,
sec=1})
  --> 10801
print(os.time{year=1970, month=1, day=1})
  --> 54000
```

(Обратите внимание, что 10 800 — это 3 часа в секундах, 54 000 — это 10 800 плюс 12 часов в секундах.)

Функция `date`, несмотря на свое имя, является своего рода противоположностью функции `time`: она преобразует число, обозначающее дату и время, обратно в какое-нибудь высокоуровневое представление. Ее первый параметр — это форматирующая строка, описывающая, какое именно представление нам нужно. Второй параметр — это дата и время в виде одного числа; по умолчанию, если его не использовать, он равен текущей дате и времени.

Чтобы получить дата-таблицу, мы воспользуемся форматирующей строкой "*t". Например, вызов os.date("*t",906000490) вернет следующую таблицу:

```
{year = 1998, month = 9, day = 16, yday = 259, wday =
4,
 hour = 23, min = 48, sec = 10, isdst = false}
```

Обратите внимание, что, кроме полей, используемых os.time, таблица, созданная os.date, также задает день недели (wday, 1 — это воскресенье) и день года (yday, 1 — это 1-ое января).

Для других форматирующих строк os.date форматирует дату как копию форматирующей строки, где заданные теги заменены информацией о дате и времени. Тег состоит из '%', за которым следует буква, как в следующих примерах:

```
print(os.date("a %A in %B"))       --> a Tuesday in
May
print(os.date("%x", 906000490))    --> 09/16/1998
```

Все представления соответствуют текущей локали. Например, для бразильской португальской локали результатом %B даст "setembro", а %x даст "16/09/98".

Следующая таблица показывает каждый тег, его смысл и значение для 16 сентября 1998 года, 23:48:10 (среда). Для числовых значений таблица также показывает их диапазон допустимых значений:

%a сокращенное название дня недели (например, Wed)
%A полное название дня недели (например, Wednesday)
%b сокращенное название месяца (например, Sep)
%B полное название месяца (например, September)
%c дата и время (например, 09/16/98 23:48:10)
%d день месяца (16) [01–31]
%H час, используя 24-часовое время 23) [00–23]
%I час, используя 12-часовое время (11) [01–12]

%j день года (259) [001–366]
%M минута (48) [00–59]
%m месяц (09) [01–12]
%p либо "am", либо "pm"
%S секунда (10) [00–60]
%w день недели (3) [0–6 = Sunday–Saturday]
%x дата (например, 09/16/98)
%X время (например, 23:48:10)
%y сокращенный год из двух цифр (98) [00–99]
%Y полный год (1998)
%% символ '%'

Если вы вызовете `date` без каких-либо аргументов, то будет использован формат `%c`, то есть полная дата и время в подходящем формате. Обратите внимание, что представления для `%x`, `%X` и `%c` зависят от локали и системы. Если вам нужно фиксированное представление, например `mm/dd/yyyy`, то используйте явную строку формата вроде "`%m/%d/%Y`".

Функция `os.clock` возвращает число затраченных программой секунд процессорного времени. Обычно она используется для замера производительности фрагмента кода:

```
local x = os.clock()
local s = 0
for i = 1, 100000 do s = s + i end
print(string.format("elapsed time: %.2f\n", os.clock()
- x))
```

23.2. Прочие системные вызовы

Функция `os.exit` завершает выполнение программы. Ее необязательный первый аргумент — это статус программы при выходе из нее. Он может быть числом (успешному выполнению соответствует ноль) или логическим значением (успешному выполнению соответствует true). Если необязательный второй

аргумент равен true, то состояние Lua закрывается посредством вызова всех финализаторов и освобождения всей памяти, используемой этим состоянием. (Обычно эта финализация не является обязательной, так как большинство операционных систем освобождает все ресурсы, занятые процессом, при выходе из него.)

Функция `os.getenv` возвращает значение переменной окружения. Она принимает имя переменной и возвращает строку с ее значением:

```
print(os.getenv("HOME"))    --> /home/lua
```

Для неопределенных переменных этот вызов возвращает nil.

Функция `os.execute` выполняет команду операционной системы; она эквивалентна функции `system` в C. Она принимает строку с командой и возвращает информацию о том, как эта команда была завершена. Первое возвращаемое значение логическое: true означает окончание программы без ошибок. Второе возвращаемое значение — это строка: "`exit`", если программа завершилась нормально, и "`signal`", если она была прервана сигналом. Третье возвращаемое значение — это статус возврата, если программа завершилась нормально, или номер сигнала, если она завершилась по сигналу. В качестве примера использования и в Windows, и в UNIX вы можете использовать следующую функцию для создания новых директорий:

```
function createDir (dirname)
  os.execute("mkdir " .. dirname)
end
```

Функция `os.execute` обладает широкими возможностями, но при этом сильно зависит от используемой системы.

Функция `os.setlocale` задает текущую локаль, используемую программой Lua. Локали определяют поведение, которое зависит от культурных и языковых различий. У функции `os.setlocale` есть два строковых параметра: имя локали и категория, которая

определяет, на какие характеристики эта локаль повлияет. У локалей есть шесть возможных категорий:

- "collate" управляет алфавитным порядком строк;
- "ctype" управляет типами отдельных символов (например, определяет, что является буквой) и преобразованием между строчными и заглавными буквами;
- "monetary" не влияет на программы на Lua;
- "numeric" управляет тем, как форматируются числа;
- "time" управляет тем, как форматируются дата и время (для функции os.date);
- "all" управляет всеми перечисленными функциями.

Категорией по умолчанию является "all", то есть если вы вызвали setlocale только с именем локали, то она будет выставлена для всех категорий. Функция setlocale возвращает имя локали и nil в случае сбоя (обычно когда система не поддерживает данную локаль).

```
print(os.setlocale("ISO-8859-1", "collate")) --> ISO-
8859-1
```

У категории "numeric" есть некоторые тонкости. Поскольку португальский и некоторые латинские языки используют для представления десятичных чисел запятую вместо точки, то локаль меняет способ, которым Lua печатает и считывает эти числа. Но локаль не влияет на то, как Lua разбирает числа внутри программы (одна из многих причин состоит в том, что у выражений вроде print(3,4) уже есть значение в Lua. Если вы пишете на Lua фрагменты кода, то здесь у вас могут быть проблемы:

```
print(os.setlocale("pt_BR"))    --> pt_BR
s = "return (" .. 3.4 .. ")"
print(s)                        --> return (3,4)
print(loadstring(s))
  --> nil [string "return (3,4)"]:1: ')' expected near
','
```

Во избежание подобных проблем, убедитесь, что ваша программа использует стандартную локаль "C" при создании фрагментов кода.

Упражнения

Упражнение 23.1. Напишите функцию, которая возвращает дату и время спустя ровно месяц от текущей даты (с применением обычного кодирования даты как числа).

Упражнение 23.2. Напишите функцию, которая получает дату и время в виде числа и возвращает число секунд, прошедших с начала дня той даты.

Упражнение 23.3. Можете ли вы использовать `os.execute`, чтобы изменить текущую директорию вашей программы Lua? Почему?

ГЛАВА 24

Отладочная библиотека

Отладочная библиотека не является отладчиком, но она предложит все необходимые примитивы для написания вашего собственного отладчика для Lua. Из соображения быстродействия официальным интерфейсом к этим примитивам является C API. Отладочная библиотека в Lua — это способ получить к ним доступ напрямую из кода Lua.

В отличие от других библиотек, вы не должны использовать отладочную библиотеку слишком часто. Во-первых, часть ее функциональности не отличается быстродействием. Во-вторых, она нарушает некоторые непреложные истины языка, например то, что вы не можете обратиться к локальной переменной вне ее лексической области видимости. Скорее всего вы решите отказаться от использования этой библиотеки в финальной версии продукта, а то и вовсе захотите ее стереть.

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

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

24.1. Интроспективные средства

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

- `source`: где была определена функция. Если эта функция была определена в строке (посредством `loadstring`), то значением `source` будет эта строка. Если функция была определена в файле, то значение `source` — имя этого файла с префиксом '`@`';
- `short_src`: короткая версия `source` (до 60 символов), полезна для сообщений об ошибках;
- `linedefined`: номер первой строки в `source`, где функция была определена;
- `lastlinedefined`: номер последней строки в `source`, где функция была определена;
- `what`: что это за функция. Возможные значения: "`Lua`", если это обычная функция Lua, "`C`", если это функция C, или "`main`", если это главная часть куска Lua;
- `name`: подходящее для функции имя;
- `namewhat`: что означает предыдущее поле. Возможные значения: "`global`", "`local`", "`method`", "`field`" и "" (пустая строка). Пустая строка означает, что Lua не нашел имени для функции;
- `nups`: количество верхних значений для этой функции;
- `activelines`: таблица, представляющая множество активных строк функции. *Активная строка* (active line) — это строка с каким-то кодом, в отличие от пустых строк и строк, состоящих только из комментариев. (Типичное

использование данной информации — это установка точек прерывания (breakpoint). Большинство отладчиков не позволяет задавать точки прерывания не на активных строках, так как они были бы недостижимы.)

- `func`: сама функция; об этом позже.

Когда `foo` является функцией C, у Lua о ней почти нет никаких данных. Для таких функций значимы лишь поля `what`, `name` и `namewhat`.

Когда вы вызываете `debug.getinfo(n)` для какого-то числа *n*, вы получаете данные о функции, активной на этом уровне стека. Например, если *n* равно 1, то вы получаете данные о функции, совершающей вызов. (Когда *n* равно 0, вы получаете данные о самой функции `getinfo`, т.е. функцию C.) Если *n* больше числа активных функций в стеке, то `debug.getinfo` возвращает nil. Когда вы опрашиваете активную функцию, вызывая `debug.getinfo` с числовым аргументом, у итоговой таблицы будет одно дополнительное поле с именем `currentline`, содержащее номер строки, на которой находится функция в данный момент. Кроме того, `func` содержит функцию, которая активна на этом уровне.

Поле `name` непростое. Как вы помните, из-за того, что функции в Lua являются значениями первого класса, функция может вообще не иметь имени или иметь несколько имен. Lua пытается найти имя функции путем просмотра кода, вызвавшем эту функцию, чтобы увидеть, как он ее вызвал. Этот метод работает лишь при вызове `getinfo` с числовым аргументом, то есть когда мы запрашиваем информацию о конкретном вызове.

Функция `getinfo` обладает низкой производительностью. Lua содержит отладочную информацию в форме, которая не ухудшает быстродействие программы; эффективный поиск информации здесь вторичен. Чтобы получить большее быстродействие, у `getinfo` есть необязательный второй параметр, который ограничивает круг поиска необходимой информации. Таким образом, данная функция

не тратит лишнее время на сбор лишней информации. Формат данного параметра является строкой, где каждая буква служит для выбора группы полей согласно следующей таблице:

```
'n' name, namewhat
'f' func
'S' source, short_src, what, linedefined, lastlinedefined
'l' currentline
'L' activelines
'u' nup
```

Следующая функция иллюстрирует использование `debug.getinfo`. Она распечатывает примитивную обратную трассировку активного стека:

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

info.currentline))
    end
  end
end
```

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

Доступ к локальным переменным

Мы можем изучать локальные переменные любой активной

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

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

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

Вызов `foo(10,20)` напечатает следующее:

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

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

Начиная с Lua 5.2, отрицательные индексы возвращают

информацию о дополнительных аргументах функции: индекс -1 соответствует первому дополнительному аргументу. В этом случае именем переменной всегда будет "`(*vararg)`".

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

Доступ к нелокальным переменным

Отладочная библиотека также позволяет обращаться к нелокальным переменным, используемым функцией Lua, при помощи `getupvalue`. В отличие от локальных переменных, нелокальные переменные, используемые функцией, существуют, даже когда функция не активна (в конце концов, в этом суть замыканий). Поэтому первый аргумент для `getupvalue` — это не уровень в стеке, а функция (точнее, замыкание). Второй аргумент — это индекс переменной. Lua нумерует нелокальные переменные в том порядке, в котором они впервые встречаются в функции, но этот порядок не важен, поскольку функция не может обратиться сразу к двум нелокальным переменным с одним и тем же именем.

Вы также можете обновлять нелокальные переменные при помощи `debug.setupvalue`. У нее, как вы уже догадались, три параметра: замыкание, имя переменной и новое значение. Как и `setlocal`, она возвращает имя переменной или nil, если индекс переменной вне допустимого диапазона.

Листинг 24.1 показывает, как мы можем получить доступ к значению любой из этих переменных по ее имени. Параметр `level` сообщает, где именно эта функция должна искать; увеличение на единицу нужно, чтобы не включать вызов к самой функции

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

Листинг 24.1. Получение значения переменной

```
function getvarvalue (name, level)
  local value
  local found = false

  level = (level or 1) + 1

  -- пробует локальные переменные
  for i = 1, math.huge do
    local n, v = debug.getlocal(level, i)
    if not n then break end
    if n == name then
      value = v
      found = true
    end
  end
  if found then return value end

  -- пробует нелокальные переменные
  local func = debug.getinfo(level, "f").func
  for i = 1, math.huge do
    local n, v = debug.getupvalue(func, i)
    if not n then break end
    if n == name then return v end
  end
```

```
    -- не найдено; получает значение из окружения
    local env = getvarvalue("_ENV", level)
    return env[name]
  end
```

Доступ к другим сопрограммам

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

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

coroutine.resume(co)
print(debug.traceback(co))
```

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

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

Данная трассировка не затрагивает вызов `resume`, поскольку эта сопрограмма и главная программа выполняются в разных стеках.

Когда сопрограмма вызывает ошибку, она не раскручивает стек. Это значит, что после ошибки мы можем его изучить. В продолжение нашего примера, если мы вновь возобновим сопрограмму, это приведет к ошибке:

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

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

вроде:

```
stack traceback:
        [C]: in function 'error'
        temp:4: in function <temp:1>
```

При этом мы можем изучать локальные переменные из сопрограммы даже после ошибки:

```
print(debug.getlocal(co, 1, 1))    --> x 10
```

24.2. Ловушки

Механизм *ловушек* (hook) из отладочной библиотеки позволяет нам зарегистрировать функцию, которая будет вызвана при наступлении определенных событий во время выполнения программы. Существует четыре вида событий, которые могут заставить сработать ловушки:

- *call* (событие вызова) происходит каждый раз, когда Lua вызывает функцию;
- *return* (событие возврата) происходит каждый раз при возврате из функции;
- *line* (событие строки) происходит, когда Lua начинает выполнение следующей строки кода;
- *count* (событие счетчика) происходит после заданного количества команд.

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

Чтобы зарегистрировать ловушку, мы вызываем функцию

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

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

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

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

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

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

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

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

```
    end
  end
```

Когда пользователь вводит «команду» `cont`, эта функция завершается. Стандартная реализация очень проста и выполняет команды в глобальном окружении, вне области видимости отлаживаемого кода. Упражнение 24.5 обсуждает более удачную реализацию.

24.3. Профилирование

Несмотря на свое имя, отладочная библиотека годится и для других задач. Часто такой задачей является профилирование. Для профилирования с учетом времени лучше использовать интерфейс C, так как затраты Lua на вызов каждой ловушки довольно велики и могут свести на нет любые замеры. Тем не менее, для профилирования на основе подсчета вызовов код Lua вполне подходит. В этом разделе мы разработаем элементарный профилировщик, который подсчитывает, сколько раз была вызвана функция во время выполнения программы.

Главными структурами данных нашей программы будут две таблицы: одна связывает функции с их счетчиками вызовов, а другая связывает функции с их именами. Индексами в обеих таблицах будут сами функции.

```
local Counters = {}
local Names = {}
```

Мы могли бы извлечь имена функций и после профилирования, но не забывайте, что мы получим лучшие результаты, если будем извлекать имена функций, пока они активны, поскольку в этом случае Lua может просматривать код, который вызывает эту функцию, чтобы найти ее имя.

Теперь мы определим функцию ловушки. Ее задачей является

получить вызванную функцию и увеличить соответствующий счетчик; при этом она собирает имена функций:

```
local function hook ()
  local f = debug.getinfo(2, "f").func
  local count = Counters[f]
  if count == nil then -- first time 'f' is called?
    Counters[f] = 1
    Names[f] = debug.getinfo(2, "Sn")
  else -- only increment the counter
    Counters[f] = count + 1
  end
end
```

Следующим шагом является запуск программы с этой ловушкой. Мы будем считать, что главный кусок программы находится в файле, и что пользователь в качестве аргумента передает профилировщику имя этого файла, например так:

```
% lua profiler main-prog
```

С этой схемой профилировщик может взять имя файла из `arg[1]`, включить ловушку и выполнить файл:

```
local f = assert(loadfile(arg[1]))
debug.sethook(hook, "c")    -- включает ловушку для
вызовов
f()                         -- выполняет главную
программу
debug.sethook()             -- выключает эту ловушку
```

Последний шаг — это показ результатов. Функция `getname` из листинга 24.2 выдает для каждой функции ее имя. Из-за того, что имена функций в Lua несколько непостоянны, мы добавим к каждой функции ее местоположение, заданное в виде пары *файл*:*номер_строки*. Если у функции нет имени, то мы печатаем лишь ее местоположение. Если это функция C, то мы используем только ее имя (так как у нее нет местоположения). После данного определения мы печатаем каждую функцию с ее счетчиком:

```
for func, count in pairs(Counters) do
  print(getname(func), count)
end
```

Листинг 24.2. Получение имени функции

```
function getname (func)
  local n = Names[func]
  if n.what == "C" then
    return n.name
  end
  local lc = string.format("[%s]:%d", n.short_src,
n.linedefined)
  if n.what ~= "main" and n.namewhat ~= "" then
    return string.format("%s (%s)", lc, n.name)
  else
    return lc
  end
end
```

Если мы применим наш профилировщик к примеру с цепью Маркова, который мы разработали в разделе 10.3, то получим результат вроде этого:

```
[markov.lua]:4 884723
write 10000
[markov.lua]:0 1
read 31103
sub 884722
[markov.lua]:1 (allwords)    1
[markov.lua]:20 (prefix)     894723
find 915824
[markov.lua]:26 (insert)     884723
random 10000
sethook 1
insert 884723
```

Данный результат указывает на то, что анонимная функция в строке 4 (которая является нашим итератором, определенным внутри `allwords`) была вызвана 884 723 раз, функция `write(io.write)` была вызвана 10 000 раз и т. д.

Есть несколько улучшений, которые могут быть внесены в этот профилировщик, например, сортировка вывода, улучшенная печать имен функций и более красивый формат вывода. Тем не менее, этот базовый профилировщик уже и так полезен и может быть использован как основа более продвинутых инструментов.

Упражнения

Упражнение 24.1. Почему рекурсия в функции `getvarvalue` (листинг 24.1) обязательно остановится?

Упражнение 24.2. Приспособьте функцию `getvarvalue` (листинг 24.1) для работы с различными сопрограммами (подобно другим функциям из отладочной библиотеки).

Упражнение 24.3. Напишите функцию `setvarvalue`, похожую на `getvarvalue` (листинг 24.1).

Упражнение 24.4. Напишите модификацию `getvarvalue` (листинг 24.1) под именем `getallvars`, которая возвращает таблицу со всеми переменными, которые видны в вызывающей функции. (Ввозвращаемая таблица не должна включать в себя переменные окружения; вместо этого она должна наследовать их из исходного окружения).

Упражнение 24.5. Напишите улучшенную версию `debug.debug`, которая выполняет заданные команды, как если бы они были выполнены в области видимости вызывающей функции. (Подсказка: выполняйте команды в пустом окружении и используйте метаметод `__index` с прикрепленной функцией `getvarvalue` для всех обращений к переменным.)

Упражнение 24.6. Улучшите предыдущий пример, добавив обновление переменных.

Упражнение 24.7. Реализуйте некоторые из предложенных улучшений для базового профилировщика из раздела 24.3.

Упражнение 24.8. Напишите библиотеку для точек прерывания

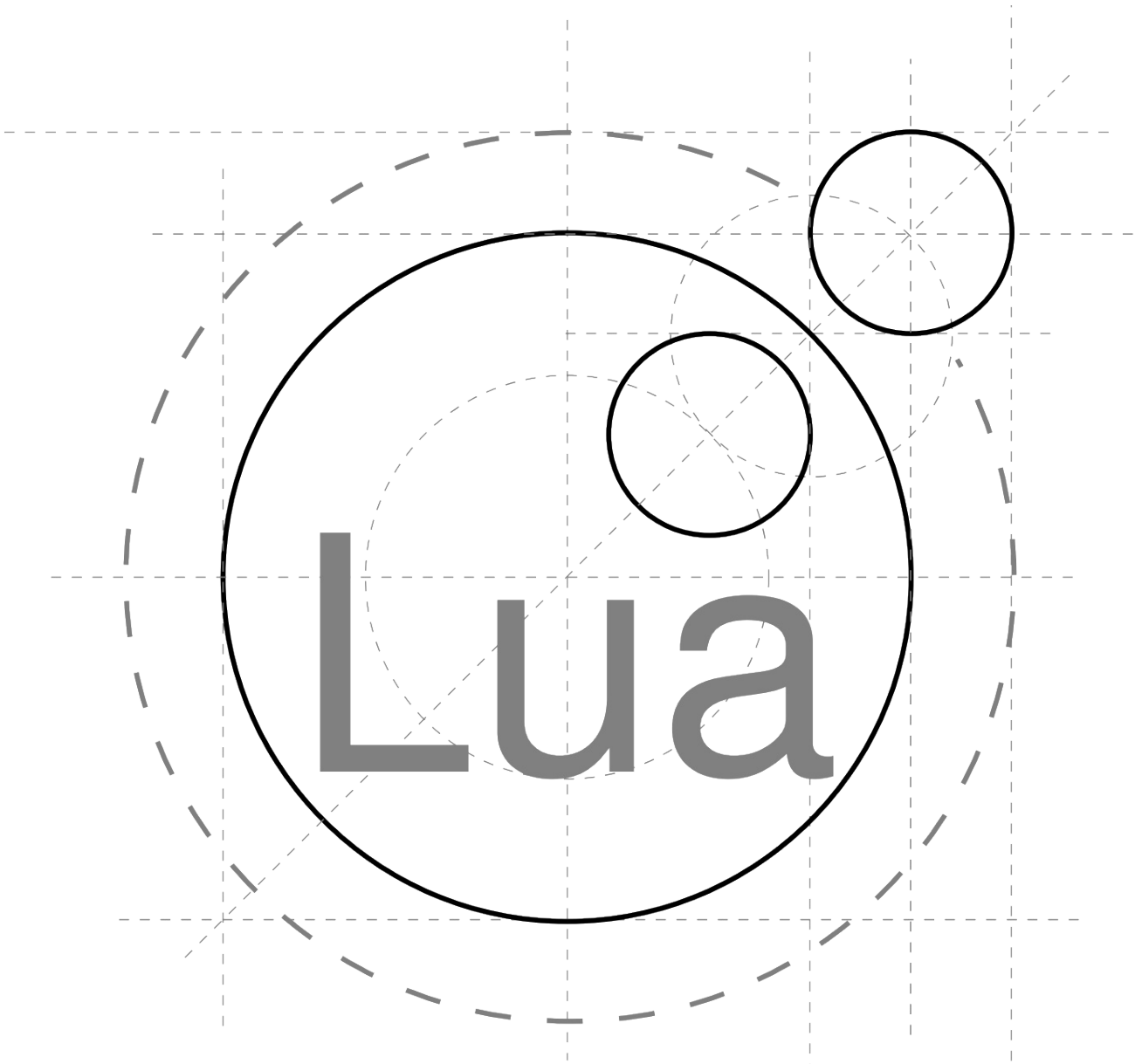
(breakpoint). Она должна предлагать как минимум две функции:

```
setbreakpoint(function, line)    --> возвращает handle
removebreakpoint(handle)
```

Точка прерывания задается функцией и строкой внутри этой функции. Когда программа сталкивается с точкой прерывания, эта библиотека должна вызывать `debug.debug`. (Подсказка: для базовой реализации используйте ловушку строки, которая проверяет, есть ли в ней точка прерывания; для улучшения быстродействия при отслеживании выполнения программы используйте ловушку вызова и включайте ловушку строки лишь тогда, когда программа выполняет заданную функцию.)

Часть IV

C API



ГЛАВА 25

Обзор C API

Lua — это *встраиваемый язык*. Это значит, что Lua — не автономный пакет, а библиотека, которую мы можем скомпоновать с другими приложениями для внедрения в них средств Lua.

Вероятно, вы задаетесь вопросом: если Lua — это не автономная программа, то как получилось, что мы до сих пор использовали ее отдельно на протяжении всей книги? Ответом на эту загадку является интерпретатор Lua (выполнимый файл `lua`). Это крошечное приложение (менее чем из пятисот строк кода), которое использует библиотеку Lua для реализации автономного интерпретатора. Автомноный интерпретатор обеспечивает взаимодействие с пользователем, принимая файлы и строки для последующей передачи их библиотеке Lua, которая и делает всю основную работу (такую как настоящее выполнение кода на Lua).

Возможность использования в качестве библиотеки для расширения приложения — это то, что делает Lua *расширяющим языком*. В то же время программа, которая использует Lua, может регистрировать новые функции в его окружении; такие функции реализуют на C (или другом языке), тем самым добавляя средства, которые не могут быть написаны непосредственно на Lua. Это то, что делает Lua *расширяемым языком*.

Эти два взгляда на Lua (как на расширяющий язык и как на расширяемый язык) соответствуют двум видам взаимодействия между C и Lua. В первом случае управление у C, а Lua — библиотека. Код C при данном виде взаимодействия мы называем *прикладным кодом*. Во втором случае управление у Lua, а C — библиотека. Здесь код C называется *библиотечным кодом*. И прикладной, и библиотечный код использует один и тот же API для

взаимодействия с Lua — так называемый C API.

C API — это набор функций, которые позволяют коду C взаимодействовать с Lua. (Примечание: Далее в этом тексте термин «функция» на самом деле означает «функция или макрос». API реализует некоторые средства в виде макросов.) Он включает в себя функции для чтения и записи глобальных переменных Lua, для вызова функций Lua, для выполнения фрагментов кода на Lua, для регистрации функций C, чтобы позже их можно было вызвать из кода Lua и т.д. Практически все, что код Lua может сделать, может быть также сделано и на C посредством C API.

C API следует принципу работы с C, который заметно отличается от принципа работы с Lua. При программировании на C мы должне заботиться о проверке типов, о восстановлении после ошибок, об ошибках выделения памяти и о некоторых источниках трудностей. Большинство функций API не проверяет правильность своих аргументов; это ваша задача — убедиться в том, что аргументы корректны перед вызовом функции. (Примечание: Вы можете компилировать код с макросом `LUA_USE_APICHECK`, который задействует некоторые проверки; эта опция особенно полезна при отладке вашего кода C. Тем не менее, некоторые ошибки, вроде недопустимых указателей, в C просто не могут быть обнаружены.) Если вы допускаете ошибки, то можете получить такие не особо информативные сообщения об ошибках, как «segmentation fault». Более того, C API делает упор на гибкость и простоту, зачастую за счет легкости использования. Типичные задачи могут потребовать нескольких вызовов API. Это может быть утомительным, но зато дает вам полный контроль над происходящим.

Целью данной главы, как ясно из ее названия, является обзор того, что вам потребуется при при использовании Lua из C. Не пытайтесь сейчас понять все детали происходящего. Мы остановимся на них позже. Однако, не забывайте, что вы всегда можете найти более подробную информацию о специфических функциях в справочном руководстве по Lua. Более того, вы можете

найти некоторые примеры использования API в самой поставке Lua. Автономный интерпретатор Lua (`lua.c`) предоставляет примеры прикладного кода, в то время как стандартные библиотеки (`lmathlib.c`, `lstrlib.c` и т.д.) снабжены примерами библиотечного кода.

С этого момента вы выступаете в роли программистов на C. Когда я говорю «вы», то имею в виду вас, как программирующего на C или пытающегося быть программистом на C.

Важным компонентом во взаимодействии между Lua и C является вездесущий виртуальный *стек*. Почти все функции API работают со значениями в этом стеке. Весь обмен данными между Lua и C происходит через этот стек. Более того, вы также можете использовать его для хранения промежуточных результатов. Он помогает разобраться с двумя принципиальными отличиями между Lua и C: первое отличие заключается в том, что в Lua есть сборка мусора, в то время как C требуется явное высвобождение памяти; второе отличие возникает из-за огромной пропасти между динамической типизацией в Lua и статической типизацией в C. Мы обсудим стек более подробно в разделе 25.2.

25.1. Первый пример

Мы начнем этот обзор с простого примера прикладного приложения: автономного интерпретатора Lua. Мы можем написать примитивный автономный интерпретатор Lua, как в листинге 25.1. Заголовочный файл `lua.h` определяет основные функции, предоставляемые Lua. Он включает в себя функции для создания нового окружения Lua, для вызова функций Lua (таких как `lua_pcall`), для чтения и записи глобальных переменных в окружении Lua, для регистрации новых функций для вызова из Lua и т.д. Все, что определено в файле `lua.h`, имеет префикс `lua_`.

Листинг 25.1. Простой автономный интерпретатор Lua

```
#include <stdio.h>
#include <string.h>
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (void) {
  char buff[256];
  int error;
  lua_State *L = luaL_newstate();                   /*
открывает Lua */
  luaL_openlibs(L);               /* открывает
стандартные библиотеки */

  while (fgets(buff, sizeof(buff), stdin) != NULL) {
    error = luaL_loadstring(L, buff) || lua_pcall(L,
0, 0, 0);
    if (error) {
      fprintf(stderr, "%s\n", lua_tostring(L, -1));
      lua_pop(L, 1);    /* выталкивает сообщение об
ошибке из стека */
    }
  }

  lua_close(L);
  return 0;
}
```

Заголовочный файл `lauxlib.h` определяет функции, предоставленные *вспомогательной библиотекой* (`auxlib`). Все их определения начинаются с `luaL_` (например, `luaL_loadstring`). Вспомогательная библиотека использует базовый API, предоставляемый `lua.h` для обеспечения абстракций более высокого уровня, в частности абстракций, используемых стандартными библиотеками. Базовый API стремится к экономичности и независимости, в то время как вспомогательная библиотека стремится к практичности для распространенных задач. Разумеется, для вашей программы тоже очень легко можно создавать другие абстракции, которые ей понадобятся. Имейте в виду, что у

вспомогательной библиотеки нет доступа ко внутренним компонентам Lua. Всю работу она выполняет посредством официального базового API. Все, что может он, может и ваша программа.

Библиотека Lua вообще не определяет никаких глобальных переменных. Она хранит все свое состояние в динамической структуре `lua_State`; все функции внутри Lua получают указатель на эту структуру в качестве аргумента. Эта реализация делает Lua реентерабельным (с возможностью повторного входа в приложение) и готовым к использованию в многонитевых приложениях.

Как следует из ее имени, функция `luaL_newstate` создает новое состояние Lua. Когда `luaL_newstate` создает новое состояние, то его окружение не содержит никаких встроенных функций, даже `print`. Чтобы сохранить размер Lua небольшим, все стандартные библиотеки представлены как отдельные пакеты, поэтому вы не обязаны их использовать, если они вам не нужны. Заголовочный файл `lualib.h` определяет функции для открытия библиотек. Функция `luaL_openlibs` открывает все стандартные библиотеки.

После создания состояния и наполнения его стандартными библиотеками пора приступить к обработке данных, вводимых пользователем. Для каждой строки, которую вводит пользователь, программа сначала вызывает `luaL_loadstring` для компиляции этого кода. Если ошибок нет, то этот вызов возвращает ноль и заталкивает получившуюся функцию в стек. (Помните, что мы обсудим этот «волшебный» стек в деталях в следующем разделе.) После этого программа вызывает `lua_pcall`, которая выталкивает функцию из стека и выполняет ее в защищенном режиме. Как и `luaL_loadstring`, функция `lua_pcall` возвращает ноль, если нет ошибок. В случае ошибки обе функции заталкивают в стек сообщение об ошибке; мы получим это сообщение при помощи `lua_tostring`, и после того, как мы его напечатаем, мы удалим его из стека при помощи `lua_pop`.

Обратите внимание, что в случае ошибки программа просто

печатает сообщение об ошибке в стандартный поток для ошибок. Настоящая обработка ошибок в C может быть довольно сложной, и то, как ее следует выполнять, зависит от типа вашего приложения. Ядро Lua никогда ничего не печатает ни в какой выходной поток; оно сообщает об ошибках посредством возвращения сообщений о них. Каждое приложение может обрабатывать эти сообщения наиболее подходящим для него способом. Чтобы данные рассуждения стали понятнее, мы на время прибегнем к простому обработчику ошибок, который печатает сообщение об ошибке, закрывает состояние Lua и производит выход из всего приложения:

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

void error (lua_State *L, const char *fmt, ...) {
  va_list argp;
  va_start(argp, fmt);
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  lua_close(L);
  exit(EXIT_FAILURE);
}
```

Позже мы еще вернемся к обработке ошибок в прикладном коде.

Поскольку вы можете компилировать Lua и как код на C, и как код на C++, `lua.h` не включает в себя этот типичный код для поправок, который бывает в некоторых других библиотеках C:

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

Если вы компилируете Lua как код C (наиболее частый случай) и

используете его в C++, вы можете включать `lua.hpp` вместо `lua.h`. Он определен следующим образом:

```
extern "C" {
#include "lua.h"
}
```

25.2. Стек

При обмене значениями между Lua и C мы сталкиваемся с двумя сложностями: несоответствие между статической и динамической системами типизации и несоответствие между автоматическим и ручным управлением памятью.

В Lua, когда мы пишем `a[k]=v`, переменные `k` и `v` могут иметь самые разные типы, даже `a` может иметь другой тип из-за применения метатаблиц. Однако, если мы хотим предложить эту операцию в C, то каждая отдельно взятая функция `settable` должна иметь фиксированный тип. Нам понадобятся десятки разных функций для этой простой операции (по одной функции на каждую комбинацию из типов трех аргументов).

Мы можем решить данную проблему, введя нечто вроде типа union из C — назовем его `lua_Value`, который может представлять все значения в Lua. Тогда мы могли бы объявить `settable` как

```
void lua_settable (lua_Value a, lua_Value k, lua_Value
v);
```

Однако, у этого решения есть два недостатка. Во-первых, может быть довольно сложно адаптировать столь комплексный тип данных под другие языки; мы разрабатывали Lua так, чтобы он легко взаимодействовал не только с C/C++, но также и с Java, Fortran, C# и другими языками. Во-вторых, Lua осуществляет сборку мусора: если мы храним таблицу Lua в переменной C, то движок Lua о таком использовании никак знать не может; он мог бы (ошибочно) предположить, что эта таблица является мусором, и

удалить ее.

Таким образом, Lua API не определяет типы подобные `lua_Value`. Вместо этого он использует абстрактный стек для обмена значениями между Lua и C. Каждый слот в этом стеке может содержать любое значение Lua. Каждый раз, когда вам нужно получить значение от Lua (например, значение глобальной переменной), вы вызываете Lua, и он заталкивает нужное значение в стек. Когда вы хотите передать значение в Lua, то вы сперва заталкиваете его в стек, и лишь затем вызываете Lua (который вытолкнет это значение из стека). Нам по-прежнему нужна одна функция для заталкивания каждого типа C в стек и другая для получения каждого типа C из стека, но зато мы избежали комбинаторный взрыв. Более того, поскольку этот стек живет внутри Lua, то сборщик мусора знает, какие значения использует C.

Практически все функции в API используют стек. Как мы уже видели в нашем первом примере, `luaL_loadstring` оставляет свой результат в стеке (либо как скомпилированный кусок, либо как сообщение об ошибке); `lua_pcall` получает функцию, которую необходимо вызвать из стека, и оставляет в нем сообщение об ошибке, если она произойдет.

Lua работает со стеком строго в соответствии с принципом LIFO (Last In, First Out — последним вошел, первым вышел). Когда вы вызываете Lua, то он меняет лишь верхнюю часть стека. У кода C больше свободы; в частности, он может просматривать любой элемент внутри стека и даже вставлять и удалять элементы на любой произвольной позиции.

Заталкивание элементов

В API содержится по одной функции заталкивания для каждого типа C, который может быть представлен в Lua: `lua_pushnil` для константы nil, `lua_pushboolean` для логических значений (целые

числа в C), `lua_pushnumber` для чисел с плавающей точкой двойной точности, `lua_pushinteger` для целых чисел со знаком, `lua_pushunsigned` для целых чисел без знака, `lua_pushlstring` для произвольных строк (указатель на `char` плюс длина) и `lua_pushstring` для строк, которые завершаются нуль-символом:

```
void lua_pushnil      (lua_State *L);
void lua_pushboolean  (lua_State *L, int bool);
void lua_pushnumber   (lua_State *L, lua_Number n);
void lua_pushinteger  (lua_State *L, lua_Integer n);
void lua_pushunsigned (lua_State *L, lua_Unsigned n);
void lua_pushlstring  (lua_State *L, const char *s,
size_t len);
void lua_pushstring   (lua_State *L, const char *s);
```

Также есть функции для заталкивания в стек функций C и значений пользовательских данных; мы обсудим их позже.

Тип `lua_Number` — это числовой тип в Lua. По умолчанию он равен double, но в некоторых дистрибутивах он может быть заменен на float или даже на long integer для адаптации под компьютеры с сильно ограниченными ресурсами. Тип `lua_Integer` — это целочисленный тип со знаком, достаточно большой, чтобы хранить в себе размер больших строк. Обычно он определен как тип `ptrdiff_t`. Тип `lua_Unsigned` (который появился в Lua 5.2) — это 32-битовый беззнаковый целочисленный тип в C; он используется библиотекой для побитовых операций и соответственными функциями.

Строки в Lua не завершаются нуль-символом; они могут содержать произвольные бинарные данные. Соответственно, их длина должна быть задана явно. Основной функцией для заталкивания строки в стек является `lua_pushlstring`, для которой требуется явно указывать длину в качестве аргумента. Для строк, завершенных нуль-символом, вы также можете использовать `lua_pushstring`, которая для вычисления длины строки использует `strlen`. Lua никогда не хранит указатели на внешние строки (или на

любой другой внешний объект, за исключением функций C, которые всегда статические). Для любой строки, которую необходимо хранить, Lua или делает копию, или повторно использует существующую. Соответственно, вы можете освободить или изменить ваш буфер, как только из этих функций вернется управление.

Когда вы заталкивате элемент в стек, то ваша обязанность — проследить, чтобы в стеке для него было достаточно места. Помните о том, что сейчас вы программист на C; Lua вас баловать не станет. Когда Lua начинает выполнение и в любой момент, когда Lua вызывает C, в стеке есть как минимум 20 свободных слотов. (Заголовочный файл `lua.h` определяет эту константу как `LUA_MINSTACK`.) Этого места более чем достаточно для большинства задач, поэтому, как правило, об этом можно даже не думать. Однако, для некоторых задач требуется больше места в стеке, в частности если у вас есть цикл, который заталкивает элементы в стек. В этих случаях вы можете вызвать функцию `lua_checkstack`, которая проверяет, достаточно ли в стеке места для ваших нужд:

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

Обращение к элементам

Для обращения к элементам в стеке API использует *индексы*. Первый помещенный в стек элемент имеет индекс 1, следующий — индекс 2, и так до самой вершины. Мы также можем обращаться к элементам стека, приняв вершину стека за отправную точку и используя отрицательные индексы. В этом случае -1 соответствует элементу на вершине стека (то есть помещенному в стек последним), -2 соответствует предыдущему элементу и т. д. Например, вызов `lua_tostring(L,-1)` возвращает значение на вершине стека как строку. Как мы увидим, в одних случаях стек удобнее индексировать с его основания (то есть используя положительные

индексы), а в других более естественно использовать отрицательные индексы.

Для проверки того, является ли элемент значением заданного типа, API предлагает семейство функций `lua_is*`, где `*` может быть любым типом Lua. Соответственно, есть функции `lua_isnumber`, `lua_isstring`, `lua_istable` и т. д. У всех этих функций один и тот же прототип:

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

В действительности `lua_isnumber` не проверяет, есть ли у значения этот конкретный тип, а проверяет, может ли значение быть преобразовано в этот тип; `lua_isstring` ведет себя аналогично: в частности, для `lua_isstring` подходит любое число.

Также существует функция `lua_type`, которая возвращает тип элемента в стеке. Каждый тип представлен константой, определенной в заголовочном файле `lua.h`: `LUA_TNIL`, `LUA_TBOOLEAN`, `LUA_TNUMBER`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA__TTHREAD`, `LUA_TUSERDATA` и `LUA_TFUNCTION`. Обычно мы используем эту функцию совместно с оператором switch. Также она оказывается полезной, когда нам нужно проверить значение на принадлежность к числам или строкам без приведения типов.

Для получения значений из стека применяются функции `lua_to*`:

```
int          lua_toboolean  (lua_State *L, int index);
const char  *lua_tolstring  (lua_State *L, int index,
size_t *len);
lua_Number   lua_tonumber   (lua_State *L, int index);
lua_Integer  lua_tointeger  (lua_State *L, int index);
lua_Unsigned lua_tounsigned (lua_State *L, int idx);
```

Функция `lua_toboolean` преобразует любое значение Lua в логическое значение C (0 или 1), следуя правилам Lua для выражений условия: nil и false ложны, все остальные значения истинны.

Допустимо вызывать любую из `lua_to*` функций, даже когда заданный элемент не обладает подходящим типом. Функция `lua_toboolean` работает для любого типа; `lua_tolstring` возвращает `NULL` для нестроковых значений. Однако, у числовых функций нет возможности сообщить о неправильном типе, поэтому в случае ошибки они просто возвращают ноль. Обычно для проверки типа вам следует вызывать `lua_isnumber`, но в Lua 5.2 ввели следующие новые функции:

```
lua_Number   lua_tonumberx   (lua_State *L, int idx,
int *isnum);
lua_Integer  lua_tointegerx  (lua_State *L, int idx,
int *isnum);
lua_Unsigned lua_tounsignedx (lua_State *L, int idx,
int *isnum);
```

Выходной параметр `isnum` возвращает булево значение, сообщающее о том, было ли значение Lua числом. (Если вам это значение не нужно, то вы можете в качестве последнего параметра передать `NULL`. Старые функции `lua_to*` теперь реализованы как макросы на основе этих функций.)

Функция `lua_tolstring` возвращает указатель на внутреннюю копию строки и хранит длину строки в позиции, заданной `len`. Вы не можете изменять эту внутреннюю копию (выставленный `const` напомнит вам об этом). Lua следит за тем, чтобы этот указатель был действителен до тех пор, пока соответствующее строковое значение находится в стеке. Когда функция C, вызванная из Lua, возвращает управление, Lua очищает ее стек; поэтому, как правило, вы никогда не должны хранить указатели на строки Lua вне функции, получившей их.

Любая строка, которую возвращает `lua_tolstring`, всегда содержит в конце дополнительный ноль, но она может содержать внутри себя и другие ноли. Настоящая длина строки возвращается через третий аргумент `len`. В частности, если предположить, что значение на вершине стека является строкой, то следующие функции

assert всегда действительны:

```
size_t l;
const char *s = lua_tolstring(L, -1, &l);    /* любая
строка Lua */
assert(s[l] == '\0');
assert(strlen(s) <= l);
```

Вы можете вызвать lua_tolstring с NULL в качестве третьего аргумента, если вам не нужна эта длина. А еще лучше воспользоваться макросом lua_tostring, который просто вызывает lua_tolstring с третьим аргументом, равным NULL.

Чтобы проиллюстрировать применение этих функций листинг 25.2 содержит полезную вспомогательную функцию, которая печатает все содержимое стека. Эта функция обходит весь стек от основания до вершины, печатая каждый элемент в соответствии с его типом. Стоки печатаются в кавычках, для чисел используется формат '%g'; для других значений (функции, таблицы и т.п.) печатается только их тип. (Функция lua_typename преобразует код типа в название типа.)

Листинг 25.2. Печать содержимого стека

```
static void stackDump (lua_State *L) {
  int i;
  int top = lua_gettop(L); /* глубина стека */
  for (i = 1; i <= top; i++) { /* повторяет для
каждого уровня */
    int t = lua_type(L, i);
    switch (t) {
      case LUA_TSTRING: { /* строки */
        printf("'%s'", lua_tostring(L, i));
        break;
      }
      case LUA_TBOOLEAN: { /* булевы значения */
        printf(lua_toboolean(L, i) ? "true" :
"false");
        break;
      }
      case LUA_TNUMBER: { /* числа */
```

```
            printf("%g", lua_tonumber(L, i));
            break;
          }
          default: { /* другие значения */
            printf("%s", lua_typename(L, t));
            break;
          }
        }
        printf(" "); /* помещает разделитель */
      }
      printf("\n"); /* конец листинга */
    }
```

Другие стековые операции

Кроме предыдущих функций, служащих для обмена данными между C и стеком, данный API также предоставляет следующие операции для общей работы со стеком:

```
int  lua_gettop    (lua_State *L);
void lua_settop    (lua_State *L, int index);
void lua_pushvalue (lua_State *L, int index);
void lua_remove    (lua_State *L, int index);
void lua_insert    (lua_State *L, int index);
void lua_replace   (lua_State *L, int index);
void lua_copy      (lua_State *L, int fromidx, int
toidx);
```

Функция `lua_gettop` возвращает число элементов в стеке, которое также является индексом верхнего элемента. Функция `lua_settop` устанавливает вершину (то есть количество элементов в стеке) на заданное значение. Если предыдущая вершина была выше новой, то функция отбрасывает с вершины эти лишние значения. В противном случае она заталкивает в стек необходимое количество nil для получения заданного размера. В частности, `lua_settop(L,0)` очищает весь стек. В функции `lua_settop` вы также можете использовать отрицательные индексы. Используя

данное средство, API предоставляет следующий макрос, который выталкивает из стека n элементов:

```
#define lua_pop(L,n) lua_settop(L, -(n) - 1)
```

Функция lua_pushvalue заталкивает в стек копию элемента с заданным индексом; lua_remove удаляет элемент с заданным индексом, сдвигая вниз все элементы поверх данной позиции, чтобы заполнить разрыв; lua_insert перемещает элемент с вершины стека в заданную позицию, сдвигая вверх все элементы над данной позицией, чтобы освободить место; lua_replace выталкивает значение с вершины стека и устанавливает его как значение элемента с заданным индексом, ничего при этом не перемещая; наконец, lua_copy копирует значение одного индекса в значение другого, не изменяя исходное значение. Обратите внимание, что следующие операции влияют лишь на пустой стек:

```
lua_settop(L, -1); /* устанавливает текущее значение
вершины стека */
lua_insert(L, -1); /* помещает элемент на вершину
стека */
lua_copy(L, x, x); /* копирует элемент на его
собственную позицию */
```

Программа в листинге 25.3 использует stackDump (определенную в листинге 25.2) для иллюстрации этих операций над стеком.

Листинг 25.3. Пример операций над стеком

```
#include <stdio.h>
#include "lua.h"
#include "lauxlib.h"

static void stackDump (lua_State *L) {
  <как в листинге 25.2>
  }

int main (void) {
```

```
  lua_State *L = luaL_newstate();

  lua_pushboolean(L, 1);
  lua_pushnumber(L, 10);
  lua_pushnil(L);
  lua_pushstring(L, "hello");

  stackDump(L);
                    /* true   10    nil    'hello' */
  lua_pushvalue(L, -4); stackDump(L);
                    /* true   10    nil    'hello'
true */
  lua_replace(L, 3); stackDump(L);
                    /* true   10    true   'hello' */
  lua_settop(L, 6); stackDump(L);
                    /* true   10    true   'hello'
nil   nil */
  lua_remove(L, -3); stackDump(L);
                    /* true   10    true    nil
nil */
  lua_settop(L, -5); stackDump(L);
                    /* true */

  lua_close(L);
  return 0;
}
```

25.3. Обработка ошибок в C API

Все структуры в Lua являются динамическими: они растут по мере необходимости и уменьшаются в размере, когда это осуществимо. Это означает, что в Lua постоянно присутствует возможность сбоя при выделении памяти. С этим может столкнуться практически каждая операция. Более того, многие операции могут вызвать и другие ошибки; например, обращение к глобальной переменной может привести к срабатыванию метаметода `__index`, который при этом может выбросить ошибку. Наконец, операции, которые выделяют память, со временем приводят к

срабатыванию сборщика мусора, который может вызвать финализаторы, которые также могут выбросить ошибки. Короче говоря, подавляющее большинство функций в Lua API может привести к ошибкам.

Вместо использования кодов ошибок для каждой операции в своем API, Lua использует исключения для уведомления об ошибках. В отличие от C++ или Java, язык C не содержит механизм обработки исключений. Чтобы обойти данное ограничение, Lua использует функцию `setjmp` из C, которая позволяет получить механизм, похожий на обработку исключений. Поэтому большинство функций API может выбросить ошибку (то есть вызвать `longjmp`) вместо возврата управления.

Когда мы пишем библиотечный код (то есть функции C, которые будут вызваны из Lua), использование `longjmp` почти так же удобно, как и использование настоящих средств обработки исключений, поскольку Lua отлавливает любую возникающую ошибку. Когда мы пишем прикладной код (то есть код C, который вызывает Lua), то мы должны обеспечить способ для перехвата подобных ошибок.

Обработка ошибок в прикладном коде

Когда ваше приложение вызывает функции из Lua API, оно подвержено ошибкам. Как мы только что обсуждали, Lua обычно сообщает об этих ошибках посредством функции `longjmp`. Однако, если нет соответствующего вызова `setjmp`, то интерпретатор не может выполнить и `longjmp`. В этом случае любая ошибка в API приводит к тому, что Lua вызывает паническую функцию (panic function), и если управление из этой функции возвращается, то происходит выход из приложения. Вы можете задать свою паническую функцию при помощи `lua_atpanic`, но такая функция мало что может сделать.

Чтобы правильно обрабатывать ошибки в вашем прикладном коде, вы должны вызывать ваш код через Lua, так как в этом случае он может установить подходящий контекст для перехвата ошибок (то есть он выполнит ваш код в контексте `setjmp`). Точно так же, как мы можем запускать код Lua в защищенном режиме при помощи `pcall`, мы можем выполнять код C посредством `lua_pcall`. Точнее, мы запаковываем код C в функцию и вызываем эту функцию через Lua, используя `lua_pcall`. (Мы подробно обсудим, как вызывать функции C из Lua в главе 27.) С такой настройкой ваш код C будет выполнен в защищенном режиме. Даже в случае ошибки выделения памяти `lua_pcall` возвращает соответствующий код ошибки, оставляя интерпретатор в рабочем состоянии.

Обработка ошибок в библиотечном коде

Lua — это *безопасный* язык. Это значит, что не зависимо от того, что вы пишите на Lua, и насколько неправильно вы это пишите, вы всегда можете понять поведение программы, не выходя за рамки самого Lua. Более того, ошибки тоже обнаруживаются и объясняются в рамках Lua. Для контраста сравните с C, где поведение многих неправильно написанных программ может быть объяснено лишь в рамках используемого оборудования (например, места ошибок в C заданы как адреса команд).

Когда вы добавляете функцию C к Lua, вы нарушаете эту безопасность. Например, такая функция, как `poke`, которая записывает произвольный байт по произвольному адресу памяти, может привести ко всем видам повреждения данных в памяти. Вы должны стремиться к тому, чтобы ваши дополнительные компоненты были безопасны для Lua и обеспечивали хорошую обработку ошибок.

Как мы ранее обсуждали, программы C должны задавать свою

обработку ошибок посредством `lua_pcall`. Тем не менее, когда вы пишете библиотечные функции для Lua, им обычно не требуется обрабатывать ошибки. Ошибки, выброшенные библиотечной функцией, будут пойманы либо при помощи `pcall` в Lua, либо при помощи `lua_pcall` в прикладном коде. Поэтому, когда функция в библиотеке C обнаруживает ошибку, она может просто вызвать `lua_error` (или, что еще лучше, — `luaL_error`, которая форматирует сообщение об ошибке и затем вызывает `lua_error`). Функция `lua_error` очищает все, что нужно очистить в Lua, и перепрыгивает обратно к защищенному вызову, с которого начиналось то выполнение, передавая при этом сообщение об ошибке.

Упражнения

Упражнение 25.1. Скомпилируйте и запустите простой автономный интерпретатор Lua (листинг 25.1).

Упражнение 25.2. Предположим, что стек пустой. Каким будет его содержимое после следующей последовательности вызовов?

```
lua_pushnumber(L, 3.5);
lua_pushstring(L, "hello");
lua_pushnil(L);
lua_pushvalue(L, -2);
lua_remove(L, 1);
lua_insert(L, -2);
```

Упражнение 25.3. Используйте простой автономный интерпретатор Lua (листинг 25.1) и функцию `stackDump` (листинг 25.2), чтобы проверить ваш ответ к предыдущему упражнению.

ГЛАВА 26

Расширение вашего приложения

Важной областью применения Lua является его использование в качестве *конфигурационного* языка. В этой главе мы покажем, как можно использовать Lua для конфигурирования программы, начав с простого примера, и развивая его для выполнения все более сложных задач.

26.1. Основы

В качестве нашей первой задачи давайте представим простой конфигурационный сценарий: у вашей программы C есть окно, и вы хотите иметь возможность задавать начальный размер окна. Ясно, что для такой простой задачи существуют и более простые решения, чем Lua, например, переменные окружения или файлы с парами имя-значение. Но даже используя простой текстовый файл, вам как-то нужно его разбирать; поэтому вы решаете использовать конфигурационный файл Lua (то есть простой текстовый файл, который является программой Lua). В простейшей форме этот текстовый файл может содержать, например, следующие строки:

```
-- задает размер окна
width = 200
height = 300
```

Теперь вы должны использовать Lua API, чтобы заставить Lua разобрать этот файл, и затем получить значения глобальных переменных `width` и `height`. Функция `load` из листинга 26.1 выполняет эту работу. Эта функция предполагает, что вы уже создали состояние Lua, следуя увиденному в предыдущей главе.

Она вызывает `luaL_loadfile` для загрузки куска из файла `fname` и затем вызывает `lua_pcall` для запуска скомпилированного куска. В случае ошибок (например, синтаксических ошибок в вашем конфигурационном файле) эти функции заталкивают сообщение об ошибке в стек и возвращают ненулевой код ошибки; затем наша программа использует `lua_tostring` с индексом -1 для получения сообщения с вершины стека. (Мы определили функцию `error` в разделе 25.1.)

Листинг 26.1. Получение пользовательской информации из конфигурационного файла

```
void load (lua_State *L, const char *fname, int *w,
int *h) {
  if (luaL_loadfile(L, fname) || lua_pcall(L, 0, 0,
0))
    error(L, "cannot run config. file: %s",
lua_tostring(L, -1));
  lua_getglobal(L, "width");
  lua_getglobal(L, "height");
  if (!lua_isnumber(L, -2))
    error(L, "'width' should be a number\n");
  if (!lua_isnumber(L, -1))
    error(L, "'height' should be a number\n");
  *w = lua_tointeger(L, -2);
  *h = lua_tointeger(L, -1);
}
```

После выполнения этого куска программе нужно получить значения глобальных переменных. Для этого она дважды вызывает `lua_getglobal`, чьим единственным параметром (кроме вездесущего `lua_state`) является имя переменной. Каждый такой вызов заталкивает соответствующее глобальное значение в стек, поэтому ширина окна будет по индексу -2, а высота по индексу -1 (на вершине). (Поскольку стек ранее был пуст, вы также можете индексировать с основания стека, то есть использовать 1 для первого значения и 2 для второго. Однако, индексируя с вершины,

вашему коду не нужно проверять, что стек пуст.) Далее наш пример использует `lua_isnumber`, чтобы проверить каждое значение на соответствие числу. Затем она вызывает `lua_tointeger` для преобразования таких значений в целые числа, назначая их на свои соответственные позиции.

Стоило ли использовать Lua для этой задачи? Как я сказал ранее, для такой простой задачи простой файл, в котором только два числа, вероятно использовать проще, чем файл Lua. Но даже в этом случае применение Lua дает некоторые преимущества. Во-первых, Lua обрабатывает за вас все синтаксические детали; ваш конфигурационный файл может даже содержать комментарии! Во-вторых, у пользователя появляется возможность выполнить с его помощью сложное конфигурирование. Например, этот скрипт может запросить у пользователя какую-то информацию или взять значение из переменной окружения для выбора подходящего размера:

```
-- конфигурационный файл
if getenv("DISPLAY") == ":0.0" then
  width = 300; height = 300
else
  width = 200; height = 200
end
```

Даже в таких простых конфигурационных сценариях трудно предвидеть, что может понадобиться пользователям; но пока скрипт определяет эти две переменные, ваша программа на C будет работать без изменений.

Окончательным доводом в пользу применения Lua будет то, что теперь с его помощью можно легко добавлять новые конфигурационные средства к вашей программе; эта легкость формирует подход, который приводит к более гибким программам.

26.2. Работа с таблицами

Давайте приспособим этот подход: теперь мы также хотим кофигурировать цвет фона для этого окна. Допустим, окончательная спефикация цвета составлена из трех чисел, каждое из которых является цветовым компонентом RGB. Обычно в C эти числа являются целыми и лежат в некотором диапазоне, например [0, 255]. В Lua, поскольку все числа являются вещественными, мы можем использовать более естественный диапазон [0, 1].

Наивным подходом было бы попросить пользователя задавать каждый компонент в отдельной глобальной переменной:

```
-- конфигурационный файл
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

У такого подхода два недостатка: он слишком громоздкий (настоящим программам могут понадобиться десятки цветов для фона окна, основного текста окна, фона меню и т. п.) и нет способа заранее определить распространенные цвета, чтобы пользователь мог потом просто написать `background=WHITE`. Во избежание этих недостатков, для представления цвета мы воспользуемся таблицей:

```
background = {r=0.30, g=0.10, b=0}
```

Применение таблиц придает вашему скрипту более понятную структуру; теперь пользователю (или приложению) станет легко предопределять цвета для дальнейшего использования в конфигурационном файле:

```
BLUE = {r=0, g=0, b=1.0}
<определения других цветов>
background = BLUE
```

Для получения этих значений в C мы можем поступить следующим образом:

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
error(L, "'background' is not a table");

red = getcolorfield(L, "r");
green = getcolorfield(L, "g");
blue = getcolorfield(L, "b");
```

Сначала мы получаем значение глобальной переменной `background` и убеждаемся в том, что это таблица, а затем используем `getcolorfield` для получения каждого компонента цвета.

Конечно, функция `getcolorfield` — это не часть API; мы должны ее определить. Мы вновь сталкиваемся с проблемой полиморфизма: может быть потенциально много версий функций `getcolorfield`, отличающихся типом ключа, типом значения, обработкой ошибок и т.д. Lua API предлагает всего одну функцию — `lua_gettable`, которая работает для всех типов. Она берет позицию таблицы в стеке, выталкивает ключ из стека и заталкивает в стек соответствующее значение. Наша закрытая `getcolorfield`, определенная в листинге 26.2, считает, что таблица находится на вершине стека, поэтому после заталкивания ключа в стек посредством `lua_pushstring` таблица будет находиться по индексу -2. Перед возвратом управления функция `getcolorfield` выталкивает из стека полученное значение, оставляя стек в том же состоянии, в котором он был перед этим вызовом.

Листинг 26.2. Отдельная реализация **`getcolorfield`**

```
#define MAX_COLOR 255

/* допустим, что таблица находится на вершине стека */
int getcolorfield (lua_State *L, const char *key) {
  int result;
  lua_pushstring(L, key); /* выталкивает key */
  lua_gettable(L, -2); /* получает background[key] */
  if (!lua_isnumber(L, -1))
    error(L, "invalid component in background color");
  result = (int)(lua_tonumber(L, -1) * MAX_COLOR);
```

```
  lua_pop(L, 1); /* удаляет число */
  return result;
}
```

Поскольку индексирование таблицы при помощи строкового ключа очень распространено, Lua 5.1 ввел специализированную версию `lua_gettable` для этого случая: `lua_getfield`. Используя эту функцию, мы можем переписать следующие две строки:

```
lua_pushstring(L, key);
lua_gettable(L, -2); /* получает background[key] */
```

как

```
lua_getfield(L, -1, key);
```

(Поскольку мы не заталкиваем строку в стек, индекс таблицы по-прежнему равен -1 в момент вызова `lua_getfield`.)

Мы расширим наш пример еще немного и введем названия цветов для пользователя. Пользователь по-прежнему сможет использовать таблицы цветов, но он также сможет использовать предопределенные названия цветов для наиболее распространенных цветов. Для реализации данной возможности нам понадобится таблица цветов в нашей программе C:

```
struct ColorTable {
  char *name;
  unsigned char red, green, blue;
} colortable[] = {
  {"WHITE", MAX_COLOR, MAX_COLOR, MAX_COLOR},
  {"RED", MAX_COLOR, 0, 0},
  {"GREEN", 0, MAX_COLOR, 0},
  {"BLUE", 0, 0, MAX_COLOR},
  <другие цвета>
  {NULL, 0, 0, 0} /* граничная метка */
};
```

Наша реализация создаст глобальные переменные с названиями цветов и проинициализирует эти переменные при помощи цветовых

таблиц. Результат такой же, как если бы пользователь добавил следующие строки в свой скрипт:

```
WHITE = {r=1.0, g=1.0, b=1.0}
RED   = {r=1.0, g=0, b=0}
<другие цвета>
```

Для задания полей таблицы мы определим вспомогательную функцию `setcolorfield`; она заталкивает в стек индекс и значение поля, а затем вызывает `lua_settable`:

```
/* предполагает, что таблица находится на вершине
стека */
void setcolorfield (lua_State *L, const char *index,
int value) {
lua_pushstring(L, index); /* key */
lua_pushnumber(L, (double)value / MAX_COLOR); /* value
*/
lua_settable(L, -3);
}
```

Подобно другим функциям API `lua_settable` работает для множества различных типов, поэтому она берет все свои операнды из стека. Она берет индекс таблицы в качестве аргумента и выталкивает ключ и значение. Функция `setcolorfield` предполагает, что перед вызовом таблица находится на вершине стека (индекс -1); после заталкивания индекса и значения в стек таблица будет находиться по индексу -3.

Lua 5.1 также ввел специализированную версию `lua_settable` для строковых ключей под названием `lua_setfield`. Используя эту новую функцию, мы можем переписать наше предыдущее определение `setcolorfield` следующим образом:

```
void setcolorfield (lua_State *L, const char *index,
int value) {
  lua_pushnumber(L, (double)value / MAX_COLOR);
  lua_setfield(L, -2, index);
}
```

Следующая функция, `setcolor`, определяет один цвет. Она создает таблицу, устанавливает соответствующие поля и присваивает эту таблицу соответствующей глобальной переменной:

```
void setcolor (lua_State *L, struct ColorTable *ct) {
  lua_newtable(L);                    /* создает
таблицу */
  setcolorfield(L, "r", ct->red);     /* table.r =
ct->r */
  setcolorfield(L, "g", ct->green);   /* table.g =
ct->g */
  setcolorfield(L, "b", ct->blue);    /* table.b =
ct->b */
  lua_setglobal(L, ct->name);         /* 'name' =
table */
}
```

Функция `lua_newtable` создает пустую таблицу и заталкивает ее в стек; вызовы `setcolorfield` задают поля этой таблицы; наконец, `lua_setglobal` выталкивает таблицу из стека и устанавливает ее как значение глобальной переменной с заданным именем.

Используя вышеуказанные функции, следующий цикл регистрирует все цвета для конфигурационного скрипта:

```
int i = 0;
while (colortable[i].name != NULL)
  setcolor(L, &colortable[i++]);
```

Помните, что приложение должно выполнить этот цикл до запуска скрипта.

Листинг 26.3. Цвета как строки или таблицы

```
lua_getglobal(L, "background");
if (lua_isstring(L, -1)) { /* значение является
строкой? */
  const char *name = lua_tostring(L, -1); /* получает
строку */
  int i; /* ищет в таблице цветов */
  for (i = 0; colortable[i].name != NULL; i++) {
    if (strcmp(colorname, colortable[i].name) == 0)
```

```
      break;
    }
  if (colortable[i].name == NULL) /* строка не найдена?
  */
    error(L, "invalid color name (%s)", colorname);
  else { /* использует colortable[i] */
    red = colortable[i].red;
    green = colortable[i].green;
    blue = colortable[i].blue;
    }
  } else if (lua_istable(L, -1)) {
    red = getcolorfield(L, "r");
    green = getcolorfield(L, "g");
    blue = getcolorfield(L, "b");
  } else
      error(L, "invalid value for 'background'");
```

Листинг 26.3 показывает другой вариант реализации именованных цветов. Вместо глобальных переменных пользователь может обозначать имена цветов при помощи строк, записывая настройки в виде `background="BLUE"`. Таким образом, `background` может быть как таблицей, так и строкой. При подобном подходе приложению не нужно что-либо делать перед запуском пользовательского скрипта. Вместо этого для получения цвета нужно выполнить немного больше работы. Когда скрипт получает значение переменной `background`, он должен проверить, является ли тип этого значения строкой, а затем отыскать эту строку в таблице цветов.

Какой вариант лучше? В программах C использование строк для обозначения опций не является хорошей практикой, поскольку компилятор не может обнаружить опечатки. Однако, в Lua сообщение об ошибке в названии цвета вероятно дойдет до того, кто пишет эту конфигурационную «программу». Различие между программистом и пользователем несколько размыто; разница между ошибкой во время компиляции и ошибкой во время выполнения не столь велика.

Со строками значение `background` могло бы быть строкой с

опечаткой; в этом случае приложение может добавить эту информацию к сообщению об ошибке. Приложение может также сравнивать строки независимо от регистра букв, так что пользователь может написать "`white`", "`WHITE`" или даже "`White`". Более того, если пользовательский скрипт небольшой, а цветов много, то было бы странно регистрировать сотни цветов (и создавать сотни таблиц и глобальных переменных), чтобы пользователь выбрал лишь некоторые из них. Со строками вы избежите данных издержек.

26.3. Вызовы функций Lua

Сильной стороной Lua является то, что конфигурационный файл может определять функции для вызова приложением. Например, вы можете написать приложение для построения графика функции и использовать Lua, чтобы задать эту функцию.

Протокол API для вызова функций прост: во-первых, вы заталкиваете функцию для вызова; во-вторых, вы заталкиваете аргументы для вызова; затем вы используете `lua_pcall` для действительного вызова функции; и, наконец, вы получаете результаты из стека.

В качестве примера допустим, что у нашего конфигурационного файла есть функция вроде этой:

```
function f (x, y)
  return (x^2 * math.sin(y)) / (1 - x)
end
```

Вы хотите на C вычислить `z=f(x,y)` для заданных `x` и `y`. При условии, что вы уже открыли библиотеку Lua и выполнили конфигурационный файл, функция `f` в листинге 26.4 инкапсулирует этот вызов.

Листинг 26.4. Вызов функции Lua из C

```
/* вызывает функцию 'f', определенную в Lua */
double f (lua_State *L, double x, double y) {
  int isnum;
  double z;

  /* заталкивает функции и аргументы */
  lua_getglobal(L, "f"); /* функция для вызова */
  lua_pushnumber(L, x);  /* заталкивает 1-ый аргумент
*/
  lua_pushnumber(L, y);  /* заталкивает 2-ой аргумент
*/

  /* производит вызов (2 аргумента, 1 результат) */
  if (lua_pcall(L, 2, 1, 0) != LUA_OK)
    error(L, "error running function 'f': %s",
  lua_tostring(L, -1));

  /* возвращает результат */
  z = lua_tonumberx(L, -1, &isnum);
  if (!isnum)
    error(L, "function 'f' must return a number");
  lua_pop(L, 1); /* выталкивает возвращенное значение
*/
  return z;
}
```

Второй и третий аргументы `lua_pcall` — это число аргументов, которые вы передаете, и число результатов, которые вы хотите получить, соответственно. Четвертый аргумент представляет собой функцию для обработки ошибок; скоро мы это обсудим. Как и в случае с присваиваниями в Lua, вызов `lua_pcall` приводит действительное число возвращенных значений к заданному вами числу, при необходимости заталкивая значения nil или отбрасывая лишние. Перед заталкиванием результатов `lua_pcall` удаляет из стека функцию и ее аргументы. Когда функция возвращает несколько значений, первое значение заталкивается первым; например, если возвращаются три значения, то первое будет по индексу -3, а последнее по индексу -1.

В случае возникновения ошибки во время своего выполнения

lua_pcall возвращает код ошибки; кроме того, она заталкивает сообщение об ошибке в стек (но по-прежнему выталкивает функцию и ее аргументы). Однако, перед заталкиванием сообщения lua_pcall вызывает функцию обработки сообщения, если она была задана. Для задания функции обработки сообщения используйте последний аргумент lua_pcall. Ноль означает, что функции обработки нет, т.е. окончательное сообщение об ошибке является при этом исходным. В противном случае этот аргумент должен быть индексом в стеке, по которому размещена функция обработки сообщения. В таких случаях обработчик должен быть помещен в стек до вызываемой функции и ее аргументов.

Для нормальных ошибок lua_pcall возвращает код ошибки LUA_ERRRUN. Два особых вида ошибок заслуживают отдельных кодов, поскольку они никогда не запускают обработчик сообщений. Первый вид — это ошибки выделения памяти. Для подобных ошибок lua_pcall всегда возвращает LUA_ERRMEM. Второй вид — это ошибки при выполнении самого обработчика сообщений. В этом случае нет никакого смысла заново вызывать обработчик сообщений, поэтому lua_pcall немедленно возвращает управление с кодом LUA_ERRERR. Lua 5.2 выделяет третий вид ошибок: когда финализатор выбрасывает ошибку, lua_pcall возвращает код LUA_ERRGCMM (ошибка в метаметоде сборщика мусора). Этот код обозначает, что ошибка не связана непосредственно с самим вызовом.

26.4. Обобщенный вызов функции

В качестве более сложного примера мы построим обертку для вызова функций Lua, используя средства vararg в C. Наша оберточная функция, давайте назовем ее call_va, принимает имя функции, которую нужно вызвать, строку, описывающую типы аргументов и результатов, затем список аргументов и, наконец,

список указателей на переменные для хранения результатов; она берет на себя все тонкости API. При помощи этой функции мы можем легко переписать наш предыдущий пример следующим образом:

```
call_va(L, "f", "dd>d", x, y, &z);
```

Строка "dd>d" означает «два аргумента типа double и один результат типа double». Этот дескриптор может использовать буквы 'd' для типа double, 'i' для целых чисел и 's' для строк; '>' отделяет аргументы от результатов. Если функция ничего не возвращает, то '>' необязателен.

Листинг 26.5 показывает реализацию функции call_va. Несмотря на ее общий вид, она идет тем же путем, что и наш первый пример: заталкивает функцию, заталкивает аргументы (листинг 26.6), производит вызов и получает результаты (листинг 26.7). Большая часть кода не нуждается в пояснении, но есть некоторые тонкости. Во-первых, ей не нужно проверять, что func является функцией; если это не так, то lua_pcall вызовет ошибку. Во-вторых, поскольку она заталкивает произвольное число аргументов, она должна проверять наличие свободного места на стеке. В-третьих, поскольку функция может вернуть строки, то call_va не может вытолкнуть результаты из стека. Это должна делать вызывающая функция после того, как она прекратит использовать получаемые время от времени строковые результаты (или после копирования их в соответственные буферы).

Листинг 26.5. Обобщенный вызов функции

```
#include <stdarg.h>

void call_va (lua_State *L, const char *func, const
char *sig, ...) {
  va_list vl;
  int narg, nres; /* число аргументов и результатов */
```

```
  va_start(vl, sig);
  lua_getglobal(L, func); /* заталкивает функцию */

  <аргументы для заталкивания (листинг 26.6)>

  nres = strlen(sig); /* число ожидаемых результатов
*/

  if (lua_pcall(L, narg, nres, 0) != 0) /* do the call
*/
    error(L, "error calling '%s': %s", func,
lua_tostring(L, -1));

  <возвращенные результаты (листинг 26.7)>

  va_end(vl);
}
```

Листинг 26.6. Выталкивание аргументов для обобщенного вызова функции

```
for (narg = 0; *sig; narg++) { /* повторяет для
каждого аргумента */

  /* проверяет пространство стека */
  luaL_checkstack(L, 1, "too many arguments");

  switch (*sig++) {

    case 'd': /* аргумент типа double */
      lua_pushnumber(L, va_arg(vl, double));
      break;

    case 'i': /* аргумент типа int */
      lua_pushinteger(L, va_arg(vl, int));
      break;

    case 's': /* аргумент типа string */
      lua_pushstring(L, va_arg(vl, char *));
      break;

    case '>': /* конец аргументов */
```

```
      goto endargs;

    default:
      error(L, "invalid option (%c)", *(sig - 1));
  }

}
endargs:
```

Листинг 26.7. Получение результатов для обобщенного вызова функции

```
nres = -nres; /* стековый индекс первого результата */
while (*sig) { /* повторяет для каждого результата */
  switch (*sig++) {

    case 'd': { /* результат типа double */
      int isnum;
      double n = lua_tonumberx(L, nres, &isnum);
      if (!isnum)
        error(L, "wrong result type");
      *va_arg(vl, double *) = n;
      break;
    }

    case 'i': { /* результат типа int */
      int isnum;
      int n = lua_tointegerx(L, nres, &isnum);
      if (!isnum)
        error(L, "wrong result type");
      *va_arg(vl, int *) = n;
      break;
    }

    case 's': { /* результат типа string */
      const char *s = lua_tostring(L, nres);
      if (s == NULL)
        error(L, "wrong result type");
      *va_arg(vl, const char **) = s;
      break;
    }

    default:
```

```
            error(L, "invalid option (%c)", *(sig - 1));
        }

    nres++;
    }
```

Упражнения

Упражнение 26.1. Напишите программу C, которая читает файл Lua, содержащий определение полностью числовой функции `f`, и строит график этой функции. (Вам не нужно делать что-нибудь этакое; программа может строить график, выводя результаты при помощи астериксов ASCII (`'*'`), как мы делали в разделе 8.1.)

Упражнение 26.2. Измените функцию `call_va` (листинг 26.5) для обработки булевых значений.

Упражнение 26.3. Допустим, есть программа, которой необходимо следить за несколькими погодными станциями. Внутри, для представления каждой станции, она использует 4-байтовую строку, и есть конфигурационный файл, который отображает каждую такую строку в URL соответствующей станции. Конфигурационный файл Lua мог бы выполнять это отображение несколькими способами:

- набор глобальных переменных, по одной для каждой станции;
- одна таблица, отображающая строковые коды в URL'ы;
- одна функция, отображающая строковые коды в URL'ы.

Обсудите плюсы и минусы каждого способа, принимая во внимание общее число станций, закономерности URL'ов (т.е. может существовать правило построения URL'ов из кодов), типы пользователей и т.д.

ГЛАВА 27

Вызываем C из Lua

Когда мы говорим, что Lua может вызывать функции C, это не значит, что Lua может вызвать любую функцию C. (Примечание: Есть пакеты, которые позволяют Lua вызывать любую функцию C, но они не переносимы и не безопасны.) Как мы видели в предыдущей главе, когда C вызывает функцию Lua, ей необходимо следовать простому протоколу для передачи аргументов и получения результатов. Аналогично, при вызове функции C из Lua она должна следовать похожему протоколу. Более того, чтобы Lua мог вызвать функцию C, мы должны зарегистрировать эту функцию, то есть должны надлежащим образом передать Lua ее адрес.

Когда Lua вызывает функцию C, Lua использует ту же разновидность стека, какую C использует для вызова функций Lua. Функция C получает свои аргументы из стека и заталкивает свои результаты в стек.

Важным понятием здесь является то, что стек не является глобальной структурой; у каждой функции есть свой собственный закрытый локальный стек. Когда Lua вызывает функцию C, первый аргумент всегда будет находиться по индексу 1 этого локального стека. Даже когда функция C вызывает код Lua, который вновь вызывает эту же (или другую) функцию C, каждый из этих вызовов видит лишь свой собственный закрытый стек с первым аргументом по индексу 1.

27.1. Функции C

В качестве первого примера давайте рассмотрим, как

реализовать упрощенную версию функции, которая возвращает синус заданного числа:

```
static int l_sin (lua_State *L) {
  double d = lua_tonumber(L, 1); /* получает аргумент
*/
  lua_pushnumber(L, sin(d)); /* заталкивает результат
*/
  return 1; /* число результатов */
}
```

Любая функция, зарегистрированная в Lua, должна иметь один и тот же прототип, определенный в файле `lua.h` как `lua_CFunction`:

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

С точки зрения C, функция C получает в качестве своего единственного аргумента состояние Lua и возвращает целое число, равное количеству значений, которое она возвращает в стек. Поэтому функции не нужно очищать стек перед заталкиванием в него своих результатов. После возвращения функцией результатов Lua автоматически сохраняет их и очищает весь ее стек.

Перед тем, как мы сможем использовать эту функцию из Lua, мы должны ее зарегистрировать. Мы творим это небольшое волшебство при помощи `lua_pushcfunction`: она получает указатель на функцию C и создает значение типа "`function`", которое представляет эту функцию внутри Lua. После регистрации функция C ведет себя внутри Lua как любая другая функция.

Чтобы на скорую руку проверить `l_sin`, нужно поместить ее код непосредственно в наш базовый интерпретатор (листинг 25.1) и добавить следующие строки прямо после вызова `luaL_openlibs`:

```
lua_pushcfunction(L, l_sin);
lua_setglobal(L, "mysin");
```

Первая строка заталкивает в стек значение типа "`function`", а вторая присваивает его значение глобальной переменной `mysin`.

После этих изменений вы можете использовать эту новую функцию `mysin` в ваших скриптах Lua. (В следующем разделе мы рассмотрим более подходящие способы компоновки новых функций C с Lua.)

В более профессионально оформленной функции вычисления синуса мы должны проверять тип ее аргумента. Здесь нам поможет вспомогательная библиотека. Функция `luaL_checknumber` проверяет, действительно ли заданный аргумент является числом: в случае ошибки она выбрасывает о ней информативное сообщение; иначе она возвращает само число. Изменения в нашей функции минимальны:

```
static int l_sin (lua_State *L) {
  double d = luaL_checknumber(L, 1);
  lua_pushnumber(L, sin(d));
  return 1; /* число результатов */
}
```

С определением выше, если вы вызовите `mysin('a')`, то получите следующее сообщение:

```
bad argument #1 to 'mysin' (number expected, got
string)
```

Обратите внимание, как `luaL_checknumber` автоматически заполняет сообщение номером аргумента (`#1`), именем функции ("`mysin`"), ожидаемым типом параметра (`number`) и настоящим типом параметра (`string`).

В качестве более сложного примера давайте напишем функцию, которая возвращает содержимое заданной директории. Lua не предоставляет эту функцию в своих стандартных библиотеках, поскольку в ANSI C нет подходящей функции для этой работы. Здесь мы будем считать, что у нас POSIX-совместимая система. Наша функция (назовем ее `dir` в Lua и `l_dir` в C) получает в качестве аргумента строку с путем к директории и возвращает массив с данными этой директории. Например, вызов `dir("/home/lua")` может вернуть таблицу `{"sre","bin","lib"}`.

В случае ошибки функция возвращает nil вместе со строкой с сообщением об ошибке. Полный код этой функции приведен в листинге 27.1. Обратите внимание на использование функции `luaL_checkstring` из вспомогательной библиотеки, которая является эквивалентом `luaL_checknumber` для строк.

(В экстремальных условиях данная реализация `l_dir` может вызвать небольшую утечку памяти. Три функции Lua, которые она вызывает, — `lua_newtable`, `lua_pushstring` и `lua_settable`, могут дать сбой из-за нехватки памяти. Если какая-либо из этих функций даст сбой, это вызовет ошибку и прервет выполнение `l_dir`, в результате чего `closedir` не будет вызвана. Как мы обсудили ранее, для большинства программ это не является большой проблемой: если у программы заканчивается память, то лучшее, что можно сделать, — это завершить ее выполнение. Тем не менее, в главе 30 мы увидим альтернативную реализацию функции для работы с директориями, в которой данная проблема устранена.)

Листинг 27.1. Функция для чтения содержимого директории

```
#include <dirent.h>
#include <errno.h>
#include <string.h>

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

static int l_dir (lua_State *L) {
  DIR *dir;
  struct dirent *entry;
  int i;
  const char *path = luaL_checkstring(L, 1);

  /* open directory */
  dir = opendir(path);
  if (dir == NULL) { /* error opening the directory?
*/
    lua_pushnil(L); /* return nil... */
```

```
    lua_pushstring(L, strerror(errno)); /* and error
message */
    return 2; /* number of results */
  }

  /* create result table */
  lua_newtable(L);
  i = 1;
  while ((entry = readdir(dir)) != NULL) {
    lua_pushnumber(L, i++); /* push key */
    lua_pushstring(L, entry->d_name); /* push value */
    lua_settable(L, -3);
  }
  closedir(dir);
  return 1; /* table is already on top */
}
```

27.2. Продолжения

При помощи `lua_pcall` и `lua_call` функция C, вызванная из Lua, может, в свою очередь, вызвать функцию Lua. Так делают некоторые функции из стандартной библиотеки: `table.sort` может вызвать функцию упорядочивания; `string.gsub` может вызвать функцию замены; `pcall` и `xpcall` вызывают функции в защищенном режиме. Если вспомнить, что главный код Lua был сам, в свою очередь, вызван из C (основной программы), то мы получаем примерно такую последовательность: C (основная программа) вызывает Lua (скрипт), который вызывает C (библиотека), который вызывает Lua (обратный вызов).

Обычно Lua обрабатывает эти последовательности вызовов без проблем; в конце концов, эта интеграция с C и является «визитной карточкой» языка. Тем не менее, есть одна ситуация, в которой подобное переплетение может вызвать затруднения: сопрограммы.

У каждой сопрограммы в Lua есть свой собственный стек, который хранит информацию об ожидающих вызовах этой сопрограммы. Точнее, стек хранит адрес возврата, параметры и

локальные переменные каждого вызова. Для вызовов функций Lua интерпретатор использует подходящую структуру данных для реализации стека — *гибкий стек* (soft stack). Однако, для вызовов функций C интерпретатор также должен использовать стек C. В конце концов, адрес возврата и локальные переменные функции C живут в стеке C.

У интерпретатора запросто может быть несколько гибких стеков, но у среды выполнения ANSI C есть лишь один внутренний стек. Поэтому сопрограммы в Lua не могут приостановить выполнение функции C: если в цепочке вызовов, начиная с `resume` и заканчивая соответственной `yield` есть функция C, то Lua не может сохранить состояние этой функции, чтобы восстановить его при следующем возобновлении. Рассмотрим следующий пример в Lua 5.1:

```
co = coroutine.wrap(function (a)
                      return pcall(function (x)

coroutine.yield(x[1])
                                        return x[3]
                                    end, a)
                    end)
print(co({10, 3, -8, 15}))
  --> false    attempt to yield across metamethod/C-
call boundary
```

Функция `pcall` — это функция C; поэтому Lua не может приостановить ее, поскольку в ANSI C нет способа приостановить функцию C и возобновить ее позже.

Lua 5.2 преодолел данные трудности при помощи *продолжений* (continuation). Lua 5.2 реализует уступку управления посредством `longjmp` тем же образом, которым он реализует ошибки. Функция `longjmp` просто отбрасывает всю информацию о функциях C в стеке C, что делает невозможным возобновление данных функций. Тем не менее, функция `foo` на C может задать продолжающую функцию `foo-c`, которая является другой функцией C, чтобы вызвать ее, когда понадобится возобновить `foo`. То есть когда интерпретатор

обнаружит, что он должен возобновить `foo`, а `longjmp` выбросила запись о `foo` из стека C, то вместо этого он вызовет `foo-c`.

Чтобы стало понятнее, давайте рассмотрим пример: реализацию `pcall`. В Lua 5.1. у этой функции был следующий код:

```
static int luaB_pcall (lua_State *L) {
  int status;
  luaL_checkany(L, 1); /* по крайней мере один
параметр */
  status = lua_pcall(L, lua_gettop(L) - 1,
LUA_MULTRET, 0);
  lua_pushboolean(L, (status == 0)); /* status */
  lua_insert(L, 1); /* status является первым
результатом */
  return lua_gettop(L); /* возвращает status + все
результаты */
}
```

Если функция, вызванная посредством `lua_pcall`, уступила управление, то возобновить `luaB_pcall` позже будет невозможно. Поэтому интерпретатор выдавал ошибку всякий раз, когда мы пытались уступить управление внутри защищенного вызова. Lua 5.2 реализует `pcall` примерно так, как показано в листинге 27.21. (Примечание: Настоящий код немного сложнее, чем показано здесь, поскольку у него есть некоторые общие части с `xpcall` и проверка на переполнение стека перед заталкиванием в него дополнительного результата в форме булевого значения.). Есть три отличия от версии Lua 5.1: во-первых, новая версия заменила вызов `lua_pcall` на вызов `lua_pcallk`; во-вторых, она объединила все, что делается после этого вызова в новую вспомогательную функцию `finishpcall`; третье отличие — это функция `pcallcont`, последний аргумент `lua_pcallk`, которая является продолжающей функцией.

Если нет никаких уступок управления, то `lua_pcallk` работает в точности как `lua_pcall`. Если есть какая-нибудь уступка управления, то все становится немного иначе. Если функция, вызванная `lua_pcall` пытается уступить управление, то Lua 5.2

вызывает ошибку, как и Lua 5.1. Однако, когда функция, вызванная `lua_pcallk`, уступает управление, то никаких ошибок нет: Lua вызывает `longjmp` и выбрасывает запись для `luaB_pcall` из стека C, но сохраняет в гибком стеке ссылку на продолжающую функцию `pcallcont`. Позже, когда интерпретатор обнаруживает, что он должен вернуться к `1иаB_pca11` (что невозможно), он вместо этого вызывает продолжающую функцию `pcallcont`.

В отличие от `luaB_pca11`, продолжающая функция `pcallcont` не может получить значение, возвращаемое `lua_pcallk`. Поэтому Lua предоставляет специальную функцию для возвращения статуса вызова: `lua_getctx`. Когда она вызвана из обычной функции Lua (что в нашем случае не происходит), `lua_getctx` возвращает `LUA_OK`. Когда она вызвана из продолжающей функции, она возвращает `lua_yield`. Продолжающая функция также может быть вызвана при некоторых ошибках; в этом случае `lua_getctx` возвращает код ошибки, который является тем самым значением, возвращаемым `lua_callk`.

Листинг 27.2. Реализация **`pcall`** с продолжениями

```
static int finishpcall (lua_State *L, int status) {
  lua_pushboolean(L, status); /* первый результат
(status) */
  lua_insert(L, 1); /* помещает первый результат в
первый слот */
  return lua_gettop(L);
}

static int pcallcont (lua_State *L) {
  int status = lua_getctx(L, NULL);
  return finishpcall(L, (status == LUA_YIELD));
}

static int luaB_pcall (lua_State *L) {
  int status;
  luaL_checkany(L, 1);
  status = lua_pcallk(L, lua_gettop(L) - 2,
LUA_MULTRET, 0,
```

```
                              0, pcallcont);
  return finishpcall(L, (status == LUA_OK));
}
```

Кроме статуса вызова, `lua_getctx` также может вернуть *контекстную информацию* (context information). Пятый параметр для `lua_pcallk` — это произвольное целое число, которое можно получить через второй параметр `lua_getctx`, который является указателем на целое число. Это целочисленное значение позволяет исходной функции передавать какую-либо произвольную информацию своему продолжению напрямую. Она может передавать дополнительную информацию через стек Lua. (Наш пример не задействует эту возможность.)

Система продолжений Lua 5.2 — это гениальный механизм для поддержки уступки управления, но это не панацея. Некоторым функциям C может понадобиться передать слишком много контекста своим продолжениям. Примеры включают в себя `table.sort`, которая использует стек C для рекурсии, и `string.gsub`, которая должна следить за захватами и буфером для своих промежуточных результатов. Хотя их возможно переписать с поддержкой уступок управления, выигрыш от этого не стоит дополнительной сложности.

27.3. Модули C

Модуль Lua — это кусок кода, который определяет некоторые функции Lua и хранит их в подходящих местах, обычно как записи в таблице. Модуль C для Lua ведет себя похожим образом. Кроме определения своих функций C, он также должен определить специальную функцию, которая исполняет роль главного куска в библиотеке Lua. Эта функция должна регистрировать все функции C из модуля и хранить их в подходящих для этого местах, обычно как записи в таблице. Подобно главному куску Lua, эта функция также должна инициализировать все, что в модуле требует

инициализации.

Lua получает функции C посредством данного процесса регистрации. Как только функция C представлена и сохранена в Lua, Lua вызывает ее через прямую ссылку на ее адрес (который мы передаем Lua, когда регистрируем эту функцию). Другими словами, Lua не зависит от имени функции, расположения пакета или правил видимости, чтобы вызвать ее, когда она зарегистрирована. Обычно модуль C имеет одну единственную открытую (`extern`) функцию, которая является функцией для открытия этой библиотеки. Все остальные функции могут быть закрытыми — объявленными в C как статические (`static`)

Когда вы расширяете Lua с помощью функций C, разработка вашего кода в качестве модуля C будет хорошей идей, даже если вы хотите зарегистрировать лишь одну функцию: рано или поздно (обычно рано) вам понадобятся и другие функции. Как обычно, вспомогательная библиотека предлагает для этого вспомогательную функцию. Макрос `luaL_newlib` берет список функций C вместе с их соответствующими именами и регистрирует их всех внутри новой таблицы. В качестве примера предположим, что мы хотим создать библиотеку с функцией `l_dir`, которую мы определили ранее. Во-первых, мы должны определить библиотечные функции:

```
static int l_dir (lua_State *L) {
  <как прежде>
}
```

Далее мы объявляем массив со всеми функциями в модуле вместе с их соответственными именами. Этот массив содержит элементы типа `luaL_Reg`, который является структурой из двух полей: имени функции (строка) и указателя на функцию.

```
static const struct luaL_Reg mylib [] = {
  {"dir", l_dir},
  {NULL, NULL} /* sentinel */
};
```

В нашем примере есть только одна функция (`l_dir`) для объявления. Последней парой в массиве всегда является `{NULL,NULL}` для обозначения его конца. Наконец, мы объявляем главную функцию, используя `luaL_newlib`:

```
int luaopen_mylib (lua_State *L) {
  luaL_newlib(L, mylib);
  return 1;
}
```

Вызов `luaL_newlib` создает новую таблицу и заполняет ее парами имя-функция из массива `mylib`. При возвращении `luaL_newlib` оставляет в стеке новую таблицу, в которой она открывала библиотеку. Функция `luaopen_mylib` затем возвращает 1, чтобы вернуть эту таблицу в Lua.

После завершения библиотеки мы должны скомпоновать ее с интерпретатором. Наиболее удобный способ добиться этого состоит в использовании средств динамической компоновки, если ваш интерпретатор Lua поддерживает эти средства. В этом случае вы должны создать динамическую библиотеку с вашим кодом (`mylib.dll` в Windows, `mylib.sob` в Linux) и поместить ее где-нибудь в пути C. После этих шагов вы можете загрузить вашу библиотеку непосредственно из Lua при помощи `require`:

```
local mylib = require "mylib"
```

Этот вызов компонует динамическую библиотеку `mylib` с Lua, находит функцию `luaopen_mylib`, регистрирует ее как функцию C и вызывает ее, открывая тем самым модуль. (Это поведение объясняет, почему `luaopen_mylib` должна иметь тот же самый прототип, что и любая другая функция C.)

Динамический компоновщик должен знать имя функции `luaopen_mylib`, чтобы найти ее. Он всегда будет искать `luaopen_` с присоединенным именем модуля. Поэтому если ваш модуль называется `mylib`, эта функция должна называться `luaopen_mylib`.

Если ваш интерпретатор не поддерживает динамическую компоновку, то вам нужно заново скомпилировать Lua с вашей новой библиотекой. Кроме этой перекомпиляции вам понадобится какой-нибудь способ сообщить интерпретатору, что он должен открывать эту библиотеку при открытии нового состояния. Простой способ это сделать — добавить `luopen_mylib` в список стандартных библиотек для открытия с помощью `luaL_openlibs` в файл `linit.c`.

Упражнения

Упражнение 27.1. Напишите на C функцию `summation`, которая вычисляет сумму на основе переменного количества числовых аргументов:

```
print(summation())                  --> 0
print(summation(2.3, 5.4))          --> 7.7
print(summation(2.3, 5.4, -34))     --> -26.3
print(summation(2.3, 5.4, {}))
  --> stdin:1: bad argument #3 to 'summation'
                (number expected, got table)
```

Упражнение 27.2. Реализуйте функцию, эквивалентную `table.pack` из стандартной библиотеки.

Упражнение 27.3. Напишите функцию, которая получает произвольное число параметров и возвращает их в обратном порядке:

```
print(reverse(1, "hello", 20))    --> 20    hello    1
```

Упражнение 27.4. Напишите функцию `foreach`, которая принимает таблицу и функцию, а затем вызывает данную функцию для каждой пары ключ-значение в этой таблице:

```
foreach({x = 10, y = 20}, print)
  --> x    10
```

```
--> y    20
```

(Подсказка: проверьте функцию `lua_next` в справочнике по Lua.)

Упражнение 27.5. Перепишите функцию `foreach` из предыдущего упражнения так, чтобы вызываемая функция могла уступать управление.

Упражнение 27.6. Создайте модуль C со всеми функциями из предыдущих упражнений.

ГЛАВА 28

Приемы написания функций C

И официальный API, и вспомогательная библиотека предоставляют несколько механизмов для помощи в написании функций C. В этой главе мы рассмотрим механизмы для обработки массивов, обработки строк и сохранения значений Lua в C.

28.1. Обработка массивов

В Lua «массив» — это всего лишь название для таблицы, используемой характерным образом. Мы можем обрабатывать массивы, используя те же функции, что мы использовали для обработки таблиц, то есть `lua_settable` и `lua_gettable`. Однако, API предоставляет особые функции для обработки массивов. Первой причиной пользоваться этими дополнительными функциями является производительность: обращение к массиву посреди внутреннего цикла алгоритма (например, для сортировки) втречается довольно часто, так что любое увеличение быстродействия этой операции может значительно повлиять на быстродействие алгоритма в целом. Другой причиной является удобство: целочисленные ключи достаточно распространены, чтобы как и строковые заслужить особое обращение.

API предоставляет две функции для работы с массивами:

```
void lua_rawgeti (lua_State *L, int index, int key);
void lua_rawseti (lua_State *L, int index, int key);
```

Описание функций `lua_rawgeti` и `lua_rawseti` несколько смущает, так как оно включает два индекса: `index` указывает, где в стеке находится таблица; `key` указывает, где в этой таблице находится

элемент. Вызов `lua_rawgeti(L,t,key)` эквивалентен следующей последовательности, когда `t` является положительным числом (иначе вы должны компенсировать появление нового элемента в стеке):

```
lua_pushnumber(L, key);
lua_rawget(L, t);
```

Вызов `lua_rawseti(L,t,key)` (опять же для положительного `t`) эквивалентен данной последовательности:

```
lua_pushnumber(L, key);
lua_insert(L, -2); /* помещает 'key' под предыдущим
значением */
lua_rawset(L, t);
```

Обратите внимание, что обе эти функции используют операции с прямым доступом. Они быстрее, и, кроме того, таблицы, используемые как массивы, редко используют метаметоды.

Листинг 28.1. Функция **`map`** на C

```
int l_map (lua_State *L) {
  int i, n;

  /* 1-ый аргумент должен быть таблицей (t) */
  luaL_checktype(L, 1, LUA_TTABLE);

  /* 2-ой аргумент должен быть функцией (f) */
  luaL_checktype(L, 2, LUA_TFUNCTION);

  n = luaL_len(L, 1); /* получает размер таблицы */

  for (i = 1; i <= n; i++) {
    lua_pushvalue(L, 2); /* push f */
    lua_rawgeti(L, 1, i); /* push t[i] */
    lua_call(L, 1, 1); /* call f(t[i]) */
    lua_rawseti(L, 1, i); /* t[i] = result */
  }

  return 0; /* без результатов */
```

```
}
```

В качестве конкретного примера использования этих функций листинг 28.1 реализует функцию отображения: она применяет заданную функцию ко всем элементам массива, заменяя каждый элемент результатом вызова. Этот пример также вводит три новые функции: `luaL_checktype`, `luaL_len` и `lua_pcall`.

Функция `luaL_checktype` (из файла `lauxlib.h`) проверяет, что заданный аргумент имеет заданный тип; в противном случае она выбрасывает ошибку.

Элементарная функция `lua_len` (не использованная в примере выше) эквивалентна операции '`#`'. Из-за метаметодов эта операция может вернуть объект любого вида, а не только числа; поэтому `lua_len` возвращает свой результат в стек. Функция `luaL_len` (использованная в примере и взятая из вспомогательной библиотеки) вызывает ошибку, если длина не является числом, в противном случае она возвращает длину как целое число C.

Функция `lua_call` выполняет незащищенный вызов. Она похожа на `lua_pcall`, но передает ошибки дальше, а не возвращает их код. Когда вы пишете главный код приложения, то вы не должны использовать `lua_call`, поскольку вам требуется ловить любые ошибки. Однако, когда вы пишете функции, использовать `lua_call` — неплохая идея; если ошибка возникнет, то мы просто оставим ее кому-нибудь, кто о ней позаботится.

28.2. Обработка строк

Когда функция C получает строковый аргумент из Lua, то существуют только два правила, которые она должна соблюдать: не выталкивать строку из стека во время обращения к ней и никогда не модифицировать эту строку.

Ситуация становится сложнее, когда функции C нужно создать

строку, чтобы вернуть ее Lua. Теперь код C должен беспокоиться о выделении-высвобождении буфера, переполнениях буфера и т.п. Тем не менее, Lua API предоставляет некоторые функции, чтобы помочь с этими задачами.

Стандартный API обеспечивает поддержку двух самых базовых строковых операций: извлечение подстроки и конкатенация строк. При извлечении подстроки помните, что базовая операция `lua_pushlstring` получает длину строки как дополнительный аргумент. Поэтому если вы хотите передать Lua подстроку строки `s`, расположенную от позиции `i` до позиции `j` (включительно), то вам всего лишь нужно сделать это:

```
lua_pushlstring(L, s + i, j - i + 1);
```

В качестве примера допустим, что вам нужна функция, которая разбивает строку по заданному разделителю (одному символу) и возвращает таблицу с подстроками. Например, вызов `split("hi:ho:there", ":")` должен вернуть таблицу `{"hi", "ho", "there"}`. Листинг 28.2 показывает простую реализацию этой функции. Ей не нужны дополнительные буферы и она не накладывает никаких ограничений на размер строк, которые она может обрабатывать. Обо всех буферах заботится Lua.

Листинг 28.2. Разбиение строки

```
static int l_split (lua_State *L) {
  const char *s = luaL_checkstring(L, 1);   /* субъект
*/
  const char *sep = luaL_checkstring(L, 2); /*
разделитель */
  const char *e;
  int i = 1;

  lua_newtable(L);                          /*
итоговая таблица */

  /* повторяет для каждого разделителя */
  while ((e = strchr(s, *sep)) != NULL) {
```

```
    lua_pushlstring(L, s, e-s);                    /*
заталкивает подстроку */
    lua_rawseti(L, -2, i++);                       /*
вставляет ее в таблицу */
    s = e + 1;                                     /*
пропускает разделитель */
  }

  /* insert last substring */
  lua_pushstring(L, s);
  lua_rawseti(L, -2, i);

  return 1;                                        /*
возвращает эту таблицу */
}
```

Для конкатенации строк Lua предоставляет в свом API характерную функцию под названием `lua_concat`. Она эквивалентна операции конкатенации `..` в Lua; она преобразует числа в строки и при необходимости вызывает метаметоды. Более того, она может за раз объединить более двух строк. Вызов `lua_concat(L, n)` соединит (и вытолкнет) `n` значений на вершине стека и заталкнет в нее результат.

Другой полезной функцией является `lua_pushfstring`:

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

Она несколько похожа на функцию `sprintf` из C тем, что создает строку по форматирующей строке и некоторым дополнительным аргументам. Однако, в отличие от `sprintf`, вам не нужно предоставлять буфер. Lua динамически создаст строку для вас настолько большой, насколько это необходимо. Эта функция заталкивает получившуюся строку в стек и возвращает указатель на нее. Вам не нужно беспокоиться о переполнении буфера.

На данный момент эта функция поддерживает только директивы для вставки следующих элементов (Примечание: Директива `%p` для

указателей появились в Lua 5.2.):

```
%s строка, завершенная нуль-символом
%d целое число (integer)
%f число Lua (double)
%p указатель
%c целое число (integer) как символ
%% символ '%'
```

Никакие модификаторы, такие как ширина или точность, ей не поддерживаются.

И `lua_concat`, и `lua_pushfstring` полезны, когда мы хотим соединить только несколько строк. Однако, если нам нужно соединить много строк (или символов) вместе, то делать это по одному за раз может быть довольно неэффективно, как мы видели в разделе 11.6. В этом случае мы можем использовать буферные средства, предоставленные вспомогательной библиотекой.

При более простом варианте применения буферные средства работают с двумя функциями: одна дает вам буфер любого размера, куда вы можете писать свою строку; другая преобразует буфер в строку Lua (Примечание: Эти две функции появились в Lua 5.2). Листинг 28.3 демонстрирует использование этих функций при помощи реализации функции `string.upper` прямо из исходного файла `lstrlib.c`. Первым шагом для использования буфера из вспомогательной библиотеки является объявление переменной с типом `luaL_Buffer`. Следующим шагом является вызов `luaL_buffinitsize` для получения указателя на буфер с заданным размером; затем вы можете свободно использовать этот буфер для создания своей строки. Последний шаг — это вызов `luaL_pushresultsize` для преобразования содержимого буфера в новую строку Lua на вершине стека. Размер в этом втором вызове — это окончательный размер строки. (Часто, как в нашем примере, этот размер равен размеру буфера, но он может быть меньше. Если вы не знаете точный размер получающейся строки, но знаете, что ее

размер ограничен, то вы можете обезопасить себя, выделив буфер большего размера.)

Листинг 28.3. Функция **`string.upper`**

```
static int str_upper (lua_State *L) {
size_t l;
size_t i;
luaL_Buffer b;
const char *s = luaL_checklstring(L, 1, &l);
char *p = luaL_buffinitsize(L, &b, l);
for (i = 0; i < l; i++)
  p[i] = toupper(uchar(s[i]));
luaL_pushresultsize(&b, l);
return 1;
}
```

Обратите внимание, что функция `luaL_pushresultsize` не получает состояние Lua в качестве своего первого аргумента. После инициализации буфер хранит ссылку на состояние, поэтому нам не нужно передавать его при вызове остальных функций для работы с буферами.

Мы также можем использовать эти буферы не зная максимальной длины получаемой строки. Листинг 28.4 показывает упрощенную реализацию функции `table.concat`. В этой функции мы сперва вызываем `luaL_buffinit` для инициализации буфера. Затем мы добавляем в буфер элементы один за другим, в этом случае используя функцию `luaL_addvalue`. Наконец, `luaL_pushresult` освобождает буфер и помещает итоговую строку на вершину стека.

Листинг 28.4. Упрощенная реализация **`table.concat`**

```
static int tconcat (lua_State *L) {
  luaL_Buffer b;
  int i, n;
  luaL_checktype(L, 1, LUA_TTABLE);
  n = luaL_len(L, 1);
```

```
  luaL_buffinit(L, &b);
  for (i = 1; i <= n; i++) {
    lua_rawgeti(L, 1, i); /* get string from table */
    luaL_addvalue(b); /* add it to the buffer */
  }
  luaL_pushresult(&b);
  return 1;
}
```

Вспомогательная библиотека предоставляет несколько функций для добавления значений к буферу: функция `luaL_addvalue` добавляет строку Lua, которая находится на вершине стека; функция `luaL_addlstring` добавляет строки с заданной длиной; функция `luaL_addstring` добавляет строку, завершенную нуль-символом, и функция `luaL_addchar` добавляет одиночные символы. Эти функции имеют следующие прототипы:

```
void luaL_buffinit   (lua_State *L, luaL_Buffer *B);
void luaL_addvalue   (luaL_Buffer *B);
void luaL_addlstring (luaL_Buffer *B, const char *s,
size_t l);
void luaL_addstring  (luaL_Buffer *B, const char *s);
void luaL_addchar    (luaL_Buffer *B, char c);
void luaL_pushresult (luaL_Buffer *B);
```

Когда вы используете буфер библиотеки `auxlib`, вам нужно учитывать один момент. После инициализации буфера он хранит некоторые промежуточные результаты в стеке Lua. Поэтому вы не можете предполагать, что вершина стека останется там, где она была перед тем, как вы начали использовать буфер. Более того, хотя вы можете использовать стек для других задач во время пользования буфером, счетчик заталкиваний-выталкиваний при каждом пользовании должен быть сбалансирован всякий раз, когда вы обращаетесь к буферу. Исключением из этого правила является функция `luaL_addvalue`, которая предполагает, что строка, которую надо добавить к буферу, была помещена на вершину стека.

28.3. Хранение состояния в функциях C

Часто функциям C нужно хранить какие-нибудь нелокальные данные, то есть данные, которые переживут вызвавшую их функцию. В C мы обычно используем глобальные (`extern`) или статические (`static`) переменные для этой цели. Однако, когда вы пишете библиотечные функции для Lua, то использование глобальных или статических переменных — не очень хороший подход. Во-первых, вы не можете хранить произвольное значение Lua в переменной C. Во-вторых, библиотека, которая использует такие переменные, не будет корректно работать с несколькими состояниями Lua.

У функции Lua есть два основных места хранения нелокальных данных: глобальные переменные и нелокальные переменные. C API также предоставляет два основных места для хранения нелокальных данных: реестр и верхние значения.

Реестр — это глобальная таблица, к которой может обратиться только код C.(Примечание: На самом деле к реестру можно обратиться и из Lua при помощи функции из отладочной библиотеки `debug.getregistry`.) Обычно реестр используется для хранения данных, которые будут использоваться сразу несколькими модулями. Если вам нужно сохранять данные только для вашего модуля или функции, то вы должны использовать верхние значения.

Реестр

Реестр всегда расположен по *псевдоиндексу*, чье значение определяется `LUA_REGISTRYINDEX`. Псевдоиндекс похож на индекс в стеке, за исключением того, что связанные с ним значения находятся не в стеке. Большинство функций в Lua API, которые принимают индексы в качестве аргументов, также принимают и

псевдоиндексы — за исключением тех функций, которые управляют стеком, например, `lua_remove` и `lua_insert`. Например, чтобы получить значение, хранящееся в реесте с ключом "`Key`", мы можем использовать следующий вызов:

```
lua_getfield(L, LUA_REGISTRYINDEX, "Key");
```

Реестр — это обычная таблица Lua. Соответственно, вы можете индексировать ее любым значением Lua, кроме nil. Однако, поскольку у всех модулей C один и тот же реестр, во избежание конфликтов вы должны с осторожностью выбирать значения для использования в качестве ключей. Строковые ключи особенно удобны, когда вы хотите разрешить другим независимым библиотекам обращаться к вашим данным, поскольку все, что им нужно знать, — это имя ключа. Для таких ключей не существует полностью безопасного метода выбора ключей, но есть некоторые хорошие правила, например, не использовать распространенные имена и начинать ваши имена с имени библиотеки или чего-то вроде этого. (Префиксы вроде `lua` или `lualib` не рекомендуются.)

Вы никогда не должны использовать числа в качестве ключей реестра, поскольку подобные ключи зарезервированы для *системы ссылок* (reference system). Эта система состоит из пары функций во вспомогательной библиотеке, которые позволяют вам хранить значения в таблице, не беспокоясь о создании уникальных имен. Функция `luaL_ref` создает новые ссылки:

```
int r = luaL_ref(L, LUA_REGISTRYINDEX);
```

Данный вызов выталкивает значение из стека, сохраняет его в таблице с новым целочисленным ключом и возвращает этот ключ. Такой ключ мы называем *ссылкой* (reference).

Как следует из названия, мы пользуемся ссылками в основном тогда, когда нам нужно хранить ссылку на значение Lua внутри структуры C. Как мы уже видели, мы никогда не должны хранить указатели на строки Lua вне функции C, которая их возвратила.

Более того, Lua даже не предлагает указатели на другие объекты, такие как таблицы или функции. Поэтому мы не можем ссылаться на объекты Lua через указатели. Вместо этого, когда нам требуются такие указатели, мы создаем ссылки и храним их в C.

Чтобы затолкать значение, связанное с ссылкой `r`, в стек, мы просто пишем следующее:

```
lua_rawgeti(L, LUA_REGISTRYINDEX, r);
```

Наконец, чтобы высвободить и значение, и ссылку, мы вызываем `luaL_unref`:

```
luaL_unref(L, LUA_REGISTRYINDEX, r);
```

После этого вызова новый вызов `luaL_ref` может снова вернуть эту ссылку.

Система ссылок трактует nil как особый случай. Каждый раз, когда мы вызываем `luaL_ref` для значения nil, вместо создания новой ссылки она возвращает ссылку на константу `lua_refnil`. Следующий вызов ничего не делает:

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

А этот вызов заталкивает nil, как и ожидалось:

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

Система ссылок также определяет константу `LUA_NOREF`, которая является целым числом, отличным от любой действующей ссылки. Она удобна, чтобы помечать ссылки как недействительные.

Другим безопасным методом создания ключей в реестре является использование в качестве ключа адреса статической переменной в вашем коде: компоновщик позаботится о том, чтобы этот адрес будет уникальным среди всех библиотек. Для использования данного варианта вам понадобится функция `lua_pushlightuserdata`, которая заталкивает в стек Lua значение,

представляющее указатель C. Следующий код показывает, как хранить и возвращать строку из реестра при помощи этого метода:

```
/* переменная с уникальным адресом */
static char Key = 'k';

/* хранит строку */
lua_pushlightuserdata(L, (void *)&Key); /* заталкивает
адресс */
lua_pushstring(L, myStr); /* заталкивает значение */
lua_settable(L, LUA_REGISTRYINDEX); /* registry[&Key]
= myStr */

/* возвращает строку */
lua_pushlightuserdata(L, (void *)&Key); /* заталкивает
адрес */
lua_gettable(L, LUA_REGISTRYINDEX); /* возвращает
значение */
myStr = lua_tostring(L, -1); /* преобразует его в
строку */
```

Мы обсудим облегченные пользовательские данные (light userdata) более подробно в разделе 29.5.

Чтобы упростить использование адресов переменных в качестве уникальных ключей, Lua 5.2 вводит две новые функции: `lua_rawgetp` и `lua_rawsetp`. Они похожи на `lua_rawgeti` и `lua_rawseti`, но вместо целых чисел они используют как ключи указатели C (переведенные в облегченные пользовательские данные). С ними мы можем переписать предыдущий код следующим образом:

```
static char Key = 'k';

/* хранит строку */
lua_pushstring(L, myStr);
lua_rawsetp(L, LUA_REGISTRYINDEX, (void *)&Key);

/* возвращает строку */
lua_rawgetp(L, LUA_REGISTRYINDEX, (void *)&Key);
myStr = lua_tostring(L, -1);
```

Обе функции используют прямой доступ. Так как у реестра нет метатаблицы, прямой доступ ведет себя так же, как и обычный доступ, но немного эффективнее.

Верхние значения

В то время как реестр предлагает глобальные переменные, механизм *верхних значений* (upvalue) реализует аналог статических переменных в C, которые видны только внутри конкретной функции. Каждый раз, когда вы создаете новую функцию C в Lua, вы можете связать с ней любое количество верхних значений; каждое верхнее значение может хранить одно значение Lua. Потом при вызове функции она получает свободный доступ к любому из ее верхних значений, используя псевдоиндексы.

Мы называем эту связь функции C со своими верхними значениями *замыканием* (closure). Замыкание C примерно соответствует замыканию Lua. В частности, вы можете создавать разные замыкания, используя один и тот же код функции, но с разными верхними значениями.

В качестве простого примера давайте напишем функцию `newCounter` на C. (Примечание: Мы определили эту же функцию на Lua в разделе 6.1.) Эта функция является фабрикой: она возвращает новую функцию `counter` при каждом своем вызове. Хотя у всех этих функций один и тот же код C, каждая из них хранит свой собственный счетчик. Эта функция-фабрика выглядит следующим образом;

```
static int counter (lua_State *L); /* предваряющее
объявление */

  int newCounter (lua_State *L) {
  lua_pushinteger(L, 0);
  lua_pushcclosure(L, &counter, 1);
  return 1;
}
```

Главной функцией здесь является `lua_pushcclosure`, которая создает новое замыкание. Ее вторым аргументом является базовая функция (в примере это `counter`), а третьим — число верхних значений (в примере это 1). Перед созданием нового замыкания мы должны затолкать в стек начальные значения для их верхних значений. В нашем примере мы заталкиваем 0 как начальное значение для единственного верхнего значения. Как и ожидалось, `lua_pushcclosure` оставляет новое замыкание в стеке, поэтому замыкание уже готово к возвращению как результат `newCounter`.

Теперь давайте рассмотрим определение `counter`:

```
static int counter (lua_State *L) {
  int val = lua_tointeger(L, lua_upvalueindex(1));
  lua_pushinteger(L, ++val);           /* новое
значение */
  lua_pushvalue(L, -1);                /* дублирует
его */
  lua_replace(L, lua_upvalueindex(1)); /* обновляет
верхнее значение */
  return 1;                            /* возвращает
новое значение */
}
```

Здесь ключевым элементом является макрос `lua_upvalueindex`, который производит псевдоиндекс верхнего значения. В частности, выражение `lua_upvalueindex(1)` возвращает псевдоиндекс первого верхнего значения выполняющейся функции. Опять же, этот псевдоиндекс выглядит как любой другой индекс стека, за исключением того, что он не живет в стеке. Поэтому вызов `lua_tointeger` возвращает текущее значение первого (и единственного) верхнего значения как число. Затем функция `counter` заталкивает в стек новое значение `++val`, делает его копию и использует одну из копий, чтобы заменить значение верхнего значения. Наконец, она возвращает другую копию как свое возвращаемое значение.

В качестве более сложного примера мы реализуем при помощи

верхних значений кортежи. *Кортеж* (tuple) — это разновидность постоянной записи с анонимными полями; вы можете получить конкретное поле по числовому индексу или можете получить все поля разом. В нашей реализации мы представим кортежи как функции, которые запоминают свои значения в своих верхних значениях. При вызове с числовым аргументом эта функция вернет то конкретное поле. При вызове без аргументов она вернет все свои поля. Следующий код иллюстрирует использование кортежей:

```
x = tuple.new(10, "hi", {}, 3)
print(x(1))    --> 10
print(x(2))    --> hi
print(x())     --> 10   hi   table: 0x8087878   3
```

В языке C мы представляем все кортежи при помощи одной и той же функции `t_tuple`, продемонстрированной в листинге 28.5. Поскольку мы можем вызвать кортеж как с числовым аргументом, так и вообще без аргументов, функция `t_tuple` использует `luaL_optint` для получения необязательного аргумента. Функция `luaL_optint` похожа на `luaL_checkint`, но она не жалуется на отсутствие аргумента — она возвращает заданное значение по умолчанию (в примере это 0).

Листинг 28.5. Реализация кортежей

```
int t_tuple (lua_State *L) {
  int op = luaL_optint(L, 1, 0);
  if (op == 0) { /* нет аргументов? */
    int i;
    /* заталкивает каждое допустимое верхнее значение
в стек */
    for (i = 1; !lua_isnone(L, lua_upvalueindex(i));
i++)
      lua_pushvalue(L, lua_upvalueindex(i));
    return i - 1; /* число значений в стеке */
  }
  else { /* получает поле 'op' */
    luaL_argcheck(L, 0 < op, 1, "index out of range");
    if (lua_isnone(L, lua_upvalueindex(op)))
```

```
      return 0; /* такого поля нет */
    lua_pushvalue(L, lua_upvalueindex(op));
    return 1;
  }
}

int t_new (lua_State *L) {
  lua_pushcclosure(L, t_tuple, lua_gettop(L));
  return 1;
}

static const struct luaL_Reg tuplelib [] = {
  {"new", t_new},
  {NULL, NULL}
};

int luaopen_tuple (lua_State *L) {
  luaL_newlib(L, tuplelib);
  return 1;
}
```

Когда мы индексируем несуществующее верхнее значение, то результатом является псевдозначение типа `LUA_TNONE`. (Когда мы обращаемся к индексу стека, находящемуся над вершиной стека, то мы также получаем псевдозначение типа `LUA_TNONE`.) Поэтому наша функция `t_tupie` использует `lua_isnone` для проверки, что у нее есть заданное верхнее значение. Однако, мы никогда не должны вызывать `lua_upvalueindex` с отрицательным индексом, поэтому нам следует проверять данное условие, когда пользователь передает индекс. Функция `luaL_argcheck` проверяет любое заданное условие, вызывая ошибку, если потребуется.

Функция для создания кортежей, `t_new` (тоже в листинге 28.5), тривиальна: поскольку все ее аргументы уже в стеке, ей всего лишь нужно вызвать `lua_pushcclosure` для создания замыкания из `t_tuple` со всеми своими аргументами в качестве верхних значений. Наконец, массив `tuplelib` и функция `luaopen_tuple` (тоже в листинге 28.5) являются стандартным кодом для создания библиотеки `tuple` с единственной функцией `new`.

Общие верхние значения

Зачастую нам нужно предоставить всем функциям библиотеки общий доступ к нескольким значениям или переменным. Хотя мы можем использовать реестр для этой цели, мы также можем использовать верхние значения.

В отличие от замыканий Lua, замыкания C не могут иметь общие верхние значения. У каждого замыкания имеются свои собственные независимые верхние значения. Однако, мы можем настроить верхние значения разных функций, чтобы они ссылались на общую таблицу, тогда эта таблица становится общим окружением, в котором у всех этих функций будет совместный доступ к данным.

В Lua 5.2 предлагает функцию, которая облегчает задачу совместного использования верхнего значения между всеми функциями библиотеки. Мы открывали библиотеки C при помощи `luaL_newlib`. Lua реализует эту функцию в качестве следующего макроса:

```
#define luaL_newlib(L,l) \
     (luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))
```

Макрос `luaL_newlibtable` просто создает таблицу для этой библиотеки. (Мы также могли бы использовать `lua_newtable`, но этот макрос использует `lua_createtable` для создания таблицы заранее определенного размера, оптимального для количества функций в данной библиотеке.) Функция `luaL_setfuncs` добавляет функции из списка `l` в эту новую таблицу, которая находится на вершине стека.

Здесь нас интересует третий параметр функции `luaL_setfuncs`. Он сообщает, сколько верхних значений будут иметь новые функции в библиотеке. Начальные значения для этих верхних значений должны находиться в стеке, как и в случае с `lua_pushcclosure`.

Таким образом, для создания библиотеки, где все функции будут совместно использовать общую таблицу как свое единственное верхнее значение, мы можем использовать следующий код:

```
/* создает библиотечную таблицу ('lib' — это ее список
функций) */
luaL_newlibtable(L, lib);
/* создает общее верхнее значение */
lua_newtable(L);
/* добавляет функции к списку 'lib' для новой
библиотеки,
   совместно используя прежнюю таблицу как верхнее
значение */
luaL_setfuncs(L, lib, 1);
```

Последний вызов также удаляет общую таблицу из стека, оставляя там только новую библиотеку.

Упражнения

Упражнение 28.1. Реализуйте на C функцию фильтрации (`filter`). Она должна получать список и предикат, а затем возвращать новый список со всеми элементами из заданного списка, которые удовлетворяют заданному предикату:

```
t = filter({1, 3, 20, -4, 5}, function (x) return x <
5 end)
  -- t = {1, 3, -4}
```

(Предикат — это всего лишь функция, которая проверяет какое-либо условие и возвращает логическое значение.)

Упражнение 28.2. Измените функцию `l_split` (из листинга 28.2) так, чтобы она могла работать со строками, содержащими нуль-символы. (Кроме прочих изменений, она также должна использовать `memchr` вместо `strchr`.)

Упражнение 28.3. Заново реализуйте функцию `transliterate` (упражнение 21.3) на C.

Упражнение 28.4. Реализуйте библиотеку с измененной функцией `transliterate` так, чтобы таблица транслитерации не передавалась как аргумент, а хранилась самой библиотекой. Ваша библиотека должна предоставить следующие функции:

```
lib.settrans (table)    -- устанавливает таблицу
транслитерации
lib.gettrans ()         -- получает таблицу
транслитерации
lib.tranliterate(s)     -- транслитерирует 's' в
соответствии с
                           текущей таблицей
```

Используйте реестр для хранения таблицы транслитерации.

Упражнение 28.5. Повторите предыдущее упражнение, используя для хранения таблицы транслитерации верхнее значение.

Упражнение 28.6. Считаете ли вы хорошим стилем разработки хранить таблицу транслитерации как часть состояния библиотеки, а не передавать ее как параметр в `transliterate`?

ГЛАВА 29

Задаваемые пользователем типы в C

В предыдущей главе мы увидели, как расширить Lua при помощи новых функций, написанных на C. Теперь мы увидим, как расширить Lua при помощи новых типов, написанных на C. Мы начнем с небольшого примера; на протяжении всей главы мы будем расширять его при помощи метаметодов и прочих вкусняшек.

Наш пример для запуска довольно прост: массив логических значений. Такая простая структура выбрана потому, что ей не требуются какие-то сложные алгоритмы, поэтому мы сможем сосредоточиться на аспектах API. Тем не менее, этот пример полезен сам по себе. Разумеется, в Lua мы можем использовать таблицы для реализации массивов логических значений. Но реализация на C, где мы храним каждую запись в одном единственном бите, использует менее 3% от памяти, используемой для таблицы.

Для нашей реализации нам понадобятся следующие определения:

```
#include <limits.h>

#define BITS_PER_WORD (CHAR_BIT*sizeof(unsigned int))
#define I_WORD(i)     ((unsigned int)(i) /
BITS_PER_WORD)
#define I_BIT(i)      (1 << ((unsigned int)(i) %
BITS_PER_WORD))
```

`BITS_PER_WORD` — это количество бит в беззнаковом целом числе. Макрос `I_WORD` вычисляет слово, которое хранит бит по заданному индексу, а макрос `I_BIT` вычисляет битовую маску для обращения к соответственному биту данного слова.

Мы будем представлять наши массивы при помощи следующей

структуры:

```
typedef struct NumArray {
  int size;
  unsigned int values[1]; /* переменная часть */
} NumArray;
```

Мы объявляем массив `values` с размером в 1 лишь в качестве заполнителя, поскольку стандарт C 89 не допускает массивы с размером 0; мы выставим его настоящий размер, когда выделим для него память. Следующее выражение вычисляет итоговый размер массива с `n` элементами:

```
sizeof(NumArray) + I_WORD(n - 1)*sizeof(unsigned int)
```

(Мы вычли единицу из `n`, поскольку начальная структура уже содержит место под один элемент.)

29.1. Пользовательские данные (`userdata`)

Наша первая задача — понять, как представить структуру `NumArray` в Lua. Специально для этого Lua предоставляет базовый тип: *пользовательские данные* (userdata). Данный тип соответствует области необрабатываемой памяти в Lua без каких-либо встроенных для нее операций, в которой мы можем хранить что-угодно.

Функция `lua_newuserdata` выделяет блок памяти заданного размера, заталкивает соответственные пользовательские данные в стек и возвращает адрес этого блока:

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

Если по какой-то причине вам нужно выделить память иным способом, то можно очень легко создать пользовательские данные размером с указатель и хранить там указатель на настоящий блок памяти. Мы увидим примеры данной техники в главе 30.

С помощью `lua_newuserdata` функция для создания новых

массивов логических значений выглядит следующим образом:

```
static int newarray (lua_State *L) {
  int i;
  size_t nbytes;
  NumArray *a;

  int n = luaL_checkint(L, 1);
  luaL_argcheck(L, n >= 1, 1, "invalid size");
  nbytes = sizeof(NumArray) + I_WORD(n -
1)*sizeof(unsigned int);
  a = (NumArray *)lua_newuserdata(L, nbytes);

  a->size = n;
  for (i = 0; i <= I_WORD(n - 1); i++)
    a->values[i] = 0; /* инициализирует массив */

  return 1; /* новые пользовательские данные уже в
стеке */
}
```

(Макрос `luaL_checkint` — это лишь приведение типа над `luaL_checkinteger`.) Как только функция `newarray` зарегистрирована в Lua, мы можем создавать новые массивы при помощи операторов наподобие `a=array.new(1000)`.

Для сохранения записи мы будем использовать вызов вроде `array.set(a,index,value)`. Позже мы увидим, как использовать метатаблицы для поддержки более удобного синтаксиса `a[index]=value`. В обоих формах записи в основе лежит одна и та же функция. Она предполагает, что индексы начинаются с 1, как это принято в Lua:

```
static int setarray (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  int index = luaL_checkint(L, 2) - 1;

  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  luaL_argcheck(L, 0 <= index && index < a->size, 2,
  "index out of range");
  luaL_checkany(L, 3);
```

```
  if (lua_toboolean(L, 3))
    a->values[I_WORD(index)] |= I_BIT(index); /* set
bit */
  else
    a->values[I_WORD(index)] &= ~I_BIT(index); /*
reset bit */
  return 0;
}
```

Поскольку Lua в качестве логического принимает любое значение, для третьего параметра мы используем `luaL_checkany`: она лишь проверяет, что для данного параметра есть значение (не важно, какое). Если мы вызовем `setarray` с некорректными аргументами, то получим подробные сообщения об ошибках:

```
array.set(0, 11, 0)
  --> stdin:1: bad argument #1 to 'set' ('array'
expected)
array.set(a, 1)
  --> stdin:1: bad argument #3 to 'set' (value
expected)
```

Следующая функция возвращает запись:

```
static int getarray (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  int index = luaL_checkint(L, 2) - 1;

  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  luaL_argcheck(L, 0 <= index && index < a->size, 2,
  "index out of range");

  lua_pushboolean(L, a->values[I_WORD(index)] &
I_BIT(index));
  return 1;
}
```

Мы определим другую функцию для возвращения размера массива:

```
static int getsize (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
```

```
  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  lua_pushinteger(L, a->size);
  return 1;
}
```

Наконец, нам нужно немного дополнительного кода для инициализации нашей библиотеки:

```
static const struct luaL_Reg arraylib [] = {
  {"new", newarray},
  {"set", setarray},
  {"get", getarray},
  {"size", getsize},
  {NULL, NULL}
};

int luaopen_array (lua_State *L) {
  luaL_newlib(L, arraylib);
  return 1;
}
```

И вновь мы воспользуемся функцией `luaL_newlib` из вспомогательной библиотеки. Она создает таблицу и заполняет ее парами имя-функция, заданными массивом `arraylib`.

После открытия библиотеки мы готовы использовать наш новый тип в Lua:

```
a = array.new(1000)
print(a)                --> userdata: 0x8064d48
print(array.size(a))    --> 1000
for i = 1, 1000 do
  array.set(a, i, i%5 == 0)
end
print(array.get(a, 10)) --> true
```

29.2. Метатаблицы

У нашей текущей реализации есть крупная дыра в безопасности. Допустим, пользователь напишет что-то вроде

`array.set(io.stdin,l,false)`. Значение в `io.stdin` — это пользовательские данные с указателем на поток (`FILE*`). Из-за того, что это пользовательские данные, `array.set` с радостью примет их как допустимый аргумент; вероятно, результатом будет повреждение памяти (если повезет, вы можете получить вместо этого сообщение о выходе индекса за границы индексации). Подобное поведение недопустимо для любой библиотеки Lua. Независимо от того, как вы используете библиотеку, она не должна повреждать данные C или приводить к сбоям с выдачей дампа ядра из Lua.

Обычным методом, чтобы отличать один тип пользовательских данных от других, является создание уникальной метатаблицы для этого типа. Каждый раз, когда мы создаем пользовательские данные, мы помечаем их при помощи соответственной метатаблицы; каждый раз, когда мы получаем пользовательские данные, мы проверяем, есть ли у них правильная метатаблица. Поскольку код Lua не может изменить метатаблицу пользовательских данных, он не сможет подделать наш код.

Нам также нужно место для хранения этой новой метатаблицы, чтобы мы могли обращаться к ней при создании новых пользовательских данных и при проверке того, что у требуемых пользовательских данных правильный тип. Как мы уже видели, есть два варианта для хранения метатаблицы: в реестре или как верхнее значение для функций в библиотеке. В Lua принято регистрировать каждый новый тип C в реестре, используя *имя типа* как индекс, а метатаблицу как его значение. Как и с любыми другими индексами реестра, во избежание конфликтов мы должны с осторожностью выбирать имя типа. В нашем примере мы будем использовать для нашего нового типа имя "`LuaBook.array`".

Как обычно, в этом нам поможет вспомогательная библиотека, предоставив некоторые функции. Эти новые вспомогательные функции, которыми мы будем пользоваться, следующие:

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

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

Функция `luaL_newmettable` создает новую таблицу (для использования в качестве метатаблицы), помещает ее на вершину стека и связывает эту таблицу с заданным именем в реестре. Функция `luaL_getmetatable` возвращает метатаблицу, связанную с именем `tname`, из реестра. Наконец, `luaL_checkudata` проверяет, что объект на заданной позиции стека является пользовательскими данными с метатаблицей, которая соответствует заданному имени. Она вызывает ошибку, если у объекта неправильная метатаблица, или это не пользовательские данные; в противном случае она возвращает адрес объекта.

Теперь мы можем начать нашу реализацию. Первым шагом будет изменение функции, которая открывает нашу библиотеку. Новая версия должна создавать метатаблицу для массивов:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_newlib(L, arraylib);
  return 1;
}
```

Следующим шагом будет изменение `newarray` таким образом, чтобы она устанавливала метатаблицу во всех создаваемых ей массивах:

```
static int newarray (lua_State *L) {
  <как прежде>
  luaL_getmetatable(L, "LuaBook.array");
  lua_setmetatable(L, -2);

  return 1; /* новые пользовательские данные уже в
стеке */
}
```

Функция `lua_setmetatable` выталкивает таблицу из стека и устанавливает ее как метатаблицу для объекта по заданному индексу. В нашем случае этим объектом являются новые пользовательские данные.

Наконец, `setarray`, `getarray` и `getsize` должны проверить, что они получили допустимый массив в качестве своего первого аргумента. Чтобы упростить им задачи, мы определим следующий макрос:

```
#define checkarray(L) \
          (NumArray *)luaL_checkudata(L, 1,
"LuaBook.array")
```

С помощью этого макроса новое определение `getsize` не вызывает затруднений:

```
static int getsize (lua_State *L) {
  NumArray *a = checkarray(L);
  lua_pushinteger(L, a->size);
  return 1;
}
```

Поскольку `setarray` и `getarray` к тому же имеют общий код для проверки индекса как своего второго аргумента, мы вынесем их общие части в следующую функцию:

```
static unsigned int *getindex (lua_State *L, unsigned
int *mask) {
  NumArray *a = checkarray(L);
  int index = luaL_checkint(L, 2) - 1;

  luaL_argcheck(L, 0 <= index && index < a->size, 2,
  "index out of range");

  /* возвращает адрес элемента */
  *mask = I_BIT(index);
  return &a->values[I_WORD(index)];
}
```

После определения `getindex` несложно написать `setarray` и

getarray:

```
static int setarray (lua_State *L) {
  unsigned int mask;
  unsigned int *entry = getindex(L, &mask);
  luaL_checkany(L, 3);
  if (lua_toboolean(L, 3))
    *entry |= mask;
  else
    *entry &= ~mask;

  return 0;
}

static int getarray (lua_State *L) {
  unsigned int mask;
  unsigned int *entry = getindex(L, &mask);
  lua_pushboolean(L, *entry & mask);
  return 1;
}
```

Теперь, если вы попытаетесь выполнить что-то вроде `array.get(io.stdin,10)`, то получите соответственное сообщение об ошибке:

```
error: bad argument #1 to 'get' ('array' expected)
```

29.3. Объектно-ориентированный доступ

Нашим следующим шагом будет преобразование нашего нового типа в объект, чтобы мы могли работать с его экземплярами при помощи обычного объектно-ориентированного синтаксиса таким образом:

```
a = array.new(1000)
print(a:size())     --> 1000
a:set(10, true)
print(a:get(10))    --> true
```

Вспомним, что форма записи `a:size()` эквивалента `a.size(a)`.

Поэтому мы должны переделать выражение `a.size` так, чтобы оно возвращало нашу функцию `getsize`. Ключевым механизмом здесь является метаметод `__index`. Lua вызывает этот метаметод для таблиц всякий раз, когда он не может найти значение для заданного ключа. Для пользовательских данных Lua вызывает его при каждом обращении, поскольку у пользовательских данных вообще нет ключей.

Предположим, что мы выполнили следующий код:

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = metaarray
metaarray.set = array.set
metaarray.get = array.get
metaarray.size = array.size
```

В первой строке кода мы создаем массив лишь для получения его метатаблицы, которую мы присваиваем `metarray`. (Мы не можем установить метатаблицу пользовательских данных из Lua, но мы можем получить ее.) Затем мы устанавливаем `metaarray.__index` равной `metaarray`. Когда мы вычисляем `a.size`, Lua не может найти ключ "`size`" в объекте `a`, поскольку этот объект является пользовательскими данными. Поэтому Lua пытается получить это значение из поля `__index` метатаблицы `a`, которая совпадает с самим `metaarray`. Но `metaarray.size` — это `array.size`, поэтому `a.size(a)` возвращает `array.size(a)`, как и требовалось.

Конечно, мы можем написать то же самое на C. Мы можем сделать еще лучше: теперь, когда массивы являются объектами со своими собственными операциями, нам больше не нужно хранить эти операции в таблице `array`. Единственной функцией, которую наша библиотека по-прежнему должна экспортировать, является функция `new` для создания новых массивов. Все прочие операции доступны только как методы. В связи с этим код C может регистрировать их напрямую.

Операции `getsize`, `getarray` и `setarray` останутся такими же, как и в нашем предыдущем подходе. Изменится только то, как мы

их зарегистрируем. Для этого нам нужно изменить код, который открывает библиотеку. Во-первых, нам потребуются два отдельных списка функций: список обычных функций и список методов.

```
static const struct luaL_Reg arraylib_f [] = {
  {"new", newarray},
  {NULL, NULL}
};
static const struct luaL_Reg arraylib_m [] = {
  {"set", setarray},
  {"get", getarray},
  {"size", getsize},
  {NULL, NULL}
};
```

Новая версия открывающей функции `luaopen_array` должна создать метатаблицу, присвоить ее своему собственному полю `__index`, зарегистрировать там все методы, а затем создать и заполнить таблицу `array`:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");

  /* metatable.__index = metatable */
  lua_pushvalue(L, -1); /* дублирует эту метатаблицу
*/
  lua_setfield(L, -2, "__index");

  luaL_setfuncs(L, arraylib_m, 0);

  luaL_newlib(L, arraylib_f);
  return 1;
}
```

Здесь мы вновь используем `luaL_setfuncs`, чтобы записать функции из списка `arraylib_m` в метатаблицу, которая находится на вершине стека. Затем мы используем `luaL_newlib` для создания новой таблицы и регистрации там функций из списка `arraylib_f` (в данном случае только `new`).

В качестве завершающего штриха мы добавим к нашему новому

типу метод `__tostring`, чтобы `print(a)` печатал "`array`" и размер массива в круглых скобках; должно получиться что-то вроде "`array(1000)`". Ниже сама функция:

```
int array2string (lua_State *L) {
  NumArray *a = checkarray(L);
  lua_pushfstring(L, "array(%d)", a->size);
  return 1;
}
```

Вызов `lua_pushfstring` форматирует строку и оставляет ее на вершине стека. Мы также должны добавить `array2string` к списку `arraylib_m`, чтобы включить ее в метатаблицу из объектов-массивов:

```
static const struct luaL_Reg arraylib_m [] = {
  {"__tostring", array2string},
  <другие методы>
};
```

29.4. Доступ как к массиву

Альтернативой объектно-ориентированной нотации является обычная нотация для доступа к нашим массивам. Вместо записи `a:get(i)` мы могли бы просто писать `a[i]`. Для нашего примера это сделать легко, поскольку наши функции `setarray` и `getarray` уже получают свои аргументы в том порядке, в котором они заданы для соответствующих метаметодов. Быстрым решением будет определение этих метаметодов прямо в нашем коде Lua:

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
metaarray.__len = array.size
```

(Мы должны выполнить этот код на нашей первоначальной реализации массивов, без модификаций для объектно-

ориентированного доступа.) Это все, что нам нужно для использования стандартного синтаксиса:

```
a = array.new(1000)
a[10] = true    -- 'setarray'
print(a[10])    -- 'getarray'    --> true
print(#a)       -- 'getsize'     --> 1000
```

Если потребуется, мы можем зарегистрировать эти метаметоды в коде C. Для этого мы снова должны изменить нашу инициализирующую функцию:

```
static const struct luaL_Reg arraylib_f [] = {
  {"new", newarray},
  {NULL, NULL}
};

static const struct luaL_Reg arraylib_m [] = {
  {"__newindex", setarray},
  {"__index", getarray},
  {"__len", getsize},
  {"__tostring", array2string},
  {NULL, NULL}
};

int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_setfuncs(L, arraylib_m, 0);
  luaL_newlib(L, arraylib_f);
  return 1;
}
```

В этой новой версии у нас опять есть лишь одна открытая функция — `new`. Все прочие функции доступны только как метаметоды для специфических операций.

29.5. Облегченные пользовательские данные

Вид пользовательских данных, который мы до сих пор

использовали, называется *полными пользовательскими данными* (ППД) (full userdata). Lua предлагает и другой вид — *облегченные пользовательские данные* (ОПД) (light userdata).

Тип ОПД представляет собой просто указатель в C (то есть значение типа `void*`). Это значение, а не объект; мы не создаем его (так же, как мы не создаем числа). Чтобы поместить ОПД в стек, мы вызываем `lua_pushlightuserdata`:

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

Несмотря на похожие имена, полные и облегченные пользовательские данные на самом деле сильно отличаются. ОПД — это не буферы, а всего лишь указатели. У них нет метатаблиц. Сборщик мусора их не удаляет, также как и числа.

Иногда мы используем ОПД как дешевую альтернативу ППД. Однако, это не типичное их применение. Во-первых, у ОПД нет метатаблиц, поэтому мы не можем знать их тип. Во-вторых, несмотря на свое название, ППД тоже довольно дешевы. Они лишь немного снижают производительность по сравнению с вызовом `malloc`.

Настоящее применение ОПД связано с проверкой на равенство. Так как тип ППД является объектом, то он равен только самому себе. Тип ОПД, с другой стороны, представляет из себя значение указателя C. И как таковой он равен любым пользовательским данным, представляющим тот же самый указатель. Таким образом, мы можем использовать ОПД для нахождения объектов C внутри Lua.

Мы уже видели типичное применение облегченных пользовательских данных как ключей в реестре (см. раздел 28.3). Там равенство ОПД было крайне важным. Каждый раз, когда мы заталкиваем ОПД в стек при помощи `lua_pushlightuserdata`, мы получаем то же значение Lua и, соответственно, ту же запись в реестре.

Другим типичным сценарием является необходимость получить

полные пользовательские данные по их адресу в C. Допустим, мы организуем связь между Lua и графической оконной системой. Тогда мы можем использовать ППД для представления окон. Каждые пользовательские данные содержат или всю структуру, представляющую окно, или только указатель на структуру, созданную системой. Когда внутри окна случается событие (например, нажатие кнопки мыши), система вызывает соответствующий обработчик, идентифицируя окно по его адресу. Чтобы передать в Lua обратный вызов, мы должны найти пользовательские данные, представляющие данное окно. Чтобы их найти, мы можем использовать таблицу, где индексы — это ОПД с адресами окон, а значения — это ППД, представляющие эти окна в Lua. Если у нас есть адрес окна, мы заталкиваем его в стек как ОПД и используем их как индекс в той таблице. (Скорее всего, у той таблицы должны быть слабые значения. Иначе те полные пользовательские данные никогда не будут удалены сборщиком мусора.)

Упражнения

Упражнение 29.1. Измените реализацию `setarray`, чтобы она принимала только логические значения.

Упражнение 29.2. Мы можем рассматривать массив логических значений как множество целых чисел (индексы с истинными значениями в массиве). Добавьте к реализации массивов логических значений функции, которые вычисляют объединение и пересечение двух массивов. Эти функции должны получать два массива и возвращать новый массив, не изменяя свои параметры.

Упражнение 29.3. Измените реализацию метаметода `__tostring`, чтобы он показывал полное содержимое массива подходящим образом. Используйте буферные средства (см. раздел 28.2) для создания итоговой строки.

Упражнение 29.4. На основе примера с массивами логических значений реализуйте библиотеку C для работы с массивами целых чисел.

ГЛАВА 30

Управление ресурсами

В нашей реализации массивов логических значений из предыдущей главы мы не беспокоились об управлении ресурсами. Этим массивам нужна лишь память. Каждые пользовательские данные, представляющие массив, обладают своей собственной памятью, которой управляет Lua. Когда массив становится мусором (то есть становится недоступным для программы), Lua со временем утилизирует его и освободит занимаемую им память.

Но жизнь не всегда так легка. Иногда объекту нужны другие ресурсы, кроме необработанной памяти, такие как дескрипторы файлов, дескрипторы окон и т. п. (зачастую эти ресурсы тоже являются памятью, но управляются другой частью системы). В подобных случаях, когда объект становится мусором и утилизируется, необходимо как-то высвободить и эти ресурсы.

Как мы уже видели в разделе 17.6, Lua предоставляет финализаторы в виде метаметода `__gc`. Чтобы проиллюстрировать применение этого метаметода в C и API в целом, в данной главе мы разработаем две привязки от Lua к внешним средствам. Первый пример — это иная реализация функции для обхода директории. Второй (и более существенный) пример — это привязка к *Expat*, парсеру XML с открытым исходным кодом. (Парсер — это программа для синтаксического анализа.)

30.1. Итератор по директории

В разделе 27.1 мы реализовали функцию `dir`, которая возвращала таблицу со всеми файлами из заданной директории. Наша новая реализация вернет итератор, который возвращает новую

запись при каждом вызове. С этой новой реализацией мы можем перебрать директорию посредством примерно такого цикла:

```
for fname in dir.open(".") do
  print(fname)
end
```

Для итерации по директории в C нам понадобится структура `DIR`. Экземпляры `DIR` нужно создавать при помощи `opendir` и явно высвобождать вызовом `closedir`. Наша предыдущая реализация `dir` хранила свой экземпляр `DIR` как локальную переменную и закрывала этот экземпляр после получения имени последнего файла. Наша новая реализация `dir` не может хранить этот экземпляр `DIR` в локальной переменной, поскольку она должна запрашивать это значение на протяжении нескольких вызовов. Более того, она не может закрыть директорию, не получив ее последнее имя; если программа прервет цикл, итератор никогда не получит ее последнее имя. Поэтому, чтобы убедиться, что экземпляр `DIR` всегда высвобожден, мы храним ее адрес в пользовательских данных и используем метаметод `__gc` для высвобождения структуры этой директории.

Несмотря на свою центральную роль в нашей реализации, этим пользовательским данным, представляющим директорию, не нужно быть видимыми из Lua. Функция `dir` возвращает итерирующую функцию, которую и видит Lua. Директория может быть верхним значением этой итерирующей функции. Таким образом, итерирующая функция обладает прямым доступом к этой структуре, но код Lua к ней доступа не имеет (и ему это не нужно).

Итого нам нужны три функции C. Во-первых, нам нужна функция `dir.open` — средство, которое Lua вызывает для создания итераторов; она должна открыть структуру `DIR` и создать замыкание итерирующей функции с этой структурой в качестве верхнего значения. Во-вторых, нам нужна итерирующая функция. В-третьих, нам нужен метаметод `__gc`, который закрывает структуру `DIR`. Как

обычно, нам также понадобится дополнительная функция для начальной подготовки, например, для создания и инициализации метатаблицы для директорий.

Давайте начнем наш код с функции `dir.open`, показанной в листинге 30.1. У нее есть один нюанс — она должна создать пользовательские данные до открытия директории. Если она сначала откроет директорию, то вызов `lua_newuserdata` приведет к ошибке доступа к памяти, потере функции и утечке структуры `DIR`. При правильном порядке структура `DIR`, как только она создана, сразу же привязывается к пользовательским данным; что бы ни случилось после этого, метаметод `__gc` со временем высвободит эту структуру.

Листинг 30.1. Фабричная функция **`dir.open`**

```
#include <dirent.h>
#include <errno.h>
#include <string.h>

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

/* предваряющее объявление для итерирующей функции */
static int dir_iter (lua_State *L);

static int l_dir (lua_State *L) {
  const char *path = luaL_checkstring(L, 1);

  /* создает объект userdata для хранения адреса DIR
*/
  DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));

  /* устанавливает его метатаблицу */
  luaL_getmetatable(L, "LuaBook.dir");
  lua_setmetatable(L, -2);

  /* пытается открыть заданную директорию */
  *d = opendir(path);
  if (*d == NULL) /* ошибка открытия директории? */
    luaL_error(L, "cannot open %s: %s", path,
```

```
    strerror(errno));

  /* создает и возвращает итерирующую функцию;
      ее единственное верхнее значение,
      пользовательские данные директории,
      уже находится на вершине этого стека */
  lua_pushcclosure(L, dir_iter, 1);
  return 1;
}
```

Следующая функция — это `dir_iter` (листинг 30.2), сам итератор. Ее код не требует долгих пояснений. Она получает адрес структуры `DIR` из ее верхнего значения и вызывает `readdir` для получения следующей записи.

Функция `dir_gc` (также в листинге 30.2) — это метаметод `__gc`. Он закрывает директорию, но при этом он должен соблюдать одну меру предосторожности: так как мы создаем пользовательские данные до открытия директории, эти пользовательские данные должны быть собраны сборщиком мусора независимо от результата `opendir`. Если `opendir` даст сбой, закрывать будет нечего.

Последняя функция в листинге 30.2, `luaopen_dir`, — это функция, которая открывает эту библиотеку с одной функцией.

В этом полном примере есть интересный нюанс. Сперва может показаться, что `dir_gc` должна проверять, что ее аргумент является директорией. Иначе пользователь-злоумышленник может вызвать ее с другим видом пользовательских данных (например, файлом), что приведет к катастрофическим последствиям. Однако, у программы на Lua нет способа обратиться к этой функции: она хранится только в метатаблице директорий, которые, в свою очередь, хранятся как верхние значения итерирующей функциии. Программы Lua не могут обращаться к этим директориям.

Листинг 30.2. Другие функции для библиотеки **`dir`**

```
static int dir_iter (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L,
```

```
lua_upvalueindex(1));
  struct dirent *entry;
  if ((entry = readdir(d)) != NULL) {
    lua_pushstring(L, entry->d_name);
    return 1;
  }
  else return 0; /* больше значений для возврата нет
*/
}

static int dir_gc (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L, 1);
  if (d) closedir(d);
  return 0;
}

static const struct luaL_Reg dirlib [] = {
  {"open", l_dir},
  {NULL, NULL}
};

int luaopen_dir (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.dir");

  /* устанавливает его поле __gc */
  lua_pushcfunction(L, dir_gc);
  lua_setfield(L, -2, "__gc");

  /* создает библиотеку */
  luaL_newlib(L, dirlib);
  return 1;
}
```

30.2. Парсер XML

Теперь мы взглянем на упрощенную реализацию привязки между Lua и Expat, которую мы назовем `lxp`. Expat — это парсер XML 1.0 с открытым исходным кодом, написанным на C. Он реализует SAX — ***S**imple **A**PI for **X**ML*. SAX — это событийно-ориентированный API. Это значит, что парсер SAX читает документ

XML и по мере чтения сообщает приложению, что он находит, при помощи обратных вызовов. Например, если мы дадим указания Expat разобрать строку наподобие "`<tag cap="5">hi</tag>`", то он сгенерирует три события: событие *начального элемента* (start-element), когда он читает подстроку "`<tag cap="5">`"; событие *текста* (также называемое событием *символьных данных* (character data)), когда он читает "`hi`"; и событие *конечного элемента* (end-element), когда он читает "`</tag>`". Каждое из этих событий вызывает подходящий *обработчик обратных вызовов* (callback handler) в приложении.

Здесь мы не будем охватывать всю библиотеку Expat. Мы сосредоточимся только на тех частях, которые иллюстрируют новые техники взаимодействия с Lua. Хотя Expat обрабатывает более дюжины различных событий, мы рассмотрим лишь те три события, которые мы увидели в предыдущем примере: начальные элементы, конечные элементы и текст. (Примечание: Пакет LuaExpat предоставляет практически полный интерфейс к Expat.)

Для этого примера нам потребуется лишь малая часть Expat API. Во-первых, нам нужны функции для создания и уничтожения парсера Expat:

```
XML_Parser XML_ParserCreate (const char *encoding);
void XML_ParserFree (XML_Parser p);
```

Аргумент `encoding` не обязателен; в нашей привязке мы используем `NULL`.

После того, как у нас есть парсер, мы должны зарегистрировать его обработчики обратных вызовов:

```
void XML_SetElementHandler(XML_Parser p,
                           XML_StartElementHandler
start,
                           XML_EndElementHandler end);

void XML_SetCharacterDataHandler(XML_Parser p,
```

```
    XML_CharacterDataHandler hndl);
```

Первая функция регистрирует обработчики для начального и конечного элементов. Вторая функция регистрирует обработчики для текста (*символьные данные* на языке XML).

Все обработчики обратных вызовов получают в качестве своего первого параметра некоторые пользовательские данные. Обработчик начального элемента также получает имя тега и его атрибуты:

```
typedef void (*XML_StartElementHandler)(void *uData,
                                        const char
*name,
                                        const char
**atts);
```

Атрибуты передаются как массив строк, завершенных нуль-символом, где каждая пара последующих строк содержит атрибут и его значение. Обработчик конечного элемента обладает лишь одним дополнительным параметром — именем тега:

```
typedef void (*XML_EndElementHandler)(void *uData,
                                      const char
*name);
```

Наконец, обработчик текста получает в качестве дополнительного параметра только текст. Эта текстовая строка не завершена нуль-символом; вместо этого она содержит явно заданную длину:

```
typedef void (*XML_CharacterDataHandler)(void *uData,
                                         const char
*s,
                                         int len);
```

Для скармливания текста Expat мы воспользуемся следующей функцией:

```
int XML_Parse (XML_Parser p, const char *s, int len,
int isLast);
```

Expat получает документ, который необходимо разобрать, по частям, через последовательные вызовы `XML_Parse`. Последний аргумент `XML_Parse`, `isLast`, сообщает Expat, была ли та часть последней в документе. Обратите внимание, что каждой части текста не требуется завершаться нуль-символом; вместо этого мы предоставляем явную длину. Функция `XML_Parse` возвращает нуль, если обнаружит ошибку разбора. (Expat также предоставляет функции для получения информации об ошибке, но для простоты мы не будем их здесь рассматривать.)

Последняя функция, которая нам нужна от Expat, позволяет задать пользовательские данные, которые будут переданы обработчикам:

```
void XML_SetUserData (XML_Parser p, void *uData);
```

Теперь давайте посмотрим, как мы можем использовать эту библиотеку в Lua. Первый подход — это прямой подход: просто экспортируем все те функции в Lua. Более удачный подход состоит в адаптации этой функциональности к Lua. Например, поскольку Lua является языком с динамической типизацией, нам не нужны разные функции для каждого вида обратного вызова. Еще лучше то, что мы можем полностью избежать функции для регистрации обратных вызовов. Вместо этого при создании парсера мы зададим таблицу обратных вызовов, которая содержит все обработчики обратных вызовов, каждый с подходящим ключом. Например, если нам потребуется напечатать структуру документа, мы можем использовать следующую таблицу обратных вызовов:

```
local count = 0

callbacks = {
  StartElement = function (parser, tagname)
    io.write("+ ", string.rep(" ", count), tagname,
"\n")
    count = count + 1
  end,
```

```
EndElement = function (parser, tagname)
  count = count - 1
  io.write("- ", string.rep(" ", count), tagname,
"\n")
  end,
}
```

Получив на вход строку "<to> <yes/> </to>", эти обработчики напечатали бы следующий вывод:

```
+ to
+    yes
-    yes
- to
```

С этим API нам не нужны функции для управления обратными вызовами. Мы управляем ими напрямую в таблице обратных вызовов. Таким образом, для всего API требуется лишь три функции: первая для создания парсеров, вторая для обработки части текста и третья для закрытия парсера. На самом деле мы реализуем последние две функции как методы объектов парсера. Типичное применение данного API могло бы выглядеть следующим образом:

```
local lxp = require"lxp"

p = lxp.new(callbacks)    -- создает новый парсер

for l in io.lines() do    -- итерирует через введенные
строки
  assert(p:parse(l))      -- разбирает эту строку
  assert(p:parse("\n"))   -- добавляет перевод строки
end

assert(p:parse())         -- завершает документ
p:close()
```

Теперь давайте обратим наше внимание на реализацию. Первое решение, которое нужно принять, — это как представить парсер в

Lua. Вполне естественно будет использовать пользовательские данные, но что нам нужно поместить внутрь них? По меньшей мере нам понадобятся настоящий парсер Expat и таблица обратных вызовов. Мы не можем хранить таблицу Lua внутри пользовательских данных (или внутри какой-либо структуры C), но Lua позволяет каждым пользовательским данным иметь *пользовательское значение* (user value), которое может быть любой таблицей Lua, связанной с ними. (Примечание: В Lua 5.1 пользовательским значением служит окружение пользовательских данных.) Мы также должны хранить состояние Lua в объекте парсера, поскольку эти объекты парсера — это все, что принимает обратный вызов Expat, а обратные вызовы нужны для вызова Lua. Поэтому определение объекта парсера следующее:

```
#include <stdlib.h>
#include "expat.h"
#include "lua.h"
#include "lauxlib.h"

typedef struct lxp_userdata {
  XML_Parser parser; /* связанный парсер expat */
  lua_State *L;
} lxp_userdata;
```

Следующим шагом является функция для создания объектов парсера, `lxp_make_parser`. Ее код приведен в листинге 30.3. Эта функция состоит из четырех важных шагов:

- Ее первый шаг следует общепринятому образцу: сначала она создает пользовательские данные; затем она предварительно инициализирует пользовательские данные согласованными значениями; и, наконец, она устанавливает свою метатаблицу. Причина этой предварительной инициализации следующая: если в ходе инициализации возникает какая-либо ошибка, то мы должны убедиться, что финализатор (метаметод `__gc`) найдет пользовательские

данные в согласованном состоянии.

- На втором шаге функция создает парсер Expat, сохраняет его в пользовательских данных и проверяет на ошибки.
- Третий шаг проверяет, что первым аргументом функции действительно является таблица (таблица обратных вызовов), и устанавливает ее как пользовательское значение для пользовательских данных.
- Последний шаг инициализирует парсер Expat. Он устанавливает пользовательские данные как объект для передачи в функции обратного вызова и устанавливает эти функции обратного вызова. Обратите внимание, что эти функции обратного вызова одни и те же для всех парсеров; в конце концов, на C невозможно динамически создавать новые функции. Вместо этого те фиксированные функции C используют таблицу обратных вызовов, чтобы решить, какие функции Lua им следует вызвать, когда придет время.

Листинг 30.3. Функция для создания объектов-парсеров XML

```
/* предваряющие объявления для функций обратного
вызова */
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts);
static void f_CharData (void *ud, const char *s, int
len);
static void f_EndElement (void *ud, const char *name);

static int lxp_make_parser (lua_State *L) {
  XML_Parser p;

  /* (1) создает объект-парсер */
  lxp_userdata *xpu = (lxp_userdata
*)lua_newuserdata(L,

sizeof(lxp_userdata));
```

```
  /* предварительно инициализирует его на случай
ошибки */
  xpu->parser = NULL;

  /* устанавливает его метатаблицу */
  luaL_getmetatable(L, "Expat");
  lua_setmetatable(L, -2);

  /* (2) создает парсер Expat */
  p = xpu->parser = XML_ParserCreate(NULL);
  if (!p)
    luaL_error(L, "XML_ParserCreate failed");

  /* (3) проверяет и хранит таблицу обратного вызова
*/
  luaL_checktype(L, 1, LUA_TTABLE);
  lua_pushvalue(L, 1);     /* помещает таблицу на
вершину стека */
  lua_setuservalue(L, -2); /* устанавливает ее как
пользовательское значение */

  /* (4) конфигурирует парсер Expat */
  XML_SetUserData(p, xpu);
  XML_SetElementHandler(p, f_StartElement,
f_EndElement);
  XML_SetCharacterDataHandler(p, f_CharData);
  return 1;
}
```

Следующим шагом является метод парсера `lxp_parse` (листинг 30.4), который разбирает часть данных XML. Он получает два аргумента: объект парсера (*self* из этого метода) и необязательную часть данных XML. При вызове без каких-либо данных он сообщает Expat, что в документе больше не осталось частей.

Листинг 30.4. Функция для разбора фрагмента XML

```
static int lxp_parse (lua_State *L) {
  int status;
  size_t len;
  const char *s;
  lxp_userdata *xpu;
```

```
  /* получает и проверяет первый аргумент (должен быть
парсером) */
  xpu = (lxp_userdata *)luaL_checkudata(L, 1,
"Expat");

  /* проверяет, что он не закрыт */
  luaL_argcheck(L, xpu->parser != NULL, 1, "parser is
closed");

  /* получает второй аргумент (строку) */
  s = luaL_optlstring(L, 2, NULL, &len);

  /* помещает таблицу обратного вызова в стек по
индексу 3 */
  lua_settop(L, 2);
  lua_getuservalue(L, 1);
  xpu->L = L; /* set Lua state */

  /* вызывает Expat для разбора строки */
  status = XML_Parse(xpu->parser, s, (int)len, s ==
NULL);

  /* возвращает код ошибки */
  lua_pushboolean(L, status);
  return 1;
}
```

Когда `lxp_parse` вызывает `XML_Parse`, последняя функция вызовет обработчики для каждого подходящего элемента, который она найдет в данной части документа. Этим обработчикам понадобится доступ к таблице обратных вызовов, поэтому `lxp_parse` заталкивает эту таблицу в стек по индексу 3 (сразу после параметров). Есть один нюанс в вызове `XML_Parse`: помните, что последний аргумент этой функции сообщает Expat, является ли данная часть текста последней. Когда мы вызываем `parse` без аргументов, `s` будет равна `NULL`, поэтому последний аргумент будет равен true.

Теперь давайте обратим наше внимание на функции обратного

вызова: `f_StartElement`, `f_EndElement` и `f_CharData`. Все три функции обладают одинаковой структурой: каждая из них проверяет, определяет ли таблица обратных вызовов обработчик Lua для своего конкретного события, и, если определяет, то подготавливает аргументы и затем вызывает этот обработчик Lua.

Давайте сначала рассмотри обработчик `f__CharData` в листинге 30.5. Его код довольно прост. Обработчик получает структуру `lxp_userdata` как свой первый аргумент, поскольку мы вызвали `XML_setuserData` при создании парсера. После получения состояния Lua обработчик может обратиться к таблице обратных вызовов в стеке по индексу 3, заданному `lxp_parse`, и к самому парсеру по индексу 1. Затем он вызывает соответствующий обработчик на Lua (когда он есть) с двумя аргументами: парсером и символьными данными (строкой).

Листинг 30.5. Обработчик символьных данных

```
static void f_CharData (void *ud, const char *s, int
len) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  /* получает обработчик */
  lua_getfield(L, 3, "CharacterData");
  if (lua_isnil(L, -1)) {        /* обработчика нет?
*/
  lua_pop(L, 1);
  return;
}

  lua_pushvalue(L, 1);           /* заталкивает парсер
('self') */
  lua_pushlstring(L, s, len);    /* заталкивает
символьные данные */
  lua_call(L, 2, 0);             /* вызывает этот
обработчик */
}
```

Обработчик `f_EndElement` довольно похож на `f_CharData`; взгляните на листинг 30.6. Он также вызывает соответствующий обработчик Lua с двумя аргументами — парсером и именем тега (снова строкой, но на этот раз завершенной нуль-символом).

Листинг 30.6. Обработчик конечных элементов

```
static void f_EndElement (void *ud, const char *name)
{
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  lua_getfield(L, 3, "EndElement");
  if (lua_isnil(L, -1)) {     /* обработчика нет? */
  lua_pop(L, 1);
  return;
}

  lua_pushvalue(L, 1);        /* заталкивает парсер
('self') */
  lua_pushstring(L, name);    /* заталкивает имя тега
*/
  lua_call(L, 2, 0);          /* вызывает этот
обработчик */
}
```

Листинг 30.7 показывает последний обработчик, `f_startElement`. Он вызывает Lua с тремя аргументами: парсером, именем тега и списком атрибутов. Этот обработчик немного сложнее остальных, поскольку ему необходимо перевести список атрибутов тега в Lua. Он использует вполне естественный перевод, строя таблицу, которая связывает имена атрибутов с их значениями. Например начальный тег наподобие

```
<to method="post" priority="high">
```

генерирует следующую таблицу атрибутов:

```
{method = "post", priority = "high"}
```

Листинг 30.7. Обработчик начальных элементов

```
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  lua_getfield(L, 3, "StartElement");
  if (lua_isnil(L, -1)) {          /* обработчика нет?
*/
    lua_pop(L, 1);
    return;
  }

  lua_pushvalue(L, 1);             /* заталкивает
парсер ('self') */
  lua_pushstring(L, name);         /* заталкивает имя
тега */

  /* создает и заполняет таблицу атрибутов */
  lua_newtable(L);
  for (; *atts; atts += 2) {
    lua_pushstring(L, *(atts + 1));
    lua_setfield(L, -2, *atts);    /* table[*atts] = *
(atts+1) */
  }

  lua_call(L, 3, 0);               /* вызывает этот
обработчик */
}
```

Последний метод для парсеров — это `close`, показанный в листинге 30.8. Когда мы закрываем парсер, нам нужно высвободить все его ресурсы, а именно структуру Expat. Помните, что из-за происходящих временами ошибок при создании у парсера может и не быть этой структуры. Обратите внимание, как мы поддерживаем парсер в согласованном состоянии по мере его закрытия, так что не будет никаких проблем, если мы попытаемся закрыть его снова, или когда сборщик мусора его финализирует. На самом деле мы будем

использовать эту функцию именно как финализатор. Это нужно для того, чтобы каждый парсер со временем обязательно освобождал свои ресурсы, даже если программист не закрыл его.

Листинг 30.8. Метод для закрытия парсера XML

```
static int lxp_close (lua_State *L) {
  lxp_userdata *xpu = (lxp_userdata
*)luaL_checkudata(L, 1, "Expat");

  /* бесплатный парсер Expat (если есть) */
  if (xpu->parser)
    XML_ParserFree(xpu->parser);
  xpu->parser = NULL;    /* предотвращает его
повторное закрытие */
  return 0;
}
```

Листинг 30.9 — это завершающий шаг: он показывает функцию `luaopen_lxp`, которая открывает библиотеку, совмещая в себе все ранее рассмотренные функции. Здесь мы используем ту же схему, что и для объектно-ориентированного примера массива логических значений в разделе 29.3: мы создаем метатаблицу, устанавливаем ее точку поля `__index` на нее саму и помещаем внутрь нее все методы. Для этого нам понадобится список с методами парсера (`lxp_meths`). Также нам нужен список функций этой библиотеки (`lxp_funcs`). Как и принято в объектно-ориентированных библиотеках, этот список содержит всего одну функцию, которая создает новые парсеры.

Листинг 30.9. Код инициализации для библиотеки **`lxp`**

```
static const struct luaL_Reg lxp_meths[] = {
  {"parse", lxp_parse},
  {"close", lxp_close},
  {"__gc", lxp_close},
  {NULL, NULL}
};
```

```
static const struct luaL_Reg lxp_funcs[] = {
  {"new", lxp_make_parser},
  {NULL, NULL}
};

int luaopen_lxp (lua_State *L) {
  /* создает метаблицу */
  luaL_newmetatable(L, "Expat");

  /* metatable.__index = metatable */
  lua_pushvalue(L, -1);
  lua_setfield(L, -2, "__index");

  /* регистрирует методы */
  luaL_setfuncs(L, lxp_meths, 0);

  /* регистрирует функции (только lxp.new) */
  luaL_newlib(L, lxp_funcs);
  return 1;
}
```

Упражнения

Упражнение 30.1. Модифицируйте функцию `dir_iter` в примере с директорией так, чтобы она закрывала структуру `DIR`, когда она достигает конца обхода. При этом изменении программе не нужно ждать сборки мусора для освобождения ресурса, который, как она знает, ей больше не нужен.

(Когда вы закрываете директорию, вы должны выставить адрес, сохраненный в пользовательских данных, на `NULL`, чтобы сообщить финализатору, что директория уже закрыта. Также функция `dir_iter` перед использованием директории должна проверять, что она не закрыта.)

Упражнение 30.2. В примере с `lxp` обработчик начальных элементов получает таблицу с атрибутами элемента. В этой таблице утерян первоначальный порядок, в котором атрибуты стояли внутри элемента. Как вы можете передать эту информацию в

обратный вызов?

Упражнение 30.3. В примере с `lxp` мы использовали пользовательские значения, чтобы связать таблицу обработчиков с пользовательскими данными, представляющими парсер. Этот выбор создал небольшую проблему, поскольку то, что получают обратные вызовы C, — это структура `lxp_userdata`, и эта структура не предоставляет прямой доступ к данной таблице. Мы решили эту проблему путем сохранения таблицы обратных вызовов по индексу стека фиксированного размера во время разбора каждого фрагмента.

Альтернативным подходом может быть связывание таблицы обратных вызовов с пользовательскими данными посредством ссылок (раздел 28.3): мы создаем ссылку на таблицу обратных вызовов и храним эту ссылку (целое число) в структуре `lxp_userdata`. Реализуйте данный подход. Не забудьте высвободить ссылку при закрытии парсера.

ГЛАВА 31

Нити и состояния

Lua не поддерживает настоящую многонитевость, то есть вытесняющие нити, использующие память совместно. Есть две причины отсутсвия данной поддержки. Первая причина состоит в том, что ANSI C не предоставляет ее, и поэтому не существует переносимого способа реализовать этот механизм в Lua. Второй и более серьезной причиной является то, что мы не считаем многонитевость хорошей идеей для Lua.

Многонитевость была разработана для низкоуровневого программирования. Механизмы синхронизации вроде семафоров и мониторов были разработаны для операционных систем (и опытных программистов), а не для прикладных программ. Крайне сложно находить и исправлять ошибки, связанные с многонитевостыо, и некоторые из них могут привести к брешам в безопасности. Более того, многонитевость может стать причиной снижения быстродействия из-за необходимости синхронизации в ряде критических мест программы, например при распределении памяти.

Проблемы с многонитевостыо возникают из-за комбинации вытесняющих нитей и общей памяти, поэтому мы можем избежать их, либо не используя вытесняющие нити, либо не используя общую память. Lua поддерживает оба подхода. Нити Lua (также известные как сопрограммы) являются совместными и поэтому избегают проблем, связанных с непредсказуемым переключением между нитями. Состояния Lua не имеют общей памяти, и поэтому формируют в Lua хорошую базу для параллельных вычислений. В данной главе мы охватим оба варианта.

31.1. Многонитевость

Нить (thread) — это суть сопрограммы в Lua. Мы рассматриваем сопрограмму как нить с удобным интерфейсом, или же мы можем рассматривать нить как сопрограмму с низкоуровневым API.

С точки зрения C, может быть удобно думать о нити как о стеке — чем на самом деле и является нить с точки зрения реализации. Каждый стек хранит информацию об ожидающих вызовах нити, а также параметрах и локальных переменных каждого вызова. Иными словами, стек содержит всю информацию, которая нужна нити для возобновления ее выполнения. Поэтому много нитей означает много независимых стеков.

Когда мы вызываем большинство функций из Lua-C API, эти функции работают с определенным стеком. Например, `lua_pushnumber` должна заталкивать число в определенный стек; `lua_pcall` нужен стек вызовов. Как Lua узнает, какой стек следует использовать? Секрет заключается в типе `lua_State`, первом аргументе этих функций, который представляет не только состояние Lua, но и нить внутри этого состояния. (Многие убеждены, что этот тип должен называться `lua_Thread`.)

Каждый раз, когда вы создаете состояние Lua, Lua автоматически создает новую нить внутри этого состояния и возвращает `lua_State`, представляющее эту нить. Эта *главная нить* (main thread) никогда не утилизируется. Она высвобождается вместе с состоянием, когда вы закрываете состояние при помощи `lua_close`.

Вы можете создавать другие нити внутри состояния путем вызова `lua_newthread`:

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

Эта функция возвращает указатель на `lua_State`, представляющее новую нить, а также заталкивает новую нить в стек как значение типа "`thread`". Например, после выполнения оператора

```
L1 = lua_newthread(L);
```

у нас будет две нити, `L1` и `L`, которые будут внутренне ссылаться на одно и то же состояние Lua. Каждая нить обладает собственным стеком. Новая нить `L1` начинает с пустого стека; у старой нити `L` есть новая нить на вершине стека:

```
printf("%d\n", lua_gettop(L1));          --> 0
printf("%s\n", luaL_typename(L, -1));    --> thread
```

За исключением главной нити, все нити подвержены сборке мусора, как и любой другой объект Lua. Когда вы создаете новую нить, то значение, которое заталкивается в стек, гарантирует, что эта нить не является мусором. Вы никогда не должны использовать нить, которая должным образом не привязана к состоянию. (Главная нить внутренне привязана с самого начала, поэтому о ней можно не беспокоиться.) Любой вызов Lua API может уничтожить непривязанную нить, даже вызов, использующий эту самую нить. Например, рассмотрим следующий фрагмент:

```
lua_State *L1 = lua_newthread (L);
lua_pop(L, 1);    /* L1 теперь является мусором для
Lua */
lua_pushstring(L1, "hello");
```

Вызов `lua_pushstring` может запустить сборщик мусора и собрать `L1` (и привести к сбою приложения), несмотря на то, что `L1` все еще используется. Во избежание этого всегда оставляйте ссылку на нити, которые вы используете, например в стеке привязанной нити или в реестре.

Как только у нас появляется новая нить, мы можем использовать ее тем же образом, что и главную. Мы можем заталкивать и выталкивать элементы из стека, мы можем использовать ее для вызова функций и т.п. Например, следующий код выполняет вызов `f(5)` в новой нити и затем перемещает результат в старую нить:

```
lua_getglobal(L1, "f");    /* допускает глобальную
функцию 'f' */
lua_pushinteger(L1, 5);
lua_call(L1, 1, 1);
lua_xmove(L1, L, 1);
```

Функция `lua_xmove` перемещает значение Lua между двумя стеками в одном и том же состоянии. Вызов наподобие `lua_xmove(F,т,n)` выталкивает `n` элементов из стека `F` и заталкивает их в `T`.

Однако, для этих целей нам не нужна новая нить; с таким же успехом мы могли бы использовать главную нить. Основной целью использования нескольких нитей является реализация сопрограмм, чтобы мы могли приостанавливать их выполнение и возобновлять их позже. Для этого нам нужна функция `lua_resume`:

```
int lua_resume (lua_State *L, lua_State *from, int
narg);
```

Для запуска выполнения сопрограммы мы используем `lua_resume` так же, как мы используем `lua_pcall`: мы заталкиваем функцию для вызова, заталкиваем ее аргументы и вызываем `lua_resume`, передавая в `narg` число аргументов. (Параметр `from` — это нить, которая совершает вызов.) Данное поведение также очень похоже на `lua_pcall`, но с тремя отличиями. Во-первых, у `lua_resume` нет параметра для числа требуемых результатов; она всегда возвращает все значения из вызванной функции. Во-вторых, у нее нет параметра для обработчика сообщений; ошибка не раскручивает стек, поэтому после нее вы можете изучить стек. В-третьих, если выполняемая функция уступает управление, то `lua_resume` возвращает особый код `LUA_YIELD` и оставляет нить в состоянии, из которого она может быть возобновлена позже.

Когда `lua_resume` возвращает `LUA_YIELD`, видимая часть стека нити содержит только значения, переданные `yield`. Вызов `lua_gettop` вернет число выработанных значений. Для перемещения этих значений в другую нить мы можем использовать

lua_xmove.

Чтобы возобновить приостановленную нить, мы вновь вызываем lua_resume. При таких вызовах Lua считает, что все значения в стеке должны быть возвращены вызовом yield. Например, если вы не трогаете стек нити между возвращением из lua_resume и следующим возобновлением, то yield вернет именно те значения, которые он выработал при уступке управления.

Обычно мы запускаем сопрограмму с функцией Lua в качестве ее тела. Эта функция Lua может вызывать другие функции, и любая из этих функций порой может уступать управление, завершая вызов lua_resume. Например, допустим, у нас есть следующие определения:

```
function foo (x) coroutine.yield(10, x) end

function foo1 (x) foo(x + 1); return 3 end
```

Теперь мы выполним этот код C:

```
lua_State *L1 = lua_newthread(L);
lua_getglobal(L1, "foo1");
lua_pushinteger(L1, 20);
lua_resume(L1, L, 1);
```

Этот вызов lua_resume вернет LUA_YIELD, чтобы сообщить, что нить уступила управление. В этот момент стек L1 содержит значения, переданные в yield:

```
printf("%d\n", lua_gettop(L1));           --> 2
printf("%d\n", lua_tointeger(L1, 1));     --> 10
printf("%d\n", lua_tointeger(L1, 2));      --> 21
```

Когда мы вновь возобновим эту нить, она продолжится с того места, где была остановлена (вызовом yield). Отсюда foo вернет управление foo1, а она, в свою очередь, вернет управление lua_resume:

```
lua_resume(L1, L, 0);
```

```
printf("%d\n", lua_gettop(L1));          --> 1
printf("%d\n", lua_tointeger(L1, 1));    --> 3
```

Этот второй вызов `lua_resume` вернет `LUA_OK`, что означает обычный возврат.

Сопрограммы также могут вызывать функции C, которые могут обратно вызывать другие функции Lua. Мы уже обсуждали, как использовать продолжения, чтобы позволить этим функциям Lua уступать управление (раздел 27.2). Сама по себе функция C также может уступать управление. В этом случае она также должна предоставить продолжающую функцию для вызова при возобновлении нити. Для уступки управления функция C должна вызвать следующую функцию:

```
int lua_yieldk (lua_State *L, int nresults, int ctx,
lua_CFunction k);
```

Мы всегда должны использовать эту функцию в операторе возврата, как, например, здесь:

```
static inf myCfunction (lua_State *L) {
...
return lua_yieldk(L, nresults, ctx, k);
}
```

Этот вызов немедленно приостанавливает выполняющуюся сопрограмму. Параметр `nresults` — это количество значений в стеке, которые нужно вернуть соответствующему `lua_resume`; `ctx` — это контекстная информация, которая должна быть передана продолжению; а `k` — это продолжающая функция. Когда сопрограмма возобновляется, управление переходит напрямую к продолжающей функции `k`. После уступки управления `myCfunction` не может больше ничего сделать; она должна делегировать всю дальнейшую работу своему продолжению.

Давайте рассмотрим типичный гипотетический пример. Пусть мы хотим написать код функции, которая читает некоторые данные,

уступая управление, пока данные не доступны. Мы можем написать эту функцию на C следующим образом:

```
int prim_read (lua_State *L) {
  if (nothing_to_read())
    return lua_yieldk(L, 0, 0, &prim_read);
  lua_pushstring(L, read_some_data());
  return 1;
}
```

Если у функции есть какие-то данные для чтения, то она читает и возвращает эти данные. В противном случае, если ей нечего читать, она уступает управление. При возобновлении нить вызовет продолжающую функцию. В этом примере продолжающей функцией является сама `prim_read`, поэтому нить снова вызывает `prim_read` и вновь попытается считать некоторые данные. (Этот образец функции, вызывающей `lua_yieldk`, которая является продолжающей функцией, встречается довольно часто.)

Если функции C нечего делать после уступки управления, то она может вызвать `lua_yieldk` без продолжающей функции или использовать макрос `lua_yield`:

```
return lua_yield(L, nres);
```

После этого вызова, когда нить снова возобновиться, управление вернется к функции, которая вызывала `myCfunction`.

31.2. Состояния Lua

Каждый вызов `luaL_newstate` (или `lua_newstate`, как мы увидим в главе 32) создает новое состояние Lua. Разные состояния Lua полностью независимы друг от друга. У них вообще нет общих данных. Это значит, что независимо от того, что происходит внутри одного состояния Lua, оно не может повредить данные другого. Это также значит, что состояния Lua не могут взаимодействовать

друг с другом напрямую; мы должны использовать в качестве посредника какой-нибудь код C. Например, если даны два состояния, `L1` и `L2`, то следующая команда затолкнет в `L2` строку с вершины стека в `L1`:

```
lua_pushstring(L2, lua_tostring(L1, -1));
```

Поскольку данные должны быть переданы посредством C, состояния Lua могут обмениваться только типами, которые могут быть представлены в C, например, строками и числами. Другие типы, такие как таблицы, должны быть сериализованы для осуществления переноса.

В системах, которые предлагают многонитевость, интересной схемой построения является создание независимого состояния Lua для каждой нити. Эта схема в результате дает нити, похожие на процессы UNIX, где мы получаем согласованность без общей памяти. В этом разделе мы разработаем прототипную реализацию многонитевости, следуя данному подходу. Для этой реализации я буду использовать нити POSIX (`pthreads`). При портировании данного кода на другие нитевые системы не должно возникнуть сложностей, так как в нем используются лишь базовые средства .

Система, которую мы собираемся построить, очень проста. Ее основным предназначением является показать использование нескольких состояний Lua в контексте многонитевости. После того, как она будет готова, вы можете реализовать на ее основе некоторые продвинутые возможности. Мы назовем нашу библиотеку `lproc`. Она предлагает всего четыре функции:

- `lproc.start(chunk)` запускает новый процесс для выполнения заданного куска (строки). Данная библиотека реализует *процесс* Lua как *нить* C плюс связанное с ней состояние Lua.
- `lproc.send(channel, val1, val2, ...)` отправляет все заданные значения (которые должны быть строками)

заданному каналу (который идентифицируют по его имени, тоже строке).

- `lproc.receive(channel)` получает значения, отправленные заданному каналу.
- `lproc.exit()` завершает процесс. Эта функция требуется только для главного процесса. Если этот процесс завершается без вызова `lproc.exit`, то вся программа прекращает свое выполнение без ожидания завершения других процессов.

Библиотека идентифицирует каналы при помощи строк и использует их для сопоставления отправителей и получателей. Операция отправки может отправить любое количество строковых значений, которые возвращаются сопоставленной с ней операцией получения. Все взаимодействие синхронно: процесс, отправляющий сообщение каналу, блокируется до тех пор, пока есть процесс, принимающий из этого канала, в то время как процесс, получающий из канала, блокируется до тех пор, пока есть процесс, отправляющий ему.

Реализация `lproc` также проста, как и ее интерфейс. Она использует два кольцевых двунаправленных списка, один для процессов, ожидающих отправки сообщения, а другой для процессов, ожидающих приема сообщения. Она использует один единственный мьютекс (объект взаимоисключающей блокировки) для управления доступом к этим спискам. У каждого процесса есть связанная с ним условная переменная. Когда процесс хочет отправить каналу сообщение, он обходит список получателей в поисках процесса, ожидающего этот канал. Если поиск успешен, то отправляющий процесс убирает найденный процесс из списка ожидания, перемещает из себя в него значения сообщения и предупреждает об этом другой процесс. В противном случае он вставляет себя в список отправителей и ожидает своей условной переменной. Получение сообщения происходит симметричным образом.

Главным элементом данной реализации является структура, представляющая процесс:

```
#include <pthread.h>
#include "lua.h"

typedef struct Proc {
  lua_State *L;
  pthread_t thread;
  pthread_cond_t cond;
  const char *channel;
  struct Proc *previous, *next;
} Proc;
```

Первые два поля представляют из себя состояние Lua, используемое процессом, и нить C, выполняющую данный процесс. Другие поля используются лишь тогда, когда процесс должен ждать подходящего отправителя или получателя. Третье поле, `cond`, — это условная переменная, которую нить использует для блокировки самой себя; четвертое поле хранит ожидаемый процессом канал; последние два поля, `previous` и `next`, используются для связывания структуры процесса в списке ожидания.

Следующий код объявляет два списка ожидания и связанный с ними мьютекс:

```
static Proc *waitsend = NULL;
static Proc *waitreceive = NULL;

static pthread_mutex_t kernel_access =
PTHREAD_MUTEX_INITIALIZER;
```

Каждому процессу нужна структура `Proc` и доступ к этой структуре всякий раз, когда скрипт вызывает `send` или `receive`. Единственный параметр, который принимают эти функции — это состояние Lua для процесса. Таким образом, каждый процесс должен хранить свою структуру `Proc` внутри своего состояния Lua. В нашей реализации каждое состояние хранит свою соответственную структуру `Proc` в реестре как полные пользовательские данные,

связанные с ключом "_SELF". Вспомогательная функция getself возвращает структуру Proc, связанную с данным состоянием:

```
static Proc *getself (lua_State *L) {
  Proc *p;
  lua_getfield(L, LUA_REGISTRYINDEX, "_SELF");
  p = (Proc *)lua_touserdata(L, -1);
  lua_pop(L, 1);
  return p;
}
```

Следующая функция, movevalues, перемещает значения из отправляющего процесса в принимающий:

```
static void movevalues (lua_State *send, lua_State
*rec) {
  int n = lua_gettop(send);
  int i;
  for (i = 2; i <= n; i++) /* переносит значения в
получателя */
    lua_pushstring(rec, lua_tostring(send, i));
}
```

Она перемещает в получателя все значения из стека отправителя, кроме самого первого, которым является канал.

Листинг 31.1 определяет функцию searchmatch, которая обходит список ожидания в поисках процесса, ожидающего заданный канал. Если функция находит такой канал, то она удаляет его из списка и возвращает его; иначе она возвращает NULL.

Листинг 31.1. Функция для поиска процесса, ожидающего заданного канала

```
static Proc *searchmatch (const char *channel, Proc
**list) {
  Proc *node = *list;
  if (node == NULL) return NULL; /* пустой список */
  do {
  if (strcmp(channel, node->channel) == 0) { /*
совпадение? */
    /* удаляет узел из этого списка */
```

```
    if (*list == node) /* этот узел является первым
элементом? */
      *list = (node->next == node) ? NULL : node-
>next;
    node->previous->next = node->next;
    node->next->previous = node->previous;
    return node;
  }
  node = node->next;
  } while (node != *list);
  return NULL; /* совпадений нет */
}
```

Последняя вспомогательная функция, определенная в листинге 31.2, вызывается, когда процесс не может найти соответствие. В этом случае процесс ставит себя в конец соответствующего списка ожидания и ожидает, пока с ним не будет сопоставлен другой процесс, который его и разбудит. (Цикл вокруг `pthread_cond_wait` защищает от случайных пробуждений, которые возможны в нитях POSIX.) Когда один процесс будит другой, он устанавливает поле `channel` другого процесса в `NULL`. Поэтому, если `p->channel` не равен `NULL`, это означает, что ни один процесс не подошел процессу `p`, и поэтому он должен ждать дальше.

Листинг 31.2. Функция для добавления процесса в список ожидания

```
static void waitonlist (lua_State *L, const char
*channel,
                                      Proc **list) {
  Proc *p = getself(L);

  /* link itself at the end of the list */
  if (*list == NULL) { /* empty list? */
    *list = p;
    p->previous = p->next = p;
  }
  else {
    p->previous = (*list)->previous;
    p->next = *list;
```

```
    p->previous->next = p->next->previous = p;
  }

  p->channel = channel;

  do { /* waits on its condition variable */
  pthread_cond_wait(&p->cond, &kernel_access);
  } while (p->channel);
}
```

Имея эти вспомогательные функции, мы можем написать `send` и `receieve` (листинг 31.3). Функция `send` начинает с проверки канала. Затем она блокирует мьютекс и ищет подходящего получателя. Если она его находит, то она перемещает свои значения в этого получателя, помечает получателя как готового к выполнению и будит его. В противном случае она ставит себя в режим ожидания. По завершении этой операции она разблокирует мьютекс и возвращается без значений в Lua. Функция `receive` аналогична, но она должна вернуть все полученные значения.

Листинг 31.3. Функции для отправки и получения сообщений

```
static int ll_send (lua_State *L) {
  Proc *p;
  const char *channel = luaL_checkstring(L, 1);

  pthread_mutex_lock(&kernel_access);

  p = searchmatch(channel, &waitreceive);

  if (p) {                    /* найден подходящий
ресивер? */
    movevalues(L, p->L);      /* перемещает значения в
получателя */
    p->channel = NULL;        /* помечает получателя как
не ожидающего */
    pthread_cond_signal(&p->cond); /* будит его */
  }
  else
    waitonlist(L, channel, &waitsend);
```

```
  pthread_mutex_unlock(&kernel_access);
  return 0;
}

static int ll_receive (lua_State *L) {
  Proc *p;
  const char *channel = luaL_checkstring(L, 1);
  lua_settop(L, 1);
  pthread_mutex_lock(&kernel_access);

  p = searchmatch(channel, &waitsend);

  if (p) {                    /* найден подходящий
отправитель? */
    movevalues(p->L, L);      /* получает значения от
отправителя */
    p->channel = NULL;        /* помечает отправителя
как не ожидающего */
    pthread_cond_signal(&p->cond); /* будит его */
  }
  else
    waitonlist(L, channel, &waitreceive);

  pthread_mutex_unlock(&kernel_access);

  /* возвращает все значения из стека, кроме канала */
  return lua_gettop(L) - 1;
}
```

Теперь давайте посмотрим, как создавать новые процессы. Новому процессу нужна новая нить POSIX, а новой нити нужно тело для выполнения. Мы определим это тело позже; ниже приведен ее прототип на основе `pthreads`:

```
static void *ll_thread (void *arg);
```

Для создания и запуска нового процесса системе нужно создать новое состояние Lua, начать новую нить, скомпилировать переданный кусок, вызвать его и в конце освободить его ресурсы. Исходная нить выполняет первые три задачи, а новая нить делает

остальное. (Для упрощения обработки ошибок система начинает новую нить только после того, как она успешно скомпилирует переданный кусок.)

Функция `ll_start` создает новый процесс (листинг 31.4). Эта функция создает новое состояние Lua `L1` и компилирует заданный кусок в этом новом состоянии. В случае ошибки функция сообщает о ней исходному состоянию `L`. Затем она создает новую нить (при помощи `pthread_create`) с телом `ll_thread`, передавая новое состояние `L1` как аргумент тела. Вызов `pthread_detach` сообщает системе, что мы не ожидаем никакого окончательного ответа от этой нити.

Листинг 31.4. Функция для создания новых процессов

```
static int ll_start (lua_State *L) {
  pthread_t thread;
  const char *chunk = luaL_checkstring(L, 1);
  lua_State *L1 = luaL_newstate();

  if (L1 == NULL)
    luaL_error(L, "unable to create new state");

  if (luaL_loadstring(L1, chunk) != 0)
    luaL_error(L, "error starting thread: %s",
    lua_tostring(L1, -1));

  if (pthread_create(&thread, NULL, ll_thread, L1) !=
0)
    luaL_error(L, "unable to create new thread");

  pthread_detach(thread);
  return 0;
}
```

Телом каждой новой нити является функция `ll_thread` (листинг 31.5). Она получает свое соответственное состояние Lua (созданное `ll_start`) с уже скомпилированным главным куском в стеке. Новая нить открывает стандартные библиотеки Lua,

открывает библиотеку `lproc` и затем вызывает свой главный кусок. В конце она уничтожает свою условную переменную (которая была создана `luaopen_lproc`) и закрывает свое состояние Lua.

Листинг 31.5. Тело для новых нитей

```
int luaopen_lproc (lua_State *L);

static void *ll_thread (void *arg) {
  lua_State *L = (lua_State *)arg;
  luaL_openlibs(L); /* open standard libraries */
  luaL_requiref(L, "lproc", luaopen_lproc, 1);
  lua_pop(L, 1);
  if (lua_pcall(L, 0, 0, 0) != 0) /* call main chunk
*/
    fprintf(stderr, "thread error: %s",
lua_tostring(L, -1));
  pthread_cond_destroy(&getself(L)->cond);
  lua_close(L);
  return NULL;
}
```

(Обратите внимание на использование `luaL_requiref` для открытия библиотеки `lproc` (Примечание: Эта функция появилась в Lua 5.2). Эта функция в чем-то эквивалентна `require`, но вместо поиска загрузчика использует заданную функцию (в нашем случае `luaopen_lproc`) для открытия библиотеки. После вызова этой открывающей функции, `luaL_requiref` регистрирует результат в таблице `package.loaded`, чтобы при будущих вызовах `require` для библиотеки она не пыталась открыть ее снова. Если последним параметром является true, то она также регистрирует эту библиотеку в соответственной глобальной переменной (в нашем случае `lproc`).)

Последняя функция в нашем модуле, `exit`, довольно проста:

```
static int ll_exit (lua_State *L) {
  pthread_exit(NULL);
  return 0;
}
```

Лишь главному процессу необходимо вызывать эту функцию по завершении, чтобы избежать немедленного окончания всей программы.

Нашим последний шагом является определение открывающей функции для модуля `lproc`. Открывающая функция `luaopen_lproc` (листинг 31.6) должна, как обычно, зарегистрировать функции модуля, но также она должна создать и инициализировать структуру `Proc` выполняемого процесса.

Листинг 31.6. Открывающая функция для модуля **`lproc`**

```
static const struct luaL_reg ll_funcs[] = {
  {"start", ll_start},
  {"send", ll_send},
  {"receive", ll_receive},
  {"exit", ll_exit},
  {NULL, NULL}
};

int luaopen_lproc (lua_State *L) {
  /* создает собственный блок управления */
  Proc *self = (Proc *)lua_newuserdata(L,
sizeof(Proc));
  lua_setfield(L, LUA_REGISTRYINDEX, "_SELF");
  self->L = L;
  self->thread = pthread_self();
  self->channel = NULL;
  pthread_cond_init(&self->cond, NULL);
  luaL_register(L, "lproc", ll_funcs); /* открывает
библиотеку */
  return 1;
}
```

Как я сказал ранее, данная реализация процессов в Lua очень простая. Есть бесконечное число улучшений, которые вы можете произвести. Здесь я кратко расскажу о некоторых из них.

Первым очевидным улучшением будет замена линейного поиска подходящего канала. Прекрасной заменой является использование хэш-таблицы для поиска канала и использование независимых

списков ожидания для каждого канала.

Другое улучшение относится к эффективности создания процесса. Создание нового состояния Lua — это быстрая операция. Однако, открытие всех стандартных библиотек происходит уже не так быстро, а большинству процессов, скорее всего, не понадобятся все стандартные библиотеки. Мы можем избежать затрат на открытие библиотеки при помощи предварительной регистрации библиотек, которую мы обсуждали в разделе 15.1. При данном подходе вместо вызова `luaL_requiref` для каждой стандартной библиотеки мы лишь помещаем функцию, открывающую библиотеку, в таблицу `package.preload`. Если процесс вызовет `require"lib"`, то тогда и только тогда `require` вызовет связанную с ней функцию для открытия библиотеки. Функция `registerlib`, в листинге 31.7, проводит эту регистрацию.

Листинг 31.7. Регистрация библиотек для открытия по запросу

```
static void registerlib (lua_State *L, const char
*name,
                                        lua_CFunction
f) {
  lua_getglobal(L, "package");
  lua_getfield(L, -1, "preload"); /* получает
'package.preload' */
  lua_pushcfunction(L, f);
  lua_setfield(L, -2, name); /* package.preload[name]
= f */
  lua_pop(L, 2); /* выталкивает таблицы 'package' и
'preload' */
}

static void openlibs (lua_State *L) {
  luaL_requiref(L, "_G", luaopen_base, 1);
  luaL_requiref(L, "package", luaopen_package, 1);
  lua_pop(L, 2); /* удаляет результаты из предыдущих
вызовов */
  registerlib(L, "io",      luaopen_io);
  registerlib(L, "os",      luaopen_os);
  registerlib(L, "table",   luaopen_table);
```

```
    registerlib(L, "string", luaopen_string);
    registerlib(L, "math",   luaopen_math);
    registerlib(L, "debug",  luaopen_debug);
  }
```

Открыть основную библиотеку — это всегда хорошая идея. Вам также понадобится библиотека для пакетов; иначе `require` не будет доступна для открытия других библиотек. (У вас даже не будет таблицы `package.preload`.) Все другие библиотеки могут быть необязательными. Поэтому вместо вызова `luaL_openlibs` мы можем вызвать нашу собственную функцию `openlibs` (также показанную в листинге 31.7) при открытии новых состояний. Всякий раз, когда процессу требуется одна на этих библиотек, он запрашивает ее явным образом, и `require` вызовет соответствующую функцию `lunopen_*`.

Другие улучшения включают в себя примитивные коммуникационные функции. Например, было бы удобно задавать лимит времени для `lproc.send` и `lproc.receive`, определяющий, сколько они должны ждать совпадения. В частности, нулевой лимит делал бы эти функции неблокирующими. В нитях POSIX мы можем реализовать данные средства при помощи `pthread_cond_timedwait`.

Упражнения

Упражнение 31.1. Как мы видели, если функция вызывает `lua_yield` (версию без продолжающей функции), то управление возвращается функции, которая ее вызвала, когда нить была вновь возобновлена. Какие значения вызывающая функция получит в качестве результатов того вызова?

Упражнение 31.2. Измените библиотеку `lproc` так, чтобы она могла отправлять и принимать другие примитивные типы, такие как логические значения и числа. (Подсказка: вам нужно изменить лишь

функцию `movevalues`.)

Упражнение 31.3. Реализуйте в библиотеке `lproc` неблокирующую операцию `send`.

ГЛАВА 32

Управление памятью

Lua выделяет все свои структуры данных динамически. Все эти структуры растут по мере надобности и со временем сокращаются или исчезают.

Lua строго следит за своим использованием памяти. Когда мы закрываем состояние Lua, то Lua явно освобождает всю его память. Более того, все объекты внутри Lua подвержены сборке мусора: не только таблицы и строки, но также функции, нити и модули (поскольку на самом деле они являются таблицами).

Способ, которым Lua управляет памятью, удобен для большинства приложений. Однако, для некоторых особых приложений может потребоваться адаптация, например для работы в условиях ограниченного объема памяти или для уменьшения пауз между сборкой мусора до минимума. Lua позволяет осуществлять подобные адаптации на двух уровнях. На нижнем уровне мы можем задать функцию выделения памяти, используемую Lua. На более высоком уровне мы можем задать некоторые параметры для управления сборщиком мусора или можем даже получить над ним прямой контроль. В данной главе мы рассмотрим эти средства.

32.1. Выделяющая функция

Ядро Lua не строит предположений о том, как выделить память. Для выделения памяти оно не вызывает ни `malloc`, ни `realloc`. Вместо этого оно осуществляет все свое выделение и освобождение памяти через одну единственную *выделяющую функцию* (allocation function), которую пользователь должен предоставить при создании состояния Lua.

Функция `luaL_newstate`, которую мы использовали для создания состояний, является вспомогательной функцией, которая создает состояние Lua с выделяющей функцией по умолчанию. Эта функция по умолчанию использует стандартные функции `malloc-realloc-free` из стандартной библиотеки C, которых (должно быть) достаточно для обычных приложений. Однако, можно довольно легко получить полный контроль над выделением памяти в Lua, создав ваше состояние при помощи примитивной функции `lua_newstate`:

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

Эта функция получает два аргумента: выделяющую функцию и пользовательские данные. Состояние, созданное таким образом, осуществляет все выделение и освобождение памяти при помощи вызовов функции `f`. (Даже сама структура `lua_state` выделяется при помощи `f`.)

Тип выделяющей функции `lua_Alioc` определен следующим образом:

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

Первый параметр — это всегда пользовательские данные, которые мы предоставили `lua_newstate`; второй параметр — это адрес блока, который мы хотим заново выделить или освободить; третий параметр — это исходный размер этого блока; и четвертый параметр — это запрашиваемый размер блока.

Lua гарантирует, что если `ptr` не равен `NULL`, то он был ранее выделен с размером `osize`.

Lua использует `NULL` для блоков нулевого размера. Когда `nsize` равен нулю, выделяющая функция должна освободить блок, на который указывает `ptr`, и вернуть `NULL`, который соответствует

блоку запрошенного размера (нулевого). Когда `ptr` равен `NULL`, функция должна выделить и вернуть блок заданного размера; если она не может выделить блок заданного размера, то она должна вернуть `NULL`. Если и `ptr` равен `NULL`, и `nsize` равен нулю, то функция ничего не делает и возвращает `NULL`.

Наконец, когда и `ptr` не равен `NULL`, и `nsize` не равен нулю, функция должна заново выделить этот блок, как это делает `realloc`, и вернуть новый адрес (который может как совпадать, так и отличаться от исходного). Опять же, в случае ошибки она должна вернуть `NULL`. Lua предполагает, что выделяющая функция никогда не дает сбоев в случае, когда новый размер меньше или равен старому размеру. (Lua сжимает некоторые структуры во время сборки мусора и потому не способен при этом восстановиться после ошибок.)

Стандартная выделяющая функция, используемая `luaL_newstate`, имеет следующее определение (извлечено напрямую из файла `lauxlib.c`):

```
void *l_alloc (void *ud, void *ptr, size_t osize,
size_t nsize) {
  if (nsize == 0) {
    free(ptr);
    return NULL;
}
else
  return realloc(ptr, nsize);
}
```

Она допускает, что `free(NULL)` не делает ничего, и что вызов `realloc(NULL,size)` эквивалентен `malloc(size)`. Стандарт ANSI C поддерживает оба этих режима.

Вы можете получить выделяющую функцию из состояния Lua посредством вызова `lua_getallocf`:

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

Если `ud` не равен `NULL`, то функция установит `*ud` в значения

пользовательских данных, используемых для данной выделяющей функции. Вы можете изменить выделяющую функцию для состояния Lua при помощи вызова `lua_setallocf`:

```
void lua_setallocf (lua_State *L, lua_Alloc f, void
*ud);
```

Имейте в виду, что любая новая выделяющая функция будет ответственна за освобождение блоков, выделенных предыдущей функцией. Чаще всего новая функция является оберткой над старой, например, для отслеживания выделений или синхронизации доступа к куче.

Внутри себя Lua не кэширует свободные блоки для их повторного использования. Она предполагает, что кэширование проводит выделяющая функция; хорошие выделяющие функции так и делают. Lua не пытается минимизировать фрагментацию памяти. Исследования показывают, что фрагментация — это больше результат плохой схемы выделения памяти, чем поведения программы; хорошие выделяющие функции не создают сильной фрагментации.

Сложно улучшить хорошо реализованную выделяющую функцию, но иногда вы можете попытаться. Например, Lua дает вам старый размер любого блока при его освобождении или выделении заново. Соответственно, специализированной выделяющей функции не нужно хранить информацию о размере блока, тем самым снижая объем требуемой памяти под каждый блок.

Другой случай, когда вы можете улучшить выделение памяти — это случай многонитевых систем. Такие системы обычно требуют синхронизации для своих выделяющих функций, так как они используют глобальный ресурс (память). Однако, доступ к состоянию Lua также должен быть синхронизирован — или, что еще лучше, ограничен всего одной нитью, как в нашей реализации `lproc` в главе 31. Поэтому если каждое состояние Lua будет выделять память из закрытого пула, выделяющая функция может

избежать расходов на дополнительную синхронизацию.

32.2. Сборщик мусора

До версии 5.0 Lua использовал простой сборщик мусора типа «пометь и почисти» (mark-and-sweep), также называемый сборщиком типа «останови мир» (stop-the-world). Это значит, что время от времени Lua прекращает интерпретировать главную программу для выполнения полного цикла сборки мусора. Каждый такой цикл состоит из трех фаз: *пометка* (mark), *уборка* (cleaning) и *очистка* (sweep).

Lua начинает фазу пометки с того, что помечает как живой свой *корневой набор* (root set), который включает в себя все объекты, к которым Lua имеет прямой доступ: реестр и главная нить. Любой объект, который хранится в живом объекте, доступен программе, и поэтому тоже помечается как живой. Фаза пометки заканчивается, когда все доступные объекты помечены как живые.

Перед началом фазы очистки Lua выполняет фазу уборки, которая связана с финализаторами и слабыми таблицами. Во-первых, она обходит все объекты, помеченные для финализации, в поисках непомеченных объектов. Эти объекты помечаются как живые (возрожденные) и помещаются в отдельный список для использования в фазе финализации. Во-вторых, Lua обходит свои слабые таблицы и удаляет из них все записи, где либо ключ, либо само значение не помечено.

Фаза очистки обходит все объекты Lua. (Чтобы данный обход был возможен, Lua хранит все создаваемые объекты с связанном списке.) Если объект не помечен как живой, Lua собирает его как мусор. В противном случае Lua снимает его пометку для подготовки к следующему циклу. Во время фазы очистки Lua также вызывает финализаторы объектов, которые были отделены во время фазы уборки.

С версией 5.1 Lua получил *инкрементальный сборщик мусора*. Этот сборщик выполняет те же шаги, что и прежний, но во время выполнения ему не нужно «останавливать мир». Вместо этого он выполняется вместе с интерпретатором. Каждый раз, когда интерпретатор выделяет некоторое количество памяти, сборщик мусора выполняет небольшой шаг. Это значит, что во время работы сборщика интерпретатор может изменить доступность объекта. Чтобы обеспечить правильность работы сборщика некоторые операции в интерпретаторе имеют барьеры, которые обнаруживают опасные изменения и поправляют пометки вовлеченных объектов.

API сборщика мусора

Lua предлагает API, который позволяет получить некоторый контроль над сборщиком мусора. Из C мы можем вызвать `lua_gc`:

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

Из Lua мы используем функцию `collectgarbage`:

```
collectgarbage(what [, data])
```

Обе функции предоставляют одну и ту же функциональность. Аргумент `what` (перечислимое значение в C, строка в Lua) определяет, что требуется делать. Возможные варианты:

- `LUA_CGSTOP ("stop")`: останавливает сборщик мусора до другого вызова `collectgarbage` (или до `lua_gc`) с опцией "`restart`".
- `LUA_GSRESTART ("restart")`: перезапускает сборщик мусора.
- `LUA_GCCOLLECT ("collect")`: осуществляет полный цикл сборки мусора, при этом все недоступные объекты собираются и финализируются. Это значение по

умолчанию для `collectgarbage`.

- `LUA_GCSTEP ("step")`: выполняет некоторую работу по сборке мусора. Объем этой работы эквивалентен тому, что сборщик мусора сделает после выделения `data` байт.
- `LUA_GCCOUNT ("count")`: возвращает количество килобайт памяти, используемой Lua в данный момент. Это количество включает в себя «мертвые», но еще не собранные объекты.
- `LUA_GCCOUNTB (`*без аргументов*`)`: возвращает дробную часть числа килобайт памяти, используемой Lua в данный момент. В C следующее выражение возвращает полный объем памяти в байтах (полагая, что это число поместится в `int`):

```
(lua_gc(L, LUA_GCCOUNT, 0) * 1024)
    + lua_gc(L, LUA_GCCOUNTB, 0)
```

  В Lua результат `collectgarbage ("count")` — это число с плавающей точкой, и полное число байт памяти может быть вычислено следующим образом:

```
collectgarbage("count") * 1024
```

  Поэтому у `collectgarbage` нет аналога для этой опции.
- `LUA_GCSETPAUSE ("setpause")`: задает параметр `pause` сборщика мусора. Это значение задается параметром `data` в процентах: когда `data` равен 100, параметр устанавливается на единицу (100%).
- `LUA_GCSETSTEPMUL ("setstepmul")`: задает параметр пошагового множителя (`stepmul`) для сборщика мусора. Эта значение также задано `data` в процентах.

Любой сборщик мусора расплачивается за память процессорным временем. В одном крайнем случае сборщик может вообще не

запуститься. Он совсем не будет расходовать процессорное время за счет огромного потребления памяти. В другом крайнем случае сборщик может производить полный цикл сборки мусора при каждом изменении графа доступности. Такая программа будет использовать самый минимум необходимой ей памяти ценой гигантского потребления процессорного времени. Настоящие сборщики мусора пытаются найти хороший баланс между этими двумя крайностями.

Как и ее выделяющая функция, сборщик мусора Lua достаточно хорош для большинства приложений. Тем не менее, в некоторых случаях сборщик мусора стоит попробовать оптимизировать. Два параметра, — `pause` и `stepmul`, предоставляют некоторое управление поведением сборщика.

Параметр `pause` управляет тем, как долго сборщик ждет между окончанием последней сборки мусора и началом новой. Значение `pause` равное нулю заставит Lua запустить новую сборку мусора сразу по завершении предыдущей. Параметр `pause` в 200% ожидает удвоения использования памяти перед тем, как начать сборку мусора. Вы можете понизить значение `pause`, если хотите пожертвовать процессорным временем ради меньшего потребления памяти. Обычно вы должны держать это значение между 0 и 200%.

Параметр `stepmul` управляет тем, как много работы сборщик мусора выполняет для каждого килобайта выделенной памяти. Чем выше это значение, тем менее инкрементальным становится сборщик мусора. Огромное значение вроде 100 000 000% заставит сборщик мусора вести себя как неинкрементальный сборщик. Значением по умолчанию является 200%. Значения, меньшие 100%, сделают сборщик настолько медленным, что он может так никогда и не завершить сборку мусора.

Другие опции `lua_gc` дают вам контроль над тем, когда запускается сборщик мусора. Игры являются типичными клиентами для данного вида контроля. Например, если вы не хотите, чтобы сборка мусора выполнялась во время определенных периодов

времени, вы можете остановить его при помощи `collectgarbage("stop")` и затем запустить снова при помощи `collectgarbage("restart")`. В системах, где возникают постоянные периоды ожидания, вы можете держать сборщик мусора остановленным и вызывать `collectgarbage("step",n)` лишь в эти периоды. Чтобы определить, как много работы нужно выполнить во время такого периода, либо экспериментально подберите подходящее значение для `n`, либо в цикле вызывайте `collectgarbage` с `n`, равным нулю, до самого истечения этого периода ожидания.

Упражнения

Упражнение 32.1. Напишите библиотеку, которая позволит скрипту ограничить общий объем памяти, используемый состоянием в Lua. Она может предлагать единственную функцию `setlimit` для задания этого ограничения.

Библиотека должна задавать свою собственную выделяющую функцию. Эта функция перед вызовом исходной выделяющей функции проверяет общий объем используемой памяти и возвращает `NULL`, если запрашиваемый объем превысит максимальное значение.

(Подсказка: эта библиотека может использовать `lua_gc` для инициализации своего счетчика памяти при запуске. Она также может использовать пользовательские данные выделяющей функции, чтобы хранить свое состояние: число байт, текущее ограничение объема памяти и т. п. Не забудьте использовать исходные пользовательские данные при вызове исходной выделяющей функции.)

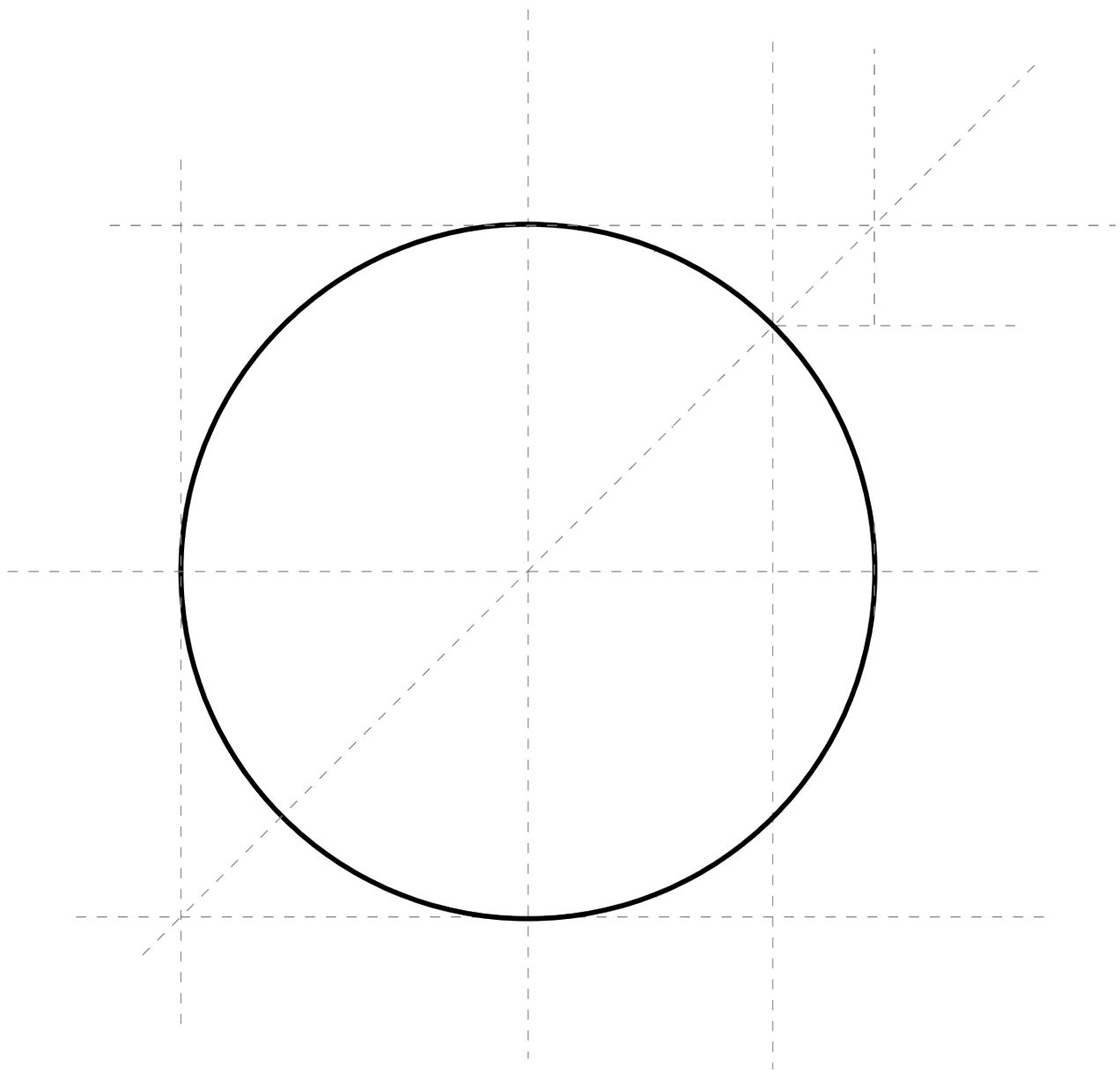
Упражнение 32.2. Для этого упражнения вам понадобится как минимум один скрипт Lua, использующий очень много памяти. Если у вас такого нет, то напишите его. (Он может быть простым,

наподобие цикла, создающего таблицы.)

- Запустите ваш скрипт с различными параметрами сборщика мусора. Насколько они влияют на быстродействие?
- Что случится, если вы установите параметр `pause` на ноль? Что случится, если вы установите его на 1000?
- Что случится, если вы установите параметр `stepmul` на ноль? Что случится, если вы установите его на 1 000 000?
- Поправьте ваш скрипт таким образом, чтобы он получил полный контроль над сборщиком мусора. Он должен держать сборщик мусора остановленным и вызывать его время от времени.
- Можете ли вы повысить быстродействие вашего скрипта с данным подходом?

Приложения

Lua 5.3



ПРИЛОЖЕНИЕ А

Переход на Lua 5.3

А.1. Изменения в языке

Главным отличием между Lua 5.2 и Lua 5.3 является введение подтипа целых чисел — **integer**. Хотя данное изменение не должно повлиять на «обычные» вычисления, некоторые из них (главным образом те, что содержат какое-либо переполнение) могут выдавать другие результаты.

Вы можете убрать влияние этих отличий путем принудительного выставления подтипа чисел с плавающей точкой **float** (в Lua 5.2 все числа были с плавающей точкой), например, записав константы с окончанием `.0` или воспользовавшись `x = x + 0.0` для преобразования переменной. (Данная рекомендация служит лишь для быстрого исправления редко встречающейся несовместимости подтипов; она не является нормой в хорошем программировании. Для хорошего программирования используйте **float** там, где вам нужны числа с плавающей точкой, и **integer** там, где вам нужны целые числа.

Преобразование числа с плавающей точкой в строку теперь добавляет суффикс `.0` к результату, если этот результат выглядит как целое число. (Например, число `2.0` будет напечатано как `2.0`, а не как `2`.) Вы всегда должны явно указывать формат, когда вам нужен конкретный формат для чисел.

(Формально это не является изменением, поскольку Lua не определяет, каким образом числа форматируются как строки, но некоторые программы ожидают конкретный формат.)

Из сборщика мусора был убран накопительный режим (generational mode). (Он был экспериментальной возможностью в

Lua 5.2.)

A.2. Изменения в библиотеках

Библиотека `bit32` была исключена. Несложно подобрать совместимую внешнюю библиотеку или, что еще лучше, заменить ее функции соответственными побитовыми операциями. (Имейте в виду, что `bit32` оперирует 32-битными целыми числами, т.е. частью **float**, в то время как побитовые операции в Lua 5.3 оперируют целыми числами Lua, у которых по умолчанию 64 бит.)

Библиотека `table` теперь распознает метаметоды для установки и получения элементов.

Итератор `ipairs` теперь распознает метаметоды, а его метаметод `__ipairs` был исключен.

Необязательным именам в `io.read` больше не нужно начинаться с `'*'`. Для совместимости Lua продолжить принимать (и игнорировать) данный символ.

Следующие функции были исключены из математической библиотеки: `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 остальных операций вы можете либо воспользоваться внешней библиотекой, либо реализовать их в Lua самостоятельно.

Искатель загрузчиков модулей C, используемый `require`, изменил способ обработки имен с номером версии. Теперь версия должна идти после имени модуля (как это принято в большинстве других инструментов). Для совместимости этот искатель по-прежнему пользуется старым форматом, когда он не может найти открывающую функцию, соответствующую этому новому стилю. (Lua 5.2 уже работал таким образом, но это изменение не было документировано.)

Вызов `collectgarbage("count")` теперь возвращает только один результат. (Вы можете вычислить второй результат из дробной части первого.)

A.3. Изменения в API

Продолжающие функции теперь принимают все, что им нужно, в виде параметров, и не нуждаются в `lua_getctx`, поэтому `lua_getctx` была убрана. Внесите соответственные изменения в ваш код.

У функции `lua_dump` появился дополнительный параметр с именем `strip`. Используйте `0` как значение данного параметра, чтобы вернуть прежнее поведение.

Функции для инъекции-проекции беззнаковых целых чисел (`lua_pushunsigned`, `lua_tounsigned`, `lua_tounsignedx`, `luaL_checkunsigned`, `luaL_optunsigned`) были исключены. Используйте их эквиваленты со знаком посредством приведения типов.

Макросы для проекции нестандартных целочисленных типов (`luaL_checkint`, `luaL_optint`, `luaL_checklong`, `luaL_optlong`) были исключены. Используйте их эквивалент посредством `lua_Integer` с приведением типа (или, если есть возможность, используйте в вашем коде `lua_Integer`).

ПРИЛОЖЕНИЕ Б

Новое в Lua 5.3

Б.1. Язык

Новое в языке:

- целые числа, по умолчанию 64-битные;
- официальная поддержка 32-битных чисел;
- побитовые операции;
- базовая поддержка UTF-8;
- целочисленное деление;
- пользовательские данные могут содержать любое значение Lua в качестве пользовательского.

Целые числа

У типа **number** два внутренних представления, или два подтипа, один называется **integer**, а другой **float**. У Lua есть строго обозначенные правила о том, когда применять каждое из этих представлений, но при этом он может автоматически переводить их друг в друга. Таким образом, у программиста есть выбор: или по большей части не обращать внимание на разницу между целыми и вещественными числами, или получить полный контроль над представлением каждого числа. Стандартная реализация Lua использует для подтипов **integer** и **float** 64-битные числа.

Официальная поддержка 32-битных чисел

Возможно скомпилировать Lua так, чтобы он по умолчанию использовал 32-битные целые числа и/или 32-битные числа с плавающей точкой (одинарной точности). Опция с одновременным включением 32 бит для целых чисел и чисел с плавающей точкой особенно привлекательна для микрокомпьютеров и встраиваемых систем. Смотрите макрос `LUA_32BITS` внутри файла `luaconf.h`.

Побитовые операции

Lua поддерживает следующие побитовые операции:

'&' побитовое И
'|' побитовое ИЛИ
'~' побитовое взаимоисключающее ИЛИ
'>>' сдвиг вправо
'<<' сдвиг влево
'~' унарное побитовое НЕ

Все побитовые операции преобразуют свои операнды в целые числа, оперируют всеми битами этих целых чисел и выдают результат в виде целого числа.

Сдвиги вправо и влево заполняют освободившиеся биты нулями. Отрицательные смещения приводят к сдвигу в обратном направлении; смещения с абсолютными значениями, которые равны или больше числа битов в целом числе, дают ноль (так как при этом задвигаются все биты).

Базовая поддержка UTF-8

Библиотека `utf8` в виде таблицы обеспечивает базовую поддержку кодировки UTF-8. Она не предлагает никакой поддержки Юникода кроме выполнения преобразования. Любая операция, которой нужно значение символа, например его классификация, лежит вне ее компетенции.

Если только не указано иначе, все функции, которые ожидают позицию байта в качестве параметра, считают, что заданная позиция либо является началом байтовой последовательности, либо равна длине обрабатываемой строки, увеличенной на единицу. Как и в строковой библиотеке, отрицательные индексы отсчитываются от конца строки.

`utf8.char (···)`

Получает ноль или более целых чисел, переводит каждое из них в его соответственную последовательность байтов для UTF-8 и возвращает строку, в которой все эти последовательности соединены.

`utf8.charpattern`

Данный образец (т.е. это строка, а не функция) равен "`[\0-\x7F\xC2-\xF4][\x80-\xBF]*`", что соответствует ровно одной байтовой последовательности UTF-8 при условии, что субъект является допустимой строкой UTF-8.

`utf8.codes (s)`

Возвращает значения таким образом, что конструкция

```
for p, c in utf8.codes(s) do body end
```

переберет все символы в строке `s`, где `p` является позицией (в байтах), а `c` является кодовой точкой каждого символа. Вызывает ошибку, если встречает любую недопустимую последовательность байтов.

`utf8.codepoint (s [, i [, j]])`

Возвращает кодовые точки (как целые числа) из всех символов в `s`, которые находятся между байтовыми позициями `i` и `j` (включительно). По умолчанию `i` равна `1`, а `j` равна `i`. Вызывает ошибку, если встречает любую недопустимую последовательность байтов.

`utf8.len (s [, i [, j]])`

Возвращает число символов UTF-8 в строке `s`, которые находятся между байтовыми позициями `i` и `j` (включительно). По умолчанию i равна `1`, а `j` равна `-1`. Если находит любую недопустимую последовательность байтов, то возвращает false и позицию первого недопустимого байта.

`utf8.offset (s, n [, i])`

Возвращает позицию (в байтах), с которой начинается кодировка символа строки `s` под номером `n` (считая от позиции `i`). Отрицательный `n` получает символы до позиции `i`. По умолчанию `i` равна `1`, когда `n` не является отрицательным, и `#s + 1` в остальных случаях, поэтому `utf8.offset(s, -n)` получит смещение `n`-ного символа от конца строки. Если заданный символ не находится внутри строки или сразу после ее конца, функция возвращает nil.

Как особый случай, когда `n` равен 0, функцию возвращает начало кодировки символа, который содержит `i`-ый по счету байт строки `s`.

Данная функция полагается на то, что `s` является допустимой строкой UTF-8.

Целочисленное деление

Для целочисленного деления введена новая операция — '`//`', а операция '`/`' служит для деления вещественного.

Вещественное деление ('`/`') всегда преобразует свои операнды в **float** и дает в результате **float**.

Целочисленное деление ('`//`') всегда преобразует свои операнды в **integer** и дает в результате **integer**. Преобразование происходит посредством округления операндов в меньшую сторону, как это делает функция `math.floor`.

Б.2. Библиотеки

Новое в библиотеках:

- итератор `ipairs` распознает метаметоды;
- библиотека `table` распознает метаметоды;
- опция `strip` в `string.dump`;
- функция `table.move`;
- функции `string.pack`, `string.unpack`, `string.packsize`.

Опция `strip` в string.dump

```
string.dump (function [, strip])
```

Возвращает строку, содержащую бинарное представление (бинарный кусок) заданной функции `function`, чтобы позже функция `load`, примененная к этой строке, вернула копию этой функции (но с новыми верхними значениями). Если параметр `strip` равен true, то бинарное представление может не включать всю отладочную информацию о функции, чтобы сэкономить место.

Функция `table.move`

```
table.move (a1, f, e, t [,a2])
```

Перемещает элементы из таблицы `a1` в таблицу `a2`. Эта функция делает то же, что и следующее множественное присваивание: `a2[t],··· = a1[f],···,a1[e]`. По умолчанию `a2` равна `a1`. Диапазон конечной таблицы `a2` может перекрывать диапазон исходной таблицы `a1`. Количество перемещаемых элементов должно укладываться в целое число Lua.

Функции `string.pack`, `string.unpack`, `string.packsize`

```
string.pack (fmt, v1, v2, ···)
```

Возвращает бинарную строку, содержащую значения `v1`, `v2` и т.д., упакованные (т.е. сериализованные в бинарной форме) в соответствии с форматирующей строкой `fmt` (см. ниже).

`string.unpack (fmt, s [, pos])`

Возвращает значения, упакованные в строке `s` (при помощи `string.pack`) в соответствии с форматирующей строкой `fmt` (см. ниже). Необязательный параметр `pos` отмечает место, с которого начинается чтение строки `s` (по умолчанию равен 1). После чтения значений эта функция также возвращает индекс первого непрочитанного байта в `s`.

`string.packsize (fmt)`

Возвращает размер строки, которая получается в результате применения `string.pack` с заданным форматом. Форматирующая строка `fmt` (см. ниже) не может иметь опции переменной длины '`s`' или '`z`'.

Форматирующая строка `fmt` является первым аргументом вышеперечисленных функций. Она описывает разбивку создаваемой или читаемой структуры. Форматирующая строка является последовательностью конверсионных опций. Конверсионные опции следующие:

'`<`' задает порядок байтов от младшего к старшему (little endian)
'`>`' задает порядок байтов от старшего к младшему (big endian)
'`=`' задает нативный порядок байтов (native endian)
'`![n]`' задает максимальное выравнивание для `n` (по умолчанию выравнивание нативное)
'`b`' **signed byte** (один символ, т.е. **char**)
'`B`' **unsigned byte** (один символ, т.е. **char**)
'`h`' **signed short** (нативный размер)
'`H`' **unsigned short** (нативный размер)
'`l`' **signed long** (нативный размер)

'L' **unsigned long** (нативный размер)
'j' lua_Integer
'J' lua_Unsigned
'T' size_t (нативный размер)
'i[n]' **signed int** размером n байт (по умолчанию нативный размер)
'I[n]' **unsigned int** размером n байт (по умолчанию нативный размер)
'f' **float** (нативный размер)
'd' **double** (нативный размер)
'n' lua_Number
'cn' строка фиксированного размера размером n байт
'z' нуль-завершенная строка
's[n]' строка, перед которой стоит ее длина, закодированная в виде **unsigned integer**, равная n байт (по умолчанию n равен size_t)
'x' один байт для создания отступа
'Xop' пустой элемент, который варавнивает в соответствии с опцией op (в остальных случаях игнорируется)
' ' (пустое пространство) игноририруется

(Элемент "[n]" означает необязательный целочисленный нумерал. За исключением отступов, пробелов и кофигураций (опции "xX <=>!"), каждая опция отвечает за свой аргумент (в string.pack) или результат (в string.unpack).

Для опций "!n", "sn", "in" и "In", n может быть любым целым числом между 1 и 16. Все целочисленные опции производят проверку на переполнение; string.pack проверяет, помещается ли заданное значение в заданный размер; string.unpack проверяет, помещается ли прочитанное значение в целое число Lua.

Любая форматирующая строка начинается, как если бы она была предварена "!1=", то есть с максимальным варавниванием, равным 1 (отсутствие выравнивания) и нативным порядком байтов.

Выравнивание работает следующим образом: Для каждой опции формт вставляет дополнительные отступы, пока данные не станут начинаться со смещения, которое кратно минимуму между размером опции и максимальным выравниванием; этот минимум должен быть степенью 2. Опции "c" и "z" не выравниваются; опция "s" следует выравниванию своего начального целого числа.

Все отступы заполняются нулями функцией `string.pack` (и игнорируются `string.unpack`).

Б.3. C API

Новое в C API:

- более простой API для продолжающих функций в C;
- функция `lua_gettable` и подобные ей возвращают тип итогового значения;
- опция `strip` в `lua_dump`;
- функции `lua_geti`, `lua_seti`, `lua_isyieldable`, `lua_numbertointeger`, `lua_rotate`, `lua_stringtonumber`.

Опция `strip` в lua.dump

```
int lua_dump (lua_State *L, lua_Writer writer, void *data, int strip);
```

Сбрасывает на диск функцию в виде бинарного куска. Получает функцию Lua с вершины стека и производит бинарный кусок, который, если загружен повторно, дает функцию, эквивалентную ранее сброшенной. По мере производства частей куска, `lua_dump` вызывает записывателя функций (см. в справочнике `lua_Writer`) с указанными данными для записи этих частей.

Если `strip` равен true, бинарное представление может не включать в себя какую-либо отладочную информацию о функции, чтобы сэкономить место.

Возвращаемое значение — это код ошибки, возвращенной последним вызовом записывателя; 0 означает отсутствие ошибок.

Данная функция не выталкивает функцию Lua из стека.

Функции

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

Заталкивает в стек значение `t[i]`, где `t` — это значение в заданном индексе. Как и в Lua, данная функция может привести к срабатыванию метаметода для события "`index`".

Возвращает тип вставленного в стек значения.

```
void lua_seti (lua_State *L, int index, lua_Integer n);
```

Выполняет действия, эквивалентные `t[n] = v`, где `t` — это значения в заданном индексе, а `v` — это значение на вершине стека.

Данная функция вытакливает значение из стека. Как и в Lua, данная функция может привести к срабатыванию метаметода для события "`newindex`".

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

Возвращает 1, если заданная сопрограмма может уступить управление, или 0 в ином случае.

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

Преобразует число Lua из **float** в **integer**. Данный макрос ожидает, что `n` обладает целочисленным значением. Если его значение лежит внутри диапазона целых чисел Lua, то оно преобразуется в целое число и присваивается `*p`. Этот макрос возвращает булево значение, указывающее на то, было ли успешно преобразование. (Обратите внимание, что без этого макроса в связи с округлениями данная проверка диапазона может не всегда давать ожидаемый результат.)

Данный макрос может вычислять свои аргументы более одного раза.

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

Циклически сдвигает элементы стека между допустимым индексом `idx` и вершиной стека. Элементы сдвигаются на `n` позиций в направлении вершины в случае положительного `n`, или на `-n` позиции в направлении основания в случае отрицательного `n`. Абсолютное значение `n` не должно быть больше размера сдвигаемой секции. Данная функция не может быть вызвана для псевдоиндекса, поскольку псевдоиндекс не является действительной позицией в стеке.

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

Преобразует нуль-завершенную строку `s` в число, заталкивает это число в стек и возвращает итоговый размер этой строки, т.е. ее длину, увеличенную на 1. Данное преобразование может выдать либо целое число, либо **integer**, либо **float**, в зависимости от лексических соглашений Lua. Эта строка может содержать знак, а также пробелы в начале и в конце. Если она не является допустимым нумералом, будет возвращен 0 и заталкивания не произойдет. (Обратите внимание, что результат может быть использован как булево значение, например, true, если преобразование пройдет успешно.)

Б.4. Автономный интерпретатор Lua

Новое в автономном интерпретаторе:

- Может быть использован как калькулятор без необходимости предварять выражение знаком '=';
- Таблица `arg` доступна всему коду.