

От создателя языка Lua

Программирование на языке **Lua**

Роберту Иерузалимски



Роберту Иерузалимски

Программирование на языке Lua

Третье издание

Programming in Lua

Third Edition

ROBERTO IERUSALIMSKY

PUC-RIO, BRAZIL
RIO DE JANEIRO

Программирование на языке Lua

Третье издание

РОБЕРТУ ИЕРУЗАЛИМСКИ



Москва, 2014

УДК 004.432.42Lua
ББК 32.973.28-018.1
ИЗО

ИЗО Иерусалимски Р.

Программирование на языке Lua. 3-е издание / пер. с англ. А. В. Боресков – М.: ДМК Пресс, 2014. – 382 с.: ил.

ISBN 978-5-94074-767-3

Книга посвящена одному из самых популярных встраиваемых языков – Lua. Этот язык использовался во многих играх и большом количестве различных приложений. Язык сочетает небольшой объем занимаемый памяти, высокое быстродействие, простоту использования и большую гибкость. Книга рассматривает практически все аспекты использования Lua, начиная с основ языка и заканчивая тонкостями расширения языка и взаимодействия с C.

Важной особенностью книги является огромный спектр охватываемых тем – практически все, что может понадобиться при использовании Lua. Также к каждой главе дается несколько упражнений, позволяющих проверить свои знания.

Книга будет полезна широкому кругу программистов и разработчиков игр. Для понимания последних глав книги необходимо знание языка C, но для большинства остальных глав достаточно базовых знаний о программировании.

УДК 004.432.42Lua
ББК 32.973.28-018.1

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-85-903798-5-0 (англ.)
ISBN 978-5-94074-767-3 (рус.)

© 2013, 2003 by Roberto Ierusalimsky
© Оформление, перевод на русский язык
ДМК Пресс, 2014



ОГЛАВЛЕНИЕ

Введение	12
Аудитория	14
О третьем издании	15
Другие ресурсы	16
Некоторые типографские соглашения	17
Запуск примеров	17
Благодарности	18
 ЧАСТЬ I	
Язык.....	19
Глава 1. Начинаем	20
1.1. Блоки	20
1.2. Некоторые лексические соглашения	22
1.3. Глобальные переменные	24
1.4. Отдельный интерпретатор	24
Упражнения	26
Глава 2. Типы и значения.....	27
2.1. Nil	28
2.2. Boolean (логические значения)	28
2.3. Числа	28
2.4. Строки	30
Литералы	30
Длинные строки	32
Приведения типов	33
2.5. Таблицы	34
2.6. Функции.....	38
2.7. userdata и нити	38
Упражнения	39
Глава 3. Выражения	40
3.1. Арифметические операторы	40
3.2. Операторы сравнения.....	41
3.3. Логические операторы.....	42

3.4. Конкатенация.....	43
3.5. Оператор длины.....	43
3.6. Приоритеты операторов	45
3.7. Конструкторы таблиц	46
Упражнения	48
Глава 4. Операторы.....	49
4.1. Операторы присваивания	49
4.2. Локальные переменные и блоки.....	50
4.3. Управляющие конструкции	52
if then else	53
while	53
repeat	54
Числовой оператор for.....	54
Оператор for общего вида	55
4.4. break, return и goto	57
Упражнения	60
Глава 5. Функции	62
5.1. Множественные результаты	64
5.2. Функции с переменным числом аргументов	68
5.3. Именованные аргументы	70
Упражнения	72
Глава 6. Еще о функциях	73
6.1. Замыкания	75
6.2. Неглобальные функции	79
6.3. Оптимизация хвостовых вызовов	82
Упражнения	83
Глава 7. Итераторы и обобщенный for	85
7.1. Итераторы и замыкания	85
7.2. Семантика обобщенного for	87
7.3. Итераторы без состояния	89
7.4. Итераторы со сложным состоянием.....	91
7.5. Подлинные итераторы (true iterators).....	93
Упражнения	94
Глава 8. Компиляция, выполнение и ошибки	96
8.1. Компиляция	96
8.2. Предкомпилированный код.....	100
8.3. Код на C	102
8.4. Ошибки	103
8.5. Обработка ошибок и исключений	105

8.6. Сообщения об ошибках и стек вызовов	106
Упражнения	108
Глава 9. Сопрограммы	110
9.1. Основы сопрограмм	110
9.2. Каналы и фильтры	113
9.3. Сопрограммы как итераторы	117
9.4. Невытесняющая многонитевость	119
Упражнения	124
Глава 10. Законченные примеры	125
10.1. Задача о восьми королевах	125
10.2. Самые часто встречающиеся слова	127
10.3. Цепь Маркова	129
Упражнения	131
ЧАСТЬ II	
Таблицы и объекты	133
Глава 11. Структуры данных	134
11.1. Массивы	134
11.2. Матрицы и многомерные массивы	135
11.3. Связанные списки	137
11.4. Очереди и двойные очереди	138
11.5. Множества и наборы	139
11.6. Строчные буферы	141
11.7. Графы	142
Упражнения	144
Глава 12. Файлы данных и персистентность	146
12.1. Файлы с данными	146
12.2. Сериализация	148
Сохранение таблиц без циклов	151
Сохранение таблиц с циклами	152
Упражнения	155
Глава 13. Метатаблицы и метаметоды	156
13.1. Арифметические метаметоды	157
13.2. Метаметоды сравнения	160
13.3. Библиотечные метаметоды	161
13.4. Метаметоды для доступа к таблице	162
Метаметод <code>__index</code>	163
Метаметод <code>__newindex</code>	164
Таблицы со значениями по умолчанию	165

Отслеживание доступа к таблице	166
Таблицы, доступные только для чтения.....	168
Упражнения	169
Глава 14. Окружение	170
14.1. Глобальные переменные с динамическими именами.....	170
14.2. Описания глобальных переменных.....	172
14.3. Неглобальные окружения	174
14.4. Использование <code>_ENV</code>	176
14.5. <code>_ENV</code> и <code>load</code>	179
Упражнения	181
Глава 15. Модули и пакеты	182
15.1. Функция <code>require</code>	184
Переименовывание модуля.....	185
Поиск по пути	186
Искатели файлов.....	187
15.2. Стандартный подход для написания модулей на Lua	188
15.3. Использование окружений	190
15.4. Подмодули и пакеты.....	192
Упражнения	193
Глава 16. Объектно-ориентированное программирование	195
16.1. Классы	197
16.2. Наследование	199
16.3. Множественное наследование	201
16.4. Скрытие	203
16.5. Подход с единственным методом	205
Упражнения	206
Глава 17. Слабые таблицы и финализаторы	208
17.1. Слабые таблицы.....	208
17.2. Функции с кэшированием	210
17.3. Атрибуты объекта.....	212
17.4. Опять таблицы со значениями по умолчанию	213
17.5. Эфемерные таблицы.....	215
17.6. Финализаторы	216
Упражнения.....	220
ЧАСТЬ III	
Стандартные библиотеки.....	221
Глава 18. Математическая библиотека.....	222

Упражнения	223
Глава 19. Библиотека для побитовых операций ...	224
Упражнения	227
Глава 20. Библиотека для работы с таблицами	228
20.1. Функции insert и remove	228
20.2. Сортировка	229
20.3. Конкатенация	230
Упражнения	231
Глава 21. Библиотека для работы со строками.....	232
21.1. Основные функции для работы со строками	232
21.2. Функции для работы с шаблонами	235
Функция string.find	235
Функция string.match	236
Функция string.gsub	236
Функция string.gsub	237
21.3. Шаблоны	238
21.4. Захваты	242
21.5. Замены	245
Кодировка URL	246
Замена табов	248
21.6. Хитрые приемы	249
21.7. Юникод	252
Упражнения	255
Глава 22. Библиотека ввода/вывода	256
22.1. Простая модель ввода/вывода	256
22.2. Полная модель ввода/вывода	260
Небольшой прием для увеличения быстродействия	261
Бинарные файлы	262
22.3. Другие операции над файлами	264
Упражнения	266
Глава 23. Библиотека функций операционной системы.....	267
23.1. Дата и время	267
23.2. Другие вызовы системы	270
Упражнения	271
Глава 24. Отладочная библиотека.....	272
24.1. Возможности по доступу (интроспекции)	272
Доступ к локальным переменным	275
Доступ к нелокальным переменным	276

Доступ к другим сопрограммам	277
24.2. Ловушки (hooks)	278
24.3. Профилирование	279
Упражнения	282

ЧАСТЬ IV

С API..... 283

Глава 25. Обзор С API 284

25.1. Первый пример	286
25.2. Стек	288
Помещение элементов на стек	290
Обращение к элементам	291
Другие операции со стеком	294
25.3. Обработка ошибок в С API	295
Обработка ошибок в коде приложения	296
Обработка ошибок в коде библиотек	296
Упражнения	297

Глава 26. Расширение вашего приложения 298

26.1. Основы	298
26.2. Работа с таблицами	300
26.3. Вызовы функций на Lua	305
26.4. Обобщенный вызов функции	307
Упражнения	309

Глава 27. Вызываем С из Lua 310

27.1. Функции на С	310
27.2. Продолжения	313
27.3. Модули на С	316
Упражнения	318

Глава 28. Приемы написания функций на С 320

28.1. Работа с массивами	320
28.2. Работа со строками	322
28.3. Сохранение состояния в функциях на С	326
Реестр	326
Значения, связанные с функцией	329
Значения, связанные с функцией, используемые несколькими функциями	332
Упражнения	333

Глава 29. Задаваемые пользователем типы в С ... 334

29.1. Пользовательские данные (userdata)	335
29.2. Метатаблицы	337

29.3. Объектно-ориентированный доступ	340
29.4. Доступ как к обычному массиву	342
29.5. Легкие объекты типа userdata (light userdata)	344
Упражнения	345
Глава 30. Управление ресурсами	346
30.1. Итератор по каталогу	346
30.2. Парсер XML	349
Упражнения	358
Глава 31. Нити и состояния	360
31.1. Многочисленные нити	360
31.2. Состояния Lua	365
Упражнения	373
Глава 32. Управление памятью	374
32.1. Функция для выделения памяти	374
32.2. Сборщик мусора	377
API сборщика мусора	378
Упражнения	380



ВВЕДЕНИЕ

Когда Вальдемар, Луис и я начали разработку Lua в 1993 году, мы с трудом могли себе представить, что Lua так распространится. 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.

Мы разработали Lua с самого начала для интегрирования с программным обеспечением, написанным на C/C++ и других распространенных языках. Эта интеграция несет с собой много преимуществ. Lua – это крошечный и простой язык, частично из-за того, что он не пытается делать то, в чем уже хорош C, например быстрое действие, низкоуровневые операции и взаимодействие с программами третьих сторон. Для этих задач Lua полагается на C. Lua предлагает то, для чего C недостаточно хорош: достаточная удаленность от аппаратного обеспечения, динамические структуры, отсутствие избыточности и легкость тестирования и отладки. Для этих целей Lua располагает безопасным окружением, автоматическим управлением памятью и хорошими возможностями для работы со строками и другими типами данных с изменяемым размером.

Часть силы Lua идет от его библиотек. И это не случайно. В конце концов, одной из главных сил Lua является расширяемость. Многие особенности языка вносят в это свой вклад. Динамическая типизация предоставляет большую степень полиморфизма. Автоматическое управление памятью упрощает интерфейсы, поскольку нет необходимости решать, кто отвечает за выделение и освобождение памяти или как обрабатывать переполнения. Функции высших порядков и анонимные функции позволяют высокую степень параметризации, делая функции более универсальными.

В большей степени, чем расширяемым языком, Lua является «склеивающим» (*glue*) языком. Lua поддерживает компонентный подход к разработке программного обеспечения, когда мы создаем приложе-

ние, склеивая вместе существующие высокоуровневые компоненты. Эти компоненты написаны на компилируемом языке со статической типизацией, таком как C/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 занимают основную часть времени выполнения итоговой программы. Lua придает итоговую форму приложению, которая, скорее всего, сильно меняется во время жизни данного программного продукта. Однако, в отличие от других «склеивающих» технологий, Lua является полноценным языком программирования. Поэтому мы можем использовать Lua не только для «склеивания» компонентов, но и для адаптации и настройки этих компонентов, а также для создания полностью новых компонентов.

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

- *Расширяемость.* Расширяемость Lua настолько велика, что многие рассматривают Lua не как язык, а как набор для построения DSL (domain-specific language, язык, созданный для определенной области, применения). Мы разрабатывали Lua с самого начала, чтобы он был расширяемым как через код на Lua, так и через код на C. Как доказательство Lua реализует большую часть своей базовой функциональности через внешние библиотеки. Взаимодействие 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. Он основан на небольшом числе понятий. Эта простота облегчает изучение. Lua вносит свой вклад в то, что его размер очень мал. 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 – один из самых быстрых языков среди скриптовых языков.

- *Портируемость.* Когда мы говорим о портируемости, мы говорим о запуске 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, большие серверы от IBM, RISC OS, Symbian OS, процессоры Rabbit, Raspberry Pi, Arduino и многое другое. 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 с ANSI C, то вам просто нужно откомпилировать Lua.

Аудитория

Пользователи Lua обычно относятся к одной из трех широких групп: те, кто используют Lua, уже встроенный в приложение, те, кто используют Lua отдельно от какого-либо приложения (standalone), и те, кто используют Lua и C вместе.

Многие используют Lua, встроенный в какое-либо приложение, например в 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 для своей области. Часто пользователи такого приложения даже не знают, что Lua – это независимый язык, адаптированный под данную область. Например, многие разработчики плагинов для Lightroom не знают о других использованиях этого языка; пользователи Nmap обычно рассматривают Lua как скриптовый язык Nmap; игроки в World of Warcraft могут рассматривать Lua как язык исключительно для данной игры.

Lua также полезен и как просто независимый язык, не только для обработки текста и одноразовых маленьких программ, но также и для различных проектов от среднего до большого размера. Для подобного использования основная функциональность Lua идет от ее библиотек. Стандартные библиотеки, например, предоставляют базовую функциональность по работе с шаблонами и другие функции для работы со строками. По мере того как Lua улучшает свою поддержку библиотек, появилось большое количество внешних пакетов. Lua Rocks, система для сборки и управления модулями для Lua, сейчас насчитывает более 150 пакетов.

Наконец, есть программисты, которые используют Lua как библиотеку для C. Такие люди больше пишут на C, чем на Lua, хотя им требуется хорошее понимание Lua для создания интерфейсов, которые являются простыми, легкими для использования и хорошо интегрированными с языком.

Эта книга может оказаться полезной всем этим людям. Первая часть покрывает сам язык, показывая, как можно использовать 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 покрывают базовые понятия, такие как управляющие структуры, в то время как остальные главы покрывают 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 сохранение (persistence), пакеты и объектно-ориентированное программирование. Именно там мы покажем всю силу языка.

Третья часть представляет стандартные библиотеки. Эта часть особенно полезна для тех, кто использует Lua как самостоятельный язык, хотя многие приложения включают частично или полностью стандартные библиотеки. В этой части каждой библиотеке посвящена отдельная глава: математической библиотеке, побитовой библиотеке, библиотеке по работе с таблицами, библиотеке по работе со строками, библиотеке ввода/вывода, библиотеке операционной системы и отладочной библиотеке.

Наконец, последняя часть книги покрывает API между Lua и C. Эта часть заметно отличается от всей остальной книги. В этой части мы будем программировать на C, а не на Lua. Для кого-то эта часть может оказаться неинтересной, а для кого-то – наоборот, самой полезной частью книги.

О третьем издании

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

Во-первых, я обновил всю книгу на Lua 5.2. Глава об окружениях (environments) была практически полностью переписана. Я также переписал несколько примеров для того, чтобы показать преимущества

использования новых возможностей, предоставляемых Lua 5.2. Однако я четко обозначил отличия от Lua 5.1, поэтому вы можете использовать книгу для работы с этой версией языка.

Во-вторых, и более важно, я добавил упражнения во все главы книги. Сложность этих упражнений варьируется от простых вопросов до полноценных небольших проектов. Некоторые примеры иллюстрируют важные аспекты программирования на Lua и так же, как и примеры, расширят ваш набор полезных приемов.

Так же, как мы поступили с первым и вторым изданиями «Programming in Lua», мы сами опубликовали третье издание. Несмотря на маркетинговые ограничения, этот подход обладает рядом преимуществ: мы сохраняем полный контроль над содержимым книги; мы сохраняем все права для предложения книги в других формах; мы обладаем свободой для выбора, когда выпустить следующее издание; мы можем быть уверены, что книга всегда будет доступна.

Другие ресурсы

Краткое руководство необходимо всем, кто хочет освоить язык. Эта книга не заменяет краткое руководство. Напротив, они дополняют друг друга. Руководство только описывает Lua. Оно не показывает ни примеров, ни объяснений для конструкций языка. С другой стороны, оно полностью описывает язык: эта книга опускает некоторые, редко используемые, «темные углы» Lua. Более того, руководство описывает язык. Там, где эта книга расходится с руководством, доверяйте руководству. Чтобы получить руководство и дополнительную информацию по Lua, посетите веб-сайт <http://www.lua.org>.

Вы также можете найти полезную информацию на сайте пользователей Lua, поддерживаемом сообществом пользователей Lua, <http://lua-users.org>. Помимо других ресурсов, он предлагает также обучающий курс (tutorial), список сторонних пакетов и документации, архив официальной рассылки по Lua.

Эта книга описывает Lua 5.2, хотя большая часть содержимого также применима к Lua 5.1 и Lua 5.0. Некоторые отличия Lua 5.2 от предыдущих версий Lua 5 четко обозначены в книге. Если вы используете более свежую версию (выпущенную после этой книги), обратитесь к руководству по поводу отличий между версиями. Если вы используете версию старше, чем 5.2, то, может, пора подумать о переходе на более новую версию.

Некоторые типографские соглашения

В этой книге строки символов (“literal string”) заключены в двойные кавычки, а одиночные символы, например ‘a’, заключены в одиночные кавычки. Строки, которые являются шаблонами, также заключены в одиночные кавычки, например ‘[%w_]*’. В книге моноширинный шрифт используется для фрагментов кода и идентификаторов. Для **зарезервированных слов** используется жирный шрифт. Большие фрагменты кода показаны с применением следующего стиля:

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

Обозначение --> показывает результат выполнения оператора или результат выражения:

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

Поскольку двойной знак минус (--) начинает комментарий в Lua, ничего не будет если вы включите такой результат вывода (вместе с -->) в свою программу. Наконец, в книге используется обозначение <--> для обозначения того, что что-то эквивалентно чему-то другому:

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

Запуск примеров

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

Сайт Lua (<http://www.lua.org>) хранит весь исходный код для интерпретатора. Если у вас есть компилятор с C и знание того, как скомпилировать C код на вашем компьютере, то вам лучше попробовать поставить Lua из исходного кода; это действительно легко. Сайт *Lua Binaries* (поищите luabinaries) предлагает уже откомпилированные интерпретаторы для всех основных платформ. Если вы используете Linux или другую UNIX-подобную систему, вы можете проверить репозиторий вашего дистрибутива; многие дистрибутивы уже предлагают готовые пакеты с Lua. Для Windows хорошим выбором является *Lua for Windows* (поищите luaforwindows), являющийся

удобным набором для работы с Lua. Он включает в себя интерпретатор, интегрированный редактор и много библиотек.

Если вы используете Lua, встроенный в приложение, как WoW или Nmap, то вам может понадобиться руководство по данному приложению (или помощь «местного гуру»), для того чтобы разобраться, как запускать ваши программы. Тем не менее Lua остается все тем же языком; большинство примеров, которые мы рассмотрим в этой книге, применимы, несмотря на то, как вы используете Lua. Однако я рекомендую начать изучение Lua с интерпретатора для запуска ваших примеров.

Благодарности

Прошло уже почти десять лет с тех пор, как я опубликовал первое издание этой книги. Многие друзья и различные организации помогли мне в этом пути.

Как всегда, Луиг Хенрик де Фигуредо и Вальдемар Селес, соавторы Lua, предлагали все варианты помощи. Андре Карригал, Аско Кауппи, Бретт Капилик, Диего Нехаб, Эдвин Морагас, Фернандо Джефферсон, Гэвин Врес, Джон Д. Рамсделл и Норман Ремси предложили неоценимые замечания и полезные взгляды для различных изданий этой книги.

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

Lightning Source, Inc. предложило надежный и эффективный вариант для печати и распространения данной книги. Без них самим издать эту книгу не получилось бы.

Центр латино-американских исследований в Стенфордском 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.

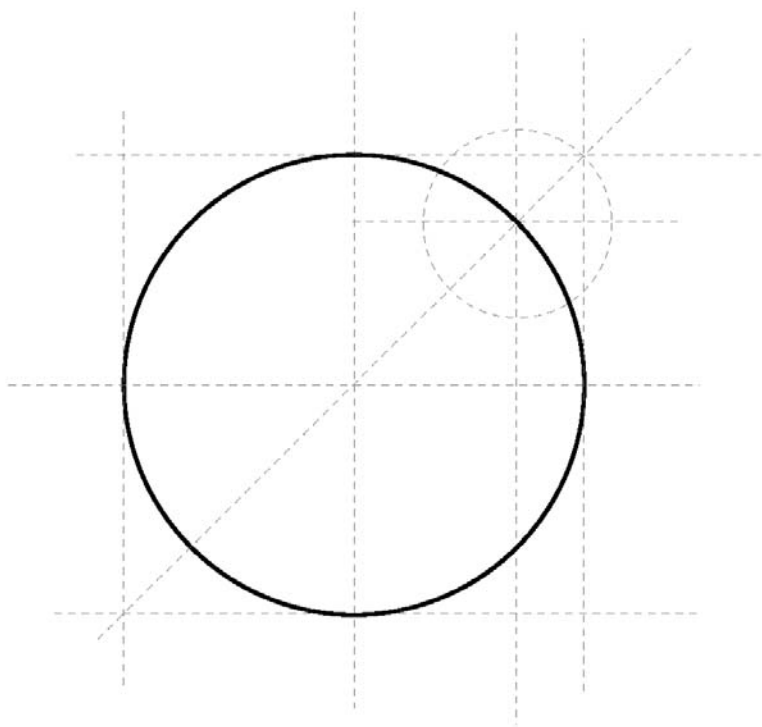
Я также хотел бы поблагодарить Pontifical Catholic University Рио де Жанейро (PUC-Rio) и Бразильский национальный исследовательский совет (CNPq) за их продолжающуюся поддержку моей работы.

Наконец, я должен выразить мою глубокую благодарность Ноэми Родригес за все виды помощи (технической, и не только) и за освещение моей жизни.



Часть I

Язык





ГЛАВА 1

Начинаем

Продолжая традицию, наша первая программа на Lua просто печатает "Hello World":

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

Если вы используете отдельный интерпретатор Lua, то все, что вам надо для запуска вашей первой программы, – это запустить интерпретатор – обычно он называется `lua` или `lua5.2` – с именем текстового файла, содержащего вашу программу. Если вы сохранили приведенную выше программу в файле `hello.lua`, то вам следует выполнить следующую команду:

```
% lua hello.lua
```

Как более сложный пример наша следующая программа определяет функцию для вычисления факториала заданного числа, спрашивает у пользователя число и печатает его факториал:

```
-- defines a factorial function
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") -- reads a number
print(fact(a))
```

1.1. Блоки

Каждый кусок кода, который Lua выполняет, такой как файл или отдельная строка в интерактивном режиме, называется *блоком* (chunk). Блок – это просто последовательность команд (или операторов).

Lua не нужен разделитель между подряд идущими операторами, но вы можете использовать точку с запятой, если хотите. Я лично использую точку с запятой только для разделения операторов, записанных в одной строке. Разбиение на строки не играет никакой роли в синтаксисе Lua; так, следующие четыре блока допустимы и эквивалентны:

```
a = 1
b = a*2
a = 1;
b = a*2;
a = 1; b = a*2
a = 1 b = a*2 -- ugly, but valid
```

Блок может состоять всего из одного оператора, как в примере «Hello World», или состоять из набора операторов и определений функций (которые на самом деле являются просто присваиваниями, как мы увидим позже), как в примере с факториалом. Блок может быть так велик, как вы хотите. Поскольку Lua используется также как язык для описания данных, блоки в несколько мегабайт не являются редкостью. Интерпретатор Lua не имеет каких-либо проблем при работе с большими блоками.

Вместо того чтобы записывать ваши программы в файл, вы можете запустить интерпретатор в интерактивном режиме. Если вы запустите lua без аргументов, то вы увидите его приглашения для ввода:

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

Соответственно, каждая команда, которую вы вводите (как, например, `print "Hello World"`), выполняется немедленно, после того как вы ее введете. Для выхода из интерпретатора, просто наберите символ конца файла (`ctrl-D` в UNI, `ctrl-Z` в Windows) или позовите функцию `exit` из библиотеки операционной системы – вам нужно набрать `os.exit()`.

В интерактивном режиме Lua обычно интерпретирует каждую строку, которую вы вводите, как отдельный блок. Однако если он обнаруживает, что строка не является законченным блоком, то он ждет продолжения ввода до тех пор, пока не получится законченный блок. Таким образом вы можете вводить многострочные определения, такие как функция `factorial`, прямо в интерактивном режиме. Однако обычно более удобным является помещать подобные определения в файл и затем вызывать Lua для выполнения этого файла.

Вы можете использовать опцию `-i` для того, чтобы заставить Lua перейти в интерактивный режим после выполнения заданного блока:

```
% lua -i prog
```

Команда вроде этой выполнит блок в файле `prog` и затем перейдет в интерактивный режим. Это особенно полезно для отладки и ручного тестирования. В конце данной главы мы рассмотрим другие опции командной строки для интерпретатора Lua.

Другим способом запускать блоки является функция `dofile`, которая немедленно выполняет файл. Например, допустим, что 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") -- load your library
> n = norm(3.4, 1.0)
> print(twice(n)) --> 7.0880180586677
```

Функция `dofile` также полезна, когда вы тестируете фрагмент кода. Вы можете работать с двумя окнами: в одном находится текстовый редактор с вашей программой (например, в файле `prog.lua`), и в другом находится консоль с запущенным интерпретатором Lua в интерактивном режиме. После того как вы сохранили изменения в вашей программе, вы выполняете `dofile("prog.lua")` в консоли для загрузки нового кода; затем вы можете начать использование нового кода, вызывая функции и печатая результаты.

1.2. Некоторые лексические соглашения

Идентификаторы (или имена) в Lua являются строками из латинских букв, цифр и знака подчеркивания, не начинающимися с цифры; например:



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

Вам лучше избегать идентификаторов, состоящих из подчеркивания, за которым следуют заглавные латинские буквы (например, `_VERSION`); они зарезервированы для специальных целей в Lua. Обычно я использую идентификатор `_` (одинокое подчеркивание) для пустых (dummy) переменных.

В старых версиях Lua понятие того, что является буквой, зависело от локали. Однако подобные буквы делают вашу программу неподходящей для запуска на системах, которые не поддерживают данную локаль. Поэтому Lua 5.2 рассматривает в качестве букв только буквы из следующих диапазонов: A-Z и a-z.

Следующие слова зарезервированы, вы не можете 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, однако **And** и **AND** – это два разных идентификатора.

Комментарий начинается с двух знаков минуса (`--`) и продолжается до конца строки. Lua также поддерживает блочный комментарий, который начинается с `--[[` и идет до следующего `]]`¹. Стандартный способ закомментировать фрагмент кода – это поместить его между `--[[` и `--]]`, как показано ниже:

```
--[[
print(10) -- no action (commented out)
--]]
```

Для того чтобы снова сделать этот код активным, просто добавьте один минус к первой строке:

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

В первом примере `--[[` в первой строке начинает блочный комментарий, и двойной минус в последней строке также находится в этом комментарии. Во втором примере `---[[` начинает обычный

¹ Блочные комментарии могут быть более сложными, как мы увидим в разделе 2.4.

ный однострочный комментарий, поэтому первая и последняя строки становятся обычными независимыми комментариями. В этом случае `print` находится вне комментариев.

1.3. Глобальные переменные

Глобальным переменным не нужны описания; вы их просто используете. Не является ошибкой обратиться к неинициализированной переменной; вы просто получите значение `nil` в качестве результата:

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

Если вы присвоите `nil` глобальной переменной, то Lua поведет себя, как будто эта переменная никогда не была использована:

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

После этого присваивания Lua может со временем вернуть себе память, занимаемую данной переменной.

1.4. Отдельный интерпретатор

Отдельный (stand-alone) интерпретатор (также называемый `lua.c` в связи с названием его исходного файла или просто `lua` по имени выполняемого файла) – это небольшая программа, которая позволяет непосредственное использование Lua. В этой секции представлены ее основные опции.

Когда интерпретатор загружает файл, то он пропускает первую строку, если она начинается с символа `#!/`. Это позволяет использовать Lua как скриптовый интерпретатор в UNIX-системах. Если вы начнете ваш скрипт с чего-нибудь вроде

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

(предполагая, что интерпретатор находится в `/usr/local/bin`) или

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

то вы можете непосредственно запускать ваш скрипт без явного запуска интерпретатора Lua.

Интерпретатор вызывается следующим образом:

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

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

Опция `-e` позволяет непосредственно задать код прямо в командной строке, как показано ниже:

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

(UNIX требует двойных кавычек, чтобы командный интерпретатор (shell) не разбирал скобки).

Опция `-l` загружает библиотеку. Как мы уже видели ранее, `-i` переводит интерпретатор в интерактивный режим после обработки остальных аргументов. Таким образом, следующий вызов загрузит библиотеку `lib`, затем выполнит присваивание `x=10` и наконец перейдет в интерактивный режим.

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

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

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

Эта 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 ее значение имеет вид *@имяфайла*, то интерпретатор запускает данный файл. Если `LUA_INIT_5_2` (или `LUA_INIT`) определена, но не начинается с символа '@', то интерпретатор предполагает, что она содержит выполнимый код на Lua и выполняет его. `LUA_INIT` дает огромные возможности по конфигурированию интерпретатора, поскольку при конфигурировании нам доступна вся мощь Lua. Мы можем загрузить пакеты, изменить текущий путь, определить свои собственные функции, переименовать или уничтожить функции и т. п.

Скрипт может получить свои аргументы в глобальной переменной `arg`. Если у нас есть вызов вида `%lua script a b c`, то интерпретатор создает таблицу `arg` со всеми аргументами командной строки перед

выполнением скрипта. Имя скрипта расположено по индексу 0, первый аргумент (в примере это "a") расположен по индексу 1 и т. д. Предшествующие опции расположены по негативным индексам, поскольку они расположены перед именем скрипта. Например, рассмотрим следующий 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 при помощи выражения ... (три точки). В главной части скрипта это выражение дает все аргументы скрипта (мы обсудим подобные выражения 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 быстрее?

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 языке нет определений типов, каждое значение несет свой собственный тип.

В Lua существует восемь базовых типов: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread* и *table*. Функция `type` возвращает тип для любого переданного 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 всегда вернет **string** вне зависимости от значения `X`, поскольку результат функции `type` всегда является строкой.

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

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

Обратите внимание на последние две строки: функции являются значения первого класса в Lua; ими можно манипулировать, как и любыми другими значениями. (Больше об этом будет рассказано в главе 6.)

Обычно когда вы используете одну и ту же переменную для значений разных типов, вы получаете отвратительный код. Однако иногда разумное использование этой возможности оказывается полезным, например использование **nil** для того, чтобы отличать нормальное возвращаемое значение от какой-либо ошибки.

2.1. Nil

Nil – это тип, состоящий из всего одного значения, **nil**, основной задачей которого является отличаться от всех остальных значений. Lua использует nil для обозначения отсутствующего значения. Как мы уже видели, глобальные переменные по умолчанию имеют значение nil до своего первого присваивания, вы также можете присвоить nil глобальной переменной, чтобы удалить ее.

2.2. Boolean (логические значения)

Тип boolean имеет два значения, **true** и **false**, которые служат для представления традиционных логических значений. Однако эти значения не монополизировать все условные значения: в Lua любое значение может представлять условие (condition). Соответствующие проверки (проверки условия в различных управляющих структурах) трактуют оба значения **nil** и **false** как ложные и все остальные значения как истинные. В частности, Lua трактует ноль и пустую строку как истину в логических условиях.

Во всей книге под ложным значением будет подразумеваться **nil** и **false**. В случае когда речь идет именно о булевых значениях, будет явно использовано значение **false**.

2.3. Числа

Тип number представляет значения с плавающей точкой, заданные с двойной точностью. В Lua нет встроенного целочисленного типа.

Некоторые опасаются, что даже такие простые операции, как увеличение на единицу (инкремент) и сравнение, могут некорректно работать с числами с плавающей точкой. Однако на самом деле это не так. Практически все платформы сейчас поддерживают стандарт IEEE 754 для представления чисел с плавающей точкой. Согласно этому стандарту, единственным возможным источником ошибок является случай, когда число не может быть точно представлено. Операция округляет свой результат, только если результат не может быть точно представлен в виде соответствующего значения с плавающей точкой. Любая операция, результат которой может быть точно представлен, будет иметь точное значение.

На самом деле любое целое число вплоть до 2^{53} (приблизительно 10^{16}) имеет точное представление в виде числа с плавающей точкой с двойной точностью (double). Когда вы используете значение с плавающей точкой с двойной точностью для представления целых чисел, нет никаких ошибок округления, за исключением случая, когда значение по модулю превосходит 2^{53} . В частности, Lua способен представлять любые 32-битовые целые значения без проблем с округлениями.

Конечно, дробные числа будут иметь проблемы с округлением. Эта ситуация не отличается от случая, когда у вас есть бумага и ручка. Если мы хотим записать $1/7$ в десятичном виде, то мы где-то должны остановиться. Если мы используем десять цифр для представления числа, то $1/7$ станет 0.142857142 . Если мы вычислим $1/7 * 7$ с десятью цифрами, то мы получим 0.999999994 , что отличается от 1. Более того, числа, которые имеют конечное представление в виде десятичных дробей, могут иметь бесконечное представление в виде двоичных дробей. Так, $12.7 - 20 + 7.3$ не равно нулю, поскольку оба числа 12.7 и 7.3 не имеют точного двоичного представления (см. упражнение 2.3).

Прежде чем мы продолжим, запомните, целые числа имеют точное представление и поэтому не имеют ошибок с округлением.

Большинство современных CPU выполняет операции с плавающей точкой так же быстро (или даже быстрее), чем с целыми числами. Тем не менее легко скомпилировать Lua так, чтобы для числовых значений использовался другой тип, например длинные целочисленные значения или числа с плавающей точкой с одинарной точностью. Это особенно полезно для платформ без аппаратной поддержки чисел с плавающей точкой, таких как, например, встроенные системы. За деталями обратитесь к файлу `luaconf.h` в исходных файлах Lua.

Мы можем записывать числа, при необходимости указывая дробную часть и десятичную степень. Примерами допустимых числовых констант являются:

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

Более того, мы можем также использовать шестнадцатеричные константы, начиная их с `0x`. Начиная с Lua 5.2 шестнадцатеричные константы также могут иметь дробную часть и двоичную степень (перед степенью ставится `'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 ее десятичное представление.)

2.4. Строки

Строки в Lua имеют обычное значение: последовательность символов. Lua поддерживает все 8-битовые символы, и строки могут содержать символы с любыми кодами, включая нули. Это значит, что вы можете хранить любые бинарные данные в виде строк. Вы также можете хранить юникодные строки в любом представлении (UTF-8, UTF-16 и т. д.). Стандартная библиотека, которая идет вместе с Lua, не содержит встроенной поддержки для этих представлений. Тем не менее вы вполне можете работать с UTF-8 строками, что мы рассмотрим в разделе 21.7.

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

```
a = "one string"
b = string.gsub(a, "one", "another") -- изменим часть строки
print(a) --> one string
print(b) --> another string
```

Строки в Lua подвержены автоматическому управлению памятью, так же как и другие объекты Lua (таблицы, функции и т. д.). Это значит, что вам не надо беспокоиться о выделении и освобождении строк; этим за вас займется Lua. Строка может состоять из одного символа или целой книги. Программы, работающие со строками в 100К или 10М символов, — не редкость в Lua.

Вы можете получить длину строки, используя в качестве префикса оператор `#` (называемый оператором длины):

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

Литералы

Мы можем помещать строки внутри одиночных или двойных кавычек:

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

Эти виды записи эквиваленты; единственным отличием является то, что внутри строки, ограниченной одним типом кавычек, вы можете непосредственно вставлять кавычки другого типа.

Обычно большинство программистов использует кавычки одного типа для одного и того же типа строк. Например, библиотека, которая работает с XML, может использовать одиночные кавычки для строк, содержащих фрагменты XML, поскольку эти фрагменты часто содержат двойные кавычки.

Строки в Lua могут содержать следующие escape-последовательности:

<code>\a</code>	звонок (bell)
<code>\b</code>	back space
<code>\f</code>	перевод страницы (form feed)
<code>\n</code>	новая строка (newline)
<code>\r</code>	возврат каретки (carriage return)
<code>\t</code>	таб (horizontal tab)
<code>\v</code>	вертикальный таб (vertical tab)
<code>\\</code>	backslash
<code>\"</code>	двойная кавычка (double quote)
<code>\'</code>	одинарная кавычка (single quote)

Следующий пример иллюстрирует 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 можем задать символ в строке при помощи его числового значения, используя конструкции вида `\ddd` и `\x`, где `ddd` – это последовательность не более чем из трех десятичных цифр, а `hh` – последовательность ровно из двух шестнадцатеричных цифр. В качестве сложного примера две строки `"a\o\n123\\""` и `"\97\o\10\04923\\""` обладают одним и тем же значением в системе, использующей ASCII: 97 – это ASCII-код для `'a'`, 10 – это код для символа перевода строки, и 49 – это код для цифры `'1'` (в этом примере мы должны записать значение 49 при помощи трех десятичных цифр `\049`, поскольку за ним следует другая цифра; иначе Lua трактовал это как код 492). Мы можем также записать ту же самую строку как `"\x61\x6c\x6f\x0a\x31\x32\x33\x22"`, представляя каждый символ его шестнадцатеричным значением.

Длинные строки

Мы можем ограничивать символьные строки при помощи двойных квадратных скобок, как мы делали это с комментариями. Строка в этой форме может занимать много строк, и управляющие последовательности в этих строках не будут интерпретироваться. Более того, эта форма игнорирует первый символ строки, если это символ перехода на следующую строку. Эта форма особенно удобна для написания строк, содержащих большие фрагменты кода, как показано ниже:

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

Иногда вы можете захотеть поместить в строку что-то вроде `a=b[c[i]]` (обратите внимание на `]]` в этом коде) или вы можете захотеть поместить в строку часть кода, где какой-то фрагмент уже закомментирован. Для работы с подобными случаями вы можете поместить любое количество знаков равенства между двумя открывающими квадратными скобками, например `[===[`. После этого строка завершится только на паре закрывающих квадратных скобок с тем же самым количеством знаков равенства (`]===]` для нашего примера). Сканер будет игнорировать пары скобок с другим количеством знаков равенства. Путем выбора подходящего количества знаков равенства вы можете заключить в строку любой фрагмент.

То же самое верно и для комментариев. Например, если вы начинаете длинный комментарий с `--[=`, то он будет продолжаться вплоть до `]=]`. Эта возможность позволяет закомментировать любой фрагмент кода, содержащий уже закомментированные фрагменты.

Длинные строки очень удобны для включения текста в ваш код, но вам не следует использовать их для нетекстовых строк. Хотя строки в Lua могут содержать любые символы, это не очень хорошая идея — использовать эти символы в своем коде: вы можете столкнуться с проблемами с вашим текстовым редактором; более того, строки вида `"\r\n"` могут превратиться в `"\n"`. Поэтому для представления про-

извольных бинарных данных лучше использовать управляющие последовательности, начинающиеся с символа "\", такие как "\x13\x01xA1\xBB". Однако это представляет проблему для длинных строк из-за получающейся длины.

Для подобных ситуаций Lua 5.2 предлагает управляющую последовательность \z: она пропускает все символы в строке до первого непробельного символа. Следующий пример иллюстрирует его использование:

```
data = "\x00\x01\x02\x03\x04\x05\x06\x07\z
        \x08\x09\x0A\x0B\x0C\x0D\x0E\x0F"
```

Находящийся в конце первой строки \z пропускает последующий конец строки и индентацию следующей строки так, что за байтом \x07 сразу же следует байт \x08 в получающейся строке.

Приведения типов

Lua предоставляет автоматическое преобразование значений между строками и числами на этапе выполнения. Любая числовая операция, примененная к строке, пытается преобразовать строку в число:

```
print("10" + 1)      --> 11
print("10 + 1")      --> 10 + 1
print("-5.3e-10"*"2") --> -1.06e-09
print("hello" + 1)   -- ERROR (cannot convert "hello")
```

Lua применяет подобные преобразования не только в арифметических операторах, но также и в других местах, где ожидается число, например для аргумента `math.sin`.

Аналогично, когда Lua ожидает получить строку, а получает число, он преобразует число в строку:

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

(Оператор `..` служит в Lua для конкатенации строк. Когда вы его записываете сразу после числа, то 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 преобразования 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 типы, несмотря на все эти преобразования. Сравнение вроде `10=="10"` дает ложное значение, поскольку `10` – это число, а `"10"` – это строка.

Если вам нужно явно преобразовать строку в число, то вы можете использовать функцию `tonumber`, которая возвращает `nil`, если строка не содержит число:

```
line = io.read()           -- прочесть строку
n = tonumber(line)         -- попробовать перевести 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

Тип таблицы соответствует ассоциативному массиву. Ассоциативный массив – это массив, который можно индексировать не только числами, но и строками или любым другим значением из языка, кроме `nil`.

Таблицы являются главным (на самом деле единственным) механизмом структурирования данных в Lua, притом очень мощным. 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 «возьми из таблицы `io` значение по ключу `read`».

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 ссылкой (указателем) на него. Lua никогда не прибегает к скрытому копированию или созданию новых таблиц. Более того, вам даже

не нужно объявлять таблицу в Lua; на самом деле даже нет способа объявить таблицу. Вы создаете таблицы при помощи специального выражения, которое в простейшем случае выглядит как `{}`;

```
a = {}          -- создать таблицу и запомнить 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 -- увеличить запись "x"
print(a["x"])    --> 11
```

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

```
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      -- только 'b' по-прежнему ссылается на таблицу
b = nil      -- на таблицу не осталось ссылок
```

Когда в программе не остается больше ссылок на таблицу, сборщик мусора в Lua со временем уничтожит таблицу и переиспользует ее память.

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

```
a = {}          -- пустая таблица
-- создать 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 в случае глобальных переменных, неинициализированные поля таблицы возвращают **nil**. Так же как и для глобальных переменных, вы можете присвоить полю таблицы **nil**, для того чтобы его уничтожить. 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 следующий «синтак-

сический сахар»: вместо `a["name"]` вы можете писать `a.name`. Таким образом, мы можем переписать последние несколько строк предыдущего примера 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 полностью эквивалентны и могут быть свободно использованы. Для читателя, однако, каждая форма может сообщать об определенном намерении. Запись через точку ясно показывает, что мы используем таблицу как запись (структуру), где у нас есть определенный набор заданных, предопределенных ключей. Другая запись подталкивает к мысли о том, что таблица может использоваться в качестве ключа любую строку и по какой-то причине в данном месте мы работаем с конкретным ключом.

Часто встречающаяся ошибка новичков заключается в том, что они путают `a.x` и `a[x]`. Первая форма на самом деле соответствует `a["x"]`, то есть обращению к таблице с ключом `"x"`. Во втором случае в качестве ключа выступает значение переменной `x`. Ниже показана разница:

```
a = {}
x = "y"
a[x] = 10    -- записать 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 просто инициализируете те элементы, которые вам нужны:

```
-- прочесть 10 строк, запоминая их в таблице
a = {}
for i = 1, 10 do
    a[i] = io.read()
end
```

Поскольку вы можете индексировать таблицу по любому значению, вы можете начать индексы в массиве 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 может быть где-то записана. Обычно мы записываем длину списка в поле с нечисловым ключом; по историческим причинам некоторые программы используют для этих целей поле "n".

Часто, однако, длина явно не задается. Поскольку любому неинициализированному полю соответствует значение **nil**, то мы можем использовать это значение для определения конца списка. Например, если вы прочли десять строк в список, то легко запомнить, что его длина равна 10, поскольку его ключами являются числа 1, 2, ..., 10. Этот подход работает только со списками, в которых нет *дыр*, которые содержат значение **nil**. Мы называем подобные списки *последовательностями* (sequence).

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

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

Поскольку мы можем индексировать таблицу значениями 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 соответствуют разным элементам таблицы. Аналогично строки "+1", "01" и "1" также соответствуют разным элементам таблицы. Когда вы не уверены насчет типа ваших индексов, 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[t tonumber(j)]) --> one value
print(a[t tonumber(k)]) --> one value
```

Если не обращать внимания на эти тонкости, то легко внести в программу трудно находимые ошибки.

2.6. Функции

Функции являются значениями первого класса в Lua: программы могут записывать функции в переменные, передавать функции как аргументы для других функций и возвращать функции как результат. Подобная возможность придает огромную гибкость языку; программа может переопределить функцию, чтобы добавить новую функциональность, или просто удалить функцию для создания безопасного окружения для выполнения фрагмента ненадежного кода (например, кода, полученного по сети). Более того, Lua предоставляет хорошую поддержку функционального программирования, включая вложенные функции с соответствующим лексическим окружением; просто подождите до главы 6. Наконец, функции первого класса играют важную 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 используем функции, написанные на C, для того чтобы получить высокое быстродействие и доступ к возможностям, недоступным непосредственно из Lua, таким как доступ к средствам операционной системы. Все стандартные библиотеки в Lua написаны на C. Они включают в себя функции для работы со строками, работы с таблицами, ввод/вывод, доступ к базовым возможностям операционной системы, математические функции и отладку.

Мы обсудим функции Lua в главе 5 и функции на C в главе 27.

2.7. userdata и нити

Тип `userdata` позволяет запоминать произвольные данные языка C в переменных Lua. У этого типа нет встроенных операций, за исключением присваивания и проверки на равенство. Значения данного типа используются для представления новых типов, созданных приложением 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.

Тип *нить* (thread) будет рассмотрен 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. Что из приведенного ниже является допустимыми числами? 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 все числа являются десятичными. Можете ли вы представить его как значение двоичной дроби? А число 5.5?

Упражнение 2.4. Как вы запишете следующий фрагмент XML в строку Lua?

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

Используйте как минимум два разных способа.

Упражнение 2.5. Допустим, вам нужно записать длинную последовательность произвольных байт как строковую константу в Lua. Как вы это сделаете? Обратите внимание на 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 представляют значения. Выражения в Lua включают числовые константы и строковые литералы, переменные, унарные и бинарные операции и вызовы функций. Выражения также включают в себя нестандартные определения функций и конструкторы для таблиц.

3.1. Арифметические операторы

Lua поддерживает стандартные арифметические операторы: бинарные `+` (сложение), `-` (вычитание), `*` (умножение), `/` (деление), `^` (возведение в степень), `%` (остаток от деления) и унарный `-` (изменение знака). Все из них работают с числами с плавающей точкой. Например, `x^0.5` вычисляет квадратный корень из `x`, а `x^(-1/3)` вычисляет обратное значение к кубическому корню из `x`.

Следующее правило определяет оператор остатка от деления:

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

Для целочисленных операндов у него стандартное значение, и результат имеет тот же знак, что и второй операнд. Для вещественных операндов у него есть некоторые дополнительные возможности. Например, `x%1` дает дробную часть `x`, а `x-x%1` дает его целую часть. Аналогично `x-x%0.01` дает `x` точно с двумя десятичными знаками после запятой:

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

В качестве другого примера использования оператора остатка от деления рассмотрим следующий пример: допустим, вы хотите узнать, будет ли транспортное средство после поворота на заданный угол сдвигаться в обратном направлении. 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 вы хотите работать с радианами вместо градусов, мы просто изменим константы в функциях:

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

Все, что нам нужно, — это операция `angle%(2*math.pi)` для приведения любого угла к интервалу $[0, 2\pi)$.

3.2. Операторы сравнения

Lua предоставляет следующие операторы сравнения:

< > <= >= == ~=

Все эти операторы всегда дают булево значение.

Оператор `==` проверяет на равенство; оператор `~=` — это отрицание равенства. Мы можем использовать оба этих оператора к любым двум значениям. Если значения имеют различные типы, то Lua считает, что они не равны. В противном случае Lua сравнивает их соответственно их типу. Значение **nil** равно только самому себе.

Lua сравнивает таблицы и объекты типа `userdata` 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 операторы порядка лишь к паре чисел или паре строк. Lua сравнивает строки в алфавитном порядке, следуя установленной для Lua локали. Например, для португальской локали Latin-1 мы получим `"acai" < "açai" < "acorde"`. Значения типов, от-

личных от строк и чисел, могут быть сравнены только на равенство (и неравенство).

При сравнении значений различных типов нужно быть аккуратным: помните, что "0" отличается от 0. Более того, $2 < 15$ очевидно истинно, но $2 < "15"$ ложно. В случае, когда вы пытаетесь сравнить строку и число, например $2 < "15"$, возникает ошибка.

3.3. Логические операторы

Логическими операторами являются **and**, **or** и **not**. Как и управляющие конструкции, логические операторы трактуют **false** и **nil** как ложные, а все остальные – как истинные значения. Оператор **and** возвращает свой первый операнд, если он ложен, иначе он возвращает свой второй операнд. Оператор **or** возвращает свой первый операнд, если он не ложен; иначе он возвращает свой второй операнд:

```
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
```

Оба оператора (**and** и **or**) использует сокращенное вычисление, то есть они вычисляют свой второй операнд, только когда это необходимо. Это гарантирует, что выражения вроде `(type(v)=="table" and v.tag=="h1")` не вызовут ошибок при их вычислении: Lua не будет пытаться вычислить `v.tag`, когда `v` не является таблицей.

Полезной конструкцией в Lua является `x=x or v`, эквивалентная следующему коду:

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

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

Другой полезной конструкцией является `(a and b) or c` или просто `a and b or c`, поскольку у оператора **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 следующего оператора:

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

Когда `x > y`, то первое выражение в операторе **and** истинно, поэтому он возвращает свое второе значение (`x`), которое всегда ис-

тинно (поскольку это число), и затем оператор **or** возвращает свой первый операнд, *x*. Если выражение *x* > *y* ложно, то результат оператора **and** также ложен, и поэтому оператор **or** возвращает свой второй операнд, *y*.

Оператор **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 оператор конкатенации как `..` (две точки). Если операнд является числом, то Lua переведет его в строку. (Некоторые языки используют для конкатенации оператор `'+'`, 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. Оператор конкатенации всегда создает новую строку, не изменяя своих операндов:

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

3.5. Оператор длины

Оператор длины работает со строками и таблицами. Со строками он дает количество байт в строке. С таблицами он возвращает длину *последовательности*, представленной таблицей.

С оператором длины связано несколько распространенных идиом для работы с последовательностями.

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

Как мы видели в предыдущей главе, оператор длины непредсказуем для списков с дырками (*nil*). Он работает только для по-

следовательностей, которые мы определили как списки без дырок. Более точно *последовательность* – это таблица, где ключи образуют последовательность 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 длины ноль.

С годами было много предложений по расширению значения оператора длины на списки с дырками, но это легче сказать, чем сделать. Проблема в том, что поскольку список – это таблица, то понятие «длины» несколько расплывчато. Например, рассмотрим список, получаемый следующим фрагментом 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 дырка по индексу 2. Однако что можно сказать о следующем примере?

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

Должны ли мы рассматривать это a как список с 10 000 элементами, где 9998 элементов равны **nil**? Теперь пусть программа делает следующее:

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

Что же произошло с длиной списка? Должна ли она быть 9999, поскольку программа удалила последний элемент? Или может быть 10 000, так как программа просто изменила значение последнего элемента на **nil**? Или же длина должна стать 1?

Другим распространенным предложением является сделать так, чтобы оператор `#` возвращал число элементов в таблице. Эта семантика ясна и хорошо определена, но не несет в себе никакой пользы. Рассмотрим все предыдущие примеры и представим, насколько полезным оказался бы подобный оператор 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 со значением **nil** не отличается от отсутствующего поля. Таким образом, предыдущая таблица неотличима от {10, 20, 30}; ее длина равна 3, а не 5.

Вы можете считать, что **nil** в конце списка – это особенный случай. Однако многие списки строятся путем добавления элементов, одного за другим. Любой список с дырками, который был построен таким образом, просто получен добавлением **nil** в свой конец.

Многие списки, которые мы используем в наших программах, являются последовательностями (например, строка файла не может быть **nil**), и поэтому большую часть времени оператор длины безопасен для использования. Если вам действительно нужно работать со списками с дырками, то вам лучше явно запоминать где-то длину списка.

3.6. Приоритеты операторов

Приоритеты операторов в Lua заданы в таблице ниже, от самого старшего к самому низшему:

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

Все бинарные операторы ассоциативны влево, за исключением **^^** (экспоненцирование) и **..** (конкатенация), которые ассоциативны вправо. 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 скобки. Это легче, чем смотреть в руководстве, и, скорее всего, потом, когда вы будете читать этот код, у вас снова возникнут сомнения.

3.7. 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 раньше это уже видели. Конструкторы также инициализируют списки. Например, оператор

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

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

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

Lua также предлагает специальный синтаксис для инициализации таблиц по полям, как в следующем примере:

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

Эта строка эквивалентна следующим командам:

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

Исходное выражение проще и быстрее, поскольку Lua сразу создаст таблицу с правильным размером.

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

```
w = {x=0, y=0, label="console"}
x = {math.sin(0), math.sin(1), math.sin(2)}
w[1] = "another field"      -- добавить ключ 1 к таблице 'w'
x.f = w                    -- добавить ключ 'f' к таблице 'x'
print(w["x"])              --> 0
print(w[1])                --> другое поле
print(x.f[1])              --> другое поле
w.x = nil                  -- удалить поле "x"
```

Однако создание таблицы сразу с правильным конструктором более эффективно и наглядно.

Мы можем смешивать эти два стиля инициализации (списком и по полям) 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

```

Эти две формы конструктора имеют свои ограничения. Например, вы не можете инициализировать поля с отрицательными индексами или с индексами, которые не являются идентификаторами. Для таких целей есть другой, более общий формат. В этом формате мы явно пишем индекс 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 более неудобен, но и более общ: рассмотренные ранее формы конструктора являются частными случаями этого более общего синтаксиса. Конструктор `{x=0,y=0}` эквивалентен `{["x"]=0,["y"]=0}`, и конструктор `{"r","g","b"}` эквивалентен `{[1]="r",[2]="g",[3]="b"}`.

Вы всегда можете поставить запятую после последней записи в конструкторе. Эти запятые необязательны:

```

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

```

Это 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 от-

деления различных секций в конструкторе, например отделения части, оформленной как записи, от части, оформленной как список:

```
{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 $2^3 \wedge 4$?
А что насчет $2^{\wedge} 3^4$?

Упражнение 3.3. Мы можем представить многочлен

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0$$

в Lua как список его коэффициентов $\{a_0, a_1, \dots, a_n\}$.

Напишите функцию, которая получает многочлен (представленный таблицей) и значение x возвращает значение полинома в x .

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, является ли значение 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 фрагмент кода? 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 с каждой управляющей последовательностью (`escape sequence`) для строк связывает ее значение. Как бы вы написали конструктор такой таблицы?



ГЛАВА 4

Операторы

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

4.1. Операторы присваивания

Присваивание – это базовое средство изменения значений переменной и поля таблицы:

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

Lua позволяет осуществлять *множественное присваивание*, которое присваивает список значений списку переменных за один шаг. Например, в операторе

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

переменная *a* получает значение 10, а переменная *b* – значение $2*x$.

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

```
x, y = y, x          -- поменять местами 'x' и 'y'
a[i], a[j] = a[j], a[i] -- поменять местами 'a[i]' и 'a[j]'
```

Lua всегда приводит количество значений к числу переменных: когда количество значений меньше, чем количество переменных, то Lua дополняет список значений соответствующим количеством **nil**, а когда количество значений больше, то лишние 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 отбрасывается
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 того чтобы проинициализировать список переменных, вы должны предоставить значение для каждой переменной:

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

В действительности большинство предыдущих примеров в чем-то искусственны. Я редко использую множественное присваивание просто для того, чтобы соединить несколько не связанных между собой присваиваний в одну строку. В частности, множественное присваивание не быстрее, чем набор соответствующих одиночных присваиваний. Тем не менее часто нам действительно нужно множественное присваивание. Мы уже видели пример, меняющий две переменные значениями. Более частое использование заключается в получении сразу нескольких значений, возвращенных функцией. Как мы обсудим в разделе 5.1, функция может вернуть сразу несколько значений. В таких случаях обычно используется множественное присваивание, чтобы получить все эти значения. Например, в присваивании `a, b = f()` вызов `f` дает два значения: первое из них записывается в `a`, а второе – в `b`.

4.2. Локальные переменные и блоки

Кроме глобальных переменных, Lua также 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 действия локальной переменной ограничена блоком, где она была объявлена. *Блок* – это тело управляющей конструкции, тело функции и блок кода (файл или строка, где переменная была объявлена):

```
x = 10
local i = 1      -- локальная в блоке
while i <= x do
    local x = i*2 -- локальная внутри блока while
    print(x)      --> 2, 4, 6, 8, ...
    i = i + 1
end
if i > 20 then
    local x      -- локальная внутри "then"
    x = 20
    print(x + 2) -- (напечатает 22, если условие выполнится)
else
    print(x)      --> 10 (глобальная)
end
print(x)          --> 10 (глобальная)
```

Обратите внимание, что этот пример не будет работать так, как ожидается, если вы его введете в интерактивном режиме. В интерактивном режиме каждая строка – это самостоятельный блок (за исключением случая, когда строка не является законченной конструкцией). Как только вы введете вторую строку примера (`local i=1`), Lua выполнит ее и начнет новый блок кода (следующая строка). К тому моменту область действия локальной переменной `i` уже завершится. Чтобы решить эту проблему, мы можем явно заключить весь этот блок между ключевыми словами **do-end**. Когда вы введете **do**, блок закончится, только когда вы введете соответствующий ему **end**, поэтому Lua не будет пытаться выполнить каждую строку как отдельный блок.

Подобные **do**-блоки оказываются полезными, когда нам нужен более точный контроль за областью действия 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 действия 'a2' и 'd' заканчивается здесь
print(x1, x2)
```

Хорошим стилем является использование локальных переменных везде, где только это возможно. Локальные переменные помогают вам избежать забивания глобального окружения ненужными именами. Более того, доступ к локальной переменной быстрее, чем доступ к глобальной. И наконец, локальная переменная перестает существовать, как только заканчивается ее область действия, позволяя сборщику мусора освободить память, занимаемую ее значением.

Lua рассматривает описания локальных переменных просто как операторы. Поэтому вы можете вставить описание локальной переменной всюду, где вы можете вставить оператор. Область действия описанных переменных начинается сразу после описания и заканчивается концом блока. Каждое описание может включать присвоение начального значения, которое действует так же, как и оператор присваивания: лишние значения отбрасываются, лишние переменные получают значение **nil**. Если в описании переменной нет присваивания, то соответствующая переменная получает значение **nil**:

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

Распространенной идиомой в Lua является следующая:

```
local foo = foo
```

Этот код создает локальную переменную `foo` и инициализирует ее значением глобальной переменной `foo`. (Локальная переменная `foo` становится видимой только после этого объявления.) Эта идиома оказывается полезной, когда блоку необходимо сохранить значение оригинальной переменной, если оно изменяется где-то далее в коде; также это ускоряет доступ к этой переменной.

Поскольку многие языки вынуждают декларировать все локальные переменные в начале блока (или процедуры), некоторые считают, что объявлять переменные в середине блока является плохой практикой. На самом деле верно обратное: объявляя переменную, только когда она действительно нужна, вам редко понадобится объявлять ее без начального значения (и поэтому вы вряд ли забудете ее проинициализировать). Более того, вы уменьшаете область действия переменной, что облегчает чтение кода.

4.3. Управляющие конструкции

Lua предоставляет небольшой и довольно традиционный набор управляющих конструкций, используя **if** для условного выполнения и **while**, **repeat** и **for** для итерирования. Все управляющие конструкции обладают явным окончанием: **end** завершает **if**, **for** и **while**, в то время как **until** завершается **repeat**.

Условное выполнение управляющей структуры может дать любое значение. Помните о том, что Lua рассматривает все значения, отличные от **false** и **nil**, как истинные. (В частности, Lua рассматривает ноль и пустую строку как истинные значения.)

if then else

Оператор **if** проверяет условие и выполнять свою *then-часть* или свою *else-часть* соответственно. Часть **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 потребности во многих **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*, то такие конструкции довольно распространены.

while

Как следует из названия, данный оператор повторяет свое тело, пока условие истинно. Как обычно, Lua сперва проверяет условие; 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. Проверка условия осуществляется после выполнения тела цикла, поэтому тело цикла будет выполнено хотя бы один раз.

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

В отличие от многих языков, в Lua в область действия локальных переменных входит условие цикла:

```
local sqr = x/2  
repeat  
    sqr = (sqr + x/sqr)/2  
    local error = math.abs(sqr^2 - x)  
until error < x/10000 -- локальная переменная 'error' видна здесь
```

Числовой оператор for

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

Числовой оператор **for** имеет следующий вид:

```
for var = exp1, exp2, exp3 do  
    <something>  
end
```

Этот цикл будет выполнять *something* для каждого значения *var* от *exp1* до *exp2*, используя *exp3* как *шаг* для увеличения *var*. Это третье выражение (*exp3*) необязательно; когда оно отсутствует, то Lua в качестве шага использует 1. В качестве типичных примеров таких циклов можно рассмотреть

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

Если вы хотите получить 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 лучше знать, чтобы использовать его хорошо. Во-первых, все три выражения вычисляются только один раз, перед началом цикла. Например, в нашем первом примере Lua выполнит $f(x)$ всего один раз. Во-вторых, управляющая переменная является локальной переменной, автоматически объявленной для оператора **for**, и она видна только внутри цикла. Типичной ошибкой является мнение, что эта переменная все еще существует после конца цикла:

```
for i = 1, 10 do print(i) end
max = i -- возможно неверно! Здесь 'i' глобальная
```

Если вам нужно значение управляющей переменной после цикла (обычно когда вы выходите раньше времени из цикла), то вы должны сохранить ее значение в другой переменной:

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

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

Оператор **for** общего вида

Оператор **for** общего вида пробегает все значения, возвращаемые итерирующей функцией:

```
-- напечатать все значения в таблице 't'
for k, v in pairs(t) do print(k, v) end
```

Этот пример использует `pairs`, удобную итерирующую функцию для обхода всей таблицы, предоставленную базовой библиотекой Lua. На каждом шаге этого цикла k получает индекс, а v — значение, связанное с этим индексом.

Несмотря на свою внешнюю простоту, оператор **for** общего вида – это очень мощная конструкция языка. С подходящими итераторами вы можете обойти практически все, что угодно, в легкочитаемой форме. Стандартные библиотеки предоставляют несколько итераторов, позволяющих нам перебирать строки файла (`io.lines`), пары из таблицы (`pairs`), элементы последовательности (`ipairs`), слова внутри строки (`string.gmatch`) и т. д.

Конечно, мы можем писать и свои собственные итераторы. Хотя использование оператора **for** в общей форме легко, задача написания функции-итератора имеет свои тонкости; мы рассмотрим эту тему позже, в главе 7.

Оператор цикла общего вида имеет две общие особенности с числовым оператором цикла: переменные цикла локальны для тела цикла, и вы никогда не должны записывать в них какие-либо значения.

Рассмотрим более конкретный пример использования оператора **for** общего вида. 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 положение в неделе. Вы можете обойти всю таблицу в поисках заданного имени. Однако, как вы скоро узнаете, вам редко понадобится искать в Lua. Более эффективным подходом будет построение обратной таблицы, например `revDays`, в которой названия дней являются индексами, а значениями являются номера дней. Эта таблица будет выглядеть следующим образом:

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

Тогда все, что вам нужно для того, чтобы найти номер дня, – это обратиться к этой обратной таблице:

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

Конечно, не нужно явно задавать эту обратную таблицу. Мы можем построить ее автоматически из исходной:

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

```
    revDays[v] = k
end
```

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

4.4. break, return и goto

Операторы **break** и **return** позволят нам выпрыгнуть прямо из блока. Оператор **goto** позволяет нам перепрыгнуть практически в любое место функции.

Мы используем оператор **break** для завершения цикла. Этот оператор прерывает внутренний цикл (**for**, **repeat** или **while**), содержащий его. Также он может использоваться для возврата из функции, поэтому вам необязательно использовать оператор **return**, если вы не возвращаете никакого значения.

Из синтаксических соображений оператор **return** может быть только последним оператором блока: другими словами, либо последним оператором, либо прямо перед **end**, **else** или **until**. В следующем примере **return** – 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, никогда бы не выполнились. Иногда на самом деле бывает полезно написать **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                                --<< SYNTAX ERROR
-- 'return' is the last statement in the next block
do return end                            -- OK
    <other statements>
end
```

Оператор **goto** переводит выполнение программы к соответствующей метке. Насчет **goto** были длинные обсуждения, некоторые люди даже сейчас считают, что они вредны для программирования и долж-

ны быть исключены из языков программирования. Однако многие языки предлагают подобный оператор, и у них есть для этого повод. Эти операторы представляют собой мощный механизм, который, будучи аккуратно использованным, способен 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, однако, более сложный: она состоит из двух двоеточий, за которыми следует имя метки, за которыми следуют еще два двоеточия, например `::name::`. Эта сложность намеренная, ее цель — заставить программиста дважды подумать, прежде чем использовать **goto**.

Lua накладывает некоторые ограничения на то, куда вы можете перескочить при помощи **goto**. Во-первых, метки следуют обычным правилам видимости, поэтому вы не можете прыгнуть прямо внутрь блока (поскольку метка внутри блока невидима снаружи его). Во-вторых, вы не можете выпрыгнуть из функции. (Обратите внимание, что первое правило исключает возможность прыгнуть *внутрь* функции.) В-третьих, вы не можете прыгнуть внутрь области действия локальной переменной.

Типичным и хорошо используемым применением оператора **goto** является эмулирование некоторой конструкции, которую вы узнали из другого языка, но которая отсутствует в Lua, например **continue**, многоуровневый **break**, **redo** и т. п. Оператор **continue** — это просто переход на метку в конце цикла, оператор **redo** перепрыгивает к началу блока:

```
while some_condition do
  ::redo::
  if some_other_condition then goto continue
  else if yet_another_condition then goto redo
  end
  <some code>
  ::continue::
end
```

Полезным нюансом в спецификациях Lua является то, что область действия локальной переменной заканчивается на последнем *непустом* операторе блока, где переменная определена; метки считаются пустыми операторами. Для того чтобы увидеть полезность этого, рассмотрим следующий фрагмент кода:

```
while some_condition do
  if some_other_condition then goto continue end
  local var = something
```

```
<some code>
::continue::
end
```

Вы можете подумать, что этот оператор **goto** перепрыгивает прямо в область действия переменной `var`. Однако метка `continue` находится после последнего непустого оператора блока, и поэтому не в области действия переменной `var`.

Оператор **goto** также полезен при написании конечных автоматов. В качестве примера листинг 4.1 является примером программы, проверяющей, содержит ли ее ввод четное количество нулей. Существуют более удачные способы написания этой программы, но данный подход весьма полезен, если вы хотите автоматически перевести конечный автомат в код на Lua (подумайте об автоматической генерации кода).

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 входит в соответствующую комнату; иначе программа печатает предупреждение. Целью является дойти от начальной комнаты до конечной комнаты.

Эта игра является типичным автоматом, где текущая комната является состоянием. Мы можем реализовать эту игру, используя один блок для каждой комнаты и оператор **goto** для перехода из одной комнаты в другую. Листинг 4.2 показывает, как можно написать простейший лабиринт из четырех комнат.

Для этой простой игры вы можете решить, что программа, управляемая данными, когда вы описываете комнаты и перемещения при помощи таблиц, является более удачным решением. Однако если в каждой комнате нас ждут свои особенности, то этот подход оказывается вполне удачным.

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
```

Листинг 4.2. Игра в 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("недопустимый ход")
        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("недопустимый ход")
        goto room2
    end
end
::room3:: do
    local move = io.read()
    if move == "north" then goto room1
    elseif move == "east" then goto room4
    else
        print("недопустимый ход")
        goto room3
    end
end
::room4:: do
    print("Поздравляем, вы выиграли!")
end
```

Упражнения

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

Упражнение 4.2. Напишите четыре различных способа реализовать безусловный цикл в Lua. Какой из них вам больше нравится?

Упражнение 4.3. Многие считают, что **repeat-until** используется редко и поэтому 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 нельзя выпрыгнуть из функции? (Подсказка: как бы вы реализовали данную возможность?)

Упражнение 4.6. Предполагая, что **goto** может выпрыгнуть из функции, объясните, что программа на листинге 4.3 должна делать. (Попытайтесь рассуждать о метке с использованием тех же самых правил, которые используются для описания области действия локальных переменных.)

Листинг 4.3. Странное (и неверное) 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 главным механизмом абстракции операторов и выражений в Lua. Функции могут выполнять определенное задание (в других языках это часто называется *procedure* или *subroutine*) или вычислить и вернуть 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 список аргументов заключен в круглые скобки, обозначающие вызов; если у функции нет аргументов, то мы все равно должны написать `()` для обозначения вызова функции. Существует специальное исключение из этого правила: если у функции всего один аргумент и этот аргумент, либо литерал (строка символов), либо конструктор таблицы, то круглые скобки необязательны:

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

Lua также предлагает специальный синтаксис для объектно-ориентированных вызовов, оператор двоеточие. Выражение вроде `o:foo(x)` — это просто способ записать `o.foo(o, x)`, то есть позвать `o.foo`, добавляя `o` как дополнительный аргумент. В главе 16 мы обсудим подобные вызовы (и объектно-ориентированное программирование) более подробно.

Программа на Lua может использовать функции, написанные как на Lua, так и на C (или любом другом языке, используемом приложением). Например, все функции из стандартной библиотеки Lua напи-

саны на С. Однако при вызове функции нет никакой разницы между функциями, написанными на Lua, и функциями, написанными на С.

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

```
-- сложить 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 add), список *параметров* и *тело*, которое является списком операторов.

Параметры работают как локальные переменные, проинициализированные значениями аргументов, переданных при вызове функции. Вы можете позвать функцию с количеством аргументов, отличающимся от ее списка параметров. Lua приведет число аргументов к числу параметров так же, как это делается во множественном присваивании: лишние аргументы отбрасываются, вместо недостающих добавляется **nil**. Например, рассмотрим следующую функцию:

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

Она обладает таким 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 (легко обнаруживаемым во время выполнения), оно также полезно, особенно для аргументов по умолчанию. Например, рассмотрим следующую функцию, увеличивающую глобальный счетчик:

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

У этой функции есть один параметр по умолчанию; если мы вызовем ее `incCount()` без аргументов, то она увеличит `count` на единицу. Когда вы вызываете `incCount()`, Lua сперва инициализирует `n` значением **nil**; оператор `or` возвращает свой второй аргумент, и в результате Lua присваивает переменной `n` значение 1.

5.1. Множественные результаты

Мало распространенной, но тем не менее очень удобной особенностью Lua является то, что функция может вернуть несколько значений. Некоторые предопределенные функции в Lua возвращают несколько значений. В качестве примера можно взять функцию `string.find`, которая ищет шаблон в строке. Эта функция возвращает два индекса, когда находит шаблон: индекс начала шаблона в строке и индекс конца шаблона. 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 сами пишем, также могут возвращать сразу несколько значений, просто перечисляя их после слова **return**. Например, функция, которая ищет максимальный элемент в последовательности, может вернуть сразу и сам максимальный элемент, и его индекс:

```
function maximum (a)
    local mi = 1                -- индекс максимального элемента
    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 значений, возвращенных функцией, к обстоятельствам ее вызова. Когда мы вызываем функцию как оператор, то Lua отбрасывает все возвращенные значения. Когда мы используем вызов функции в выражении, то Lua сохраняет только первое значение. Мы получим все значения, только когда вызов функции является последним (или единственным) выражением в списке выражений. В Lua эти списки встречаются в четырех конструкциях: множественное присваивание, аргументы для вызова функции, конструктор таблицы и оператор **return**. Чтобы проиллюстрировать все эти случаи, мы рассмотрим следующие определения функций:

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

В множественном присваивании вызов функции как последнее (или единственное) выражение использует столько результатов, сколько нужно для соответствия списку 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, но не так много, как требуется, то в качестве недостающих значений используется **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, дает ровно одно значение:

```
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 идут как аргументы на вход второго вызова. Мы уже видели примеры этой конструкции с функцией `print`. Поскольку функция `print` может получать переменное число аргументов, оператор `print(g())` печатает все значения, 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` оказывается внутри выражения, Lua приводит число возвращенных значений к одному; поэтому в последней строке конкатенация использует только `"a"`.

Если мы запишем `f(g(x))` и `y f` фиксированное число аргументов, то Lua приводит число возвращенных значений к числу аргументов `f`, как мы уже видели ранее.

Конструктор таблицы также использует все значения, возвращенные функцией, без каких-либо изменений:

```

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

```

Как всегда, это поведение встречается, только если вызов является последним выражением в списке; вызовы в любых других местах дают ровно по одному значению:

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

Наконец, оператор `return f()` возвращает все значения, которые вернула `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))      -- (нет значений)
print(foo(3))      -- (нет значений)

```

Вы можете «заставить» вызов вернуть только одно значение, заключив его в дополнительную пару круглых скобок:

```

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

```

Будьте внимательны: оператор **return** не требует скобок вокруг возвращаемого значения. Поэтому выражение 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 на вход массив и возвращает все элементы этого массива, начиная с 1:

```

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

```

Важным использованием `unpack` является обобщенный механизм вызова функции. Обобщенный механизм позволяет вам вызывать любую функцию с любыми аргументами динамически. В ANSI C, например, не существует способа построить обобщенный вызов. Вы можете объявить функцию, которая получает переменное число аргументов

(при помощи `stdarg.h`), и вы можете вызывать различные функции, используя указатели на функции. Однако вы не можете позвать функцию с переменным числом аргументов: каждый раз, когда вы пишете на С, у вас фиксированное число аргументов, и каждый аргумент имеет фиксированный тип. В Lua если вы хотите позвать функцию `f` с переменным числом аргументов из массива `a`, то можете просто написать следующее:

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

Вызов функции `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 оператор длины, для того чтобы узнать, сколько элементов следует вернуть, поэтому он работает только с последовательностями. Если нужно, то его можно явно ограничить:

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

Хотя функция `unpack` написана на С, мы можем записать 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, в `i` записывается 1, а в `n` записывается длина последовательности. Затем функция возвращает `t[1]` вместе с всеми результатами `unpack(t, 2, n)`, что, в свою очередь, возвращает `t[2]` и все результаты вызова `unpack(t, 3, n)` и т. д., останавливаясь после `n` элементов.

5.2. Функции с переменным числом аргументов

Функция в Lua может иметь произвольное число аргументов (*variadic*). Например, мы уже вызывали функцию `print` с одним, двумя и большим числом аргументов. Хотя `print` определена в C, мы и на Lua можем писать функции с переменным числом аргументов.

В качестве следующего примера функция ниже 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 обозначают, что эта функция имеет переменное число аргументов. Когда мы вызываем такую функцию, Lua собирает все ее аргументы в список; мы называем эти собранные аргументы *дополнительными аргументами* функции. Функция может получить доступ к своим дополнительным опять при помощи трех точек, теперь уже как в качестве выражения. В нашем примере выражение {...} дает массив со всеми собранными аргументами. Функция перебирает элементы этого массива для того, чтобы найти их сумму.

Мы называем выражение ... *выражением с переменным числом аргументов* (vararg expression). Оно ведет себя как функция, возвращающая много 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 дополнительных аргументов (или **nil**, если таких аргументов нет).

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

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

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

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

Тем, кому нравится механизм передачи параметров в Perl, это понравится.

Функция, показанная ниже, просто возвращает все переданные аргументы:

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

Следующая функция ведет себя так же, как и `foo`, за исключением того, что перед ее вызовом она печатает сообщение со всеми переданными аргументами:

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

Это довольно полезный трюк для отслеживания всех вызовов к заданной функции.

Давайте рассмотрим еще один полезный пример. Lua предоставляет отдельные функции для форматирования текста (`string.format`) и для записи текста (`io.write`). Довольно просто объединить эти две функции в одну функцию с переменным числом аргументов:

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

Обратите внимание на присутствие параметра `fmt` перед точками. Функции с переменным числом аргументов могут иметь любое количество фиксированных параметров перед частью с переменным числом параметров. Lua присваивает первые значения этим переменным; остальные (если есть) идут как дополнительные параметры. Ниже мы покажем несколько примеров вызовов и соответствующих параметров:

Вызов	Параметры
<code>fwrite()</code>	<code>fmt = nil</code> , нет дополнительных параметров
<code>fwrite("a")</code>	<code>fmt = "a"</code> , нет дополнительных параметров
<code>fwrite("%d%d", 4, 5)</code>	<code>fmt = "%d%d"</code> , дополнительные 4 и 5

(Обратите внимание, что вызов `fwrite()` приведет к ошибке, поскольку `string.format` требует строку как свой первый аргумент.)

Для обхода всех дополнительных параметров функция может использовать выражение `{...}` для того, чтобы собрать их всех в таблицу, как мы это сделали в определении функции `add`.

В редких случаях, когда переданные аргументы могут принимать значение **nil**, таблица, созданная при помощи `{ ... }`, не будет настоящей последовательностью. Например, не существует способа для того, чтобы по этой таблице узнать, были ли в конце списка аргументов **nil**¹. Для этих случаев Lua предлагает функцию `table.pack`¹. Эта функция получает произвольное число аргументов и возвращает новую таблицу, содержащую все свои аргументы, как и `{ ... }`, но в этой таблице будет дополнительное поле `n`, содержащее полное число ее аргументов. Следующая функция использует `table.pack` для того, чтобы среди ее аргументов были значения **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
```

Запомните, однако, что `{ ... }` быстрее и чище, чем `table.pack`.

5.3. Именованные аргументы

Механизм передачи параметров в Lua является *позиционным*: когда мы вызываем функцию, то соответствие между аргументами и формальными параметрами осуществляется по их положению. Первый аргумент дает значение первому параметру и т. д. Иногда, однако, полезно указать параметр по имени. Для того чтобы проиллюстрировать это, 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 параметра:

```
-- неверно
rename(old="temp.lua", new="templ.lua")
```

В Lua нет непосредственной поддержки для этого синтаксиса, но мы можем добиться требуемого эффекта путем небольшого синтак-

¹ Эта функция появилась только в Lua 5.2.

сического изменения. Идея заключается в том, чтобы собрать все аргументы в таблицу и использовать эту таблицу как единственный аргумент функции. Специальный синтаксис, который Lua предоставляет для вызова функции, с конструктором таблицы как единственным аргументом, поможет нам добиться этого:

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

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

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

Этот способ передачи параметров особенно полезен, когда у функции много аргументов и большинство из них необязательные. Например, функция, которая создает новое окно в библиотеке GUI, может иметь десятки аргументов, большинство из которых необязательные, и лучше всего их передать, используя имена:

Листинг 5.1. Функция с именованными необязательными параметрами

```
function Window (options)
    -- check mandatory options
    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", -- значение по умолчанию
        options.border -- значение по умолчанию false (nil)
    )
end

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


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 числа комбинаций: $C(n, m) = C(n-1, m-1) + C(n-1, m)$. Для получения всех $C(n, m)$ комбинаций из n элементов в группы размера m вы сперва добавляете первый элемент к результату и затем генерируете все $C(n-1, m-1)$ комбинаций из оставшихся элементов в оставшихся местах. 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 о функциях

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

Что значит, что функции являются «значениями первого класса»? Это значит что в Lua функция – это значение, обладающее тем же правами, что и стандартные значения вроде чисел и строк. Мы можем сохранять функции в переменных (локальных и глобальных) и в таблицах, мы можем передавать функции как аргументы и возвращать их из других функций.

Что значит для функций «лексическая область действия»? Это значит, что функции могут обращаться к переменным, содержащим их функции¹. Как мы увидим в этой главе, это вроде безобидное свойство дает огромную мощь языку, поскольку позволяет применять в Lua многие могущественные приемы из мира функционального программирования. Даже если вы совсем не интересуетесь функциональным программированием, все равно стоит немного узнать о том, как использовать эти возможности, поскольку они могут сделать вашу программу меньше и проще.

Несколько смущающим понятием в Lua является то, что функции, как и другие значения, являются анонимными; у них нет имен. Когда мы говорим об имени функции, например `print`, мы имеем в виду переменную, которая содержит данную функцию. Как и с любой другой переменной, содержащей любое другое значение, мы можем манипулировать этими переменными многими разными способами. Следующий пример, хотя и несколько надуман, показывает возможные примеры:

```
a = {p = print}
a.p("Hello World")  --> Hello World
print = math.sin     -- 'print' теперь ссылается на синус
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.)

¹ Это также значит, что Lua полностью содержит в себе лямбда-исчисление.

Если функции являются значениями, то существуют ли выражения, которые создают функции? Да. В частности, стандартный способ создать функцию в Lua, как, например,

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

это просто пример того, что мы называем *синтаксическим сахаром*; это просто более красивый способ написать следующий код:

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

По этому определению функции – это на самом деле оператор (присваивания), который создает значение типа “function” и присваивает его переменной. Мы можем рассматривать выражение `function(x) body end` как конструктор функции, точно так же, как `{ }` является конструктором таблицы. Мы называем результат выполнения подобных конструкторов *анонимной функцией*. Хотя мы часто присваиваем функции глобальным переменным, давая им что-то вроде имени, бывают случаи, когда функции остаются анонимными. Давайте рассмотрим несколько примеров.

Библиотека `table` предоставляет функцию `table.sort`, которая получает таблицу и сортирует ее элементы. Подобная функция должна предоставлять бесконечные вариации порядка сортировки: по возрастанию и по убыванию, числовой или по алфавиту, по какому ключу и т. д. Вместо попытки предоставить все возможные опции `sort` предоставляет дополнительный параметр, который является *функцией упорядочения*: функция, которая получает два аргумента и определяет, должен ли первый элемент идти перед вторым в отсортированном списке. Например, допустим, что у нас есть следующая 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 вы хотите отсортировать таблицу по полю `name` в обратном алфавитном порядке, то вы можете просто записать:

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

Посмотрите, как удобно было использовать анонимную функцию в этом операторе.

Функция, которая получает другую функцию как аргумент, является тем, что мы называем *функцией высшего порядка*. Функции высшего порядка являются удобным программным механизмом, и использование анонимных функций для создания их функциональных аргументов является большим источником гибкости. Однако запомните, что функции высших порядков не являются чем-то особенным, они просто следствие способности Lua работать с функциями как значениями первого класса.

Для того чтобы проиллюстрировать использование функций высших порядков, мы напомним упрощенное определение часто встречающейся функции высшего порядка, производной. Следуя неформальному определению, производная функции f в точке x — это значение $(f(x + d) - f(x))/d$, когда d становится бесконечно малой. В соответствии с этим определением мы можем написать 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
```

Получив функцию f , вызов `derivative(f)` вернет приближенное значение ее производной, которое является другой 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 являются значениями первого класса в Lua, мы можем запоминать их не только в глобальных переменных, но и в локальных переменных и полях таблиц. Как мы увидим дальше, использование функций в полях таблицы — это ключевой компонент некоторых продвинутых возможностей Lua, таких как модули и объектно-ориентированное программирование.

6.1. Замыкания

Когда мы пишем функцию, заключенную внутри другой функции, то она имеет полный доступ к локальным переменным окружающей ее функции; мы называем это *лексической областью действия* (lexical scoping). Хотя это правило видимости может показаться очевидным,

на самом деле это не так. Лексическая область видимости вместе с функциями, которые являются объектами первого класса, является очень мощной концепцией в языке программирования, но многие языки этого не поддерживают.

Давайте начнем с простого примера. Пусть у вас есть список имен студентов и таблица, сопоставляющая им их оценки; вы хотите отсортировать список студентов по их оценкам, студенты с более высокими оценками должны идти раньше. 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]           -- сравнить оценки
end)
```

Теперь допустим, что вы хотите создать функцию для решения данной задачи:

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

Интересной особенностью в этом примере является то, что анонимная функция, передаваемая функции `sort`, обращается к параметру `grades`, который является локальным для заключающей их функции `sortbygrade`. Внутри этой анонимной функции `grades` не является ни глобальной переменной, ни локальной переменной, а тем, что мы называем *нелокальной переменной*. (По историческим причинам для обозначения нелокальных переменных в Lua также используется термин *upvalue*.)

Почему это так интересно? Потому что функции являются значениями первого класса, и поэтому они могут *выйти* из начальной области действия своих переменных. Рассмотрим следующий пример:

```
function newCounter ()
    local i = 0
    return function ()
        i = i + 1
        return i
    end
end

c1 = newCounter()
```

-- анонимная функция

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

В этом коде анонимная функция ссылается на нелокальную переменную `i` для учета значения. Однако к тому времени, как мы позовем анонимную функцию, переменная `i` уже выйдет из своей области видимости, поскольку функция, которая создала эту переменную (`newCounter`), уже завершится. Тем не менее Lua правильно обрабатывает эту ситуацию, используя понятие *замыкания* (*closure*). Проще говоря, замыкание – это функция плюс все, что ей нужно для доступа к нелокальным переменным. Если мы снова позовем `newCounter`, то она создаст новую локальную переменную `i`, поэтому мы получим новое замыкание, работающее с этой новой переменной:

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

Таким образом, `c1` и `c2` – это разные замыкания одной и той же функции, и каждая использует свою независимо instantiated локальную переменную `i`.

На самом деле в Lua значением является замыкание, а не функция. Функция – это просто прототип для замыкания. Тем не менее мы будем использовать термин «функция» для обозначения замыкания всегда, когда это не будет приводить к путанице.

Замыкания оказываются очень удобным инструментом во многих случаях. Как мы уже видели, они оказываются удобными в качестве аргументов функций высших порядков, таких как `sort`. Замыкания также полезны для функций, которые строят другие функции, как функция `newCounter` в нашем примере или же функция для нахождения производной; этот механизм позволяет программам на Lua использовать продвинутые методы из мира функционального программирования. Замыкания также удобны для различных *вызываемых функций* (*callback*). Типичный пример возникает, когда вы создаете различные кнопки в своей библиотеке для создания GUI. У каждой кнопки есть своя функция, которая должна быть вызвана, когда пользователь нажимает на эту кнопку; обычно нужна, чтобы разные кнопки приводили к различным действиям. Например, калькулятору нужно десять кнопок, по одной на каждую цифру. Вы можете их создать с помощью подобной функции:

```
function digitButton (digit)
  return Button{ label = tostring(digit),
```

```
        action = function ()
            add_to_display(digit)
        end
    end
```

В этом примере мы предполагаем, что `Button` – это функция из библиотеки, которая создает новые кнопки; `label` – это метка кнопки; `action` – это замыкание, которое нужно позвать, когда кнопка будет нажата. Замыкание может быть заметно спустя длительное время после того, как `digitButton` выполнилась, и после того, как локальная переменная `digit` вышла из области своей видимости, но тем не менее замыкание все равно может к ней обращаться.

Замыкания также оказываются полезными в совсем другом случае. Поскольку функции хранятся в обычных переменных, мы можем легко переопределять функции в Lua, включая даже стандартные. Эта возможность является одной из причин, почему Lua столь гибок. Часто, когда вы переопределяете функцию, вам все равно нужна старая функция. Например, вы хотите переопределить функцию `sin`, чтобы она работала с градусами вместо радиан. Эта новая функция преобразует свой аргумент и затем зовет исходную функцию `sin` для выполнения работы. Ваш код при этом может выглядеть, как показано ниже:

```
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 использовать этот же подход для создания безопасных окружений, также называемых *песочницами* (*sandbox*). Безопасные окружения крайне важны при выполнении кода из ненадежных источников, таких как Интернет. Например, 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)
    <check access>
  end
  io.open = function (filename, mode)
    if access_OK(filename, mode) then
      return oldOpen(filename, mode)
    else
      return nil, "access denied"
    end
  end
end
end
```

Что делает этот пример особенно приятным, так, что после этого переопределения нет абсолютно никакого способа программе позвать исходный `open`, кроме как через новую версию с контролем. Небезопасная версия хранится в локальной переменной внутри замыкания, не достижима никак снаружи. С этим подходом вы можете строить песочницы для Lua на самом Lua, получая при этом в качестве плюсов простоту и гибкость. Вместо какого-то универсального решения для всех проблем Lua предоставляет мета-механизм, так что вы можете подогнать свое окружение под ваши цели.

6.2. Неглобальные функции

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

Мы уже видели различные примеры функций, хранимых в полях 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
```


Конечно, мы также можем использовать конструкторы:

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

Более того, Lua также предоставляет еще один синтаксис для подобных функций:

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

Когда мы запоминаем 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 блок как функцию, блок может определять локальные функции, которые видны только из блока. Лексическая область видимости гарантирует, что другие функции из пакета могут использовать эти локальные функции:

```
local f = function (<params>  
  <body>  
end  
local g = function (<params>  
  <some code>  
  f() -- 'f' здесь видна  
  <some code>  
end
```

Lua также поддерживает следующий синтаксический сахар для локальных функций:

```
local function f (<params>  
  <body>  
end
```

При определении рекурсивных локальных функций возникает тонкость. Наивный подход здесь не работает. Рассмотрим следующее определение:

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

Когда Lua компилирует вызов `fact(n-1)` в теле функции, то локальная функция `fact` еще не определена. Поэтому данное определение попытается позвать глобальную функцию `fact`, а не локальную. Мы можем решить эту проблему, сперва определив локальную переменную и затем уже определяя 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 в момент, когда функцию определяют, ничего не значит; к тому моменту, когда функция будет выполняться, она уже получит правильное значение.

Когда Lua «раскрывает» свой синтаксический сахар для локальных функций, она не использует «наивный» путь. Вместо этого определение, как показано ниже:

```
local function foo (<params>) <body> end,
```

переходит в

```
local foo; foo = function (<params>) <body> end
```

Поэтому мы можем спокойно использовать этот синтаксис для рекурсивных функций.

Конечно, этот прием не работает, если у вас не прямая рекурсия — две функции вызывают друг друга. В таких случаях нужно явно записать соответствующие описания локальных переменных:

```
local f, g — описали локальные переменные

function g ()
  <some code> f() <some code>
end
function f ()
  <some code> g() <some code>
end
```

В этом примере для функции `f` нельзя написать `local function f`, поскольку в подобном случае Lua создаст новую локальную переменную `f`, оставив старую (на которую ссылается `g`) неинициализированной.

6.3. Оптимизация хвостовых вызовов

Другой интересной особенностью функций в Lua является то, что Lua выполняет оптимизацию хвостовых вызовов. (Это значит, что Lua поддерживает оптимизацию *хвостовой рекурсии* (tail recursion), хотя это здесь и не связано непосредственно с рекурсией, см. упражнение 6.3.)

Хвостовой вызов (tail call) – это фактически **goto**, выглядящее как вызов функции. Хвостовой вызов случается, когда функция вызывает другую функцию как последнее свое действие. 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 нет необходимости возвращаться в вызывающую функцию, когда вложенный вызов завершился. Поэтому после хвостового вызова программе нет необходимости хранить какую-либо информацию о вызывающей функции на стеке. Когда вызов *g* завершается, управление непосредственно переходит к точке, где была вызвана *f*. Реализации некоторых языков, например интерпретатор Lua, используют этот факт и не выделяют дополнительно места на стеке для хвостового вызова. Мы говорим, что эти реализации поддерживают *устранение хвостовых вызовов* (tail-call elimination).

Поскольку хвостовые вызовы не используют места на стеке, количество вложенных хвостовых вызовов, которое программа может выполнить, просто ничем не ограничено. Например, 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.

Тонким моментом в устранении хвостовых вызовов является вопрос о том, что же является хвостовым вызовом. Для некоторых вполне очевидных кандидатов требование о том, что вызывающая функция больше ничего не делает после вызова, не выполняется. Например, в следующем коде вызов функции *g* не является хвостовым.

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

Проблема в этом примере – в том, что после вызова `g` функция `f` должна отбросить результаты `g` перед возвратом. Аналогично все следующие вызовы также не удовлетворяют условию:

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

В Lua только вызов вида `return func(args)` является хвостовым. Однако и `func`, и ее аргументы могут быть сложными выражениями, поскольку Lua выполнит 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 получает функцию `f` и возвращает приближенное значение ее интеграла.

Упражнение 6.2. В упражнении 3.3 вам надо было написать функцию, которая получает многочлен (представленный таблицей) и значение переменной и возвращает значение многочлена для этой переменной. Напишите функцию, которая получает многочлен и возвращает 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, поддерживающий оптимизацию хвостовых вызовов, называется *поддерживающим хвостовую рекурсию* (properly tail recursive), если оптимизация хвостовых вызовов поддерживается только для рекурсивных вызовов. (Без рекурсивных вызовов максимальная глубина вызовов статически определена.)

Покажите, что это не справедливо в языке типа Lua: напишите программу, которая реализует неограниченную глубину вызовов без использования рекурсии. (*Подсказка:* см. раздел 8.1.)

Упражнение 6.4. Как мы видели, хвостовой вызов – это замаскированное `goto`. Используя эту идею, перепишите код для

игры в лабиринт из раздела 4.4, используя хвостовые вызовы. Каждый блок должен стать новой функцией, и каждый **goto** становится хвостовым вызовом.



ГЛАВА 7

Итераторы и обобщенный **for**

В этой главе мы покажем, как писать итераторы для обобщенного **for** (оператор **for** общего вида). Начиная с простых итераторов, мы узнаем, как использовать всю силу обобщенного **for** для написания простых и более эффективных итераторов.

7.1. Итераторы и замыкания

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

Любой итератор должен где-то сохранять свое состояние между вызовами, чтобы знать, где он находится и как продолжать дальше. Замыкания являются великолепным механизмом для этой задачи. Напомним, что замыкание – это функция, которая обращается к одной или нескольким локальным переменным из своего окружения. Эти переменные сохраняют свои значения между последовательными вызовами замыкания, помогая тем самым замыканию понять, где оно находится в своем пути. Конечно, для создания нового замыкания мы также должны создать его нелокальные переменные. Поэтому построение замыкания обычно включает в себя сразу две функции: само замыкание и *фабрику*, функцию, которая создает замыкание вместе с окружающими его переменными.

В качестве примера давайте напишем простой итератор для списка. В отличие от `ipairs`, этот итератор не будет возвращать индекс каждого элемента, а только 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 (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 обозначает конец итераций.

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

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

Однако гораздо легче использовать обобщенный оператор **for**. В конце концов, он был создан именно для подобного итерирования:

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

Обобщенный **for** выполняет всю закулисную работу для итерирования: он внутри хранит итерирующую функцию, поэтому нам не нужна переменная `iter`, он вызывает итератор для каждой новой итерации, и он завершает итерирование, когда итератор возвращает `nil`. (В следующей секции мы увидим, что обобщенный **for** делает даже больше, чем только это.)

В качестве более продвинутого примера, – листинг 7.1 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 шаблон `'%w+'`, которому удовлетворяет один или более алфавитно-цифровых символов. Если этот вызов находит слово, то функция обновляет текущую позицию на первый символ

после слова и возвращает это слово¹. Иначе итератор читает следующую строку и повторяет поиск. Если больше строк нет, он возвращает **nil**, чтобы сообщить о конце обхода.

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. Это не является проблемой, гораздо чаще конечные пользователи, программирующие на Lua, не пишут свои итераторы, а используют итераторы, предоставленные 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 ()          -- функция-итератор
    while line do             -- повторять, пока есть строки
      local s, e = string.find(line, "%w+", pos)
      if s then               -- нашли слово?
        pos = e + 1           -- следующая позиция после слова
        return string.sub(line, s, e) -- вернуть слово
      else
        line = io.read()      -- слово не найдено; пробуем след. строку
        pos = 1               -- начинаем с начала строки
      end
    end
    return nil -- больше нет строк, конец обхода
  end
end
```

7.2. Семантика обобщенного for

Одним из недостатков рассмотренных выше 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 замыкания несравнима с ценой чтения целого файла. Однако

¹ Функция `string.sub` извлекает подстроку из `line` между заданными позициями; мы детально разберем ее в разделе 21.2.

в некоторых ситуациях это может оказаться существенным. В таких случаях мы можем использовать сам обобщенный `for` для хранения состояния. В этом разделе мы увидим, какие возможности по хранению состояния предлагает обобщенный `for`.

Мы видели, что обобщенный `for` во время цикла хранит итерирующую функцию внутри себя. На самом деле он хранит три значения: итерирующую функцию, *неизменяемое состояние* (invariant state) и *управляющую переменную*. Теперь давайте обратимся к деталям.

Синтаксис обобщенного `for` приводится ниже:

```
for <var-list> in <exp-list> do
  <body>
end
```

Здесь *var-list* – это список из одного или нескольких имен переменных, разделенных запятыми, а *exp-list* – 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 также 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 не равно `nil`, поскольку когда она становится равной `nil`, цикл завершается.

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

После этой инициализации **for** вызывает итерирующую функцию с двумя аргументами: инвариантным состоянием и управляющей переменной. (С точки зрения оператора **for**, это инвариантное состояние вообще не имеет никакого смысла. Оператор **for** только передает значение состояния с шага инициализации вызову итерирующей функции.) Затем **for** присваивает значения, возвращенные итерирующей функцией, переменным, объявленным в списке переменных. Если первое значение (присваиваемое управляющей переменной) равно **nil**, то цикл завершается. Иначе **for** выполняет свое тело и снова зовет итерирующую функцию, повторяя процесс.

Более точно конструкция вида

```
for var_1, ..., var_n in <explist> do <block> end
```

эквивалента следующему коду:

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

Поэтому если наша итерирующая функция – это f , неизменяемое состояние s и начальное состояние для управляющей переменной есть 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 получают дополнительные значения, возвращаемые f .

7.3. Итераторы без состояния

Как подразумевает его название, такой итератор не хранит в себе какого-либо состояния. Поэтому мы можем использовать один и тот же итератор без состояния во многих циклах, избегая тем самым создания новых замыканий.

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

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

Состояние этого итератора – это таблица, которую мы перебираем (неизменяемое состояние, сохраняющее свое значение на протяжении цикла), и текущий индекс (управляющая переменная). И `ipairs` (фабрика), и сам итератор очень просты, мы могли бы записать 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)` для цикла **for**, она получает три значения: итерирующую функцию `iter`, а как инвариантное состояние и ноль в качестве начального значения для управляющей переменной. Затем Lua вызывает `iter(a, 0)`, что дает `1, a[1]` (если только `a[1]` уже не **nil**). На следующей итерации вызывается `iter(a, 1)`, что возвращает `2, a[2]`, и т. д. до первого элемента, равного **nil**.

Функция `pairs`, которая перебирает все элементы таблицы, похожа, за исключением того, что итерирующая функция – это функция `next`, которая является стандартной функцией в Lua:

```
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 больше нет пар, то `next` возвращает **nil**.

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

```
for k, v in next, t do
```

```
<loop body>
end
```

Вспомним, что `for` приводит свой список выражений к трем значениям, в качестве которых здесь выступают `next`, `t` и **`nil`**; это именно то, что получается при вызове `pairs`.

Итератор для обхода связанного списка является другим 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
```

Здесь мы используем начало списка как инвариантное состояние (второе значение, возвращаемое `traverse`) и текущий узел в качестве управляющей переменной. Когда итерирующая функция `getnext` будет первый раз вызвана, `node` будет равен **`nil`**, и поэтому функция вернет `list` как первый узел. В последующих вызовах `node` будет уже не равно **`nil`**, и поэтому итератор вернет `node.next`, как и ожидается. 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 переменные инвариантного состояния и управляющей переменной. Простейшим решением является использование замыканий. Альтернативным решением будет запаковать все, что нужно

итератору, в таблицу и использовать эту таблицу как инвариантное состояние для цикла. Используя таблицу, итератор можем хранить так много данных, как ему нужно. Более того, он может менять эти данные так, как он хочет. Хотя состояние – все время одна и та же таблица (поэтому она инварианта), содержимое таблицы может меняться на протяжении цикла. Поскольку такие итераторы хранят все свои данные в состоянии, обычно они игнорируют второй аргумент, предоставляемый обобщенным циклом `for` (переменная цикла).

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

Функция, начинающая цикл, 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                        -- повторяем, пока есть строки
                                                -- ищем следующее слово
        local s, e = string.find(state.line, "%w+", state.pos)
        if s then                             -- нашли слово?
                                                -- обновить положение
            state.pos = e + 1
            return string.sub(state.line, s, e)
        else -- word not found
            state.line = io.read()            -- пробуем следующую строку...
            state.pos = 1                     -- ... с начала
        end
    end
    return nil                                -- больше строк нет, завершаем цикл
end
```

Всегда, когда это возможно, вам следует пытаться написать итераторы без состояния, такие, которые хранят все свое состояние в переменных цикла `for`. С ними вы не создаете новых объектов, когда начинаете цикл. Если эта модель не подходит, то вам следует попробовать замыкания. Кроме того, это более красиво, замыкание обычно является более эффективным в качестве итератора, чем таблица: 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, с использованием сопрограмм. Это является наиболее мощным решением, но оно несколько дороже.

7.5. Подлинные итераторы (true iterators)

Термин «итератор» несколько неточен, поскольку на самом деле итерирует не итератор, а цикл **for**. Итераторы только предоставляют последовательные значения для итерирования. Может быть, более удачным термином был бы «генератор», но термин «итератор» уже получил широкое распространение в таких языках, как Java.

Тем не менее существует другой способ построения итераторов, где итераторы действительно осуществляют итерирование. Когда мы используем такие итераторы, мы не пишем цикл; вместо этого мы просто зовем итератор с аргументом, описывающим, что итератор должен делать на каждой итерации. Более точно итератор получает в качестве аргумента функцию, которую он вызывает внутри своего цикла.

В качестве примера давайте еще раз перепишем итератор `allwords` с использованием этого подхода:

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

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

```
allwords(print)
```

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

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

```
end)
print(count)
```

Та же самая задача, записанная при помощи итераторов ранее рассмотренного стиля, 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 очень популярны в старых версиях Lua, когда в языке еще не было оператора **for**. Как они соотносятся с итераторами ранее рассмотренного стиля? Оба стиля имеют примерно одни и те же накладные расходы: один вызов функции на итерацию. С другой стороны, проще писать итератор с использованием подлинных итераторов (хотя мы можем получить эту же легкость при помощи сопрограмм). С другой стороны ранее рассмотренный стиль более гибкий. Во-первых, он позволяет два и более параллельных итерирования. (Например, рассмотрим случай обхода сразу двух файлов, сравнивая их слово за словом.) Во-вторых, он позволяет использование **break** и **return** внутри цикла. С подлинными итераторами **return** возвращает из анонимной функции, но не из цикла. Поэтому я обычно использую традиционные (то есть рассмотренные ранее) итераторы.

Упражнения

Упражнение 7.1. Напишите итератор `fromto` – такой, что следующие два цикла оказываются эквивалентными:

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

Можете ли вы реализовать это при помощи итератора без состояния?

Упражнение 7.2. Добавьте параметр шаг к предыдущему упражнению. Можете ли вы по-прежнему реализовать это при помощи итератора без состояния?



Упражнение 7.3. Напишите итератор `uniqewords`, который возвращает все слова из заданного файла без повторений.

(Подсказка: начните с кода `allwords` на листинге 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 в промежуточную форму перед его выполнением. (На самом деле многие интерпретируемые языки делают то же самое.) Наличие файла компиляции может звучать странным по отношению к интерпретируемому языку вроде Lua. Однако основной чертой интерпретируемых языков является не то, что они не компилируются, а то, что возможно (и легко) выполнять сгенерированный на лету код. Мы можем 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 блоков кода на Lua, но `dofile` – на самом деле вспомогательная функция: всю тяжелую работу выполняет `loadfile`. Как и `dofile`, `loadfile` загружает блок кода на Lua из файла, но он не выполняет этот блок. Вместо этого он только компилирует этот блок и возвращает откомпилированный блок как функцию. Более того, в отличие от `dofile`, `loadfile` не вызывает ошибки, а просто возвращает код ошибки, так что мы можем сами обработать эти ошибки. Мы можем определить `dofile` следующим образом:

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

Обратите внимание на использование функции `assert` для того, чтобы *вызывать ошибки* (raise error), если `loadfile` обрабатывает с ошибкой.

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 удобным способом. Более того, если нам нужно выполнить файл несколько раз, то мы можем один раз позвать `loadfile` и несколько раз позвать возвращенную им функцию. Этот подход гораздо дешевле, чем несколько раз вызывать `dofile`, поскольку файл компилируется всего один раз.

Функция `load` похожа на `loadfile`, за исключением того, что она берет блок кода не из файла, а из строки¹. Например, рассмотрим следующую строку:

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

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

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

Функция `load` довольно мощная, и мы должны использовать ее с осторожностью. Это также дорогая функция (по сравнению с некоторыми альтернативами) и может привести к коду, который очень тяжело понять. Перед тем как вы ее используете, убедитесь, что нет более простого способа решить задачу.

Если вы хотите быстрый и грязный `dostring` (то есть загрузить и выполнить блок), вы можете непосредственно использовать результат `load`:

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

Однако, если есть хотя бы одна синтаксическая ошибка, то `load` вернет `nil` и окончательной ошибкой будет что-то вроде *"attempt to call a nil value"*. Для более ясной обработки ошибок используйте `assert`:

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

Обычно нет никакого смысла звать функцию `load` для литерала (то есть явно заданной строки в кавычках). Например, следующие две строки примерно эквивалентны:

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

¹ В Lua 5.1 функция `loadstring` выполняет роль `load`.

Однако вторая строка гораздо быстрее, поскольку Lua скомпилирует функцию вместе с окружающим ее блоком. В первой строке вызов `load` включает отдельную компиляцию.

Поскольку `load` не компилирует с учетом лексической области действия, то рассмотренные две строки могут быть не совсем эквивалентны. Для того 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 переменной `i`, как и ожидалось, однако функция `f` работает с глобальной `i`, поскольку `load` всегда компилирует свои блоки в глобальном окружении.

Наиболее типичным использованием `load` является выполнение внешнего кода, то есть фрагментов кода, приходящих извне вашей программы. Например, вы можете хотеть построить график функции, заданной пользователем; пользователь вводит код функции, а вы затем используете `load` для того, чтобы выполнить его. Обратите внимание, что `load` ожидает получить блок, то есть операторы. Если вы хотите вычислить выражение, то вы можете дописать к началу выражения `return`, что даст вам оператор, возвращающий значение данного выражения. Посмотрим 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`, — это обычная функция, вы можете звать ее много 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 видной извне блока)
  print(string.rep("*", f()))
end
```

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

Мы также можем позвать функцию `load`, передав ей в качестве аргумента *читающую функцию* (reader function). Читающая функция

может возвращать блок кода частями; `load` вызывает эту функцию до тех пор, пока она не вернет **nil**, обозначающий конец блока. Например, следующий вызов эквивалентен `loadfile`:

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

Как мы увидим в главе 22, вызов `io.lines(filename, "*L")` возвращает функцию, которая, будучи вызванной, возвращает следующую строку из файла². Таким образом, `load` будет читать блок из файла строка за строкой. Следующий вариант похож, но более эффективен:

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

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

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

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

Как и любая другая функция, блоки могут определять свои 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 не пользоваться 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 описание `"local x = ..."` в начало блока, чтобы определить `x` как локальную переменную. Когда мы вызываем `f` с аргументом `i`, этот аргумент становится значением выражения `...`.

Функция `load` никогда не вызывает ошибки, в случае ошибки она просто возвращает **nil** и сообщение об ошибке:

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

² Опции для `io.lines` появились только в Lua 5.2.

Более того, у этих функций нет никакого побочного эффекта. Они только компилируют блок во внутреннее представление и возвращают результат как анонимную функцию. Распространенной ошибкой является то, что предполагается, что загрузка блока определяет функции (определенные в этом блоке). В Lua определения функций – это присваивания; и как таковые они происходят во время выполнения, а не во время компиляции. Например, допустим, что у нас есть файл `foo.lua` со следующим содержанием:

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

Затем мы выполняем команду

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

После этой команды `foo` откомпилирована, но еще не определена. Чтобы определить ее, мы должны выполнить блок:

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

В серьезных программах, которым нужно выполнять внешний код, вы должны обрабатывать все ошибки, возникающие при загрузке блока. Более того, вы можете захотеть запустить новый блок в защищенном окружении, чтобы избежать неприятных побочных эффектов. Мы подробно обсудим окружения в главе 14.

8.2. Предкомпилированный код

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

Простейшим путем получения предкомпилированного файла – также называемым в Lua *бинарным блоком* – является использование программы `luac`, которая входит в стандартную поставку. Например, следующий вызов создает файл `prog.lc` с предкомпилированной версией файла `prog.lua`:

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

Интерпретатор может затем выполнить этот файл, как и нормальный код на Lua, работая так же, как и с исходным файлом:

```
$ lua prog.lua
```

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

Мы можем написать простую замену `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 на Lua и возвращает ее предкомпилированный код как строку, правильно оформленную для ее загрузки в Lua.

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

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

(Мы не будем обсуждать внутренности Lua в этой книге; если вас интересует информация об этих кодах операций, то поиск в Интернете по словам `"lua opcode"` даст вам достаточно точную информацию.)

Код в предкомпилированной форме не всегда меньше исходного кода, но он загружается быстрее. Еще одним плюсом является то, что это дает вам защиту от случайных изменений в исходниках. Однако, в отличие от исходного кода, злонамеренно измененный бинарный код может привести к падению интерпретатора Lua или даже выполнить предоставленный пользователем машинный код. При запуске обычного кода вам беспокоиться нечего. Однако вам следует избегать запуска ненадежного кода в предкомпилированной форме. У функции `load` есть специальная опция именно для этой задачи.

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
```

Кроме обязательного первого аргумента, у `load` есть еще три необязательных аргумента. Вторым аргументом является имя блока, которое используется только в сообщениях об ошибках. Четвертый аргумент – это окружение, его мы подробно рассмотрим в главе 14. Третий аргумент – это именно то, что нас сейчас интересует; он управляет тем, какие типы блоков могут быть загружены. Если этот аргумент присутствует, то он должен быть строкой: строка `"t"` позволяет загружать лишь текстовые блоки, строка `"b"` позволяет загружать только бинарные (предкомпилированные) блоки, и строка `"bt"` (значение аргумента по умолчанию) позволяет загружать блоки обоих типов.

8.3. Код на C

В отличие от кода, написанного на Lua, код на C должен быть слинкован с приложением перед его использованием. В ряде популярных операционных систем простейшим путем добиться этого является использование возможности динамической линковки. Однако данная возможность не часть спецификаций ANSI C; поэтому нет переносимого пути для реализации этого.

Обычно Lua не включает возможности, которые не могут быть реализованы в ANSI C. Однако с динамической линковкой ситуация отличается. Мы можем рассматривать ее как основу всех других возможностей: как только она у нас есть, мы сразу можем подгружать любую возможность, которая сейчас не в Lua. Поэтому в данном случае Lua отказывается от правил переносимости и реализует динамическую линковку для ряда платформ. Стандартная реализация предлагает эту возможность для Windows, Mac OS X, Linux, FreeBSD, Solaris и большинства других реализаций UNIX. Перенос этой возможности на другие платформы не должен быть сложным; обратитесь к вашему дистрибутиву. (Для проверки этого выполните `print(package.loadlib("a", "b"))` из командной строки Lua и посмотрите на результат. Если он сообщает о несуществующем файле, то у вас есть поддержка динамической линковки. В противном случае сообщение об ошибке скажет, что данная возможность не поддерживается или не установлена.)

Lua предоставляет все возможности динамической линковки через одну функцию, `package.loadlib`. Эта функция получает два строковых аргумента: полный путь к библиотеке и имя функции из

этой библиотеки. Поэтому ее типичный вызов выглядит, как показано ниже:

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

Функция `loadlib` загружает указанную библиотеку и подключает ее к Lua. Однако он не вызывает заданную функцию. Вместо этого он возвращает функцию на C как функцию на 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 передать полный путь к библиотеке и правильное имя функции (включая подчеркивания в начале, добавляемые компилятором). Часто мы загружаем библиотеки на C, используя `require`. Эта функция ищет библиотеку и использует `loadlib` для загрузки инициализирующей функции для библиотеки. При вызове эта инициализирующая функция строит и возвращает таблицу с функциями из этой библиотеки, как делает обычная Lua-библиотека. Мы обсудим `require` в разделе 15.1 и дополнительную информацию о библиотеках на C в разделе 27.3.

8.4. Ошибки

*Errare humanum est*³. Поэтому мы должны обрабатывать ошибки так хорошо, как мы можем. Поскольку Lua – это язык для расширения, часто встраиваемый в приложение, мы не можем просто упасть или завершить работу в случае возникновения ошибки. Вместо этого, когда возникает ошибка, Lua прерывает выполнение текущего блока и возвращает управление в приложение.

Любая неожиданная ситуация, с которой сталкивается Lua, приводит к *вызову ошибки* (*raises an error*). Ошибки возникают, когда вы (точнее, ваша программа) не можете сложить значения, которые не являются числами, индексировать не таблицу и т. п. (Вы можете изменить это поведение, используя *метатаблицы*, как мы увидим позже.) Вы также можете явно вызвать ошибки при помощи вызова функции `error` сообщением об ошибке в качестве аргумента. Обычно эта функция является правильным способом сообщить об ошибке в вашем коде:

```
print "введите число:"  
n = io.read("*n")  
if not n then error("неверный ввод") end
```

³ Человеку свойственно ошибаться. (лат.)

Этот способ вызова `error` настолько распространен, что для этого в Lua есть встроенная функция `assert`:

```
print "введите число:"  
n = assert(io.read("*n"), "неверный ввод")
```

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 об ошибке, не обязателен. Однако имейте в виду, что `assert` – это обычная функция. Как и со всеми функциями, Lua всегда перед ее вызовом вычисляет ее аргументы. 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 если `n` – это число. Поэтому в подобных случаях может быть лучше использовать явный тест.

Когда функция обнаруживает непредвиденную ситуацию (*исключение*), она может пойти двумя путями: вернуть код ошибки (обычно **nil**) или вызвать ошибку, используя `error`. Не существует жестких правил для выбора между этими двумя вариантами, но мы можем предложить общий совет: исключение, которое легко обходится, должно вызывать ошибку; иначе следует вернуть код ошибки.

Например, давайте рассмотрим функцию `sin`. Как она должна себя вести, если ее аргументом является таблица? Предположим, она возвращает код ошибки. Если нам нужно проверять на ошибки, то мы можем написать что-то вроде

```
local res = math.sin(x)  
if not res then -- error?  
    <error-handling code>
```

Однако мы можем легко изменить это исключение *перед* вызовом функции:

```
if not tonumber(x) then -- x не число?  
    <error-handling code>
```

Часто мы не проверяем ни аргумент, ни результат вызова `sin`, если аргумент – не число, то это значит, что что-то не так в нашей программе. 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 надо себя вести, если попросили открыть

несуществующий файл? В этом случае не существует простого способа проверки на вызов на исключение перед вызовом этой функции. Во многих системах единственным способом проверить, что файл существует, является попробовать открыть его. Поэтому если `io.open` не может открыть файл по какой-то внешней причине (например, «файл не существует» или «нет прав!»), то она просто возвращает `nil` вместе с сообщением об ошибке. В этом случае у вас есть шанс обработать ситуацию подходящим образом, например попросив другое имя файла:

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

Если вы не хотите обрабатывать подобную ситуацию, но тем не менее все равно хотите быть в безопасности, то вы можете просто использовать `assert`:

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

Это типичная идиома для Lua: если `io.open` завершится с ошибкой, 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 при вызове `assert`.

8.5. Обработка ошибок и исключений

Для многих приложений вам не нужно делать какую-либо обработку ошибок в Lua; всю обработку делает само приложение. Вся работа Lua начинается с вызова приложением, обычно заключающимся в выполнении блока. Если возникает ошибка, то этот вызов возвращает код ошибки и приложение может соответствующим образом отреагировать. В случае отдельного интерпретатора его главный цикл просто печатает сообщение об ошибке и продолжает работать дальше.

Однако если вам надо обрабатывать ошибки в Lua, то вы должны *использовать функцию* `pcall` (protected call) для инкапсуляции своего кода.

Допустим, вы хотите выполнить фрагмент кода на Lua и поймать любую ошибку, возникающую при его выполнении. Вашим первым шагом будет заключить этот фрагмент кода в функцию; довольно часто для этого используются анонимные функции. Затем вы вызываете эту функцию, используя `pcall`:

```
local ok, msg = pcall(function ()
    <some code>
    if unexpected_condition then error() end
    <some code>
    print(a[i])    -- возможна ошибка: 'a' может не быть таблицей
    <some code>
end)
if ok then        -- не возникло ошибки при выполнении защищенного кода
    <regular code>
else              -- защищенный код вызвал ошибку; обработать ее
    <error-handling code>
end
```

Вызов `pcall` вызывает свой первый аргумент в *защищенном режиме*, так что перехватываются все ошибки во время выполнения функции. Если нет никаких ошибок, то вызов `pcall` возвращает **true** и все значения, возвращенные функцией. Иначе он возвращает **false** и сообщение об ошибке.

Несмотря на свое название, сообщение об ошибке не обязано быть строкой: вызов `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 надо для обработки исключений в Lua. Мы выбрасываем исключение при помощи `error` и перехватываем его при помощи `pcall`. Сообщение об ошибке идентифицирует тип ошибки.

8.6. Сообщения об ошибках и стек вызовов

Хотя в качестве сообщения об ошибке мы можем использовать значение любого типа, обычно сообщения об ошибках – это строки, опи-

сывающие, что пошло не так. В случае возникновения внутренней ошибки (например, попытка индексировать не таблицу) сообщение об ошибке генерирует Lua; иначе сообщением об ошибке становится значение, переданное функции `error`. Когда сообщение об ошибке является строкой, то 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
```

Сообщение об ошибке содержит имя файла (в примере это `stdin`) и номер строки в нем (в примере это 1).

Функция `error` имеет второй дополнительный параметр, который сообщает *уровень*, где надо сообщать ошибку; вы используете этот параметр, чтобы обвинить кого-то другого в случае ошибки. Например, вы написали функцию, которая сразу же проверяет, что она была вызвана надлежащим образом:

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

Затем кто-то вызывает вашу функцию с неправильным аргументом:

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

В этом случае Lua показывает на вашу функцию – в конце концов, это именно она вызвала `error`, – а не настоящего виновника, того, кто вызвал ее с неправильным аргументом. Для того чтобы это исправить, мы можем передать `error`, что ошибка, о которой вы сообщаете, возникла на уровне 2 в стеке вызовов (уровень 1 – это ваша функция):

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

Часто при возникновении ошибки мы хотим получить больше отладочной информации, чем просто место, где она возникла. Как минимум мы хотим стек вызовов, приведший к ошибке. Когда `pcall` возвращает сообщение об ошибке, она разрушает часть стека (часть от нее до момента возникновения ошибки). Соответственно, если мы хотим получить стек вызовов, то мы должны построить его до возврата из `pcall`. Для этого Lua предоставляет функцию `xpcall`. Кроме функции, которую нужно вызвать, она получает второй аргумент, *функцию обработки ошибки*. В случае возникновения ошибки Lua вызывает эту функцию обработки ошибки перед очисткой стека, поэтому она может использовать отладочную библиотеку для получения любой дополнительной информации об ошибке. Двумя наиболее распространенными обработчиками ошибок являются `debug.debug`, предоставляющий вам командную строку в Lua, чтобы вы могли сами посмотреть, что происходило в момент возникновения ошибки; и `debug.traceback`, который строит расширенное сообщение об ошибке, включающее в себя стек вызовов⁴.

Именно последнюю функцию использует самостоятельный интерпретатор для печати сообщений об ошибках.

Упражнения

Упражнение 8.1. Часто бывает нужно добавить код к началу загружаемого блока. (Мы уже видели пример в этой главе, когда мы добавляли код к **return**.) Напишите функцию `loadwithprefix`, которая работает как `load`, за исключением того, что она добавляет свой дополнительный аргумент к началу загружаемого блока.

Как и оригинальная функция `load`, `loadwithprefix` должна принимать блоки, представленные как строками, так и читающими функциями. Даже в случае, когда оригинальный блок является строкой, `loadwithprefix` не должна явно конкатенировать переданный аргумент с блоком. Вместо этого она должна вызвать `load` с соответствующей читающей функцией, которая сперва возвратит переданный аргумент, а потом — блок.

Упражнение 8.2. Напишите функцию `multiload`, которая обобщает `loadwithprefix`, получая на вход список читающих функций, как в следующем примере:

⁴ В главе 24 мы больше узнаем об этих функциях и об отладочной библиотеке.

```
f = multiload("local x = 10;",  
             io.lines("temp", "*L"),  
             " print(x)")
```

Для приведенного выше примера `multiload` должна загрузить блок, эквивалентный конкатенации строки `"local..."` с содержимым файла `temp` и строки `"print(x)"`. Как и функция `loadwithprefix`, данная функция сама ничего конкатенировать не должна.

Упражнение 8.3. Функция `string.rep` в листинге 8.2 использует *алгоритм двоичного умножения* (binary multiplication algorithm) для конкатенации n копий заданной строки s . Для любого фиксированного n мы можем создать специализированную версию `string.rep`, раскрывая цикл в последовательность команд `r=r..s` и `s=s..s`. В качестве примера для $n=5$ мы получаем 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 на Lua с соответствующими командами `r=r..s` и `s=s..s` и затем использовать `load` для получения итоговой функции. Сравните быстродействие функции `string.rep` и полученной вами функции.

Упражнение 8.4. Можете ли вы найти такое значение для p , что выражение `pcall(pcall,f)` вернет **false** как первое значение?



ГЛАВА 9

Сопрограммы

Сопрограмма похожа на нить (в смысле многонитевости): это поток выполнения со своим стеком, своими локальными переменными и своим *указателем команд* (instruction pointer); но он разделяет глобальные переменные и почти все остальное с другими сопрограммами. Основное отличие между нитями и сопрограммами – это то, что программа с несколькими нитями выполняет все эти нити параллельно. Сопрограммы работают совместно: в любой момент времени программа выполняет только одну из своих сопрограмм, и эта выполняемая сопрограмма приостанавливает свое выполнение, только когда явно попросит это.

Сопрограммы – это очень мощное понятие. И поэтому многие из их применений довольно сложны. Не волнуйтесь, если вы не поймете некоторые из примеров из этой главы при первом чтении. Вы можете дочитать до конца книги и вернуться сюда позже. Но, пожалуйста, вернитесь, это будет хорошо проведенное время.

9.1. Основы сопрограмм

Луа хранит все функции для работы с сопрограммами в таблице `coroutine`. Функция `create` создает новые сопрограммы. У нее всего один аргумент – функция, которую сопрограмма будет выполнять. Она возвращает значение типа `thread`, которое представляет из себя созданную 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: приостановлена (`suspended`), выполняется (`running`), уничтожена (`dead`) и нормальном (`normal`). Мы можем узнать состояние сопрограммы при помощи функции `status`:

```
print(coroutine.status(co)) --> suspended
```

Когда мы создаем сопрограмму, то она начинает в приостановленном состоянии; сопрограмма не начинает автоматически выполнять свое тело, когда мы ее создаем. Функция `coroutine.resume` продолжает (начинает) выполнение сопрограммы, меняя ее состояние из приостановленной в выполняемую:

```
coroutine.resume(co) --> hi
```

В этом первом примере тело сопрограммы просто печатает “hi” и прекращает выполнение, оставляя сопрограмму в уничтоженном состоянии:

```
print(coroutine.status(co)) --> dead
```

До сих пор сопрограммы выглядели просто как сложный способ вызова функций. Настоящая сила сопрограмм идет от функции `yield`, которая позволяет выполняемой сопрограмме приостановить свое выполнение так, что оно может быть продолжено позже. Давайте рассмотрим простой пример:

```
co = coroutine.create(function ()
  for i = 1, 10 do
    print("co", i)
    coroutine.yield()
  end
end)
```

Теперь, когда мы продолжаем выполнение этой функции, она начинает свое выполнение и выполняется до первого `yield`:

```
coroutine.resume(co) --> co 1
```

Если мы сейчас проверим ее статус, то увидим, что данная сопрограмма приостановлена и, следовательно, мы можем снова продолжить ее выполнение:

```
print(coroutine.status(co)) --> suspended
```

С точки зрения сопрограммы, вся деятельность, которая происходит, пока сопрограмма приостановлена, происходит внутри вызова `yield`. Когда мы продолжим выполнение сопрограммы, она возвращается из вызова `yield` и продолжает свое выполнение до следующего вызова `yield` или окончания работы сопрограммы:

```
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` цикл завершается и завершается выполнение функции без печати чего-либо. Если мы попытаемся снова продолжить ее выполнение, то `resume` вернет **false** и сообщение об ошибке:

```
print(coroutine.resume(co))
--> false cannot resume dead coroutine
```

Обратите внимание, что `resume` выполняет тело сопрограммы в защищенном режиме. Поэтому в случае возникновения какой-либо ошибки при выполнении сопрограммы Lua не будет показывать сообщения об ошибке, а просто вернет управление вызову `resume`.

Когда сопрограмма продолжает выполнение другой сопрограммы, то он не приостанавливается; в конце концов, мы не можем продолжить ее выполнение. Однако она и не является выполняемой, поскольку выполняемой сопрограммой является другая сопрограмма. Поэтому ее статус называется *нормальным*.

Полезной возможностью в Lua является то, что пара `resume` – `yield` может обмениваться данными. Первый вызов `resume` (у которого нет ожидающего его вызова `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 вызову `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 аргументы, переданные в соответствующий вызов `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 завершает свое выполнение, все значения возвращенные ее главной функцией, передаются как результат `resume`:

```
co = coroutine.create(function ()
    return 6, 7
end)
print(coroutine.resume(co)) --> true 6 7
```

Обычно мы редко используем все эти возможности 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, прежде чем мы пойдем дальше. Lua предлагает то, что называется *асимметричными сопрограммами*. Это значит, что у нее есть функция для приостановки выполнения сопрограммы и другая функция для продолжения выполнения приостановленной сопрограммы. В некоторых языках есть *симметричные сопрограммы*, когда есть только одна функция для передачи управления от одной сопрограммы другой.

Некоторые называют асимметричные сопрограммы полусопрограммами (не будучи симметричными, они не являются со-). Однако другие используют тот же термин полусопрограммы для обозначения ограниченной реализации сопрограмм, где сопрограмма может приостановить свое выполнение, только когда она не вызывает ни одну другую функцию, то есть когда у нее нет ожидающих вызовов. Другими словами, только главное тело такой сопрограммы может вызвать `yield`. Примером подобных полусопрограмм являются *генераторы* в Python.

В отличие от разницы между симметричными и несимметричными сопрограммами, разница между сопрограммами и генераторами (как реализовано в Python) гораздо глубже; генераторы просто не достаточно мощные, чтобы реализовать некоторые интересные конструкции, которые мы можем сделать с нормальными сопрограммами. Lua предлагает полноценные несимметричные сопрограммы. Те, кто предпочитают симметричные сопрограммы, могут реализовать их на основе асимметричных возможностей Lua. Это не сложно. (Фактически каждая передача управления выполняет `yield`, за которым следует `resume`.)

9.2. Каналы и фильтры

Одним из наиболее важных примеров использования сопрограмм является задача потребителя-производителя (*producer-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()      -- произвести новое значение
    send(x)                  -- послать его потребителю
  end
end
function consumer ()
  while true do
    local x = receive()      -- получить значение от производителя
    io.write(x, "\n")        -- потребить его
  end
end
```

(В этой реализации и производитель, и потребитель выполняются вечно. Однако их легко можно изменить для остановки, когда больше нет данных.) Задача здесь заключается в том, чтобы соединить вызовы `send` и `receive`. Это типичный пример задачи «у кого главный цикл». И производитель, и потребитель активны, у каждого есть свои главные циклы, и каждый предполагает, что другой – это вызываемый сервис. Для этого конкретного примера можно легко изменить структуру одной из функций, развернув ее цикл и сделав ее пассивной стороной. Однако в реальных случаях подобное изменение может быть далеко не так легко.

Сопрограммы предоставляют идеальный механизм для соединения производителя и потребителя, поскольку пара `resume-yield` переворачивает обычное соотношение между вызывающим и вызываемым. Когда сопрограмма вызывает `yield`, она не вызывает новую функцию; вместо этого она возвращает управление из текущего вызова (`resume`). Аналогично вызов `resume` не начинает новую функцию, а завершает вызов `yield`. Это именно то, что нам нужно для соединения `send` и `receive` таким образом, что каждый действует так, как будто главным является именно он, а второй является подчиненным. Поэтому `receive` продолжает выполнение производителя, так что он может произвести новое значение; и `send` возвращает это значение обратно потребителю:

```
function receive ()
  local status, value = coroutine.resume(producer)
  return value
end
```

```
function send (x)
  coroutine.yield(x)
end
```

Конечно, производитель также должен быть сопрограммой:

```
producer = coroutine.create(
  function ()
    while true do
      local x = io.read() -- произвести новое значение
      send(x)
    end
  end)
end)
```

При таком дизайне программа начинает с вызова потребителя. Когда потребителю нужно значение, он возобновляет работу производителя, который выполняется до тех пор, пока у него не будет готового значения, которое он передает потребителю, и не останавливается до тех пор, пока потребитель снова не продолжит его выполнение. Таким образом, мы получаем то, что называется дизайн, *управляемый потребителем* (consumer-driven). Другим вариантом было бы написать программу, используя дизайн, *управляемый производителем*, где потребитель является сопрограммой.

Мы можем расширить этот дизайн при помощи фильтров, которые являются заданиями, находящимися между производителем и потребителем и выполняющими преобразование данных. *Фильтр* – это производитель и потребитель в одно и то же время, поэтому он продолжает выполнение производителя для получения нового значения и использует `yield` для передачи этого 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. В конце нам просто надо создать компоненты, соединить их и начать выполнение итогового потребителя:

```
p = producer()
f = filter(p)
consumer(f)
```

Или еще проще:

```
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()      -- произвести новое значение
            send(x)
        end
    end)
end
function filter (prod)
    return coroutine.create(function ()
        for line = 1, math.huge do
            local x = receive(prod) -- получить новое значение
            x = string.format("%5d %s", line, x)
            send(x)                 -- послать его потребителю
        end
    end)
end
function consumer (prod)
    while true do
        local x = receive(prod)     -- получить новое значение
        io.write(x, "\n")           -- потребить новое значение
    end
end

```

Листинг 9.2. Функция для получения всех перестановок из n элементов a

```

function permgen (a, n)
    n = n or #a      -- по умолчанию 'n' - это размер 'a'
    if n <= 1 then   -- ничего не надо делать?
        printResult(a)
    else
        for i = 1, n do
            -- поместить i-й элемент в конец
            a[n], a[i] = a[i], a[n]
            -- создать все перестановки остальных элементов
            permgen(a, n - 1)
            -- восстановить i-й элемент
            a[n], a[i] = a[i], a[n]
        end
    end
end

```

Если вы подумали о каналах (pipe) в UNIX, то вы не одиноки. В конце концов, сопрограммы – это вариант *невывесняющей многозадачности* (non-preemptive multitasking). С каналами каждая задача выполняется как отдельный процесс; с сопрограммами каждая задача

выполняется как отдельная сопрограмма. Каналы предоставляют буфер между пишущим (производителем) и читающим (потребителем), поэтому возможна некоторая свобода в их относительных скоростях. Для канала это важно, поскольку цена переключения между процессами высока. С сопрограммами цена переключения между задачами намного меньше (примерно как вызов функции), поэтому пишущий и читающий могут идти нога в ногу.

9.3. Сопрограммы как итераторы

Мы можем рассматривать итераторы цикла как отдельный пример производителя-потребителя: итератор производит значения, потребляемые телом цикла. Поэтому является вполне естественным использовать сопрограммы для написания итераторов. Действительно, сопрограммы являются мощным инструментом для этой цели. Опять, ключевой особенностью является их способность перевернуть отношение между вызывающим и вызываемым. С этой особенностью мы можем писать итераторы, не волнуясь хранить состояние между последовательными вызовами итератора.

Для того чтобы проиллюстрировать этот вариант использования, давайте напишем итератор для перебора всех перестановок заданного массива. Написание подобного итератора не так легко, но довольно просто написать рекурсивную функцию, строящую все эти перестановки. Идея проста: по очереди поместить каждый элемент в конец массива и рекурсивно сгенерировать все оставшиеся перестановки. Код приведен в листинге 9.2. Для того чтобы он работал, мы должны написать соответствующую функцию `printResult` и вызвать `permgen` с надлежащими 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
```

После того как генератор готов, его очень легко преобразовать в итератор. Во-первых, мы заменим `printResult` на `yield`:

```
function permgen (a, n)
  n = n or #a
  if n <= 1 then
    coroutine.yield(a)
  else
    <как ранее>
```

Затем мы определяем фабрику, которая будет запускать генератор внутри сопрограммы, и создаем итерирующую функцию. Для получения следующей перестановки итератор просто продолжает выполнение сопрограммы:

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

После этого мы легко можем перебирать все перестановки массива при помощи оператора **for**:

```
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 типичный для Lua паттерн, который прячет возобновление сопрограммы внутри функции. Этот паттерн настолько распространен, что Lua предоставляет специальную функцию для него: `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 продолжает выполнение этой сопрограммы. В отличие от `resume`, она не возвращает код ошибки как первое значение; вместо этого она вызывает ошибку. Используя `wrap`, мы можем записать `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 продолжения ее выполнения. Однако она менее гибкая. Не существует способа проверить статус 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 предоставляют вариант совместной многонитевости. Каждая сопрограмма эквивалентна нити. Пара `yield-resume` переключает управление от одной нити к другой нити. Однако, в отличие от обычной многонитевости, сопрограмма не является вытесняющими (*preemptive*). Пока сопрограммы 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 вытесняемости. Вам не нужно беспокоиться об ошибках синхронизации, поскольку вся синхронизация явная. Вам нужно только убедиться в том, что сопрограмма вызывает `yield` вне критической области кода.

Однако с невытесняющей многонитевостью, как только какая-то нить вызывает блокирующую операцию, вся программа блокируется до тех пор, пока эта операция не завершится. Для большинства приложений это недопустимо, что ведет к тому, что многие программисты не рассматривают сопрограммы как альтернативу традиционной многонитевости. Как мы увидим здесь, у этой проблемы есть интересное (и очевидное притом) решение.

Давайте рассмотрим типичную многонитевую задачу: мы хотим скачать несколько файлов через HTTP. Для скачивания нескольких файлов мы сначала должны разобраться, как скачать один файл. В этом примере мы рассмотрим разработанную Диего Нехабом библиотеку *LuaSocket*. Для того чтобы скачать файл, надо сперва установить соединение с сайтом, содержащим данный файл, получить 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 сайт 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 сайт закрывает соединение, мы печатаем оставшиеся данные и выходим из цикла.

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 один файл за раз. Однако этот последовательный подход, когда мы начинаем читать файл только после того, как закончим с предыдущим файлом, слишком медленный. При чтении файла по сети программа проводит основную часть времени, ожидая данных. Более точно, она проводит большую часть времени блокированная в вызове `receive`. Поэтому программа может выполняться значительно быстрее, если будет сразу скачивать все файлы. Тогда, когда у соединения нет готовых данных, программа может читать данные из другого соединения. Ясно, что сопрограммы предоставляют удоб-

ный способ для организации этих одновременных скачиваний. Мы создаем новую нить для каждого скачиваемого файла. Когда у нити нет готовых данных, она передает управление диспетчеру, который вызывает другую нить.

Для того чтобы переписать программу с использованием сопрограмм, нам для начала надо переписать скачивающий код как функцию. Результат приведен в листинге 9.3. Поскольку нам не интересно содержимое файла, функция считает и печатает размер файла вместо записи файла в стандартный вывод (когда у нас несколько нитей читают сразу несколько файлов, на выходе получилась бы полная мешанина).

Листинг 9.3. Код для скачивания страницы из сети

```
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 получившемся коде мы используем вспомогательную функцию (`receive`) для получения данных из соединения. При последовательном подходе код бы выглядел 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 нет необходимых данных, то она вызывает `yield`. Новый код выглядит следующим образом:

```
function receive (connection)
  connection:settimeout(0)      -- не блокировать
  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 завершилась без выполнения запроса. В этом случае нить передает управление другой нити. 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 данные.

Листинг 9.4. содержит код диспетчера и дополнительный код. Таблица `threads` содержит список всех активных нитей для диспетчера. Функция `get` обеспечивает, что каждый скачиваемый файл скачивается в отдельной нити. Сам диспетчер – фактически это просто цикл, который перебирает все нити, запуская их на выполнение одну за другой. Также он удаляет из списка те нити, которые уже завершили скачивание. Цикл останавливается, когда больше не остается нитей.

Листинг 9.4. Диспетчер

```
threads = {} -- список всех работающих нитей
function get (host, file)
  -- создать сопрограмму
  local co = coroutine.create(function ()
    download(host, file)
  end)
  -- вставить ее в список
  table.insert(threads, co)
end
function dispatch ()
  local i = 1
  while true do
    if threads[i] == nil then -- больше нет нитей?
      if threads[1] == nil then break end -- список пуст?
      i = 1 - перезапустить цикл
    end
    local status, res = coroutine.resume(threads[i])
    if not res then -- нить завершила скачивание?
      table.remove(threads, i)
    else
      i = i + 1 -- переходим к следующей нити
    end
  end
end
```

Наконец, главная процедура 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-xhtml11-20020801/xhtml11.pdf")
get(host, "/TR/REC-html32.html")
get(host, "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt")

dispatch()    -- main loop
```

На моем компьютере скачивание этих четырех файлов с использованием сопрограмм занимает 6 секунд. С последовательным скачиванием это занимает больше, чем вдвое (15 секунд).

Несмотря на данную оптимизацию, эта последняя реализация еще 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 готовых для чтения данных, диспетчер постоянно переключается с нити на нить только для того, чтобы убедиться в том, что нет готовых данных. В результате эта реализация занимает почти в 30 раз больше времени CPU, чем последовательная версия.

Для того чтобы избежать подобной ситуации, мы можем использовать функцию `select` из библиотеки `LuaSocket`: она позволяет заблокировать программу, находящуюся в ожидании, пока изменится статус в группе соединений. Изменения в реализации незначительны: нам нужно изменить только диспетчер, как показано в листинге 9.5. В цикле новый диспетчер собирает в таблице `timedout` соединения, для которых нет готовых данных. (Помните, что `receive` передает подобные соединения функции `yield`, таким образом снова запуская их). Если ни у одного соединения нет готовых данных, то диспетчер вызывает `select` для ожидания того, когда хотя бы у одного из этих соединений изменится статус. Эта окончательная реализация работает так же быстро, как и предыдущая. Однако она использует лишь немногим более времени CPU, 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 нитей нет?
      if threads[1] == nil then break end
      i = 1                        -- начать цикл сначала
      timedout = {}
    end
    local status, res = coroutine.resume(threads[i])
```

```
if not res then                -- нить закончила свою работу?
    table.remove(threads, i)
else                           -- вышло время ожидания
    i = i + 1
    timeout[#timeout + 1] = res
    if #timeout == #threads then -- все нити блокированы?
        socket.select(timeout)
    end
end
end
end
end
```

Упражнения

Упражнение 9.1. Используйте сопрограммы для того, чтобы преобразовать упражнение 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 (невывесняющая многонитевость).

Упражнение 9.3. Реализуйте функцию `transfer` на Lua. Если подумать о том, что вызовы `resume-yield` аналогичны вызову функции и возврату из нее, то эта функция будет как `goto`: она прерывает текущую сопрограмму и возобновляет любую другую сопрограмму, переданную как аргумент.

(Подсказка: используйте аналог процедуры `dispatch` для управления вашими сопрограммами. Тогда `transfer` передаст управление диспетчеру, сообщая о том, какую следующую сопрограмму нужно запустить, и диспетчер вызовет для нее `resume`).



ГЛАВА 10

Законченные примеры

Заканчивая введение в язык, мы покажем три простые, но законченные программы. Первая программа решает задачу о восьми королевах. Вторая программа печатает самые часто встречающиеся слова в тексте. Последний пример – это реализация цепи Маркова, описанная Керниганом и Пайком 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*: нужно так расставить восемь королей на шахматной доске, чтобы ни одна из королей не была под боем.

Первым шагом решения данной задачи нужно заметить, что у каждого решения должно быть ровно по одной королеве в каждой строке. Таким образом, мы можем представить решение как массив из восьми чисел, по одному для каждой строки; каждое число сообщает нам, в каком столбце расположена королева в соответствующей строке. Например, массив {3, 7, 2, 1, 8, 6, 5, 4} обозначает, что одна королева находится в строке 1 в столбце 3, другая – в строке 2 в столбце 7 и т. д. (Обратите внимание, что это не является допустимым решением; например, королева в строке 3 в столбце 2 нападает на королеву в строке 4 в столбце 1). Также обратите внимание, что любое решение должно быть перестановкой чисел от 1 до 8, так как решение должно содержать по одной королеве в каждом столбце.

Полная программа приведена в листинге 10.1. Первая функция – это `isplaceok`, которая проверяет, что заданная позиция на доске не попадает под бой ранее размещенных королей. Помня, что не может быть двух королей на одной строке, эта функция проверяет, что нет двух королей на одном столбце или одной диагонали с заданным положением.

Листинг 10.1. Программа для восьми королей

```

local N = 8      -- размер доски

-- проверяет, что положение (n,c) не находится под боем
local function isplaceok (a, n, c)
  for i = 1, n - 1 do -- для каждой ранее 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 -- положение под боем
    end
  end
  return true -- не под боем
end

-- напечатать 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

-- добавить к доске 'a' всех королей от 'n' до 'N'
local function addqueen (a, n)
  if n > N then -- все королевы были размещены?
    printsolution(a)
  else -- попытаться разместить n-ую королеву
    for c = 1, N do
      if isplaceok(a, n, c) then
        a[n] = c -- поместить n-ую королеву в столбец 'c'
        addqueen(a, n + 1)
      end
    end
  end
end

-- запустить программу
addqueen({}, 1)

```

Далее у нас идет функция `printsolution`, которая печатает шахматную доску. Она просто обходит всю доску, печатая 'X' в местах с королевой и '-' во всех остальных местах. Каждый результат выглядит, как показано ниже:

```

X - - - - -
- - - - X - -
- - - - - - X
- - - - - X -
- - X - - - -
- - - - - X -
- X - - - - -
- - - X - - -

```

Последняя функция `addqueen` является сердцем программы. Сначала она проверяет, является ли решение законченным, и если да, то печатает это решение. В противном случае она перебирает все столбцы; для каждого незанятого столбца программа помещает туда королеву и рекурсивно пытается разместить оставшихся королев.

10.2. Самые часто встречающиеся слова

Наш следующий пример — это простая программа, которая читает текст и печатает самые часто встречающиеся слова из этого текста.

Главная структура данных данной программы — это просто таблица, которая сопоставляет каждому слову его частоту. С использованием этой структуры данных у программы есть три основные задачи:

- Прочитать текст, посчитав число вхождений каждого слова.
- Отсортировать список слов по убыванию частоты встречаемости каждого слова.
- Напечатать первые n элементов из отсортированного списка.

Для чтения текста мы можем использовать итератор `allwords`, который мы разобрали в разделе 7.1. Для каждого слова, которое мы читаем, мы увеличиваем соответствующий счетчик:

```

local counter = {}
for w in allwords do
    counter[w] = (counter[w] or 0) + 1
end

```

Следующая задача — это отсортировать список слов. Однако, как внимательный читатель мог заметить, у нас нет списка слов! Однако его легко создать, используя слова, которые являются ключами в таблице `counter`:

```

local words = {}
for w in pairs(counter) do

```



```
words[#words + 1] = w
end
```

Листинг 10.2. Программа для печати самых часто встречающихся 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
```

Теперь, когда у нас есть список, мы можем отсортировать его при помощи функции `table.sort`, которую мы обсуждали в главе 6:

```
table.sort(words, function (w1, w2)
  return counter[w1] > counter[w2] or
    counter[w1] == counter[w2] and w1 < w2
end)
```

Полная программа приведена в листинге 10.2. Обратите внимание на использование сопрограмм в итераторе `auxwords`. В последнем цикле, печатающем результат, программа считает, что ее первый аргумент – это число слов, которое нужно напечатать, и использует значение 10, если аргументов передано не было.

10.3. Цепь Маркова

Наш последний пример – это реализация *цепи Маркова*. Программа генерирует псевдослучайный текст на основании того, какие слова могут следовать за последовательностью из n предыдущих слов в тексте. Для этой реализации мы будем считать, что n равно 2.

Первая часть читает основной текст и строит таблицу, которая для каждых двух слов дает список всех слов, которые могут за ними следовать в основном тексте. После построения таблицы программа использует ее для построения случайного текста, где каждое слово следует за двумя предыдущими с той же вероятностью, что и в базовом тексте. Как результат мы получаем текст, который случаен, но не совсем. Например, применив его к английскому тексту данной книги, мы получим тексты 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, соединяя два слова при помощи пробела:

```
function prefix (w1, w2)
  return w1 .. " " .. w2
end
```

Мы будем использовать строку NOWORD (" \backslash n") для инициализации префиксных слов и обозначения конца текста. Например, для текста «the more we try the more we do» таблица следующих далее слов будет выглядеть, как показано ниже:

```
{ ["\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 того чтобы вставить в таблицу новое слово, мы будем использовать следующую функцию:

```

function insert (index, value)
  local list = statetab[index]
  if list == nil then
    statetab[index] = {value}
  else
    list[#list + 1] = value
  end
end
end

```

Она сначала проверяет, что у данного префикса уже есть список; если нет, то создает новый список с переданным словом. Иначе она вставляет переданное слово 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-w2`, и затем обновляем значения `w1` и `w2`.

После построения таблицы программа начинает строить текст, состоящий из MAXGEN слов. Для начала она задает значения переменным `w1` и `w2`. Затем для каждого префикса она случайно выбирает следующее слово из списка допустимых слов, печатает это слово и обновляет значения `w1` и `w2`. Листинги 10.3 и 10.4 содержат полную программу. В отличие от нашего предыдущего примера с наиболее часто встречающимися словами, здесь мы используем реализацию `allwords`, основанную на замыканиях.

Листинг 10.3. Дополнительные определения для программы с цепью Маркова

```

function allwords ()
  local line = io.read() -- текущая строка
  local pos = 1          -- текущая положение в строке
  return function ()     -- итерирующая функция
    while line do        -- повторять пока остались строки
      local s, e = string.find(line, "%w+", pos)
      if s then          -- нашли слово?
        pos = e + 1      -- обновить положение
        return string.sub(line, s, e) -- вернуть слово
      else
        line = io.read() -- слово не найдено; попробуем след. строку
        pos = 1          -- начать с начала строки
      end
    end
  end
  return nil            -- больше строк нет, конец обхода
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 для цепи Маркова

```
local N = 2
local MAXGEN = 10000
local NOWORD = "\n"

-- построить 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)

-- сгенерировать текст
w1 = NOWORD; w2 = NOWORD      -- инициализировать
for i = 1, MAXGEN do
    local list = statetab[prefix(w1, w2)]
    -- выбрать случайное слово 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 восьми королевами, чтобы она останавливалась после печати первого решения.

Упражнение 10.2. Альтернативной реализацией задачи о восьми королевах может быть построение всех перестановок чисел от 1 до 8 и проверка, какие из них допустимы. Измените програм-

му для использования 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`.)

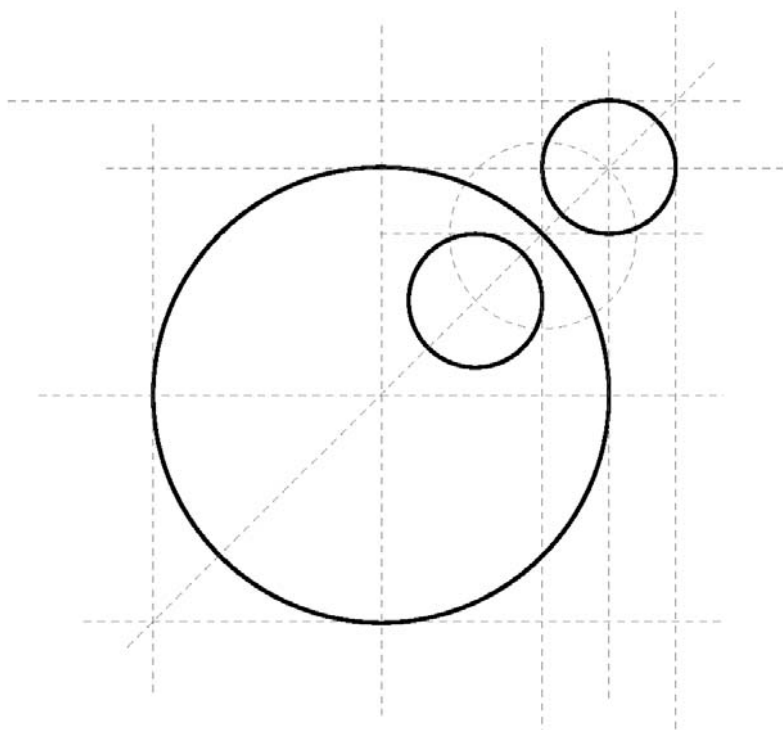
Упражнение 10.3. Когда мы применяем программу определения самых часто встречаемых слов, то обычно самыми часто встречаемыми словами оказываются короткие неинтересные слова вроде артиклей и предлогов. Измените программу так, чтобы она пропускала слова, состоящие из менее чем четырех букв.

Упражнение 10.4. Обобщите алгоритм цепи Маркова так, чтобы можно было использовать любой размер в качестве длины префикса.



Часть II

ТАБЛИЦЫ И ОБЪЕКТЫ





ГЛАВА 11

Структуры данных

Таблицы в Lua – это не просто структура данных, – это основная и единственная структура данных. Все структуры, которые предлагают другие языки, – массивы, записи, списки, очереди, множества – могут быть представлены в Lua при помощи таблиц. Более того, таблицы в 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 в Lua (и иногда мы это делаем), таблицы – гораздо более мощные, чем массивы и списки; многие алгоритмы с использованием таблиц становятся почти тривиальными. Например, мы редко используем поиск в Lua, поскольку таблицы предоставляют прямой доступ к значениям различных типов.

Требуется время, чтобы понять, как эффективно использовать таблицы в Lua. В этой главе я покажу, как реализовать типичные структуры данных при помощи таблиц, и приведу примеры их использования. Мы начнем с массивов и списков не потому, что они понадобятся нам для других структур, но поскольку большинство программистов уже знакомы с ними. Мы уже видели основу этого материала в предыдущих главах, но я также повторю его здесь.

11.1. Массивы

Мы реализуем массивы в Lua, просто индексируя таблицы целыми числами. Таким образом, массивы не имеют фиксированного размера и растут по необходимости. Обычно при инициализации массива мы неявно задаем 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
```

Оператор длины (`#`) использует это для определения длины массива:

```
print(#a) --> 1000
```

Вы можете начать массив с нуля или любого другого значения:

```
-- создаем массив с индексами от -5 до 5
a = {}
for i = -5, 5 do
    a[i] = 0
end
```

Однако обычно в Lua принято начинать массивы с индекса 1. Библиотеки Lua следуют этому соглашению; так же как и оператор длины. Если ваши массивы не начинаются с 1, то вы не сможете использовать эти возможности языка.

Мы можем использовать конструктор для создания и инициализации массива одним выражением:

```
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, можно создать матрицу из нулей размером M на N при помощи следующего кода:

```
mt = {}          -- создать матрицу
for i = 1, N do
    mt[i] = {}    -- создать строку
    for j = 1, M do
        mt[i][j] = 0
    end
end
```

Поскольку таблицы являются объектами в Lua, для создания матрицы нужно явно создавать каждую строку. С одной стороны, это более громоздко, чем просто объявить матрицу, как это делается в

языках C и Pascal. С другой – это дает больше гибкости. Например, вы можете создать треугольную матрицу, изменив цикл `for j=1, M do...end` в предыдущем фрагменте кода на `for j=1, i do...end`. С этим кодом треугольная матрица будет использовать только половину памяти по сравнению с исходным примером.

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 добавить второй индекс. С этим подходом следующий код создаст нашу матрицу из нулей размером M на N :

```
mt = {} -- создать 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, просто соединив эти строки с некоторым символом между ними. Например, вы можете создать матрицу `m` со строковыми индексами `s` и `t` при помощи следующего кода `m[s..":":t]`, при условии, что `s` и `t` не содержат двоеточия; в противном случае пары вроде `("a:", "b")` и `("a", ":b")` обе дадут один и тот же индекс `"a::b"`. Когда сомневаетесь, то вы можете использовать управляющий символ вроде `'\0'` для разделения индексов.

Довольно часто приложения используют *разреженную матрицу*, то есть матрицу, где большинство элементов – либо 0, либо *nil*. Например, вы можете представить граф при помощи его матрицы связности, в которой значение в позиции m, n равно x , если между узлами m и n есть соединение ценой x . Когда эти узлы не соединены, то значение в позиции m, n равно *nil*. Для того чтобы представить граф с десятью тысячами узлов, где каждый узел имеет около пяти соседей, вам нужна матрица со ста миллионами возможных элементов, но только пятьдесят тысяч из них будут не равны *nil* (пять ненулевых столбцов для каждой строки, соответствующих пяти соседям). Многие книги по структурам данных детально обсуждают, как можно реализовать подобные разреженные матрицы, не тратя на них 400 Мб памяти, но вам редко понадобятся подобные приемы при программировании на Lua. С нашим первым представлением (таблица таблиц) вам понадобятся десять тысяч таблиц, каждая из которых содержит около пяти элементов, то есть всего около пятидесяти тысяч значений. При

втором представлении у нас будет одна таблица с пятьюдесятью тысячами элементов. Какое бы представление вы не использовали, вам понадобится память только для элементов, отличных от *nil*.

При работе с разреженными матрицами мы не можем использовать оператор длины из-за дырок (значений *nil*) между элементами. Однако это не большая потеря; даже если бы мы могли его использовать, то делать этого не стоило бы. Для большинства операций было крайне неэффективно перебирать все эти пустые элементы. Вместо этого мы можем использовать `pairs` для обхода только элементов, отличных от *nil*. Например, для того 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
```

Обратите внимание, однако, что ключи не имеют какого-то определенного порядка в таблице, поэтому итерирование при помощи `pairs` не гарантирует, что мы посетим все столбцы по возрастанию. Для некоторых задач (например, нашего предыдущего примера) это не проблема. Для других целей вы можете использовать отличные представления, например связанные списки.

11.3. Связанные списки

Поскольку таблицы являются динамическими сущностями, то реализовать связанные списки в Lua довольно легко. Каждый узел представлен таблицей, и ссылки являются просто полями таблицы, которые содержат ссылки на другие таблицы. Например, давайте реализуем простейший список, где каждый узел содержит два поля, `next` и `value`. Корнем списка является обычная переменная:

```
list = nil
```

Для того чтобы вставить элемент со значением `v` в начало списка, мы делаем:

```
list = {next = list, value = v}
```

Для обхода списка мы можем написать:

```
local l = list
while l do
```

```
<visit l.value>
l = l.next
end
```

Другие варианты списков, например двунаправленные или круговые 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 как (неограниченный) массив.

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, передвигая остальные элементы массива. 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
```

Для того чтобы не загрязнять глобальное пространство имен, мы определим все операции по работе со списком внутри таблицы, которую мы назовем `List` (таким образом, мы создадим *модуль*). Тогда мы можем переписать наш последний пример следующим образом:

```
List = {}
function List.new ()
    return {first = 0, last = -1}
end
```

Теперь мы можем вставлять и удалять элементы с любого конца 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    -- позволим сборщику мусора его удалить
    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    -- позволим сборщику мусора его удалить
    list.last = last - 1
    return value
end
```

Если вы будете использовать эту структуру классическим способом, 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 спокойно можете индексировать их с 1 до 20 или с 16 777 216 до 16 777 236. Поскольку Lua использует числа с двойной точностью для представления чисел, ваша программа может работать на протяжении двухсот лет, делая по миллиону вставок в секунду, прежде чем возникнет проблема с переполнением.

11.5. Множества и наборы

Предположим, вы хотите перебрать все идентификаторы, используемые в программе; каким-то образом вам нужно отфильтровывать зарезервированные слова. Некоторые программисты на C могут попытаться использовать для представления множества зарезервированных слов массив строк и затем для проверки того, является ли заданное слово зарезервированным, осуществлять поиск в этом массиве. Для ускорения поиска вы можете даже использовать бинарное дерево для представления множества.

В Lua эффективным и простым способом представления множеств будет использовать элементы как *индексы* в таблице. Тогда вместо поиска, содержит ли таблица заданное слово, можно просто попро-

бовать проиндексировать таблицу этим словом и посмотреть, равен ли полученный результат *nil*. Для примера мы можем использовать следующий код:

```
reserved = {
  ["while"] = true,    ["end"] = true,
  ["function"] = true, ["local"] = true,
}

for w in allwords() do
  if not reserved[w] then
    <do something with 'w'>    -- 'w' незарезервированное слово
  end
end
```

(Поскольку эти слова зарезервированы в Lua, то мы не можем использовать их в качестве идентификаторов; например, мы не можем записать `while=true`. Вместо этого мы пишем `["while"]=true`.)

Вы можете также использовать более ясную инициализацию при помощи дополнительной функции, которая строит множество:

```
function Set (list)
  local set = {}
  for _, l in ipairs(list) do set[l] = true end
  return set
end
reserved = Set{"while", "end", "function", "local", }
```

Наборы, также называемые *мультимножествами*, отличаются от обычных множеств тем, что каждый элемент может встречаться несколько раз. Простое представление наборов в Lua похоже на предыдущее представление для множеств, но с каждым ключом связан соответствующий счетчик. Для того чтобы вставить элемент, мы увеличиваем его счетчик:

```
function insert (bag, element)
  bag[element] = (bag[element] or 0) + 1
end
```

Для удаления элемента мы уменьшаем его счетчик:

```
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 не равен нулю.

11.6. Строчные буферы

Пусть вы работаете с текстом и читаете файл строка за строкой. Тогда ваш код может выглядеть 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 в 1 Мб занимает 1,5 минуты на моем старом компьютере¹.

Почему это так? Чтобы понять, что происходит, представим, что мы находимся внутри цикла; каждая строка состоит из 20 байтов, и мы уже прочли 2500 строк, поэтому `buff` – это 50 Кб строка. Когда Lua соединяет `buff..line.."\n"`; она выделяет новую строку в 50 020 байт и копирует 50 000 байт из `buff` в эту новую строку. Таким образом для каждой новой строки 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) Lua уже переместил более 2 Мб памяти. Когда Lua завершит чтение 350 Кб, уже будет перемещено в памяти 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, это не типичная проблема. Для маленьких строк приведенный выше цикл отлично работает. Для чтения всего файла Lua предоставляет `io.read("*a")`, данный вызов читает сразу весь файл. Однако иногда мы сталкиваемся с этой проблемой. Для борьбы с подобной проблемой Java использует структуру `StringBuffer`. В Lua в качестве строкового буфера мы можем использовать таблицу. Ключом к этому подходу является функция `table.concat`, которая возвращает результат конкатенации всех строк из заданного списка. При помощи `concat` мы можем переписать наш предыдущий код следующим образом:

```
local t = {}
for line in io.lines() do
    t[#t + 1] = line .. "\n"
```

¹ «Мой старый компьютер» – это одноплатный 32-битовый Pentium 3 ГГц. Все быстродействие для этой книги измерялось на этом компьютере.

```
end
local s = table.concat(t)
```

Этот алгоритм требует менее 0,5 секунды для чтения того же самого файла, который требовал почти минуту с ранее использованным кодом. (Несмотря на это, для чтения всего файла лучше использовать `io.read` с опцией `"*a"`.)

Мы можем сделать даже лучше. Функция `concat` берет на вход второй необязательный аргумент, который является разделителем, который будет вставляться между строками. Используя этот сепаратор, мы можем избавиться от необходимости вставлять каждый раз символ `'\n'`:

```
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 символ `'\n'`. Эта последняя операция конкатенации копирует получаемую строку, что может потребовать заметного времени. Не существует способа заставить `concat` вставить дополнительный разделитель, но мы можем легко добиться этого, просто добавив в `t` пустую строку:

```
t[#t + 1] = ""
s = table.concat(t, "\n")
```

Дополнительный символ `'\n'`, который `concat` добавит перед последней строкой, — это то, что нам нужно.

11.7. Графы

Как и любой разумный язык, Lua предлагает различные реализации для графов, каждый из которых лучше подходит для своего типа алгоритмов. Здесь мы рассмотрим простую объектно-ориентированную реализацию, в которой мы будем представлять узлы как объекты (точнее, таблицы, конечно) и дуги как ссылки между узлами.

Мы будем представлять каждый узел как таблицу с двумя полями: `name`, являющийся именем узла, и `adj`, являющийся множеством узлов, соединенных с данным. Поскольку мы будем читать граф из текстового файла, нам будет нужен способ найти узел по его имени. Для этого мы будем использовать дополнительную таблицу. Функция `name2node`, получив имя узла, будет возвращать данный узел:

```
local function name2node (graph, name)
  local node = graph[name]
  if not node then
    -- узла еще нет, создать новый
    node = {name = name, adj = {}}
    graph[name] = node
  end
  return node
end
```

Листинг 11.1 содержит функцию, которая будет строить граф. Она читает файл, где каждая строка содержит имена двух узлов, обозначая что есть дуга, ведущая от первого узла ко второму. Для каждой строки она использует функцию `string.match`, для того чтобы разбить строку на два имени, далее находит соответствующие узлы (создавая их при необходимости) и соединяет их.

Листинг 11.1. Чтение графа из файла

```
function readgraph ()
  local graph = {}
  for line in io.lines() do
    -- разбить строку на два имени
    local namefrom, nameto = string.match(line, "(%S+)%s+(%S+)")
    -- найти соответствующие узлы
    local from = name2node(graph, namefrom)
    local to = name2node(graph, nameto)
    -- добавить 'to' к списку связей узла 'from'
    from.adj[to] = true
  end
  return graph
end
```

Листинг 11.2 иллюстрирует алгоритм, использующий подобные графы. Функция `findpath` ищет путь между двумя узлами, используя обход в глубину. Ее первый параметр – это текущий узел; второй задает желаемый 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 посещен?
    return nil                 -- здесь пути нет
end
visited[curr] = true           -- пометим узел как посещенный
path[#path + 1] = curr         -- добавим к пути
if curr == to then             -- цель?
    return path
end
                                -- попробуем 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              -- удалить 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 были бы равны нулю, если очередь пуста.

Упражнение 11.2. Повторите упражнение 10.3, только, вместо того чтобы использовать длину как критерий для отбрасывания слова, теперь программа должна прочесть из специального файла список слов, которые нужно пропускать.

Упражнение 11.3. Измените структуру графа так, чтобы она содержала метку для каждой дуги. Каждая дуга также должна быть представлена при помощи объекта с двумя полями: меткой и узлов, на который она показывает. Вместо множества соседних узлов каждый узел должен содержать список дуг, исходящих из данного узла.

Измените функцию `readgraph` так, чтобы она из каждой строки файла читала два имени узлов и метку (считая, что метка это число).

Упражнение 11.4. Используйте представление графа из предыдущего упражнения, где метка каждой дуги представляет собой расстояние между соединяемыми ей узлами. Напишите функцию, которая находит кратчайший путь между двумя узлами. (*Подсказка:* Используйте алгоритм Дейкстры.)



ГЛАВА 12

Файлы данных и персистентность

При работе с файлами с данными обычно гораздо проще писать файлы, 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 всех типов данных, который корректный файл с данными может содержать, программа также должна разумно обрабатывать и плохие файлы. Поэтому написание корректно работающих процедур для чтения данных всегда сложно.

В этой главе мы увидим, как можно использовать Lua для того, чтобы устранить весь код по чтению данных из наших программ, просто записывая данные в подходящем формате.

12.1. Файлы с данными

Конструкторы таблиц представляют интересную альтернативу форматам данных. При помощи небольшой дополнительной работы при записи данных чтение становится тривиальным. Подход заключается в том, чтобы писать наш файл с данными как программу на Lua, которая при выполнении создает необходимые данные.

Как обычно, для того чтобы было ясно, давайте рассмотрим пример. Если наш файл с данными находится в определенном формате, например CSV 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 каждую запись как конструктор 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{code}` – это то же самое что и `Entry({code})`, то есть вызов функции `Entry` с таблицей в качестве единственного аргумента. Поэтому приведенный выше фрагмент данных – это на самом деле программа на Lua. Для того чтобы прочесть такой файл, нам просто нужно выполнить его с надлежащим образом определенной функцией `Entry`. Например, следующая программа считает число записей в файле:

```
local count = 0  
function Entry () count = count + 1 end  
dofile("data")  
print("число записей: " .. count)
```

Следующая программа строит множество всех имен авторов, найденных в файле, и печатает их (не обязательно в том же порядке, в котором они встретились в файле):

```
local authors = {} -- множество авторов  
function Entry (b) authors[b[1]] = true end  
dofile("data")  
for name in pairs(authors) do print(name) end
```

Обратите внимание на подход, использованный в этих фрагментах кода: функция `Entry` выступает в роли функции обратного вызова (callback), которая вызывается во время выполнения `dofile` для каждой записи в файле.

Когда нас не волнует размер файла, мы можем в качестве нашего представления использовать пары имя–значение¹:

```
Entry{  
  author = "Donald E. Knuth",  
  title = "Literate Programming",  
  publisher = "CSLI",  
  year = 1992  
}  
Entry{  
  author = "Jon Bentley",
```

¹ Если этот формат напоминает вам BibTeX, то это не случайность. Формат BibTeX был одним из источников, определившим вид конструкторов в Lua.

```
title = "More Programming Pearls",  
year = 1990,  
publisher = "Addison-Wesley",  
}
```

Этот формат – это то, что мы называем *самоописывающий формат* данных, поскольку каждый фрагмент данных содержит краткое описание его значения. Самоописывающие данные более читаемы (как минимум людьми), чем CSV или другой компактный формат; их легко редактировать при необходимости; и они позволяют нам вносить небольшие изменения в базовый формат без необходимости изменять файлы с данными. Например, если мы добавим новое поле, то нам нужно только изменить читающую программу, предоставив значение по умолчанию, когда поле не указано.

При помощи формата имя–значение наша программа для составления списка авторов становится, как показано ниже:

```
local authors = {}          -- множество для 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 только изменить 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, приведенная выше программа для составления списка авторов обрабатывает 1 Мб данных за одну десятую секунды². И это не случайно. Описание данных было одним из главных приложений Lua с момента создания, и мы уделяем много внимания тому, чтобы ее компилятор быстро работал для больших программ.

12.2. Сериализация

Часто нам нужно сериализовать какие-то данные, то есть перевести данные в поток байтов или символов, который мы можем записать в файл или послать по сети. Мы можем представлять сериализованные данные как код на Lua таким образом, что при выполнении этого кода он восстанавливает сохраненные значения для выполняющей его программы.

² Для моего старого компьютера.

Обычно если мы хотим восстановить значение глобальной переменной, то наш блок кода будет чем-то вроде `varname=exp`, где *exp* – 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 <другие случаи>
  end
end
```

При записи числа в десятичном виде есть риск потерять точность. В Lua 5.2 можно использовать шестнадцатеричный формат, для того чтобы избежать подобной проблемы:

```
if type(o) == "number" then
  io.write(string.format("%a", o))
```

При использовании этого формата ("`%a`") прочитанное значение будет состоять из точно тех же самых битов, что и исходное.

Для строки наивным подходом было бы что-то вроде следующего:

```
if type(o) == "string" then
  io.write("'", o, "'")
```

Однако если строка содержит специальные символы (такие как кавычки или `\n`), то получившийся код уже не будет программой на Lua.

Вам может показаться, что эту проблему можно решить, изменив тип кавычек:

```
if type(o) == "string" then
  io.write("[", o, "]")
```

Однако будьте осторожны. Если вы попытаетесь сохранить что-то вроде `"]]..os.execute('rm *')..[["` (например, передав данную строку в качестве адреса), то получившийся блок кода будет:

```
varname = [ [ ] ]..os.execute('rm *')..[ [ ] ]
```

В результате вы получите неприятный сюрприз при попытке прочесть такие «данные».

Простейшим способом записать строку безопасно будет использование опции `"%q"` из функции `string.format`. Она окружает строку двойными кавычками и безопасным образом представляет двойные кавычки и некоторые другие символы внутри строки:

```
a = 'a "problematic" \\string'
print(string.format("%q", a))    --> "a \"problematic\" \\string"
```

Используя эту возможность, наша функция `serialize` может выглядеть 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 <другие случаи>
  end
end
```

Начиная с версии 5.1 Lua предлагает другой способ записи строк безопасным образом, при помощи записи `[=[...]=]` для длинных строк. Однако этот способ записи в основном предназначен для написанного пользователем кода, когда мы никоим образом не хотим менять строку символов. В коде, генерируемом автоматически, легче использовать `"%q"` из `string.format`.

Если же вы тем не менее хотите использовать подобную запись для автоматически генерируемого кода, то вам нужно обратить внимание на некоторые детали. Первой является то, что вам нужно подобрать правильное количество знаков равенства. Хорошим вариантом является число больше, чем встречается в исходной строке. Поскольку строки, содержащие большое количество знаков равенства, не являются редкостью (например, комментарии, разделяющие блоки кода), то 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 игнорирует символ `'\n'` в начале длинной строки; простейшим способом борьбы с этим является добавление символа `'\n'`, который будет отброшен.

Листинг 12.1. Вывод произвольной строки символов

```
function quote (s)
  -- найти максимальную длину последовательности знаков равенства
  local n = -1
  for w in string.gmatch(s, "[=]*") do
    n = math.max(n, #w - 2) -- -2 для удаления ']'
  end
  -- создать строку с '\n'+1 знаком равенства
  local eq = string.rep("=", n + 1)
  -- построить итоговую строку
  return string.format(" [%s[\n%s]%s] ", eq, s, eq)
end
```

Функция `quote` из листинга 12.1 является результатом наших предыдущих замечаний. Она получает на вход произвольную строку и возвращает отформатированную строку как длинную строку. Вызов `string.gmatch` создает итератор для перебора всех последовательностей вида ``]=*]'` (то есть закрывающей квадратной скобки, за которой следуют ноль или больше знаков равенства, за которыми следует еще одна закрывающая квадратная скобка) в строке³. Для каждого вхождения обновляется значение `n`, равное максимальному числу уже встреченных знаков равенства. После цикла мы используем функцию `string.rep`, для того чтобы повторить знак равенства `n+1` раз, то есть на один больше, чем максимальное количество, встреченное в строке. Наконец, функция `string.format` заключает `s` между парами квадратных скобок с надлежащим числом знаков равенства и добавляет дополнительные пробелы вокруг строки и символ ``\\n'` в начале 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. Нет одного алгоритма, который бы подходил для всех случаев. Простые таблицы не только требуют более простых алгоритмов, но и получающиеся при этом файлы могут быть визуально приятней.

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
```

³ Мы обсудим шаблоны строк в главе 21.

Наша следующая попытка представлена в листинге 12.2. Несмотря на свою простоту, эта функция выполняет вполне приличную работу. Она даже обрабатывает вложенные таблицы (то есть таблицы внутри других таблиц) до тех пор, пока структура таблицы является деревом (то есть нет общих подтаблиц и циклов). Небольшим визуальным улучшением будет добавление пробелов для инdentации вложенных таблиц (см. упражнение 12.1).

Предыдущая функция предполагает, что все ключи в таблице являются валидными идентификаторами. Если в таблице есть числовые ключи или строки, которые не являются 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 улучшением мы увеличили надежность нашей функции за счет визуальной наглядности получающегося файла. Рассмотрим следующий вызов:

```
serialize{a=12, b='Lua', key='another "one"'}
```

Результатом этого вызова при использовании первой версии функции `serialize` будет следующий код:

```
{
  a = 12,
  b = "Lua",
  key = "another \"one\"",
}
```

Сравните с результатом использования второй версии:

```
{
  ["a"] = 12,
  ["b"] = "Lua",
  ["key"] = "another \"one\"",
}
```

Мы можем получить и надежность, и красивый вид, проверяя в каждом случае, нужны ли квадратные скобки; опять мы оставим это улучшение в качестве упражнения.

Сохранение таблиц с циклами

Для обработки таблиц в общем случае (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 для сохранения и имя. Более того, мы должны отслеживать имена уже сохраненных таблиц, для того чтобы переиспользовать их, когда мы обнаруживаем цикл. Для этого мы будем использовать дополнительную таблицу. Эта таблица будет использовать таблицы в качестве индексов и их имена в качестве хранимых 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") -- используем его имя
    else
      saved[value] = name -- сохранить имя для следующего раза
      io.write("{}\n") -- создать новую таблицу
      for k,v in pairs(value) do -- сохранить ее поля
        k = basicSerialize(k)
        local fname = string.format("%s[%s]", name, k)
        save(fname, v, saved)
      end
    end
  end
  error("cannot save a " .. type(value))
end
```

Итоговый код показан в листинге 12.3. Мы пока придерживаемся ограничения, что таблицы, которые мы хотим сохранять, содержат лишь числа и строки в качестве ключей. Функция `basicSerialize` сериализует эти базовые типы. Следующая функция, `save`, выполняет всю тяжелую работу. Параметр `saved` – это таблица, которая отслеживает уже сохраненные таблицы. Например, если мы построим таблицу следующим образом:

```

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 гарантирует, что любой элемент, необходимый для построения таблицы уже определен.

Если мы хотим сохранить несколько значений с общими частями, то мы можем вызвать функцию `save` в той же самой таблице `saved`. Например, рассмотрим следующие две таблицы:

```

a = {{ "one", "two"}, 3}
b = {k = a[1]}

```

Если мы сохраним их независимо, 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 в получившемся результате 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 обычно, в Lua существует несколько других вариантов. Среди них мы можем сохранить значение без выдачи ему глобального имени (например, блок строит локальное значение и возвращает его), мы можем обрабатывать функции (путем построения дополнительной таблицы, связывающей каждую функцию с ее именем) и т. д. Lua дает вам силу; вы строите механизмы.

Упражнения

Упражнение 12.1. Измените код из листинга 12.2, чтобы он выравнивал вложенные таблицы.

(Подсказка: добавьте дополнительный параметр функции `serialize`, содержащий строку выравнивания.)

Упражнение 12.2. Измените код из листинга 12.2 так, 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 когда это необходимо.

Упражнение 12.4. Измените код предыдущего упражнения так, чтобы он использовал конструкторы всегда, когда это возможно. Например, он должен представить 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, ..., пока они не равны *nil*. Обратите внимание на то, что не нужно их снова сохранять при обходе остальной части таблицы.)

Упражнение 12.5. Подход, заключающийся в отказе от использования конструкторов, при сохранении таблиц с циклами, слишком радикальный. Можно сохранить таблицу в более приятном виде, используя конструкторы в общем случае и затем используя присваивания только для обработки общих таблиц и циклов.

Заново реализуйте функцию `save` с использованием этого подхода. Добавьте к ней все, что вы уже реализовали в предыдущих упражнениях.



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. п. Однако мы не можем складывать таблицы, не можем сравнивать функции и не можем вызвать строку. Если только мы не используем метатаблицы.

Метатаблицы позволяют изменить поведение значения в случае, когда мы сталкиваемся с неожиданной операцией. Например, при помощи метатаблиц мы можем определить, как Lua должен вычислить выражение $a+b$, где a и b – это таблицы. Когда Lua пытается сложить две таблицы, то он проверяет, есть ли хотя бы в одной из них метатаблица и содержит ли эта метатаблица поле `__add`. Если Lua находит это поле, то он вызывает соответствующее значение – так называемый *метаметод*, который должен быть функцией, – для вычисления суммы.

Каждое значение в Lua может иметь связанную с ним метатаблицу. Таблицы и значения типа `userdata` хранят индивидуальные значения для каждого экземпляра; значения остальных типов используют одну общую таблицу на каждый тип. Lua всегда создает новые таблицы без метатаблиц:

```
t = {}  
print(getmetatable(t))      --> nil
```

Мы можем использовать функцию `setmetatable`, для того чтобы задать или изменить метатаблицу для любой таблицы:

```
t1 = {}  
setmetatable(t, t1)  
print(getmetatable(t) == t1)  --> true
```

Непосредственно из Lua мы можем устанавливать метатаблицы только для таблиц; для работы с метатаблицами значений других ти-

пов мы должны использовать код на С¹. Мы позже увидим в главе 21, что библиотека для работы со строками устанавливает метатаблицы для строк. Все остальные типы по умолчанию не имеют метатаблиц:

```
print(getmetatable("hi"))      --> table: 0x80772e0
print(getmetatable("xuxu"))    --> table: 0x80772e0
print(getmetatable(10))        --> nil
print(getmetatable(print))     --> nil
```

Любая таблица может быть метатаблицей любого значения; группа связанных между собой таблиц может разделять общую метатаблицу, которая задает их общее поведение; таблица может быть метатаблицей сама для себя так, что она описывает свое собственное поведение.

13.1. Арифметические метаметоды

В этом разделе мы рассмотрим простой пример для того, чтобы объяснить, как использовать метатаблицы. Пусть мы используем таблицы для представления множеств с функциями для вычисления объединения, пересечения и т. д., как показано в листинге 13.1. Для того чтобы не засорять глобальное пространство имен, мы будем хранить эти функции в таблице `Set`.

Листинг 13.1. Простая реализация множеств

```
Set = {}
-- создать новое множество, взяв значения 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
```

¹ Главной причиной для такого ограничения служит желание ограничить слишком частое использование метатаблиц. Опыт с предыдущими версиями Lua показал, что подобные глобальные изменения часто ведут к неперейсуществующему коду.

```

        res[k] = b[k]
    end
    return res
end
-- представить множество как строку
function Set.toststring (set)
    local l = {} -- список, куда будут помещены все элементы
    for e in pairs(set) do
        l[#l + 1] = e
    end
    return "{" .. table.concat(l, ", ") .. "}"
end
-- напечатать множество
function Set.print (s)
    print(Set.toststring(s))
end

```

Теперь мы будем использовать оператор сложения ('+') для вычисления объединения двух множеств. Для этого мы сделаем так, что все таблицы, представляющие множества, будут иметь одну общую метатаблицу. Эта метатаблица определит, как таблицы должны реагировать на оператор 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, создающей множества `Set.new`. В новой версии этой функции будет одна дополнительная строка, которая для создаваемых таблиц устанавливает `mt` как метатаблицу:

```

function Set.new (l) -- 2-ая 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 метатаблицу:

```

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, поле `__add`, которое определяет, как нужно выполнять сложение:

```
mt.__add = Set.union
```

После этого всегда, когда Lua будет пытаться сложить два множества, она будет вызывать функцию `Set.union`, передавая оба операнда в качестве аргументов.

С метаметодом мы можем использовать оператор сложения для выполнения объединения множеств:

```
s3 = s1 + s2
Set.print(s3)           --> {1, 10, 20, 30, 50}
```

Аналогично мы можем определить оператор умножения для выполнения пересечения множеств:

```
mt.__mul = Set.intersection

Set.print((s1 + s2)*s1)  --> {10, 20, 30, 50}
```

Для каждого арифметического оператора существует соответствующее имя поля в метатаблице. Кроме `__add` и `__mul`, также 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 задания оператора конкатенации.

Когда мы складываем два множества, то вопрос о том, какую метатаблицу взять, не возникает. Однако мы можем записать выражение, в котором участвуют два значения с разными метатаблицами, например как показано 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 соответствующее значение в качестве метаметода независимо от второго значения; иначе если второе значение имеет метатаблицу с полем `__add`, то Lua использует это значение в качестве метаметода; в противном случае возникает ошибка. Таким образом, в последнем примере будет вызван `Set.union`, так же как и для выражений `10+s` и `"hello"+s`.

Lua не беспокоится по поводу смешивания типов, однако это важно для нашего приложения. Например, если мы выполним `s=s+8`, то мы получим ошибку внутри `Set.union`:

```
bad argument #1 to 'pairs' (table expected, got number)
```


Если мы хотим получать более точные сообщения об ошибках, то мы должны явно проверять типы операндов перед выполнением 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 ранее>

Помните, что второй аргумент функции `error` (2 в нашем случае) направляет сообщение об ошибке туда, где данная операция была вызвана.

13.2. Метаметоды сравнения

Метаблицы также позволяют придать смысл операторам сравнения при помощи метаметодов `__eq` (*равно*), `__lt` (*меньше, чем*) и `__le` (*меньше или равно, чем*). Нет специальных метаметодов для трех оставшихся операций сравнения: Lua переводит $a \sim b$ в $\text{not } (a == b)$, $a > b$ в $b < a$ и $a \geq b$ в $b \leq a$.

До версии 4.0 Lua переводил все операции упорядочивания в одну, переводя $a \leq b$ в $\text{not } (b < a)$. Однако такой перевод некорректен, когда мы имеем дело с *частичным упорядочиванием*, 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*). В соответствии со стандартом IEEE 754 NaN представляет неопределенные значения, например $0/0$. Согласно стандарту, любое сравнение, включающее в себя NaN, должно быть ложным. Это значит, что $\text{NaN} \leq x$ всегда ложно, но и $x < \text{NaN}$ также ложно. Из этого следует, что перевод $a \leq b$ в $\text{not } (b < a)$ неверен в этом случае.

В нашем примере с множествами мы имеем дело с похожей проблемой. Очевидным (и полезным) значением для `<=` для множеств является входжение множества: $a \leq b$ означает, что a — это подмножество b . С этим значением опять возможно, что $a \leq b$ и $b < a$ ложны; таким образом, нам нужны отдельные реализации для `__le` (*меньше или равно*) и `__lt` (*меньше, чем*):

```
mt.__le = function (a, b) -- входжение 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 вложение множеств:

```
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, у которых есть полное упорядочение, мы можем не определять метаметод `__le`. При его отсутствии Lua использует `__lt`.

Сравнение на равенство также обладает некоторыми ограничениями. Если у двух объектов разные базовые типы или метаметоды, то операция сравнения на равенство вернет *false*, даже не вызывая метаметодов. Таким образом множество всегда будет отличаться от числа, независимо от того, что возвращает метаметод.

13.3. Библиотечные метаметоды

До сих пор мы видели метаметоды, определенные в самой Lua. Виртуальная машина сама проверяет, содержат ли значения, соединенные операцией, метатаблицы с соответствующими метаметодами. Однако поскольку метатаблицы являются обычными таблицами, то их может использовать любой. Поэтому часто библиотеки определяют 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 вывода. Однако при форматировании произвольного значе-

ния `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 как строки. Поэтому нам нужно только выставить поле `__tostring` в 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 метатаблицы. Предположим, что вы хотите защитить ваши множества так, что пользователи не смогут ни увидеть, ни изменить их метатаблицы. Если задать поле `__metatable` в метатаблице, то `getmetatable` вернет значение этого поля, а вызов `setmetatable` приведет к возникновению ошибки:

```
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 добавить обход для объектов, не являющихся таблицами).

13.4. Метаметоды для доступа к таблице

Метаметоды для арифметических операций и операций сравнения определяют поведение для ситуаций, которые иначе приводили бы к возникновению ошибок. Они не изменяют обычного поведения языка. Но Lua также предоставляет способ для того, чтобы изменить поведение таблиц в двух обычных случаях, чтения и изменения несуществующего поля в таблице.

Метаметод `__index`

Я ранее уже сказал, что когда мы обращаемся к отсутствующему полю в таблице, то результатом является *nil*. Это так, но это не вся правда. На самом деле подобное обращение приводит к тому, что интерпретатор ищет метаметод `__index`: если такого метода нет, что обычно и бывает, то возвращается *nil*; иначе результат предоставляет данный метаметод.

Стандартным примером здесь является наследование. Пусть мы хотим создать несколько таблиц, описывающих окна. Каждая таблица должна задать 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 окна, задавая только те значения, которые отличаются от значений по умолчанию. Первым вариантом является конструктор, заполняющий отсутствующие поля. Вторым вариантом является организовать окна таким образом, чтобы они *наследовали* любое отсутствующее поле от базового прототипа. Для начала мы объявим прототип и конструктор, который создает новые окна, обладающие общей метатаблицей:

```
-- создать прототип со значениями по умолчанию
prototype = {x = 0, y = 0, width = 100, height = 100}
mt = {}      -- создать метатаблицу
-- объявить 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 мы создадим новое окно и обратимся к отсутствующему полю:

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

Луа определяет, что у `w` нет требуемого поля, но есть метатаблица с полем `__index`. Поэтому Луа вызывает этот метаметод с аргументами `w` (таблица) и `"width"` (отсутствующее поле). Метаметод обращается с этим полем к прототипу и возвращает полученное значение.

Использование метаметода `__index` для наследования в Lua так распространено, что Lua предоставляет упрощенный вариант. Несмотря на название *метод*, метаметод `__index` не обязан быть функцией: например, он может быть таблицей. Когда он является функцией, то Lua вызывает его, передавая таблицу и отсутствующий ключ как аргументы, как мы уже видели. Когда это таблица, то Lua просто выполняет обращение к этой таблице. Поэтому в нашем предыдущем примере мы могли просто определить `__index` следующим образом:

```
mt.__index = prototype
```

Теперь, когда Lua будет искать метаметод `__index`, то он найдет значение `prototype`, которое является таблицей. Соответственно, Lua выполняет обращение к этой таблице, то есть осуществляет аналог `prototype["width"]`. Это обращение и дает требуемый результат.

Использование таблицы в качестве метаметода `__index` дает простой и быстрый способ реализации обычного (не множественного) наследования. Функция является более дорогостоящим вариантом, но и предоставляет при этом больше гибкости: мы можем реализовать множественное наследование, кэширование и многое другое. Мы обсудим эти формы наследования в главе 16.

Когда мы хотим обратиться к таблице без вызова метаметода `__index`, то мы используем функцию `rawget`. Вызов `rawget(t, i)` осуществляет непосредственное обращение к таблице `t`, то есть обращение без использования метатаблиц. Выполнение непосредственного обращения не ускорит ваш код (цена вызова функции уничтожит все, что можно выиграть), но иногда он оказывается необходимым, как мы увидим позже.

Метаметод `__newindex`

Метаметод `__newindex` является аналогом метаметода `__index`, но только он работает для записи значений в таблицу. Когда вы присваиваете значение отсутствующему полю в таблице, то 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, выполняющая непосредственный доступ, минуя метаметоды: `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 различные довольно мощные конструкции, такие как таблицы, доступные только для чтения, таблицы со значениями по умолчанию и наследование для объектно-ориентированного программирования. В этой главе мы увидим некоторые из таких применений. Объектно-ориентированному программированию отведена отдельная глава.

Таблицы со значениями по умолчанию

Значение по умолчанию для любого поля в обычной таблице – это *nil*. Легко изменить это поведение 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 вызов к отсутствующему полю в `tab` вызовет его метаметод `__index`, который вернет ноль (значение `d` для этого метаметода).

Функция `setDefault` создает новое замыкание и новую метатаблицу для каждой таблицы, которой нужно значение по умолчанию. Это может оказаться дорогостоящим, если у нас много таблиц, которым нужны значения по умолчанию. У метатаблицы значение по умолчанию `d` «зашиито» в ее метаметод, поэтому мы не можем использовать одну и ту же метатаблицу для всех таблиц. Для того 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 не думать о возможных конфликтах по именам, то мы можем использовать ключ вроде `"__"` для нашего поля:

```
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` только один раз, вне функции `setDefault`.

Если мы хотим гарантировать уникальность ключа, то это довольно легко обеспечить. Все, что нам нужно, – это создать новую таблицу и использовать ее в качестве ключа:

```
local key = {} -- unique key
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 ключами являются сами таблицы, а значениями – значения по умолчанию. Однако для корректной реализации такого подхода нам нужен специальный тип таблиц, называемых *слабыми таблицами* (weak table), поэтому мы здесь не будем использовать данный подход; мы вернемся к этому в главе 17.

Другим вариантом является запоминать метатаблицы, за счет чего мы можем переиспользовать метатаблицы, соответствующие одному и тому же значению по умолчанию. Однако это также требует использования слабых таблиц, поэтому нам придется подождать до главы 17.

Отслеживание доступа к таблице

И `__index`, и `__newindex` работают только в случае, когда в таблице нет соответствующего значения. Поэтому единственный способ отслеживать весь доступ к таблице – это держать ее пустой. Таким образом, если мы хотим отслеживать весь доступ к таблице, то нам нужно создать специальную *proxy*-таблицу для исходной таблицы. Она будет пустой 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 использовать следующий код:

```
t = {} -- исходная таблица создана где-то

-- создадим закрытый доступ к ней
local _t = t
-- создадим проху
t = {}
-- создадим 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 -- изменение исходной таблицы
end
}
setmetatable(t, mt)
```

Этот код отслеживает каждый доступ к t:

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

Если мы хотим иметь возможность обходить такую таблицу, то нам нужно создать в проху-таблице метаметод `__pairs`:

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

Также можно создать что-то похожее для `__ipairs`.

Если мы хотим отслеживать доступ к нескольким таблицам, то нам не нужно для каждой из них создавать отдельную метатаблицу. Вместо этого мы можем как-нибудь связать проху-таблицу с исходной и использовать одну общую метатаблицу для всех проху-таблиц. Это похоже на задачу связывания таблицы со значением по умолчанию, которые мы рассматривали ранее. Например, можно хранить исходную таблицу в специальном поле проху-таблицы, используя для этого специальный ключ. В результате мы приходим к следующему коду:

```
local index = {} -- создать уникальный ключ
local mt = { -- создать метатаблицу
  __index = function (t, k)
    print("*access to element " .. tostring(k))
    return t[index][k] -- обращение к исходной таблице
  end,
  __newindex = function (t, k, v)
    print("*update of element " .. tostring(k) ..
      " to " .. tostring(v))
    t[index][k] = v -- изменение исходной таблицы
  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, когда мы хотим отслеживать таблицу `t`, все что нам нужно, — это выполнить `t=track(t)`.

Таблицы, доступные только для чтения

Легко использовать понятие проху-таблиц для создания таблиц с доступом только на чтение. Все, что нам нужно, — это вызвать ошибку каждый раз, когда мы ловим попытку изменить таблицу. Для метаметода `__index` мы можем использовать саму исходную таблицу вместо функции, так как нам не нужно отслеживать все чтения из нее; быстрее и эффективнее перенаправлять такие запросы сразу к исходной таблице. Это потребует, однако, новой метатаблицы для каждой проху-таблицы с полем `__index`, указывающим на исходную таблицу:

```

function readOnly (t)
    local proxy = {}
    local mt = { -- создать 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 таблицы, доступной только на чтение, 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 $a-b$ — это множество всех элементов из a , которые не содержатся в b .)

Упражнение 13.2. Определите метаметод `__len` так, что `#s` возвращает число элементов в s .

Упражнение 13.3. Дополните реализацию `iproxy`-таблиц 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 доступ к таблице более дорогим, но создание таких таблиц более дешевым, так как все таблицы, доступные только для чтения, могут иметь одну общую метатаблицу. Перепишите функцию `readOnly` с использованием данного подхода.



ГЛАВА 14

Окружение

Луа хранит все свои глобальные переменные в обычной таблице, называемой *глобальным окружением* (global environment). (Точнее, Луа хранит свои «глобальные» переменные в нескольких окружениях, но мы для простоты будем это вначале игнорировать.) Одним из преимуществ этого подхода является то, что он упрощает внутреннюю реализацию Луа, поскольку нет необходимости в специальной структуре данных для хранения глобальных переменных. Другим преимуществом является то, что мы можем работать с этой таблицей так же, как и с любой другой таблицей. Для упрощения такой работы Луа 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 присваивания достаточно для доступа к и установки значения глобальной переменной. Однако часто нам бывает нужен вариант метапрограммирования, когда мы хотим работать с глобальной переменной, имя которой содержится в другой переменной или вычисляется в ходе работы. Чтобы получить значение такой переменной, многие программисты пытаются использовать что-то вроде следующего фрагмента кода:

```
value = loadstring("return " .. varname)()
```

Если `varname` равно `x`, то в результате конкатенации мы получим `"return x"`, что при выполнении даст нам желаемый результат. Од-

нако этот код включает в себя создание и компиляцию нового блока кода, что является дорогостоящим. Вы можете добиться того же самого при помощи следующего кода, который более чем на порядок более эффективен, чем ранее рассмотренный код:

```
value = _G[varname]
```

Поскольку окружение – это обычная таблица, то вы можете просто обращаться к нему по ключу (имени переменной). Похожим образом можно также присвоить значение переменной, имя которой вычисляется динамически, при помощи кода `_G[varname]=value`. Однако будьте осторожны: некоторые программисты так радуются подобной возможности, что заканчивают написанием кода вроде `_G["a"]=_G["var1"]`, что является просто сложным вариантом `a=var1`.

Обобщением предыдущей задачи является использование имен полей в динамических именах, например `"io.read"` или `"a.b.c.d"`. Однако если мы напишем `_G["io.read"]`, то мы точно не получим поле `read` из таблицы `io`. Но мы можем написать функцию `getfield`, такую что `getfield("io.read")` вернет ожидаемое значение. Эта функция представляет из себя цикл, который начинается с `_G` и дальше последовательно перебирает поля:

```
function getfield (f)
  local v = _G -- начать с таблицы глобальных переменных
  for w in string.gmatch(f, "[%w_]+") do
    v = v[w]
  end
  return v
end
```

Мы используем функцию `gmatch` из библиотеки `string` для того, чтобы обойти все слова в `f` (слово – это последовательность букв, цифр и знака подчеркивания).

Соответствующая функция для установки значений полей является более сложной. Присваивание вроде `a.b.c.d=v` эквивалентно следующему коду:

```
local temp = a.b.c
temp.d = v
```

То есть мы должны извлечь имя без последней компоненты и затем отдельно обработать последнюю компоненту. Функция `setfield` выполняет это и также создает вспомогательные таблицы в пути, если они не существуют:

```
function setfield (f, v)
  local t = _G -- начинаем с таблицы глобальных переменных
```

```

for w, d in string.gmatch(f, "[%w_]+)(%.?)") do
    if d == "." then      -- не последнее имя?
        t[w] = t[w] or {} -- создает таблицу, если ее нет
        t = t[w]          -- получаем таблицу
    else                  -- последнее имя
        t[w] = v          -- выполняем присваивание
    end
end
end
end

```

В переменной `w` запоминается имя поля, и, возможно, следующая за ним точка запоминается в переменной `d`¹. Если за именем не следует точка, то это последнее имя.

Используя ранее рассмотренные функции, следующий код создает глобальную таблицу `t`, таблицу `t.x` и затем присваивает `10 t.x.y`:

```

setfield("t.x.y", 10)
print(t.x.y) --> 10
print(getfield("t.x.y")) --> 10

```

14.2. Описания глобальных переменных

В Lua глобальным переменным не нужны описания. Хотя это и удобно для небольших программ, в больших программах всего одна опечатка может привести к трудно обнаруживаемым ошибкам. Однако при желании мы можем изменить это поведение. Поскольку Lua хранит глобальные переменные в обычной таблице, то мы можем использовать метатаблицы для изменения поведения при обращении к глобальным переменным.

Первый подход просто отслеживает любые обращения 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:

¹ Мы рассмотрим использование шаблонов в главе 21.

```
> print(a)
stdin:1: attempt to read undeclared variable a
```

Однако как мы будем объявлять глобальные переменные? Одним вариантом является использование `rawset`, который не использует метаметоды:

```
function declare (name, initval)
  rawset(_G, name, initval or false)
end
```

(Конструкция ***or false*** нужна затем, чтобы глобальная переменная получила значение, отличное от ***nil***.)

Более простым вариантом является ограничить присваивания новым глобальным переменным только внутри функций, позволяя присваивания на внешнем уровне блока.

Для проверки того, что присваивание происходит в главном блоке, нам нужно использовать отладочную библиотеку. Вызов `debug.getinfo(2, "S")` возвращает таблицу, у которой поле `what` говорит о том, является ли функция, вызвавшая метаметод, главным блоком, обычной функцией или С-функцией. (Мы обсудим `debug.getinfo` более подробно в главе 24.) Используя эту функцию, 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 в С-коде, так как обычно в этом коде авторы знают, что они делают.

Для проверки того, что такая переменная существует, мы не можем просто сравнить ее с ***nil***, поскольку если она ***nil***, то обращение приведет к ошибке. Вместо этого мы используем функцию `rawget`, которая не использует метаметод:

```
if rawget(_G, var) == nil then
  -- 'var' is undeclared
  ...
end
```

Сейчас наш подход не допускает глобальных переменных со значением ***nil***, поскольку они автоматически будут считаться необъявленными. Но это легко исправить. Все, что нам нужно, – это дополнитель-

ная таблица, содержащая имена описанных переменных. При вызове метаметода он по этой таблице проверяет, описана ли эта переменная. Похожий код приведен в листинге 14.1. Теперь даже присваивания `x=nil` достаточно, чтобы объявить глобальную переменную.

Цена обоих решений крайне незначительна. При первом решении, при нормальной работе метаметод вообще не вызывается. При втором решении метаметоды могут быть вызваны, когда программа обращается к переменной, значение которой равно ***nil***.

Стандартная поставка Lua содержит модуль `strict.lua`, который реализует проверку обращений к глобальным переменным, аналогичную рассмотренному нами коду. Хорошей привычкой является использовать его при написании кода на Lua.

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) -- do the actual set
  end,
  __index = function (_, n)
    if not declaredNames[n] then
      error("attempt to read undeclared variable "..n, 2)
    else
      return nil
    end
  end,
end,
})
```

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 следовать соответствующей политике. Если вы хотите использовать библиотеку, которая использует глобальные переменные без их объявления, то вам не повезло.

В Lua глобальные переменные не обязаны быть действительно глобальными. Мы можем даже сказать, что в Lua нет глобальных переменных. Это может звучать странно, поскольку с самого начала книги мы использовали глобальные переменные. Очевидно, что Lua очень старается создать иллюзию наличия глобальных переменных. Давайте посмотрим, как Lua создает эту иллюзию².

Начнем с понятия свободных имен. *Свободное имя* – это имя не привязано к явному описанию, то есть не встречается внутри области действия локальной переменной (или переменной цикла *for*, или параметра) с этим именем. Например, оба имени `var1` и `var2` – это свободные имена в следующем блоке:

```
var1 = var2 + 3
```

В отличие от того, что было сказано ранее, свободное имя не относится к глобальной переменной (по крайней мере, не непосредственно). Вместо этого Lua переводит любое свободное имя `var` в `_ENV.var`. Поэтому предыдущий блок 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. Компилятор снова жульничает. Я уже говорил, что Lua рассматривает каждый блок как анонимную функцию. На самом деле Lua компилирует наш исходный блок в следующий код:

```
local _ENV = <some value>
return function (...)
  _ENV.var1 = _ENV.var2 + 3
end
```

То есть Lua компилирует любой блок кода в присутствии предопределенного значения с именем `_ENV`.

Обычно когда мы загружаем блок кода, то функция `load` инициализирует это предопределенное значение ссылкой на глобальное окружение. Поэтому наш исходный блок становится эквивалентным следующему блоку:

```
local _ENV = <the global environment>
return function (...)
  _ENV.var1 = _ENV.var2 + 3
end
```

² Обратите внимание, что этот механизм был одной из тех частей Lua, которые появились с версии 5.1 до версии 5.2. Следующее обсуждение относится только к Lua 5.2 и очень мало применимо к предыдущим версиям.

Результатом всех этих присваиваний является то, что поле `var1` из глобального окружения получает значение поля `var2` плюс 3.

На первый взгляд это может показаться несколько запутанным способом работать с глобальным окружением. Я не буду утверждать, что это простейший способ, но он достигает гибкости, которую трудно получить более простой реализацией.

Прежде чем мы продолжим, давайте сформулируем, как Lua 5.2 работает с глобальными переменными:

- Lua компилирует любой блок с использованием значения `_ENV`.
- Компилятор переводит любое свободное имя `var` в `_ENV.var`.
- Функция `load` (или `loadfile`) инициализирует значение `_ENV` ссылкой на глобальное окружение.

В конце концов, все не так уж и сложно.

Некоторых это смущает, поскольку они пытаются найти какую-то магию, стоящую за этими правилами. Нет тут никакой магии. В частности, первые два правила полностью делаются компилятором. За исключением того, что величина `_ENV` известна компилятору, она является обычной переменной. За исключением компиляции, `_ENV` не имеет какого-то специального смысла в Lua³. Аналогично перевод из `var` в `_ENV.var` — это просто синтаксическая замена без скрытого смысла. В частности, после этого перевода `_ENV` будет относиться к той переменной `_ENV`, которая видна в данном фрагменте кода, исходя из правил видимости.

14.4. Использование `_ENV`

В этом разделе мы рассмотрим некоторые пути использования той гибкости, которая привносится переменной `_ENV`. Имейте в виду, что каждый из этих примеров должен быть запущен как отдельный, самостоятельный блок кода. Если вы будете вводить строка за строкой в интерпретаторе, то каждая строка становится отдельным блоком и получает свою переменную `_ENV`. Для выполнения фрагмента кода как отдельного блока вам нужно либо запустить его как файл, либо в интерактивном режиме поместить внутрь пары ***do-end***.

Поскольку `_ENV` — это обычная переменная, то мы можем присваивать ей и читать ее так же, как и любую другую переменную. Присваи-

³ Если быть честными до конца, то Lua использует это имя для сообщений об ошибках, поскольку она возвращает ошибку, включающую переменную `_ENV.x` как ошибку с переменной `global x`.

вание `_ENV=nil` запретит любой доступ к глобальным переменным на протяжении оставшейся части блока. Это может быть полезным для контроля того, какие переменные ваш код использует:

```
local print, sin = print, math.sin
_ENV = nil
print(13)           --> 13
print(sin(13))      --> 0.42016703682664
print(math.cos(13)) -- error!
```

Любое присваивание свободному имени приведет к аналогичной ошибке.

Мы можем явно обращаться к `_ENV` для того, чтобы обойти локальные переменные:

```
a = 13           -- global
local a = 12
print(a)         --> 12 (local)
print(_ENV.a)    --> 13 (global)
```

Конечно, главным использованием `_ENV` является изменение окружения, используемого фрагментом кода. Как только вы изменили ваше окружение, все обращения к глобальным переменным будут использовать новую таблицу:

```
-- изменить текущее окружение на пустую таблицу
_ENV = {}
a = 1    -- создать 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 глобальным переменным, включая `print`. Поэтому вам сперва нужно заполнить его некоторыми полезными значениями, например старым окружением:

```
a = 15           -- создаем глобальную переменную
_ENV = {g = _G}  -- изменяем текущее окружение
a = 1           -- создаем 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 есть функция `print`.

Мы можем переписать предыдущий пример, используя имя `_G` вместо `g`:

```
a = 15           -- создаем глобальную переменную
_ENV = {_G = _G} -- изменяем текущее окружение
```

```

a = 1                -- создаем поле в _ENV
_G.print(a)          --> 1
_G.print(_G.a)       --> 15

```

Для Lua имя `_G` – это такое же имя, как и все остальные. Его отличительной чертой является только то, что когда Lua создает глобальную таблицу, то она присваивает ее переменной с именем `_G`. Для Lua не важно текущее значение этой переменной. Но обычно принято использовать одно и то же имя, когда мы обращаемся к глобальной переменной, как мы это делали в переписанном примере.

Другой способ заполнить ваше новое окружение – это наследование:

```

a = 1
local newgt = {}    -- создать новое окружение
setmetatable(newgt, {__index = _G})
_ENV = newgt        -- установить его
print(a)            --> 1

```

В этом коде новое окружение наследует `print` и `a` из старого окружения. Однако любое присваивание идет в новую таблицу. Тем самым нет опасности по ошибке изменить глобальное окружение, хотя его все равно можно изменить 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 блока, обращаются к `_ENV` так же, как и к любой другой внешней переменной:

```

_ENV = {_G = _G}
local function foo ()
  _G.print(a)      -- компилируется в `'_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 доступ к свободным именам будет идти через нее:

```
a = 2
do
  local _ENV = {print = print, a = 14}
  print(a)      --> 14
end
print(a)        --> 2 (назад к исходной _ENV)
```

Поэтому несложно построить функцию с собственным (закрытым) окружением:

```
function factory (_ENV)
  return function ()
    return a      -- "глобальная" а
  end
end
f1 = factory{a = 6}
f2 = factory{a = 7}
print(f1())      --> 6
print(f2())      --> 7
```

Функция `factory` создает простые замыкания, которые возвращают значение и локальных переменных `a`. Когда замыкание создано, то видимая переменная `_ENV` — это параметр `_ENV` из содержащей функции `factory`; поэтому замыкание использует эту переменную для доступа к 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 уже упоминал, `load` обычно инициализирует значение `_ENV` загруженного блока указателем на глобальное окружение. Однако у `load` есть необязательный четвертый параметр, который задает значение для `_ENV`. (Функция `loadfile` также имеет аналогичный параметр.)

В качестве примера пусть 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 с пустым окружением `env`. Более важно, что все его определения пойдут именно в это окружение. Конфигурационный файл не может повлиять на что-либо еще, даже по ошибке. Даже зловредный код не может причинить много вреда. Он может выполнить DoS-атаку, тратя время CPU и память, но ничего больше.

Иногда вам может понадобиться выполнить блок несколько раз, каждый раз с другой таблицей окружения. В этом случае дополнительный аргумент у `load` нам не помогает. Вместо этого у нас есть два варианта.

Первый вариант — это использовать функцию `debug.setupvalue` из отладочной библиотеки. Как следует из имени, `setupvalue` позволяет нам изменить любое входящее значение (`upvalue`) заданной функции. Следующий код иллюстрирует его использование:

```
f = loadfile(filename)  
...  
env = {}  
debug.setupvalue(f, 1, env)
```

Первый аргумент при вызове `setupvalue` — это функция, второй — это индекс значения, и третий — это новое значение. Для нашего использования второй аргумент всегда равен единице: когда функция является результатом `load` или `loadfile`, Lua гарантирует, что будет всего одно значение и это значение есть `_ENV`.

Небольшим минусом данного решения является зависимость от отладочной библиотеки. Эта библиотека нарушает некоторые стандартные предположения насчет программ. Например, `debug.setupvalue` нарушает правила видимости Lua, которые гарантируют, что переменная не может быть увидена за пределами своей области видимости.

Другим способом запускать блок с различными окружениями является небольшое изменение блока при его загрузке. Представьте себе, что мы добавляем следующую строку к началу загружаемого блока:

```
_ENV = ...;
```

Вспомним из раздела 8.1, что Lua любой блок рассматривает как функцию с переменным числом аргументов. Поэтому эта строка при-

своит переменной `_ENV` первый аргумент блока, устанавливая его как окружение. После загрузки блока мы вызываем получающуюся функцию, передавая желаемое окружение как первый аргумент. Следующий фрагмент кода иллюстрирует эту идею, используя функцию `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, обеспечивает слишком мало контроля, так как она допускает такие поля, как `math?sin` или `string!!!gsub`. Перепишите ее так, чтобы она воспринимала в качестве разделителя только одну точку. (Для этого упражнения вам может понадобиться информация из главы 21.)

Упражнение 14.2. Объясните в деталях, что происходит в следующей программе и какой 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 какой 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 устанавливает каких-либо соглашений. Вместо этого Lua предоставляет механизмы, которые достаточно мощны для групп разработчиков для реализации тех соглашений, которые им подходят. Однако этот подход плохо работает для модулей. Одной из основных целей системы модулей является позволить различным людям совместно использовать код. Отсутствие общей политики мешает этому совместному использованию.

Начиная с версии 5.1, Lua определил набор соглашений для модулей и пакетов (пакет – это набор модулей). Эти соглашения не требуют каких-либо дополнительных возможностей от языка; программисты могут их реализовать, используя то, что мы уже в языке видели: таблицы, функции, метатаблицы и окружения. Программисты могут использовать другие соглашения. Однако другие соглашения могут привести к тому, что нельзя будет использовать чужие модули и свои модули не могут быть использованы в чужих программах.

С точки зрения пользователя, *модуль* – это некоторый код (на Lua или на C), который может быть загружен при помощи `require` и который создает и возвращает таблицу. Все, что модуль экспортирует, будь это 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 отдельный интерпретатор (доступный в виде командной строки) заранее загружает все стандартные библиотеки при помощи кода, эквивалентного следующему:

```
math = require "math"
string = require "string"
...
```

Эта загрузка позволяет нам использовать обычную запись `math.sin`.

Очевидным плюсом от использования таблиц для реализации модулей является то, что мы можем работать с модулями так же, как и с таблицами, и использовать для этого всю силу Lua. В большинстве языков модули не являются значениями первого класса (то есть они не могут быть запомнены в переменных, переданы как аргументы функциям и т. п.), поэтому этим языкам нужны специальные механизмы для каждой возможности, которую они хотят предложить для модулей. В Lua вы получаете эти возможности бесплатно.

Например, существует несколько способов вызвать функцию из модуля. Обычным способом является следующий:

```
local mod = require "mod"
mod.foo()
```

Пользователь может установить любое локальное имя для модуля:

```
local m = require "mod"
m.foo()
```

Также можно предоставить альтернативные имена для отдельных функций:

```
local m = require "mod"
local f = mod.foo
f()
```

Приятной стороной этих возможностей является то, что они не требуют специальной поддержки от языка. Они используют только то, что язык и так предоставляет.

Распространенной жалобой на `require` является то, что эта функция не позволяет передать аргумент загружаемому модулю. Например, математический модуль мог бы получить аргумент, позволяющий выбирать между использованием градусов или радиан:

```
-- плохой код
local math = require("math", "degree")
```

Проблемой является то, что одной из главных целей `require` является избегать загрузки уже загруженного модуля. Как только модуль загружен, он будет переиспользован любой частью программы, которая в нем нуждается. Поэтому при использовании параметров возникла бы проблема, если бы понадобился тот же самый модуль, но с другими параметрами:

```
-- плохой код
local math = require("math", "degree")
```



```
-- где-то в другом месте той же программы
local math = require("math", "radians")
```

В случае если вы действительно хотите, чтобы ваш модуль поддерживал параметры, лучше создать явную функцию для их задания:

```
local mod = require"mod"
mod.init(0, 0)
```

Если инициализирующая функция возвращает сам модуль, то мы можем писать код вроде следующего:

```
local mod = require"mod".init(0, 0)
```

Другой вариант – сделать так, чтобы модуль возвращал функцию для инициализации и уже эта функция возвращала бы таблицу модуля:

```
local mod = require"mod"(0, 0)
```

В любом случае помните, что модуль загружается всего один раз; модуль сам должен разрешать инициализации с конфликтами.

15.1. Функция `require`

Функция `require` пытается свести к минимуму предположения о том, что является модулем. Для `require` модуль – это просто какой-то код, который определяет некоторые значения (такой как 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"modname"`. Первым шагом `require` является проверка по таблице `package.loaded`, не загружен ли данный модуль уже. Если это так, то `require` возвращает соответствующее значение. Поэтому, как только модуль загружен, другие вызовы, требующие загрузки этого модуля, просто вернут то же значение без выполнения какого-либо кода.

Если модуль еще не загружен, то `require` ищет файл на Lua с именем модуля. Если он находит такой Lua файл, то он его загружает при помощи `loadfile`. Результатом этого является функция, которую мы называем *загрузчиком*. (Загрузчик – это функция, которая при вызове возвращает модуль.)

Если `require` не может найти файл на Lua с именем модуля, то она ищет библиотеку на C с именем модуля. Если она находит соответствующую библиотеку на C, то она загружает ее при помощи `package.loadlib` (которую мы обсудили в разделе 8.3) и ищет функцию с именем `luaopen_modname`¹. В этом случае загрузчик является результатом `loadlib`, то есть функцией `luaopen_modname`, выглядящей как функция на Lua.

Независимо от того, является ли модуль файлом на Lua или библиотекой на C, у `require` теперь есть для него загрузчик. Для окончательной загрузки модуля `require` вызывает загрузчик с двумя аргументами: именем модуля и именем файла с загрузчиком. (Большинство модулей просто игнорируют эти аргументы.) Если загрузчик возвращает какое-либо значение, то `require` возвращает это значение и запоминает его в таблице `package.loaded` для того, чтобы всегда возвращать именно это значение для этого модуля. Если загрузчик ничего не возвращает, то `require` ведет себя так же, как если бы модуль вернул *true*. Без этого уточнения, последующие вызовы `require` снова бы выполняли этот модуль.

Для того чтобы заставить `require` загрузить указанный модуль еще раз, мы просто стираем запись об этом модуле из таблицы `package.loaded`:

```
package.loaded.<modname> = nil
```

В следующий раз, когда понадобится этот модуль, `require` проделает всю необходимую работу еще раз.

Переименовывание модуля

Обычно в качестве имени модуля мы используем их изначальные имена, но иногда мы должны переименовать модуль, чтобы избежать конфликта по именам. Типичной ситуацией является загрузка разных версий одного и того же модуля, например для тестирования. Модули на Lua не имеют внутри себя зашитых имен, поэтому обычно достаточно просто переименовать соответствующий `.lua`-файл. Однако мы не можем отредактировать бинарную библиотеку для изменения имени ее функции `luaopen_*`. Для того чтобы поддерживать подобные переименования, есть маленькая хитрость внутри `require`: если имя модуля содержит минус, то `require` отрезает часть имени вплоть до знака минуса при создании имени функции `luaopen_*`. Например, если имя модуля – это `a-b`, то `require` ожидает, что соответст-

1 В разделе 27.3 мы обсудим как загружать библиотеки на C.

вующая функция будет называться `luaopen_b`, а не `luaopen_a-b` (что по-любому не будет 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"`, `require` найдет переименованный файл `v1-mod` и внутри этого файла найдет функцию с именем `luaopen_mod`.

Поиск по пути

При поиске файла на Lua `require` использует путь для поиска, который несколько отличается от обычных путей для поиска. Типичный путь – это список каталогов, где нужно искать заданный файл. Однако в ANSI C (абстрактной платформой, на которой выполняется Lua) нет понятия каталога. Поэтому путь, используемый `require`, – это список *шаблонов*, 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 соответствующим именем; если нет, то переходит на следующий шаблон. Шаблоны с пути разделены при помощи точки с запятой (символ, редко используемый в именах файлов в современных операционных системах). Например, если путь равен

```
?;?.lua;c:\windows\?;/usr/local/lua/?/?.lua,
```

то вызов `require("sql")` попытается открыть следующие файлы:

```
sql
sql.lua
c:\windows\sql
/usr/local/lua/sql/sql.lua
```

В качестве специальных символов функция `require` использует только точку с запятой (как разделитель компонент) и вопросительный знак; все остальное, включая разделители в пути и расширения файлов, определяется самим путем.

Путь, который `require` использует для поиска файлов на Lua, – это всегда текущее значение переменной `package.path`. При запуске Lua она инициализирует эту переменную значением следующей переменной окружения `LUA_PATH_5_2`. Если эта переменная окружения не установлена, то Lua пытается использовать переменную окружения с именем `LUA_PATH`. Если они обе не определены, то Lua использует

путь по умолчанию, задаваемый на этапе компилирования². При использовании переменных окружения Lua подставляет путь по умолчанию вместо любой подстроки ";;". Например, если `LUA_PATH_5_2` равна `"mydir/?.lua;;"`, то окончательный путь будет шаблоном `"mydir/?.lua"`, за которым следует путь по умолчанию.

Путь для поиска библиотек на C работает таким же образом, но значение пути 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 для UNIX-систем бывает

```
./?.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 шаблон будет похож на следующий:

```
.\?.dll;C:\Program Files\Lua502\dll\?.dll
```

Функция `package.searchpath` реализует все эти соглашения для поиска библиотек. Она получает имя модуля и путь и ищет файл, следуя описанным выше правилам. Она возвращает или имя первого найденного файла, или *nil* и сообщение об ошибке, описывающее все файлы, которые она попыталась 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 файлов

В действительности `require` несколько сложнее, чем мы описали. Поиск файла на Lua и поиск C-библиотеки – это просто два частных случая более общего понятия *искателя файла* (*searcher*). Искатель файла – это просто функция, которая получает имя модуля и возвращает загрузчик для этого блока или *nil*, если она не может найти ни одного.

Массив `package.searchers` содержит список искателей файлов, которые использует `require`. При поиске модуля `require` вызывает каждый искатель по очереди, передавая ему имя модуля, до тех пор,

² В Lua 5.2 параметр командной строки `-E` предотвращает использование переменных окружения и приводит к использованию пути, заданного при компиляции.

пока не найдет загрузчик для модуля. Если поиск завершится впустую, то `require` вызывает ошибку.

Использование списка для управления поиском модуля придает большую гибкость функции `require`. Например, если вы хотите хранить модули сжатыми в `zip`-файлы, то все, что вам для этого нужно, — это предоставить соответствующую искомую-функцию и добавить ее к списку. Однако чаще всего программам все же не нужно изменять значение `package.searchers`. В конфигурации по умолчанию искомый функций на Lua и искомый библиотек на C, которые мы описали выше, занимают вторую и третью позиции. Перед ними стоит искомый уже загруженных модулей (`preload searcher`).

Этот искомый позволяет ввести произвольную функцию для загрузки модуля. Он использует таблицу `package.preload` для сопоставления именам модулей загрузочных функций. При поиске этот искомый просто ищет заданное имя в таблице. Если он находит функцию, то он возвращает ее как загрузчик модуля. Иначе он возвращает *nil*. Этот искомый предоставляет способ обрабатывать некоторые нетипичные случаи. Например, библиотека на C, статически прилинкованная к Lua, может зарегистрировать свою функцию `luaopen_` так, что она будет вызвана, только когда (и если) пользователю понадобится этот модуль. Таким образом, программа не тратит времени на открытие модуля, если он не используется.

По умолчанию `package.searchers` включает в себя четвертую функцию, которая нужна для подмодулей. Мы рассмотрим их в разделе 15.4.

15.2. Стандартный подход для написания модулей на Lua

Простейший способ создать модуль на Lua действительно 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, как мы определяем функцию `inv` как закрытую, просто объявляя ее внутри блока.

Некоторым не нравится завершающий оператор `return`. Одним из способов устранить его является запись таблицы модуля непосредственно в `package.loaded`:

```
local M = {}  
package.loaded[...] = M  
    <as before>
```

Имейте в виду, что `require` вызывает загрузчик, передавая имя модуля как первый аргумент. Поэтому выражение переменного числа аргументов ... дает в результате именно это имя. 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*). Однако я предпочитаю возвращать таблицу, поскольку это выглядит аккуратнее.

Другим способом записи модуля является определение всех функций как локальных и построение таблицы в конце, как в листинге 15.2. В чем преимущества этого подхода? Вам не нужно начинать каждое имя с `M.` или чего-то похожего; существует явный список экспортируемых функций; вы определяете и используете экспортируемые и внутренние функции абсолютно одинаково внутри модуля. В чем заключаются недостатки этого подхода? Список экспортируемых функций находится в конце модуля, а не в его начале, где он был бы более удобен в качестве быстрой справки; и список для экспорта избыточен, так как нужно каждое имя записать дважды. (Этот последний недостаток может стать преимуществом, поскольку он позволяет функциям иметь разные имена снаружи модуля и внутри него, но я думаю, что программисты редко этим пользуются.) Мне лично нравится данный стиль.

Однако 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)
```

Листинг 15.1. Простой модуль для комплексных чисел

```
local M = {}
function M.new (r, i) return {r=r, i=i} end
-- определяем 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

```

Листинг 15.2. Модуль с явным списком экспортируемых функций

```

local function new (r, i) return {r=r, i=i} end
-- определяем константы 'i'
local i = complex.new(0, 1)
<other functions follow the same pattern>
return {
    new = new,
    i = i,
    add = add,
    sub = sub,
    mul = mul,
    div = div,
    tostring = tostring,
}

```

15.3. Использование окружений

Одним из недостатков рассмотренных методов для создания модулей является то, что очень легко засорить глобальное пространство имен, например просто забыв `local` в описании локального ресурса.

Окружения предоставляют интересный подход к созданию модулей, который решает эту проблему. Если у модуля есть свое окружение, то не только все функции попадут в эту таблицу, но также и все глобальные переменные. Поэтому мы можем определить все открытые функции как глобальные, и они автоматически попадут в соответствующую таблицу. Все, что нужно сделать модулю, — так это присвоить эту таблицу переменной `_ENV`. После этого, когда мы определяем функцию `add`, она автоматически становится `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 модуля без какого-либо префикса. В предыдущем коде `add` обращается к `new` из своего окружения, то есть на самом деле обращается к `M.new`.

Этот метод является хорошим способом создания модулей, требующим очень небольшой работы от программиста. Префиксы вообще не нужны. Нет никакой разницы между вызовом экспортируемой и закрытой функций. Если программист забывает вставить `local`, то он не засоряет глобальное пространство имен; вместо этого закрытая функция просто становится экспортируемой.

Тем не менее обычно я предпочитаю один из двух ранее рассмотренных методов. Хотя они могут потребовать чуть больше работы, тем не менее код получается более понятным. Для того чтобы не создать глобальную величину по ошибке, я просто присваиваю `_ENV` значение ***nil***. После этого любая попытка создать глобальную величину просто вызывает ошибку.

Чего при этом не хватает, так это доступа к другим модулям. После того как мы изменили значение `_ENV`, мы потеряли доступ ко всем предыдущим глобальным переменным. Есть несколько способов вернуть этот доступ, каждый со своими плюсами и минусами.

Одним вариантом является использование наследования:

```
local M = {}  
setmetatable(M, {__index = _G})  
_ENV = M
```

(Вам нужно вызвать `setmetatable` перед присваиванием `_ENV`, понятно почему?) При использовании этого подхода модуль получает прямой доступ к любой глобальной переменной, при очень небольшой цене такого доступа. Любопытным последствием этого решения является то, что ваш модуль теперь содержит все глобальные переменные. Например, кто-то, использующий ваш модуль, теперь может вызывать стандартную функцию для вычисления синуса при помощи `complex.math.sin(x)`. (Подобная особенность есть также и в языке Perl.)

Другим быстрым способом доступа к другим модулям является введение локальной переменной, содержащей глобальное 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 нет использования метаметодов.

Более строгим подходом является определение в качестве локальных переменных только тех функций или модулей, которые вам нужны:

```
-- настройка модуля
local M = {}
-- раздел импорта:
-- возьмите снаружи все, что нужно этому модулю
local sqrt = math.sqrt
local io = io
-- с этого места доступ наружу невозможен
_ENV = nil -- or _ENV = M
```

Этот подход требует больше работы, но он явно документирует зависимости вашего модуля. Также он приводит к коду, который выполняется немного быстрее, чем в ранее рассмотренных случаях, из-за использования локальных переменных.

15.4. Подмодули и пакеты

Lua допускает использование иерархических имен модулей, используя точку для разделения уровней. 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`, то `require` сперва ищет в таблице `package.loaded` и затем в таблице `package.preload`, используя полное имя “`mod.sub`” в качестве ключа; в этом случае точка является таким же символом, как и любой другой.

Однако при поиске файла, задающего этот подмодуль, `require` переводит точку в другой символ, обычно системный разделитель в пути (то есть ‘/’ для UNIX и ‘\’ для Windows). После этого преобразования `require` ищет получающемся имя, как и любое другое имя. Например, путь ‘/’ – это разделитель пути, и у нас есть следующий путь:

```
./?.lua;/usr/local/lua/?.lua;/usr/local/lua/?.init.lua
```

Вызов `require(“a.b”)` попыбует 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 отдельном каталоге. 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`, где каталог `p` содержится в соответствующем месте.

Разделитель пути, используемый Lua, задается во время компиляции и может быть любой строкой (вспомните, что Lua ничего не знает про каталоги). Например, системы без иерархических каталогов 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 на 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 на C с именем `a.b` должна назвать свою инициализирующую функцию `luaopen_a_b`. Мы также можем использовать здесь прием с минусом, но с более сложным результатом. Например, пусть у нас есть библиотека на C с именем `a` и мы хотим сделать ее подмодулем `mod`, тогда мы можем переименовать соответствующий файл в `mod/v-a`. При вызове `require "mod.v-a"` вызов `require` правильно найдет новый файл `mod/v-a`, так же как и функцию `luaopen_a` внутри него.

Также у `require` есть один дополнительный искатель для загрузки подмодулей на C. Когда он не может найти ни Lua-файл, ни C-файл для подмодуля, этот искатель опять ищет в пути для C, но на этот раз ищет имя пакета. Например, если программа хочет загрузить подмодуль `a.b.c`, то этот искатель просто будет искать `a`. Если он найдет библиотеку на C для этого имени, то `require` будет искать в этой библиотеке соответствующую функцию, в нашем случае `luaopen_a_b_c`. Эта возможность позволяет размещать несколько подмодулей вместе в одной библиотеке на C, каждая со своей инициализирующей функцией.

С точки зрения Lua, подмодули в одном пакете не имеют явной связи. Загрузка модуля `a` не приводит к загрузке любого из ее подмодулей; также загрузка `a.b` не загружает автоматически `a`. Конечно, при реализации пакета разработчик вправе задать эти связи при желании. Например, модуль `a` может явно потребовать загрузки как кого-то конкретного (или всех) своего подмодуля.

Упражнения

Упражнение 15.1. Перепишите код в листинге 13.1 как отдельный модуль.

Упражнение 15.2. Что случится при поиске библиотеки, если путь содержит фиксированную компоненту (то есть компоненту, не содержащую знака вопроса)? Может ли такое поведение быть полезным?

Упражнение 15.3. Напишите искатель, который одновременно ищет файлы на Lua и библиотеки на C. Например, путь для этого искателя может быть чем-то вроде:

```
./?.lua;./?.so;/usr/lib/lua5.2/?.so;/usr/share/lua5.2/?.lua
```

(Подсказка: используйте `package.searchpath` для поиска соответствующего файла, затем попытайтесь загрузить его, сначала при помощи `loadfile`, затем при помощи `package.loadlib`.)

Упражнение 15.4. Что случится, если вы установите метатаблицу для `package.preload` при помощи метаметода `__index`? Может ли это быть полезным?



ГЛАВА 16

Объектно-ориентированное программирование

Таблица в Lua является объектом более чем в одном смысле. Подобно объектам, у таблицы есть состояние. Подобно объектам, у таблицы есть идентичность (*self*), которая не зависит от ее значений; в частности, две таблицы с одинаковыми значениями являются разными объектами, 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 методы. У таблиц также могут быть свои методы, как показано ниже:

```
Account = {balance = 0}
function Account.withdraw (v)
    Account.balance = Account.balance - v
end
```

Это определение создает новую функцию и запоминает ее в поле `withdraw` объекта `Account`. Затем мы можем позвать ее, как показано ниже:

```
Account.withdraw(100.00)
```

Функция подобного типа – это почти то, что мы называем *методом*. Однако использование глобального имени `Account` внутри функции является плохой практикой. Во-первых, эта функция будет работать только для данного конкретного объекта. Во-вторых, даже для этого объекта ровно до тех пор, пока этот объект записан в этой 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 гибким вариантом является использование *получателя* операции. Для этого нашему методу понадобится дополнительный аргумент со значением получателя. Этот параметр обычно имеет имя *self* или *this*:

```
function Account.withdraw (self, v)
  self.balance = self.balance - v
end
```

Теперь, когда мы вызываем метод, мы должны указать, с каким объектом 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)
```

Это использование параметра *self* является ключевым в любом объектно-ориентированном языке. В большинстве объектно-ориентированных языков данный механизм частично скрыт от программиста, поэтому этот параметр не нужно явно объявлять (хотя внутри метод по-прежнему можно использовать — *self* или *this*). Lua также может скрывать этот параметр при помощи *оператора двоеточия*. Мы можем переписать предыдущее определение метода следующим образом:

```
function Account:withdraw (v)
  self.balance = self.balance - v
end
```

Тогда вызов метода будет выглядеть следующим образом:

```
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 метод при использовании синтаксиса с точкой и позвать его, используя синтаксис с двоеточием, и 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 операции над этим состоянием. Им не хватает системы классов, наследования и возможности скрыть свои переменные (состояние). Давайте сначала разберемся с первой задачей: как мы можем создать различные объекты с одинаковым поведением? Например, как мы можем создать несколько счетов?

16.1. Классы

Класс выступает как шаблон для создания объектов. Большинство объектно-ориентированных языков предлагают понятие класса. В таких языках каждый объект является экземпляром какого-то конкретного класса. В Lua нет понятия класса; каждый объект определяет свое поведение и свои данные. Однако это совсем не сложно — эмулировать классы в Lua, идя по пути прототипных языков вроде Self или NewtonScript. В этих языках у объектов нет классов. Вместо этого каждый объект может иметь прототип, который является объектом, в котором первый объект ищет операции, которые он не знает. 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 выступают в качестве места, в которое помещается поведение, общее 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* выступил как прототип для *a*, — это следующее:

```
setmetatable(a, {__index = b})
```

После этого *a* будет искать в *b* все операции, которых он не знает. Установить *b* в качестве класса для *a* — это на самом деле практически то же самое.

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 операции от `Account` при помощи метаметода `__index`. В качестве небольшой оптимизации мы можем не создавать отдельные метатаблицы для каждого из объектов; вместо этого мы будем использовать саму таблицу `Account`:

```
function Account:new (o)
    o = o or {} -- создать таблицу, если пользователь не передал ее
    setmetatable(o, self)
    self.__index = self
    return o
end
```

(Когда мы вызываем `Account:new`, то `self` равно `Account`; поэтому мы могли бы явно использовать `Account` вместо `self`. Однако использование `self` очень нам при пригодится в следующем разделе, когда мы введем наследование.) Что произойдет, 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` будет выставлена `Account` (параметр `self` при вызове `Account:new`) в качестве метатаблицы. Затем, когда мы вызываем `a:deposit(100.00)`, мы на самом деле вызываем `a.deposit(a, 100.00)`; двоеточие – это просто синтаксический сахар. Однако Lua не может найти запись `deposit` в таблице `a`; поэтому Lua ищет запись `__index` в метатаблице. Ситуация выглядит примерно следующим образом:

```
getmetatable(a).__index.deposit(a, 100.00)
```

Метатаблицей `a` является `Account` и `Account.__index` – это также `Account` (поскольку метод `new` выполнил `self.__index=self`). Поэтому предыдущее выражение сводится к

```
Account.deposit(a, 100.00)
```

То есть Lua вызывает исходную функцию `deposit`, но передавая `a` в качестве параметра `self`. Таким образом, новый счет `a` унаследовал функцию `deposit` от `Account`. Таким же образом он наследует все поля от `Account`.

Наследование работает не только для методов, но также и для других полей, которых нет 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
```

Когда мы вызовем у `b` метод `deposit`, то этот вызов будет эквивалентен следующему коду (поскольку *self* равно `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 соответствующего метаметода, так как у `b` теперь есть свое 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).

Пусть у нас есть базовый класс `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 класс `SpecialAccount`, позволяющий покупателю снять больше, чем есть на его балансе. Мы начинаем с пустого класса, который наследует все операции от своего базового класса:

```
SpecialAccount = Account:new()
```


До этого момента `SpecialAccount` является просто экземпляром `Account`. Однако интересное случается дальше:

```
s = SpecialAccount:new{limit=1000.00}
```

`SpecialAccount` наследует `new` от `Account`, как и все остальные 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` равно `SpecialAccount`. Поэтому `s` наследует от `SpecialAccount`, который, в свою очередь, наследует от `Account`. Теперь если мы выполним

```
s:deposit(100.00),
```

то Lua не сможет найти поле `deposit` в `s`, поэтому он будет искать его в `SpecialAccount`, там его он также не найдет и будет далее искать в `Account`, где он и найдет исходную реализацию этого метода.

Что делает `SpecialAccount` особенным, это то, что мы можем переопределить любой метод, унаследованный от его родительского класса. 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)`, то Lua не обратится в `Account`, поскольку она до этого найдет новый метод `withdraw` в классе `SpecialAccount`. Так как `s.limit` равно `1000.00` (мы задали это поле при создании `s`), то программа осуществит снятие, оставляя в результате `s` с отрицательным балансом.

Интересной особенностью объектов в Lua является то, что вам не нужно создавать новый класс для задания нового поведения. Если изменить поведение нужно всего для одного объекта, то мы можем реализовать это изменение непосредственно в этом объекте. Например, если счет `s` представляет особого клиента, чей предел всегда равен 10% от текущего баланса, то мы можем изменить всего лишь один счет:

```
function s:getLimit ()
    return self.balance * 0.10
end
```

После этого вызов `s:withdraw(200.0)` выполнит метод `withdraw` из класса `SpecialAccount`, но когда `withdraw` вызовет `s:getLimit`, то будет вызвано ранее введенное определение этой функции.

16.3. Множественное наследование

Поскольку объекты не являются базовыми примитивами, в Lua есть несколько способов использовать объектно-ориентированное программирование. Подход, который мы только что видели, использующий метаметод `__index`, является, наверное, лучшей комбинацией простоты, скорости и гибкости. Однако есть и другие реализации, которые могут оказаться более подходящими для каких-то определенных случаев. Сейчас мы увидим альтернативную реализацию, которая допускает множественное наследование в Lua.

Ключевым в этой реализации является использование функции в качестве метаполя `__index`. Напомним, что когда у метатаблицы данной таблицы есть поле `__index`, то Lua вызовет эту функцию всякий раз, когда не сможет найти ключ в исходной таблице. В этом случае `__index` может искать отсутствующий ключ в любом количестве родителей.

Множественное наследование означает, что у класса может быть более одного суперкласса (родительского класса). Поэтому мы уже не можем использовать такую функцию, как ранее, для создания дочерних классов. Вместо этого мы определим функцию `createClass`, которая получает в качестве аргументов родительские классы (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 принадлежит одному классу, который и используется для поиска методов. Поэтому взаимоотношение между классом и суперклассами отличается от взаимоотношения между классами и его экземплярами (созданными объектами). В частности, класс не может одновременно быть метатаблицей для его экземпляров и дочерних классов. В листинге 6.1 мы используем класс как метатаблицу для созданных экземпляров и создаем отдельную таблицу в качестве метатаблицы класса.

Листинг 16.1. Реализация множественного наследования

```

-- ищем 'k' в списке таблиц 'plist'
local function search (k, plist)
  for i = 1, #plist do
    local v = plist[i][k] -- попробовать i-й 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 родителей
  setmetatable(c, {__index = function (t, k)
    return search(k, parents)
  end})
  -- подготовить 'c' в качестве метатаблицы его экземпляров
  c.__index = c
  -- определить новый конструктор для этого нового класса
  function c:new (o)
    o = o or {}
    setmetatable(o, c)
    return o
  end
  return c -- вернуть новый класс
end

```

Давайте проиллюстрируем использование `createClass` при помощи небольшого примера. Пусть у нас есть наш старый класс `Account` и класс `Named` с 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 дочерним классом и `Account`, и `Named`, мы просто вызовем `createClass`:

```
NamedAccount = createClass(Account, Named)
```

Мы создаем и используем экземпляры этого класса, как и ранее:

```

account = NamedAccount:new{name = "Paul"}
print(account:getname()) --> Paul

```

Теперь давайте посмотрим, как работает последний оператор. Lua не может найти метод `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, то Lua вызывает ее. Эта функция сперва ищет `"getname"` в `Account` и, не найдя его там ищет в `Named`, где она и находит отличное от *nil* значение, которое и становится окончательным результатом.

Конечно, из-за сложности такого поиска быстроедействие для множественного наследования отличается от быстрогодействия для простого наследования. Простым способом улучшить это быстроедействие является скопировать наследованные методы в дочерние классы. С использованием этого подхода метаметод `__index` будет выглядеть следующим образом:

```
setmetatable(c, {__index = function (t, k)
    local v = search(k, parents)
    t[k] = v -- сохранить 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). Недостатком является то, что сложно изменить определения методов, когда система работает, поскольку эти изменения не переносятся вдоль цепочки наследования.

16.4. Скрытие

Многие считают возможность скрытия 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 или его метод видны снаружи. В языке Smalltalk все переменные скрыты, а все методы доступны снаружи. Simula, первый объектно-ориентированный язык, не предоставляет подобной защиты для полей и методов.

Дизайн объектов для Lua, который мы ранее рассматривали, не предоставляет механизмов скрытия. Частично это является следствием нашего использования таблиц для представления объектов. Кроме того, Lua избегает избыточности и искусственных ограничений. Если вы не хотите обращаться к полям внутри объекта, просто *не делайте этого*.

Тем не менее другой целью Lua является гибкость, она предоставляет метамеханизмы, позволяющие эмулировать многие возможности. Хотя базовый дизайн объектов для Lua и не предусматривает механизмов скрытия, мы можем реализовать объекты другим способом, так чтобы получить контроль за доступом. Хотя эту возможность программисты используют нечасто, будет полезным узнать о ней, поскольку это приоткрывает некоторые интересные аспекты Lua и может быть хорошим решением и для других задач.

Основная идея альтернативного дизайна – это представлять каждый объект при помощи двух таблиц: одна – для его состояния и другая – для его операций (его интерфейс). Обращение к объекту идет через вторую таблицу, то есть через операции, образующие его интерфейс. Для того чтобы избежать несанкционированного доступа, таблица, предоставляющая его состояние, не хранится в поле другой таблицы, она доступна только через замыкания внутри методов. Например, чтобы представлять банковский счет при помощи этого дизайна, мы будем 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 хранения внутреннего состояния объекта и запоминает ее в локальной переменной `self`. Затем функция создает методы для объекта. Наконец, функция создает и возвращает внешний объект, который сопоставляет имена методов их реализациям. Ключевым здесь является то, что эти методы не получают `self` как дополнительный параметр; вместо этого они непосредственно обращаются к `self`. Поскольку дополнительного аргумента нет, то мы не используем синтаксис с двоеточием для работы с объектом. Мы вызываем их методы просто как обычные функции:

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance()) --> 60
```

Этот дизайн обеспечивает полную скрытость для всего, что хранится в таблице `self`. После возвращения из функции `newAccount` нет никакого способа получить непосредственный доступ к этой таблице. Хотя наш пример хранит всего одну переменную в закрытой таблице, мы можем хранить все закрытые части объекта в этой таблице. Мы можем также определить закрытые методы: они похожи на открытые, но мы не помещаем их в интерфейс. Например, наши счета могут предоставлять дополнительный 10%-ный кредит при балансе выше определенной величины, но мы не хотим, чтобы пользователи имели доступ к деталям вычислений. 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
end
```

<как ранее>

Опять нет никакого способа непосредственно позвать функцию `extra`.

16.5. Подход с единственным методом

Частным случаем предыдущего подхода для объектно-ориентированного программирования является случай, когда у объекта всего один метод. В подобном случае нам не нужно создавать интерфейсную таблицу; мы можем просто вернуть этот метод в качестве представления объекта. Если это выглядит немного странно, давайте вспомним раз-

дел 7.1, где мы создавали итерирующие функции, хранящие свое состояние как замыкания. Итератор, хранящий свое состояние, ничем не отличается от объекта с единственной функцией.

Другим интересным случаем объектов с единственным методом является случай, когда этот метод на самом деле выполняет различные задачи в зависимости от определенного 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 довольно просто:

```
d = newObject(0)
print(d("get")) --> 0
d("set", 10)
print(d("get")) --> 10
```

Эта реализация объектов довольно эффективна. Синтаксис `d("set", 10)` хотя и выглядит странно, всего на два символа длиннее, чем традиционный `d:set(10)`. Каждый объект использует одно замыкание, что дешевле одной таблицы. Здесь нет наследования, но зато мы имеем полную закрытость: единственный способ обратиться к состоянию объекта заключается в использовании его единственного метода.

Tcl/Tk использует похожий подход для своих виджетов. Имя виджета в Tk обозначает функцию (*команду виджета*), которая может выполнять различные типы операций 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 закрытость для объектов – это реализовать их с использованием прокси (proxy) (см. раздел 13.4). Каждый объект представлен пустой таблицей (прокси). Внутренняя таблица устанавливает соответствие между этими пустыми таблицами и таблицами, несущими состояние объекта. Эта внутренняя таблица не доступна снаружи, но методы используют ее для перевода своего параметра `self` на 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 управление памятью. Программы создают объекты (таблицы, нити и т. п.), но нет функции для уничтожения объектов. Lua автоматически уничтожает объекты, которые становятся мусором, при помощи *сборки мусора*. Это освобождает вас от основной части работы с памятью и, что более важно, освобождает от большинства ошибок, связанных с этой деятельностью, таких как висящие ссылки и утечки памяти.

Использование сборщика мусора означает, что у Lua нет проблем с циклами. Вам не нужно никаких специальных действий при использовании циклических структур данных; они автоматически освобождаются, как и любые другие данные. Однако иногда даже умному сборщику мусора нужна ваша помощь. Ни один сборщик мусора не позволит вам забыть обо всех проблемах об управлении ресурсами, такими как внешние ресурсы.

Слабые таблицы и финализаторы – это механизмы, которые вы можете использовать в Lua для того, 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 сам угадать, что является мусором по вашему мнению. Типичным примером является стек, реализованный как массив, со ссылкой на вершину стека. Вы знаете, что данные лежат только от начала массива и до этого индекса (вершины стека), но Lua этого не знает. Если вы снимаете элемент с вершины стека,

просто уменьшая индекс вершины, то оставшийся в массиве объект не является мусором для Lua. Аналогично любой объект, на который ссылается глобальная переменная, также не является мусором для Lua, даже если вы его никогда не будете использовать. В обоих случаях вам (точнее, вашей программе) следует записать **nil** в соответствующие переменные (или элементы массива), для того чтобы избежать появления неуничтожаемых объектов.

Однако просто убрать ссылки не всегда достаточно. В некоторых случаях нужно дополнительное взаимодействие между вашей программой и сборщиком мусора. Типичным примером является набор всех активных объектов определенного типа (например, файлов) в вашей программе. Задача кажется простой: все, что вам требуется, – это добавить каждый новый объект к этому набору. Однако как только объект становится частью набора, он уже никогда не будет уничтожен! Даже если на него никто и не ссылается, то набор все равно ссылается на него. Lua не может знать, что эта ссылка не должна препятствовать уничтожению этого объекта, если только вы не скажете Lua об этом.

Слабые таблицы – это тот механизм, который вы используете в Lua, для того чтобы сказать, что ссылка не должна препятствовать уничтожению объекта. *Слабая ссылка* – это такая ссылка на объект, которая не учитывается сборщиком мусора. Если все ссылки, указывающие на объект, являются слабыми, то данный объект освобождается, и все эти слабые ссылки уничтожаются. Lua реализует слабые ссылки при помощи слабых таблиц: *слабая таблица*, – это такая таблица, все ссылки которой являются слабыми. Это значит, что если объект хранится только внутри слабой таблицы, то сборщик мусора рано или поздно уничтожит данный объект.

Таблицы хранят ключи и значения, и те, и другие могут быть объектами любого типа. В нормальных условиях сборщик мусора не уничтожает объекты, которые являются ключами и ссылками в доступной таблице. И ключи, и значения являются *сильными* ссылками, то есть они предотвращают уничтожение тех объектов, на которые они указывают. В слабой таблице и ключи, и значения могут быть слабыми. Это значит, что существуют три типа слабых таблиц: таблицы со слабыми ключами, таблицы со слабыми значениями и полностью слабые таблицы, где и ключи, и значения являются слабыми. Независимо от типа таблицы, при уничтожении ключа или значения вся запись удаляется из таблицы.

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 поведение слабых таблиц:

```
a = {}
b = {__mode = "k"}
setmetatable(a, b)      -- теперь у 'a' слабые ключи
key = {}                 -- создаем первый ключ
a[key] = 1
key = {}                 -- создаем второй ключ
a[key] = 2
collectgarbage()         -- заставляем сборщик мусора удалить мусор
for k, v in pairs(a) do print(v) end
--> 2
```

В этом примере второе присваивание `key={}` уничтожает ссылку на первый ключ. Вызов `collectgarbage` заставляет сборщик мусора удалить весь мусор. Поскольку нет больше ссылок на первый ключ, то этот ключ и соответствующая запись в таблице удаляются. Второй ключ по-прежнему хранится в переменной `key`, поэтому он не удаляется.

Обратите внимание, что только объекты могут быть удалены из слабой таблицы. Такие значения, как числа и логические значения, не удаляются. Например, если мы вставим числовой ключ в таблицу `a` (из нашего предыдущего примера), то сборщик мусора никогда его не удалит. Конечно, если значение, соответствующее числовому ключу, хранится в таблице со слабыми значениями, то вся соответствующая запись целиком удаляется из таблицы.

Со строками есть определенная тонкость: хотя строки и удаляются сборщиком мусора, с точки зрения реализации они отличаются от остальных объектов. Другие объекты, такие как таблицы и нити, создаются явно. Например, когда Lua выполняет выражение `{}`, то он создает новую таблицу. Однако создает ли Lua новую строку при выполнении `"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 может быть удалена из слабой таблицы (кроме случая, когда удаляется связанное с ней значение).

17.2. Функции с кэшированием

Распространенным программистским приемом является получение выигрыша во времени за счет проигрыша по памяти. Вы можете

ускорить функцию, кэшируя ее результаты, так что когда позже вы вызовете эту же функцию с теми же аргументами, функция сможет использовать сохраненное в кэше значение.

Представьте себе сервер, получающий запросы в виде строк, содержащих код на Lua. Каждый раз при получении запроса сервер выполняет `load` для полученной строки и затем вызывает полученную функцию. Однако `load` — это дорогая функция, и некоторые команды серверу могут много раз повторяться. Вместо постоянного вызова `load` каждый раз, когда сервер получает команду вроде `"closeconnection()"`, сервер может запомнить результат `load` во вспомогательной таблице. Перед вызовом `load` сервер проверяет, нет ли уже значения, соответствующего данной строке. Если он не может найти соответствующее значение, то тогда (и только тогда) сервер вызывает `load` и запоминает результат в этой таблице. Мы можем реализовать это поведение при помощи следующей функции:

```
local results = {}
function mem_loadstring (s)
  local res = results[s]
  if res == nil then          -- результата нет?
    res = assert(load(s))    -- вычислить новый результат
    results[s] = res         -- сохранить результат
  end
  return res
end
```

Выигрыш от этой схемы может быть очень значительным. Однако также она может вызвать большие потери памяти. Хотя некоторые команды повторяются снова и снова, многие другие команды встречаются только один раз. Со временем таблица `results` собирает все команды, которые сервер когда-либо получал, и соответствующий им код; со временем 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 значения (фактически все):

```
local results = {}
setmetatable(results, {__mode = "v"}) -- значения будут слабыми
function mem_loadstring (s)
  <как ранее>
```

На самом деле, поскольку индексы всегда являются строками, мы можем сделать эту таблицу полностью слабой, если мы этого хотим:

```
setmetatable(results, {__mode = "kv"})
```

Техника кэширования также полезна, чтобы гарантировать уникальность объектов определенного типа. Например, пусть мы представляем цвета как таблицы с полями `red`, `green` и `blue`. Простейшая фабрика цветов будет создавать новую таблицу каждый раз, когда мы к ней обращаемся:

```
function createRGB (r, g, b)
  return {red = r, green = g, blue = b}
end
```

Используя кэширование, мы можем переиспользовать таблицы для одних и тех же цветов. Для создания уникального ключа для каждого цвета мы просто соединяем компоненты цвета при помощи некоторого разделителя:

```
local results = {}
setmetatable(results, {__mode = "v"}) -- значения будут 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 на равенство при помощи стандартного оператора сравнения, поскольку двум одновременно существующим одинаковым цветам всегда будет соответствовать одинаковая таблица. Обратите внимание, что одинаковый цвет может быть представлен разными таблицами в разные моменты времени, поскольку время от времени сборщик мусора будет опустошать таблицу `results`. Однако пока данный цвет используется, он не может быть удален из `results`. Поэтому если цвет существует достаточно долго, чтобы быть сравненным с другим цветом, его представление также будет существовать столь же долго.

17.3. Атрибуты объекта

Другим интересным использованием слабых таблиц является связывание атрибутов с объектами. Существует бесконечное число ситуаций, когда нам может понадобиться привязать некоторый атрибут к объекту: имена к функциям, значения по умолчанию к таблицам, размеры к массивам и т. д.

Когда объект является таблицей, то мы можем запомнить атрибут в самой таблице, выбрав подходящий уникальный ключ. Как мы уже видели, простой и надежный способ создать уникальный ключ – создать новый объект (обычно таблицу) и использовать его в качестве ключа. Однако если объект не является таблицей, то этот подход уже не годится. Даже для таблиц нам может понадобиться не хранить атрибут в самой таблице. Например, мы можем захотеть сделать подобный атрибут закрытым или мы не хотим влиять на то, как таблица перебирается. Во всех этих случаях нам нужен другой способ связывания атрибутов с объектами.

Конечно, отдельная таблица предоставляет идеальный способ привязывания атрибутов объектам (не случайно, что таблицы иногда называют *ассоциативными массивами*). Мы можем использовать объекты как ключи, а их атрибуты – как значения. Такая таблица может хранить атрибуты объектов любого типа, так как Lua позволяет использовать объекты любого типа в качестве ключей таблицы. Более того, атрибуты, хранимые в отдельной таблице, не влияют на другие объекты и могут быть закрытыми, так же как и сама таблица.

Однако это решение обладает огромным недостатком: как только мы использовали объект в качестве ключа в таблице, он уже не может быть удален сборщиком мусора. Lua не может удалить объект, который используется в качестве ключа. Если мы используем обычную таблицу, для того чтобы привязать к функциям их имена, то ни одна из этих функций никогда не будет удалена. Как вы можете предположить, мы можем избежать этого недостатка при помощи слабых таблиц. Однако на этот раз нам понадобятся слабые ключи. Использование слабых ключей не мешает сборщику мусора удалять эти ключи, когда на них не остается больше ссылок. С другой стороны, у таблицы не могут быть слабые значения; иначе атрибуты существующих объектов могли бы быть удалены.

17.4. Опять таблицы со значениями по умолчанию

В разделе 13.4 мы рассмотрели, как можно работать со значениями по умолчанию, отличными от *nil*. Мы показали один подход и заметили, что два других подхода требуют использования слабых таблиц, поэтому рассказ о них мы отложили на потом. Теперь пора вернуться к этой теме. Как вы увидите, эти два подхода к реализации значений по

умолчанию на самом деле являются частными случаями уже рассмотренных подходов, а именно атрибутов объектов и кэширования.

В первом подходе мы используем слабые таблицы, для того чтобы связать с 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 `defaults` не использовал слабые ключи, то все таблицы со значениями по умолчанию существовали бы всегда.

Во втором решении мы используем разные метатаблицы для разных значений по умолчанию, но при этом мы переиспользуем одну и ту же метатаблицу, когда мы снова используем то же самое значение по умолчанию. Это типичный случай кэширования:

```
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 -- запомнить
    end
    setmetatable(t, mt)
end
```

В этом случае мы используем слабые значения, для того чтобы неиспользуемые метатаблицы могли бы быть собраны сборщиком мусора.

Какое из этих двух решений является лучшим? Как обычно, это зависит от использования. Оба решения обладают примерно одинаковой сложностью и одинаковым быстродействием. Первое решение требует нескольких слов памяти для каждой таблицы со значением по умолчанию (на запись в `defaults`). Второе решение требует нескольких десятков слов памяти на каждое уникальное значение по умолчанию (новая таблица, новое замыкание плюс 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

Интересный случай возникает, когда в таблице со слабыми ключами значение ссылается на его собственный ключ.

Этот случай гораздо более распространен, чем может показаться. Типичным примером является фабрика, возвращающая функции. Подобная фабрика получает объект и возвращает функцию, которая при вызове вернет этот объект:

```
function factory (o)
  return function () return o end
end
```

Эта фабрика является хорошим кандидатом для кэширования, для того чтобы не создавать новые замыкания, когда уже есть подходящее, уже созданное замыкание:

```
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 — (соответствующая функция), связанная с объектом, находящимся в `mem`, — ссылается на свой собственный ключ (сам объект). Хотя ключи являются слабыми в этой таблице, но значения слабыми не являются. При стандартной интерпретации слабых таблиц ничто не будет удалено из кэширующей таблицы. Поскольку значения не являются слабыми, то всегда есть сильная ссылка на каждую функцию. Каждая функция ссылается на свой объект, то есть всегда есть сильная ссылка на каждый объект. Поэтому эти объекты не могут быть удалены, несмотря на использование слабых ключей.

Однако подобная интерпретация не всегда очень полезна. Большинство людей ожидает, что значение в таблице доступно только через соответствующий ключ. Поэтому мы можем рассматривать подобный сценарий как случай цикла, где замыкание ссылается на объект, который (через кэширующую таблицу) сам ссылается на это замыкание.

Lua 5.2 решает данную проблему при помощи 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. Более точно рассмотрим запись (k, v) в эфемерной таблице. Ссылка на v является сильной, только если есть сильная ссылка на k . В противном случае запись со временем удаляется из таблицы, даже если v ссылается (непосредственно или опосредованно) на k .

17.6. Финализаторы

Хотя целью сборщика мусора является удаление объектов Lua, он также может помочь программе освобождать внешние ресурсы. Для этих целей различные языки программирования предлагают механизм финализаторов. *Финализатор* – 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 вызываем сборку мусора при помощи вызова `collectgarbage`. Во время сборки мусора Lua обнаруживает, что данная таблица не является доступной и вызывает ее финализатор (метаметод `__gc`).

Тонким моментом в Lua является пометчение объекта для финализации. Мы помечаем объект для финализации, когда задаем для него метатаблицу с ненулевым полем `__gc`. Если мы не пометим объект, то он не будет финализирован. Большая часть кода, который мы пишем, будет работать, однако 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 мы потом и добавляем поле `__gc` метатаблице, Lua не рассматривает это присваивание как нечто особенное, поэтому объект и не будет помечен. Как мы уже сказали, это редко бывает проблемой; обычно метатаблица не изменяется после того, как она была назначена метатаблицей.

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` помечается для финализации. Нет никакой проблемы в том, чтобы задать метаметод позже; Lua вызывает финализатор, только если он является функцией.

Когда сборщик мусора уничтожает 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.

Распространенным заблуждением является мнение о том, что ссылки между уничтожаемыми объектами могут повлиять на порядок, в котором они будут финализированы. 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 не накладывают никакого порядка на финализацию.

Другим тонким моментом, связанным с финализаторами, является *восстановление*. Когда финализатор вызывается, то он получает финализируемый объект в качестве параметра. Таким образом, объект снова становится живым, по крайней мере на время финализации. Я называю это *временным восстановлением*. Во время выполнения финализатора ничего не мешает ему запомнить объект например в глобальной переменной, таким образом, что объект останется доступным после завершения финализатора. Я называю это *постоянным восстановлением*.

Восстановление должно быть транзитивным. 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 перед финализацией B. Lua должен восстановить и A, и B перед вызовом финализатора.

Из-за восстановления объекты с финализаторами восстанавливаются в два прохода. Вначале сборщик мусора обнаруживает, что объект с финализатором недостижим (на него никто не ссылается), тогда он восстанавливает этот объект и добавляет его к очереди для 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 гарантировать, что весь мусор в вашей программе действительно собран, то вы должны позвать `collectgarbage` дважды; второй вызов уничтожит объекты, которые были финализированы во время первого вызова.

Финализатор для каждого объекта выполняется ровно один раз, поскольку Lua помечает уже финализированные объекты. Если объект не был удален до конца работы программы, то Lua позовет его в самом конце. Эта возможность позволяет реализовать в Lua аналог `atexit` функций, то есть функций, которые вызываются непосредственно перед завершением работы программы. Все, что для этого нужно, – это создать таблицу с финализатором и запомнить ссылку на нее где-нибудь, например в глобальной переменной:

```
_G.AA = {__gc = function ()
-- your 'atexit' code comes here
print("finishing Lua program")
end}
setmetatable(_G.AA, _G.AA)
```

Другой интересной возможностью является возможность вызывать определенную функцию каждый раз, когда Lua завершает цикл сборки мусора. Поскольку финализатор вызывается ровно один раз, то необходимо в финализаторе создать новый объект для вызова следующего финализатора:

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

Взаимодействие объектов с финализаторами и слабых таблиц содержит тонкий момент. 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 -- добавим 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 слабыми ключами (как мы это рассмотрели в разделе 17.3) и финализаторам может понадобиться обращение к этим атрибутам. Однако мы используем таб-

лицы со слабыми значениями для переиспользования существующих объектов; в этом случае финализируемые объекты больше не нужны.

Упражнения

Упражнение 17.1. Напишите код, для того чтобы проверить, действительно ли Lua использует эфемерные таблицы. (Не забудьте вызвать `collectgarbage` для сборки мусора.) По возможности проверьте ваш код как в Lua 5.1, так и в Lua 5.2.

Упражнение 17.2. Рассмотрим первый пример из разделе 17.6, создающий таблицу с финализатором, печатающим сообщение при вызове. Что произойдет, если программа завершится без вызова сборки мусора? Что случится, если программа вызовет `os.exit`? Что случится, если программа завершит свое выполнение с ошибкой?

Упражнение 17.3. Пусть вам нужно реализовать кэширующую таблицу для функции, получающей строку и возвращающей строку. Использование слабой таблицы не позволит удалять записи, поскольку слабые таблицы не рассматривают строки как удаляемые объекты. Как вы можете реализовать кэширование в этом случае?

Упражнение 17.4. Объясните вывод следующей программы:

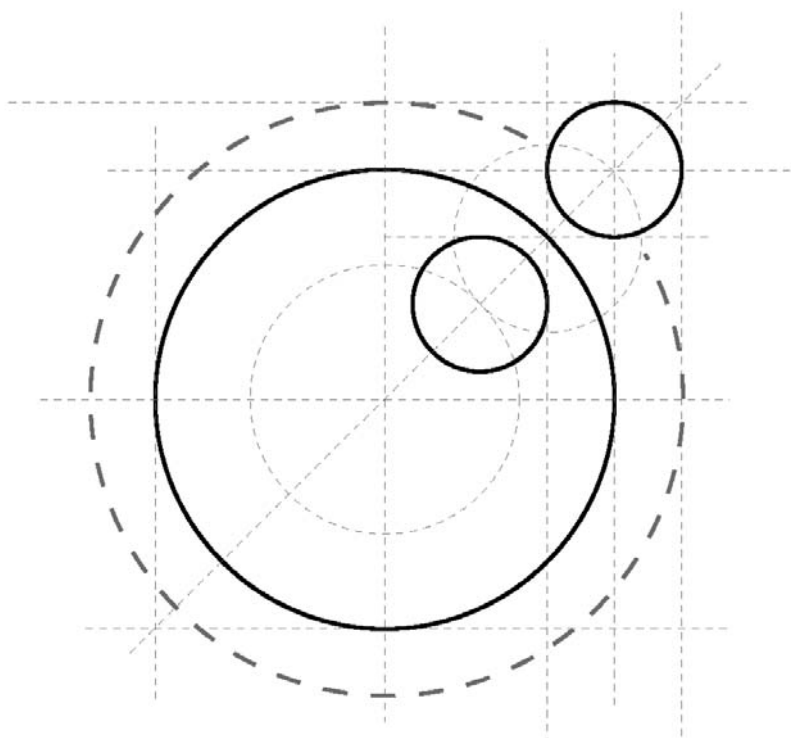
```
local count = 0
local mt = {__gc = function () count = count - 1 end}
local a = {}
for i = 1, 10000 do
    count = count + 1
    a[i] = setmetatable({}, mt)
end

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 стандартной библиотеке, моей целью является не дать полную спецификацию каждой функции, а показать, какую функциональность предоставляет каждая библиотека. Я могу опустить некоторые специфические опции или поведение для ясности изложения. Главной целью является зажечь ваше любопытство, которое затем может быть удовлетворено чтением документации по Lua.

Библиотека `math` содержит стандартный набор математических функций, таких как тригонометрические (`sin`, `cos`, `tan`, `asin`, `acos` и т. п.), экспоненцирование и логарифмирование (`exp`, `log`, `log10`), округление (`floor`, `ceil`), `min`, `max`, функции для генерации псевдослучайных чисел (`random`, `randomseed`) и переменные `pi` и `huge` (последнее является наибольшим представимым 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. Мы можем вызывать ее тремя разными способами. Когда мы вызываем ее без аргументов, она возвращает вещественное псевдослучайное число в диапазоне `[0, 1)`. Когда мы вызываем ее с единственным аргументом,

целым n , то она возвращает псевдослучайное целое число x , лежащее между 1 и n . Наконец, мы можем вызвать ее с двумя целочисленными аргументами l и u , тогда она вернет псевдослучайное целое число, лежащее между l и u .

Вы можете задать «затравку» (seed) для генератора псевдослучайных чисел при помощи функции `randomseed`; ее единственным числовым аргументом является «затравка». Обычно при начале работы программы генератор псевдослучайных чисел инициализируется 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 оказывается весьма полезным, но в игре все время будете получать одно и то же. Стандартным приемом для борьбы с этим является использование текущего времени в качестве «затравки» при помощи вызова `math.randomseed(os.time())`. Функция `os.time()` возвращает число, представляющее текущее время, обычно в виде числа секунд, прошедших с определенной даты.

Функция `math.random` использует функцию `rand` из стандартной библиотеки языка C. В некоторых реализациях возвращает числа с не очень хорошими статистическими свойствами. Вы можете обратиться к независимым дистрибутивам в поисках более удачного генератора псевдослучайных чисел. (Стандартная поставка Lua не включает в себя подобного генератора из-за 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 конуса по его высоте и углу между его образующей и осью.

Упражнение 18.3. Реализуйте другой генератор псевдослучайных чисел для Lua. Поищите хороший алгоритм в Интернете. (Вам может понадобиться библиотека для побитовых операций.)

Упражнение 18.4. Используя функцию `math.random`, напишите функцию для получения псевдослучайных чисел с гауссовским распределением.

Упражнение 18.5. Напишите функцию для перемешивания заданного списка. Убедитесь, что все варианты равновероятны.



ГЛАВА 19

Библиотека для побитовых операций

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 не равно -1)? Или может быть $2^{64}-1$ (число, которое нельзя точно представить при помощи значения типа `double`)?

Для того чтобы избежать 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 для 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=-1`. Более того, некоторые побитовые библиотеки используют различные соглашения для своих результатов. По-

этому 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-то код>
```

Побитовая библиотека также определяет операции для сдвига и вращения бит: `lshift` для сдвига налево; `rshift` и `arshift` 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.

```
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, начиная с 0 и 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. Вторым 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 числа единичных бит в двоичном представлении числа.

Упражнение 19.3. Напишите функцию для проверки того, является ли двоичное представление числа палиндромом.

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

Библиотека для работы с таблицами

Библиотека `table` содержит в себе дополнительные функции, позволяющие работать с таблицами как с массивами. Она предоставляет функции для вставки и удаления элементов из списка, для сортировки элементов массива и для конкатенации всех строк в массиве.

20.1. Функции `insert` и `remove`

Функция `table.insert` вставляет элемент в заданное место массива, сдвигая остальные элементы, для того чтобы освободить место. Например, если `t` — это массив `{10, 20, 30}`, то после вызова `table.insert(t, 1, 15)` `t` будет равен `{15, 10, 20, 30}`. Специальным (и довольно частым) случаем является вызов `insert` без указания положения, тогда элемент вставляется в самый конец массива и сдвига элементов не происходит. В качестве примера следующий код читает ввод строку за строкой, запоминая все строки в массиве:

```
t = {}
for line in io.lines() do
    table.insert(t, line)
end
print(#t) --> (число прочтенных строк)
```

В Lua 5.0 этот прием довольно распространен. В более поздних версиях я предпочитаю использовать `t[#t+1]=line`, для того чтобы добавить строку к массиву.

Функция `table.remove` удаляет (и возвращает) элемент из заданного места массива, сдвигая при этом следующие элементы массива. Если при вызове положение внутри массива не было указано, то удалится последний элемент массива.

При помощи этих двух функций довольно легко реализовать стеки, очереди и двойные очереди. Мы можем инициализировать

подобные структуры как `t={}`. Операция добавления элемента эквивалентна `table.insert(t,x)`; операция удаления элемента эквивалентна `table.remove(t)`. Вызов `table.insert(t,1,x)` добавляет элемент в другой конец соответствующей структуры, а вызов `table.remove(t,1)` соответственно удаляет элемент из этого конца. Две последние операции не особенно эффективны, так как они должны перемещать все элементы массива в памяти. Однако поскольку в библиотеке `table` эти функции реализованы на С, то они не являются слишком дорогими и хорошо работают для небольших массивов (до нескольких сот элементов).

20.2. Сортировка

Другой полезной функцией для работы с массивами является `table.sort`; мы уже видели ее ранее. Она принимает в качестве аргументов массив и опционально функцию для сравнения. Эта функция принимает на вход два аргумента и должна вернуть *true*, если первый элемент должен идти перед вторым. Если эта функция не указана, то функция сортировки использует стандартный оператор `<`.

Типичная путаница происходит, когда программист пытается отсортировать индексы в таблице. В таблице индексы образуют множество, в котором нет никакого упорядочения. Если вы хотите их отсортировать, то вам надо скопировать их в массив и отсортировать этот массив. Давайте рассмотрим пример. Пусть вы прочли входной файл и построили таблицу, которая для каждого имени функции содержит строку, в которой эта функция была определена: что-то вроде следующего:

```
lines = {
  luaH_set = 10,
  luaH_get = 24,
  luaH_present = 48,
}
```

И теперь вам нужно напечатать эти функции в алфавитном порядке. Если вы обойдете эту таблицу при помощи функции `pairs`, то имена окажутся в произвольном порядке. Вы не можете их явно отсортировать, поскольку эти имена являются ключами таблицы. Однако если вы поместите их в массив, то тогда уже этот массив можно отсортировать. Поэтому вам сначала нужно создать массив с этими именами, затем отсортировать его и уже потом 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 Lua в массивах нет никакого упорядочения (массивы – это на самом деле таблицы). Поэтому мы навязываем упорядочение при работе с индексами, которые можно упорядочить. Именно поэтому вам лучше обходить массив при помощи `ipairs`, а не `pairs`. Первая из этих функций устанавливает порядок ключей 1, 2, 3, ..., в то время как вторая просто использует произвольный порядок из таблицы.

В качестве более продвинутого решения мы можем написать итератор для обхода таблицы, использующей заданный порядок ключей. Необязательный параметр `f` задает этот порядок. Этот итератор сначала сортирует ключи в отдельный массив, а затем уже обходит этот массив. На каждом шаге он возвращает ключ и соответствующее значение из исходного массива:

```
function pairsByKeys (t, f)  
  local a = {}  
  for n in pairs(t) do a[#a + 1] = n end  
  table.sort(a, f)  
  local i = 0  
  return function () -- итерирующая функция  
    i = i + 1  
    return a[i], t[a[i]]  
  end  
end
```

При помощи этого итератора легко напечатать имена функций в алфавитном порядке:

```
for name, line in pairsByKeys(lines) do  
  print(name, line)  
end
```

20.3. Конкатенация

В разделе 11.6 мы уже видели функцию `table.concat`. Она берет на вход список строк и возвращает результат конкатенации всех этих строк. Необязательный второй аргумент задает строку-разделитель. Также есть еще два необязательных аргумента, которые задают индексы первой и последней конкатенируемых строк.

Следующая функция является интересным обобщением `table.concat`. Она может принимать на вход 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 `rconcat` рекурсивно вызывает себя для обработки вложенных списков. Затем она вызывает `table.concat` для объединения промежуточных результатов.

```
print(rconcat({"a", {" nice"}}, " and", {" long"}, {" list"}))
--> a nice and long list
```

Упражнения

Упражнение 20.1. Перепишите функцию `rconcat` так, чтобы для нее можно было задать строку-разделитель:

```
print(rconcat({{"a", "b"}, {"c"}}, "d", {}, {"e"}}, ";")
--> a;b;c;d;e
```

Упражнение 20.2. Проблемой `table.sort` является то, что эта сортировка не является устойчивой (stable sort), то есть элементы, которые сортирующая функция считает равными, могут поменять свой порядок в процессе сортировки. Как можно реализовать устойчивую сортировку в Lua?

Упражнение 20.3. Напишите функцию для проверки того, является ли заданная таблица допустимой последовательностью.



ГЛАВА 21

Библиотека для работы со строками

Непосредственные возможности работы со строками интерпретатора Lua довольно ограничены. Программа может создавать строки, соединять их и получать длину строки. Но она не может извлекать подстроки или исследовать их содержимое. Подлинная мощь для работы со строками идет из ее библиотеки для работы со строками.

Библиотека для работы со строками доступна как модуль `string`. Начиная с Lua 5.1, функции также экспортируются как методы строк (используя метатаблицы). Так, перевод строки в заглавные буквы можно записать как `string.upper(s)` или `s:upper()`. Выбирайте сами.

21.1. Основные функции для работы со строками

Некоторые функции для работы со строками в библиотеке крайне просты: вызов `string.len(s)` возвращает длину строки `s`. Она эквивалентна `#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 строки с заглавными буквами, замененными в строчные; все остальные символы не меняются. (Функция `string.upper` переводит строчные буквы в заглавные.) В качестве примера, если вы хотите отсортировать строки вне зависимости от заглавных/строчных букв, вы можете использовать следующий фрагмент кода:

```
table.sort(a, function (a, b)
    return a:lower() < b:lower()
end)
```

Вызов `string.sub(s, i, j)` возвращает подстроку `s`, начиная с `i`-го символа и заканчивая `j`-м (включительно). В Lua первый символ строки имеет индекс 1. Вы можете также использовать негативные индексы, которые отсчитываются от 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 начало строки длиной в `j`; `string.sub(s, j, -1)` (или просто `s:sub(j)`, поскольку значением по умолчанию для последнего аргумента является `-1`) возвращает конец строки, начиная с `j`-го символа; и `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 значения строки, а возвращает новую строку. Типичной ошибкой является использовать что-то вроде `s:sub(2, -2)` и ожидать, что это изменит значение строки `s`. Если вы хотите изменить значение переменной, то вы должны присвоить ей новое значение:

```
s = s:sub(2, -2)
```

Функции `string.char` и `string.byte` переводят между символами и их внутренними числовыми представлениями. Функция `string.char` берет на вход целые числа, преобразует каждое из них в символ и возвращает строку, построенную из всех этих символов. Вызов `string.byte(s, i)` возвращает внутреннее числовое представление `i`-го символа строки `s`; второй аргумент необязателен, вызов `string.byte(s)` возвращает внутреннее числовое представление первого 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 мы использовали отрицательный индекс для обращения к последнему символу строки.

Начиная с Lua 5.1 функция `string.byte` поддерживает третий, необязательный аргумент. Вызов `string.byte(s, i, j)` возвращает

численные представления сразу всех символов, находящихся между индексами *i* и *j* (включительно):

```
print(string.byte("abc", 1, 2)) --> 97 98
```

Значением по умолчанию для *j* является *i*, поэтому вызов без третьего аргумента возвращает *i*-й символ. Вызов `{s:byte(1,-1)}` создает таблицу с кодами всех символов строки *s*. По этой таблице мы можем получить исходную строку при помощи 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` является мощным инструментом для форматирования строк, обычно для вывода. Она возвращает отформатированную версию от своих аргументов (поддерживается произвольное число аргументов), используя описание, заданное своим первым аргументом, так называемой *строкой формата*. Для этой строки существуют правила, похожие на правила для функции `printf` из стандартной библиотеки языка C: она состоит из обычного текста и *указателей*, которые управляют, где и как поместить каждый аргумент в результирующей строке. Указатель состоит из символа '%', за которым следует символ, задающий, как отформатировать аргумент: 'd' для десятичных чисел, 'x' для шестнадцатеричных чисел, 'o' для восьмеричных, 'f' для чисел с плавающей точкой, 's' для строк, также есть еще некоторые другие варианты. Между '%' и символом могут находиться другие опции, задающие форматирование, такие как число 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` задает число с плавающей точкой с четырьмя цифрами после десятичной точки. Во втором примере `%02d` обозначает десятичное число как минимум из двух цифр, при необходимости дополненное нулями; `%2d` без нуля будет дополнять число пробелами. За полным описанием этих опций обратитесь к справочному руководству по Lua или обратитесь к руководству по языку C, так как Lua использует библиотеку языка C для выполнения всей тяжелой работы здесь.

21.2. Функции для работы с шаблонами

Наиболее мощными функциями в библиотеке для работы со строками являются функции `find`, `match` и `gsub` (глобальная подстановка) и `gmatch` (глобальный поиск). Они все основаны на *шаблонах*.

В отличие от ряда других скриптовых языков, Lua не использует для работы с шаблонами ни синтаксис POSIX, ни синтаксис из языка Perl. Основной причиной для этого решения является размер: типичная реализация регулярных выражений POSIX занимает более 4000 строк кода. Это больше размера всех стандартных библиотек Lua, взятых вместе. Для сравнения реализация работы с шаблонами в Lua занимает менее 600 строк. Конечно, реализация работы с шаблонами в Lua уступает полноценной реализации POSIX. Тем не менее работа с шаблонами в Lua является мощным инструментом и включает в себя некоторые возможности, которые трудно соотнести со стандартными реализациями POSIX.

Функция *string.find*

Функция `string.find` ищет заданный шаблон внутри строки. Простейшим случаем шаблона является слово, которое соответствует своей копии. Например, шаблон `'hello'` будет искать подстроку `"hello"` внутри всей заданной строки. При нахождении шаблона `find` возвращает два значения: индекс, начиная с которого начинается совпадение, и индекс, где совпадение заканчивается. Если совпадение не найдено, то 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 поиск шаблона завершился успешно, то мы можем вызвать `string.sub` с возвращенными значениями, для того чтобы получить часть исходной строки, удовлетворяющей шаблону. Для простых шаблонов такой строкой будет сам шаблон.

У функции `string.find` есть необязательный третий параметр: индекс, задающий, с какого места внутри строки следует начать поиск.

Этот параметр оказывается полезным, когда мы хотим получить все вхождения шаблона: в этом случае мы вызываем функцию поиска неоднократно, каждый раз начиная поиск после позиции, в которой было найдено предыдущее совпадение. В качестве примера следующий код строит таблицу с позициями всех символов `'\n'` внутри строки:

```
local t = {} -- таблица для хранения индексов
local i = 0
while true do
    i = string.find(s, "\n", i+1) -- ищем следующее вхождение
    if i == nil then break end
    t[#t + 1] = i
end
```

Позже мы увидим более простой способ записи подобных циклов, используя итератор `string.gmatch`.

Функция *string.match*

Функция `string.match` похожа на `string.find` в том смысле, что она также ищет вхождения шаблона в строке. Однако, вместо того чтобы возвращать позиции, где был найден шаблон, она возвращает часть строки, удовлетворяющую шаблону:

```
print(string.match("hello world", "hello")) --> hello
```

Для простых шаблонов вроде `'hello'` эта функция не имеет смысла. Она показывает свою мощь, когда используется со сложными шаблонами, 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 как значение шаблона `'%d+/%d+/%d+'`, так и более сложное использование `string.match`.

Функция *string.gsub*

Функция `string.gsub` имеет три обязательных параметра: строку, шаблон и строку для замены. Она используется для замены всех вхождений шаблона в исходную строку на заданную строку:

```
s = string.gsub("Lua is cute", "cute", "great")
print(s) --> Lua is great
s = string.gsub("all llii", "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, простым способом посчитать число пробелов в строке является

```
count = select(2, string.gsub(str, " ", " "))
```

Функция *string.gmatch*

Функция `string.gmatch` возвращает функцию, которая перебирает все вхождения шаблона 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, шаблон `'%a+'` соответствует вхождению одного или большего числа букв (то есть слов). Поэтому цикл обойдет все слова внутри строки, запоминая их в таблице `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 найдено
end
```

Первым шагом будет замена всех точек на разделитель в пути, который считается равным `'\'`. (Как мы увидим далее, точка имеет специальное значение в шаблонах. Для сопоставления с точкой мы

должны записать '%.'). Далее функция перебирает все компоненты пути, где для каждой компоненты выполняется замена всех вопросительных знаков на имя модуля и проверяется, существует ли такой файл. Если да, то функция закрывает этот файл и возвращает его имя.

21.3. Шаблоны

Вы можете сделать шаблоны более полезными при помощи *классов символов*. Класс символов – это элемент в шаблоне, который может соответствовать любому символу из заданного множества. Например, класс %d соответствует любой цифре. Следовательно, можно искать дату в формате dd/mm/yyyy при помощи шаблона %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 пробела
%l	Строчные буквы
%p	Символы пунктуации
%s	Пробельные символы
%u	Строчные буквы
%w	Буквы и цифры
%x	Шестнадцатеричные цифры

Если в качестве имени класса использовать соответствующую заглавную букву, то она соответствует дополнению класса (то есть всем символам, не входящим в класс). Например, '%A' соответствует всем небуквам:

```
print(string.gsub("hello, up-down!", "%A", "."))
--> hello..up.down. 4
```

(4 не являются частью результирующей строки. Это второе значение, возвращаемое gsub, полное число выполненных замен. Я буду

далее опускать это число в следующих примерах, печатающих результат вызова `gsub()`.)

Некоторые символы, называемые *магическими символами*, имеют специальное значение внутри шаблона. Магическими символами являются

() . % + - * ? [] ^ \$

Символ `'\%'` используется для вставки этих символов в шаблон. Так, `'\%.'` соответствует точке; `'%%'` соответствует самому символу `'\%'`. Вы можете использовать подобным образом `'\%'` не только с магическими символами, но и с любыми не алфавитно-цифровыми символами. Когда сомневаетесь, лучше используйте `'\%'`.

Для парсера Lua шаблоны — это просто обычные строки. Они подчиняются тем же правилам, что и остальные строки. Только функции для работы с шаблонами рассматривают их как шаблоны, и только эти функции используют специальное значение символа `'\%'`. Для помещения кавычек внутрь шаблона используются те же самые приемы, что используются для помещения кавычек внутри других строк.

Вы также можете создавать свои классы, группируя при этом различные классы и символы внутри квадратных скобок. Например, класс `'[%w_]'` соответствует алфавитно-цифровым символам и символу подчеркивания; класс `'[01]'` соответствует двоичным цифрам; класс `'[%[]]'` соответствует квадратным скобкам. Для того чтобы посчитать число гласных в тексте, вы можете использовать следующий код:

```
nvow = select(2, string.gsub(text, "[AEIOUaeiou]", ""))
```

В подобные классы вы можете также включать диапазоны символов, записывая первый и последний символы, разделенные знаком минус. Я редко этим пользуюсь, поскольку все наиболее часто используемые диапазоны уже определены; например, `'[0-9]'` — это то же самое, что и `'%d'`, а `'[0-9a-zA-F]'` — это то же самое, что и `'%x'`. Однако если вам понадобятся восьмеричные цифры, то вы можете использовать `'[0-7]'` вместо `'[01234567]'`, вы также можете получить дополнение любого класса, поставив в начале символ `'^'`: так, шаблон `'^[0-7]'` находит любой символ, который не является восьмеричной цифрой, а `'^[^n]'` соответствует любому символу, отличному от `'\n'`. Однако помните, что для встроенных классов гораздо проще использовать вариант с большой буквой: `'%S'` проще, чем `'[^%s]'`.

Шаблоны могут стать более полезными, если использовать модификаторы для задания числа повторений и необязательных частей. Шаблоны в Lua предлагают четыре таких модификатора:

+	1 или более повторений
*	0 или более повторений
-	0 или более коротких повторений
?	Необязательно (0 или 1 раз)

Модификатор '+' соответствует одному или большему числу символов заданного класса. Он всегда вернет самое длинное вхождение шаблона. Например, шаблон '%a+' обозначает одну или более букв, то есть слово:

```
print(string.gsub("one, and two; and three", "%a+", "word"))  
--> word, word word; word word
```

Шаблон '%d+' соответствует одной или большему числу цифр (целому числу без знака):

```
print(string.match("the number 1298 is even", "%d+")) --> 1298
```

Модификатор '*' похож на '+', но он также допускает нулевое число вхождений символов из данного класса. Часто используется для обозначения необязательных пробелов между частями шаблона. Например, для шаблона, соответствующего паре скобок (возможно, с пробелами между ними), можно использовать следующий шаблон: '%(s*)': шаблон '%s*' соответствует нулю или большему числу пробелов между скобками. (Скобки также имеют специальное значение в шаблонах, поэтому мы их задаем, используя символ '%'.) В качестве другого примера шаблон '['_a]['_w]*' соответствует идентификаторам внутри программы на Lua: начинается с пробела или подчеркивания, за которым идет ноль или большее количество подчеркиваний и алфавитно-цифровых символов.

Подобно '*', модификатор '-' также соответствует нулю или большему количеству символов заданного класса. Однако вместо соответствия самой длинной последовательности он соответствует самой короткой последовательности. Иногда между ними нет никакой разницы, но обычно они дают разные результаты. Например, если вы попытаетесь найти идентификатор при помощи шаблона '['_a]['_a]-', то вы получите только первый символ идентификатора, поскольку '['_a]-' соответствует пустой последовательности. С другой стороны, допустим, вы хотите найти комментарии в програм-

ме на С. Большинство попытается использовать шаблон ``/%*. %*/'` (то есть ``/*'`, за которым следует любая последовательность символов, за которой следует `*/`). Однако поскольку `.*'` будет пытаться соответствовать наибольшему количеству символов, то первый ``/*'` закроется только самым последним `*/` в программе:

```
test = "int x; /* x */ int y; /* y */"
print(string.match(test, "/%*. %*/"))
--> /* x */ int y; /* y */
```

Шаблон ``. -'` захватит наименьшее количество символов, необходимое для первого `*/`, и даст, таким образом, желаемый результат:

```
test = "int x; /* x */ int y; /* y */"
print(string.gsub(test, "/%*.- %*/", ""))
--> int x; int y;
```

Последний модификатор ``?'` соответствует необязательно присутствующему символу. Например, пусть мы хотим найти число в тексте, которое может содержать необязательный знак. Шаблон ``[+]?%d+'` успешно справляется с работой, находя такие последовательности, как `"-12"`, `"23"` и `"+1009"`. Класс ``[+]'` соответствует либо символу `+`, либо символу `-`; следующий за ним знак ``?'` делает этот символ необязательным.

В отличие от других систем, в Lua модификатор может быть применен только к классу символов; нельзя группировать шаблоны под одним знаком модификатора. Например, нет шаблона, соответствующего необязательному слову (если только оно не состоит из одного символа). Обычно это ограничение можно обойти при помощи продвинутых приемов, которые мы увидим в конце этой главы.

Если шаблон начинается с символа `^^`, то он будет сопоставляться только с началом строки. Аналогично если шаблон заканчивается символом ``$'`, то он будет сопоставляться только с концом строки. Вы можете использовать оба этих символа для создания шаблонов. Например, следующий тест проверяет, начинается ли строка с цифры:

```
if string.find(s, "^%d") then ...
```

Следующий тест проверяет, что строка является числом, без других символов в начале или конце:

```
if string.find(s, "[+]?%d$") then ...
```

Символы `^^` и ``$'` обладают этим смыслом, только когда встречаются соответственно в начале или в конце строки. Иначе они выступают как обычные символы.

Другим элементом в шаблоне является `'%b'`. Мы записываем его как `'%bxу'`, где x и y — это два различных символа; символ x выступает как открывающий символ, а y — как закрывающий. Например, шаблон `'%b ()'` соответствует части строки, которая начинается с `' ('` и заканчивается `')`:

```
s = "a (enclosed (in) parentheses) line"
print(string.gsub(s, "%b()", "")) --> a line
```

Обычно мы используем этот шаблон в виде `'%b()'`, `'%b[]'`, `'%b{}'` или `'%b<>'`, но вы можете использовать в качестве разделителей любые символы.

Наконец, элемент `'%f[char-set']'` является *паттерном границы*. Он определяет место, в котором следующий символ содержится в классе *char-set*, а предыдущий — нет:

```
s = "the anthem is the theme"
print(s.gsub("%f[%w]the%f[%W]", "one"))
--> one anthem is one theme
```

Шаблон `'%f[%w]'` соответствует границе между неалфавитно-цифровым и алфавитно-цифровым символами, и шаблон `'%f[%W]'` соответствует границе между алфавитно-цифровым символом и неалфавитно-цифровым символом. Поэтому заданный шаблон соответствует строке `"the"` как целому слову. Обратите внимание, что множество символов мы должны записать внутри квадратных скобок, даже когда это всего один класс.

Положения перед первым и после последнего трактуются как содержащие символ с кодом 0. В предыдущем примере первое `"the"` начинается с границы между нулевым символом (не в классе `'[%w]'`) и `'t'` (в классе `'[%w]'`).

Шаблон границы был реализован в Lua 5.1, но не документирован. Официальным он стал только в Lua 5.2.

21.4. Захваты

Механизм *захвата* позволяет шаблону запомнить части строки, удовлетворяющие частям шаблона для последующего использования. Вы можете указать захват, записав части шаблона, которые вы хотите захватить, внутри круглых скобок.

Когда в шаблоне 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
```

Шаблон `'%a+'` задает непустую последовательность букв; шаблон `'%s*'` задает, возможно, пустую, последовательность пробелов. Поэтому в примере выше весь шаблон задает последовательность букв, за которой следует последовательность пробелов, за которой следует знак равенства `'='`, за которой опять следует последовательность пробелов, за которой следует другая последовательность букв. У обеих последовательностей букв их соответствующие шаблоны заключены в круглые скобки, поэтому они будут захвачены при соответствии. Ниже приводится 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 шаблона элемент `'%d'`, где *d* — это цифра, соответствует копии *d*-ой захваченной строки. В качестве примера рассмотрим случай, когда вы хотите внутри строки найти подстроку, заключенную в обычные или двойные кавычки. Вы можете попробовать шаблон `'[\"'].-[\"']'`, 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 этой проблемы мы можем 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)               --> "
```

Первое захваченное значение — это сам символ кавычки, и второе захваченное значение — это подстрока между кавычками (подстрока, удовлетворяющая `'.-'`).

В качестве другого похожего примера мы можем взять шаблон, соответствующий длинным строкам в Lua:

```
%[(=*)%[(.-)%]%1%]
```

Он соответствует открывающей квадратной скобке, за которой следует ноль или большее число знаков равенства, за которой следует другая открывающая квадратная скобка, за которой следует что угодно (сама строка), за которой следует закрывающая квадратная скобка, за которой следует то же самое количество знаков равенства, за которой следует еще одна закрывающая квадратная скобка:

```
p = "%[(=*)%[(.)]%1%"
s = "a = [[[[ something ]] ]==] ]=]; print(a)"
print(string.match(s, p)) --> = [[ something ]] ]==]
```

Первый захват — это последовательность знаков равенства (в примере только один знак); второе захваченное значение — это сама строка.

Также захваченные значения могут использоваться в заменяющей строке в `gsub`. Как и шаблон, заменяющая строка может содержать элементы `'%d'`, которые заменяются на соответствующие захваченные значения при выполнении подстановки. В частности, элемент `'%0'` соответствует всей части строки, удовлетворяющей шаблону. (Обратите внимание, что символ `'%'` в строке замены должен записываться как `'%%'`.) Еще один пример:

```
print(string.gsub("hello Lua!", "%a", "%0-%0"))
--> h-he-el-ll-lo-o L-Lu-ua-a!
```

Следующий пример переставляет соседние символы:

```
print(string.gsub("hello Lua", "(.)(.)", "%2%1")) --> ehll ouLa
```

В качестве более полезного примера давайте напишем простой преобразователь формата, который получает на вход строку с командами в стиле LaTeX и переводит их в XML-формат:

```
\command{some text} --> <command>some text</command>
```

Если мы запретим вложенные команды, то следующий вызов `string.gsub` выполняет работу:

```
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.
```

(Далее мы увидим, как обрабатывать вложенные команды.)

Другим полезным примером является удаление пробелов из начала и конца строки:

```
function trim (s)
    return (string.gsub(s, "^%s*(.)%s*$", "%1"))
end
```

Обратите внимание на аккуратное использование форматов. Два якоря (`'^'` и `'$'`) гарантируют, что мы получим всю строку. Поскольку `'\.'` старается выбрать самую короткую строку, то два шаблона `'%s*'` захватывают все пробелы по краям. Также обратите внимание, что поскольку `gsub` возвращает два значения, то мы используем круглые скобки для отбрасывания лишнего (числа замен).

21.5. Замены

Вместо строки в качестве третьего аргумента `string.gsub` мы можем использовать функцию или таблицу. При использовании функции `string.gsub` вызывает функцию каждый раз, когда находит удовлетворяющую шаблону подстроку; аргументами каждого вызова являются захваченные значения, и возвращенное функцией значение используется в качестве заменяющей строки. Когда третьим аргументом является таблица, функция `string.gsub` обращается в таблицу, используя первое захваченное значение как ключ и полученное значение из таблицы как заменяющую строку. Если полученное от функции или таблицы значение – это *nil*, то для данного совпадения замены не производится.

В качестве первого примера рассмотрим выполнение простой подстановки – каждое вхождение `$varName` заменяется 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?
```

Для каждого совпадения с шаблоном `'$(%w+)'` (знак доллара, за которым следует имя переменной) функция `gsub` ищет соответствующую переменную в `_G`, найденное значение заменяет вхождение шаблона в строку. Когда в таблице нет соответствующей переменной, то замены не производится:

```
print(expand("$othername is $status, isn't it?"))
--> $othername is great, isn't it?
```

Если вы не уверены в том, что соответствующие переменные имеют строковые значения, то вы можете попробовать применить `tostring` к этим значениям. В этом случае в качестве заменяющего 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 для каждого совпадения с шаблоном `'$ (%w+)'` `gsub` вызывает заданную функцию, передавая имя как аргумент; функция возвращает значение для замены.

В последнем примере мы возвращаемся к преобразованию формата команд. Мы опять хотим преобразовывать команды из стиля LaTeX'a (`\example{text}`) в стиль XML (`<example>text</example>`), но на этот раз мы будем обрабатывать вложенные команды. Следующая функция использует рекурсию для решения нашей задачи:

```
function toxml (s)
  s = string.gsub(s, "\\((%a+)(%b{1})\"", function (tag, body)
    body = string.sub(body, 2, -2) -- убрать скобки
    body = toxml(body) -- обработка вложенных команд
    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 будем использовать *кодирование URL*, которое применяется HTTP для передачи параметров в URL. Это кодирование заменяет специальные символы (такие как `'='`, `'&'` и `'+'`) на `'%xx'`, где `xx` – это шестнадцатеричный код символа. После этого он заменяет пробелы на `'+'`. Например, строка `"a+b = c"` будет закодирована как `"a%2Bb+%3D+c"`. Также имя каждого параметра и его значение со знаком равенства между ними добавляются к итоговой строке, переменные отделяются друг от друга символом `'&'`. 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 пусть мы хотим раскодировать такой URL и записать каждое значение в таблицу по своему имени. Следующая функция выполняет подобное 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 '+' на пробел. Второй находит закодированные шестнадцатеричным представлением символы и для каждого такого символа вызывает анонимную функцию. Эта функция переводит шестнадцатеричное представление 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 используем функцию `gmatch`. Поскольку и имя, и значение не могут содержать символов '&' и '=', то мы можем использовать шаблон '^&=+' :

```
cgi = {}
function decode (s)
  for name, value in string.gmatch(s, "([^&]=([^&=]+)") do
    name = unescape(name)
    value = unescape(value)
    cgi[name] = value
  end
end
```

Вызов функции `gmatch` находит пары вида `name=value`. Для каждой такой пары итератор возвращает захваченные значения (выделенные скобками в шаблоне) как значения поля `name` и `value`. Тело цикла просто вызывает `unescape` для обеих этих строк и записывает соответствующую 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 (для функции `format` используется опция "%02X", гарантирующая получение строки из двух цифр), и затем заменяет пробелы на символ '+' :

```
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, и строит получающуюся 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

```

Замена табов

Пустой захват `()` в Lua имеет специальное значение. Вместо того чтобы не захватывать ничего (что совершенно не нужно), этот шаблон захватывает текущее положение внутри строки как число:

```
print(string.match("hello", "()ll()")) --> 3 5
```

(Обратите внимание, что результат этого примера отличается от вызова `string.find`, поскольку положение второго захваченного значения идет после найденного шаблона.)

Красивым примером использования этой возможности является замена символов табуляции соответствующим числом пробелов:

```

function expandTabs (s, tab)
    tab = tab or 8 -- размер таба (по-умолчанию 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

```

Вызов `gsub` находит все символы табуляции внутри строки, захватывая их положение. Для каждого символа табуляции внутренняя функция использует это положение, для того чтобы вычислить количество пробелов, которое нужно, чтобы получить позицию, являющуюся кратной значению `tab`: она сначала вычитает один для перевода позиции, начиная с нуля, и затем добавляет `corr` для учета ранее встреченных табов (замена каждого символа табуляции влияет на положения последующих символов). Затем вычисляется поправка для следующего символа табуляции: минус один для удаляемого таба плюс `sp` для учета добавляемых пробелов. Наконец, она возвращает строку с соответствующим числом пробелов.

Для полноты давайте рассмотрим, как можно обратить эту операцию, заменяя пробелы символами табуляции. На первый взгляд также можно использовать пустые захваты для работы с положения-

ми внутри строки, но существует более простое решение: на каждом восьмом символе мы будем вставлять пометку внутрь строки. Затем, когда перед этой пометкой идут пробелы, мы будем заменять соответствующую последовательность 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 свою работу с замены всех имеющихся символов табуляции на пробелы. Затем она строит вспомогательный шаблон и использует его для добавления пометки (управляющего символа `\1`) после каждых `tab` символов. Далее все последовательности пробелов, за которыми следует пометка, заменяются на символ табуляции. Наконец, все пометки удаляются.

21.6. Хитрые приемы

Шаблоны – это очень мощный инструмент для работы со строками. Вы можете выполнить много сложных операций всего несколькими вызовами `string.gsub`. Однако, как и всякую другую силу, ее надо использовать аккуратно.

Использование шаблонов не заменяет парсер. Для быстрых решений (*quick-and-dirty*) вы можете использовать шаблоны для работы с исходным кодом, но получившиеся решения, скорее всего, не будут обладать высоким качеством. В качестве примера давайте рассмотрим шаблон, который мы использовали для поиска комментарием в программе на C: `"/%*.-%*/'`. Если у вас в программе есть строка, содержащая `"/*`, 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 содержанием встречаются довольно редко, и для ваших личных целей подобный шаблон, скорее всего, будет работать. Но вы не можете распространять программу с подобной ошибкой.

Обычно шаблоны работают в Lua довольно эффективно: моему старому компьютеру Pentium нужно всего 0,3 секунды, для того что-

бы найти все слова в тексте размером 4,4 Мб (850К слов). Но всегда лучше предпринять некоторые предосторожности. Всегда лучше делать шаблон как можно более точным; неточные шаблоны медленнее точных. Простым примером является использование `'(. -) %$'` для получения всей подстроки до первого вхождения знака доллара. Если в строке есть знак доллара, то все хорошо; но давайте допустим, что в строке вообще нет ни одного знака доллара. Тогда алгоритм сначала попытается получить подстроку, удовлетворяющую шаблону, начиная с первой позиции внутри строки. Дальше он будет двигаться вдоль всей строки в поисках знака доллара. Когда строка закончится, то мы получим несовпадения с шаблоном *только для первой позиции* внутри строки. Затем алгоритм выполнит то же самое, начиная уже со второй позиции внутри строки, и т. д. Таким образом, мы получим квадратичную сложность по времени, занимая более 4 минут на моем Pentium для строки из 100К символов. Вы можете легко исправить эту ситуацию, привязав шаблон к началу строки при помощи `^^ (. -) %$'`. При использовании такой привязки выполнение занимает всего одну сотую секунды.

Также будьте очень аккуратны с *пустыми шаблонами*, то есть шаблонами, которым удовлетворяет пустая строка. Например, если вы попробуете искать имена при помощи шаблона `'%a*'`, то вы везде будете находить имена:

```
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.

Никогда не следует писать шаблон, который начинается или заканчивается с `'-'`, поскольку ему будет удовлетворять пустая строка. Для этого модификатора обычно нужно что-то вокруг него, для того чтобы его ограничить. Аналогично шаблоны, включающие в себя `'.*'`, также довольно коварны, поскольку эта конструкция может захватить гораздо больше, чем вы планировали.

Иногда проще использовать сам Lua для построения шаблонов. Мы уже использовали этот прием в функции, преобразующей пробелы в символы табуляции. В качестве другого примера давайте рассмотрим, как мы можем найти строки более чем из 70 символов. Такая строка – это последовательность из 70 или большего числа символов, отличных от `'\n'`. Одиночный символ, отличающийся от `'\n'` принадлежит классу `'[^\\n]'`. Соответственно, мы можем получить

шаблон для длинной строки, повторив шаблон для символа 70 раз и добавив шаблон для нуля или большего числа следующих далее символов. Вместо того, чтобы явно написать этот шаблон, мы можем создать его при помощи `string.rep`:

```
pattern = string.rep("[^\\n]", 70) .. "[^\\n]*"
```

В качестве другого примера пусть вы хотите сделать поиск, нечувствительный к регистру букв. Для этого можно заменить каждую букву `x` в шаблоне на класс `[xX]`, то есть класс, включающий в себя и строчную, и заглавную версии буквы. Мы можем автоматизировать это преобразование при помощи следующей 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 просто нужно заменить каждое вхождение `s1` на `s2`, без учета всяких магических символов. Если обе строки явно заданы в тексте, то вы легко можете сами добавить все необходимые преобразования для магических символов, но если это переменные, то вам понадобятся дополнительные `gsub` для выполнения этой работы:

```
s1 = string.gsub(s1, "(%W)", "%%1")
s2 = string.gsub(s2, "%%", "%%%")
```

В строке, в которой мы ищем, мы заменяем все неалфавитно-цифровые символы, в строке замены мы заменяем только символ `%`.

Другим полезным приемом для работы с шаблонами является выполнение специальной обработки строки перед началом основной работы. Пусть мы хотим перевести в прописные все буквы, содержащиеся внутри двойных кавычек, но при этом внутри самой строки могут быть `\"`:

```
follows a typical string: "This is \"great\"!".
```

Одним из подходов для подобных случаев является кодирование входной строки. Например, давайте заменим `\"` на `\1`. Однако если в исходном тексте уже содержался символ `\1`, то у нас проблема. Простым способом выполнить кодирование и избежать этой проблемы является замена всех последовательностей `\"` на `\\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"` могла прийти только из нашего кодирования, поскольку любое `"\\ddd"` в исходной строке также было бы закодировано. 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 больше не содержит `"\\\""`, то мы можем смело использовать шаблон `"\".-%\""`:

```
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 записав это короче:

```
print(decode(string.gsub(code(s), "\".-%\"", string.upper)))
```

21.7. Юникод

На данный момент библиотека для работы со строками не содержит явной поддержки юникода. Однако несложно реализовать некоторые простые задачи по работе с юникод-строками, закодированные в UTF-8 без использования дополнительных библиотек.

Основной кодировкой для юникода в Web является UTF-8. Из-за ее совместимости с ASCII эта кодировка также очень хорошо подходит для Lua. Эта совместимость обеспечивает то, что ряд приемов по работе со строками без каких-либо изменений будет работать с UTF-8.

UTF-8 представляет каждый символ юникода различным числом байт. Например, символ `'А'` он представляет одним байтом, 65; символ Алеф, имеющий в юникоде код 1488, представлен двухбайтовой последовательностью 215-144. UTF-8 представляет все символы из ASCII как ASCII, то есть одним байтом со значением, меньшим 128.

Все остальные символы представляются последовательностями байт, где первый байт лежит в диапазоне [194, 244] и идущие далее байты лежат в диапазоне [128, 191]. Точнее, диапазон первого байта для двухбайтовых последовательностей это [194, 223], для трехбайтовых последовательностей [224, 239] и для четырехбайтовых последовательностей [240, 244]. Подобная схема гарантирует, что последовательность для любого символа никогда не встретится внутри последовательности для другого символа. Например, байт, меньший, чем 128, никогда не встретится в многобайтовой последовательности; он всегда представлен своим ASCII-символом.

В Lua вы можете читать, записывать и хранить строки в UTF-8 как обычные строки. Строчные константы (литералы) также могут содержать внутри себя UTF-8. (Конечно, вы, скорее всего, захотите редактировать ваш файл как файл в UTF-8.) Операция конкатенации выполняется корректно 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 функций операционной системы и библиотека для ввода-вывода являются на самом деле просто интерфейсами к операционной системе, поэтому их поддержка UTF-8 зависит от поддержки UTF-8 в самой системе. 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 дополнительные библиотеки или модификация стандартных библиотек Lua.

Давайте посмотрим, как функции из библиотеки для работы со строками 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 каждая из этих функций считает, что один символ – это один байт.

Функции `string.format` и `string.rep` без всяких проблем работают со строками в UTF-8, за исключением опции `'%c'`, которая подразумевает, что один символ – это один байт. Функции `string.len` и `string.sub` корректно работают со строками в UTF-8, но при этом индексы уже относятся не к символам, а к байтам. Довольно часто это именно то, что нужно. Но мы можем легко посчитать количество символов, как мы скоро увидим.

Для функций для работы с шаблонами их применимость зависит от шаблона. Простые шаблоны работают без всяких проблем, поскольку представление одного символа никогда не может встретиться внут-

при представления другого символа. Классы символов и множества символов работают только для ASCII-символов. Например, шаблон "%s" работает для строк в UTF-8, но он будет соответствовать только пробелам ASCII и не будет соответствовать пробелам в юникоде, таким как неразбиваемый пробел (U+00A0), разделителю параграфов (U+2029) или монгольскому Г+180E.

Некоторые шаблоны 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]", ""))
```

В этом примере gsub убирает второй, третий и четвертые байты, оставляя в результате по одному байту на каждый символ.

Аналогично 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 показывает некоторые приемы для работы с UTF-8 строками в Lua. Конечно, для выполнения этих примеров вам нужна платформа, где print поддерживает UTF-8.

К сожалению, больше Lua ничего предложить не может. Адекватная поддержка юникода требует огромных таблиц, которые плохо соотносятся с маленьким размером Lua. У юникода много особенностей. Практически невозможно абстрагировать какое-либо понятие из конкретных языков. Даже понятие того, что есть символ, весьма нечетко, поскольку нет взаимно-однозначного соответствия между закодированными в юникоде символами и графемами (то есть символами с диакритическими пометками и «полностью игнорируемыми» символами). Другие вроде бы базовые понятия, такие как что есть символ, также различаются для разных языков.

Чего, на мой взгляд, не хватает в Lua, так это функций для перевода между UTF-8 и юникодом и проверкой правильности строк в UTF-8. Возможно, они войдут в следующую версию Lua. Для остальных вещей лучшим вариантом будет использование внешней библиотеки вроде Slnunicode.

Листинг 21.1. Примеры работы с UTF-8 в Lua

```
local a = {}
a[#a + 1] = "Nähdään"
a[#a + 1] = "ação"
```

```

a[#a + 1] = "ÃØĖĖĐ"
local l = table.concat(a, ";")
print(l, #(string.gsub(l, "[\128-\191]", "")))
--> Nähdään;ação;ÃØĖĖĐ 18
for w in string.gmatch(l, "[^;]+") do
  print(w)
end
--> Nähdään
--> ação
--> ÃØĖĖĐ
for c in string.gmatch(a[3], "[\128-\191]*") do
  print(c)
end
--> Ã
--> Ø
--> Ė
--> Ė
--> Đ

```

Упражнения

Упражнение 21.1. Напишите функцию `split`, которая получает строку и шаблон-разделитель и возвращает последовательность блоков, разделенных разделителем:

```

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. Шаблоны `'%D'` и `'[^%d]'` эквивалентны. А что насчет шаблонов `'[^%d%u]'` и `'[%D%U]'`?

Упражнение 21.3. Напишите функцию для транслитерации. Эта функция получает строку и заменяет каждый символ в этой строке другим символом в соответствии с таблицей, заданной вторым аргументом. 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 реверсирует строку в UTF-8.

Упражнение 21.5. Напишите функцию транслитерации для UTF-8.



ГЛАВА 22

Библиотека ввода/вывода

Библиотека ввода/вывода предоставляет две различные модели для работы с файлами. Простая модель использует *текущий входной* и *текущий выходной* файлы, и все ее операции происходят над этими файлами. Полная модель использует явные указатели на файлы; она опирается на объектно-ориентированный подход, который определяет все операции как методы над указателями на файлы.

Простая модель удобная для простых вещей; мы использовали ее на протяжении всей книги. Но ее недостаточно для более гибкой работы с файлами, например для одновременного чтения или одновременной записи сразу в несколько файлов. Для этого нам нужна полная модель.

22.1. Простая модель ввода/вывода

Простая модель выполняет все свои операции над двумя текущими файлами. Библиотека при инициализации использует стандартный ввод (`stdin`) как входной файл по умолчанию и стандартный вывод (`stdout`) как выходной файл по умолчанию. Таким образом, когда мы выполняем что-то вроде `io.read()`, что мы читаем из стандартного ввода.

Мы можем изменить эти текущие файлы при помощи функций `io.input` и `io.output`. Вызов `io.input(filename)` открывает заданный файл для чтения и устанавливает его в качестве входного файла по умолчанию. Начиная с этого момента, весь ввод будет идти из этого файла до следующего вызова `io.input`; `io.output` работает аналогично, но уже для вывода. В случае ошибки обе функции вызывают ошибку. Если вы хотите явно обрабатывать ошибки, то вам нужна полная модель.

Функция `write` проще, чем `read`, поэтому мы сперва рассмотрим ее. Функция `io.write` получает произвольное число строковых аргументов и записывает их в выходной файл по умолчанию. Она преобразует числа в строки, используя стандартные правила 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)` выполняет то же самое, используя при этом меньше ресурсов, так как он избегает операции конкатенации.

Используйте `print` для небольших программ или для отладки и `write` 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 символов вроде символов табуляции или перехода на следующую строку. Кроме того, функция `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:

<code>"*a"</code>	Читает весь файл
<code>"*l"</code>	Читает следующую строку (без символа перевода строки)
<code>"*L"</code>	Читает следующую строку (с символом перевода строки)
<code>"*n"</code>	Читает число
<code>num</code>	Читает строку, состоящую из не более чем <i>num</i> 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, то простым способом написания фильтров на Lua является прочесть весь файл в строку, выполнить обработку строки (обычно при помощи `gsub`) и затем записать строку на вывод:

```
t = io.read("*a")           -- прочесть весь файл
t = string.gsub(t, ...)     -- выполнить работу
io.write(t)                 -- записать файл
```

В качестве примера следующий фрагмент кода – это законченная программа для кодирования содержимого файла в MIME *quoted-printable*. Каждый не ASCII-байт кодируется как `=xx`, где `xx` – это шестнадцатеричное значение байта. Для целостности кодирования сам символ равенства 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)
```

Шаблон, используемый в `gsub`, находит все байты от 128 до 255, включая знак равенства.

Вызов `io.read("*l")` читает следующую строку из текущего входного файла без символа перевода строки (`'\n'`); вызов `io.read("*L")` аналогичен, но только он возвращает символ перевода строки (если он присутствовал). Когда мы достигаем конца файла, функция возвращает *nil* (так как больше нет строк). Шаблон `"*l"` для функции `read` является значением по умолчанию. Обычно я использую этот шаблон, только когда естественно обрабатывает файл строка за строкой; в противном случае я предпочитаю сразу прочесть весь файл при помощи `"*a"` или читать его блоками, как мы увидим позже.

В качестве простого примера использования этого шаблона 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 для того, чтобы перебирать весь файл, строка за строкой, лучше использовать итератор `io.lines`. Например, мы можем написать законченную программу для сортировки строк файла следующим образом:

```
local lines = {}
-- читаем строки в таблицу 'lines'
for line in io.lines() do lines[#lines + 1] = line end
-- сортируем
table.sort(lines)
-- записываем 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 много чисел из файла, то отсутствие промежуточных строк улучшает 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`, передав сразу несколько опций; для каждого аргумента функция вернет соответствующее значение. Пусть у вас есть файл, содержащий по три числа на каждую строку:

```
6.0 -3.23 15e12
4.3 234 1000001
...
```

Теперь вам нужно напечатать максимум для 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 стандартных шаблонов, вы можете вызвать `read`, передав в качестве аргумента число *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 следующая программа демонстрирует эффективный способ (для Lua, конечно) скопировать файл из `stdin` в `stdout`:

```
while true do
  local block = io.read(2^13) -- размер буфера 8K
  if not block then break end
  io.write(block)
end
```

Как отдельный случай `read(0)` работает как проверка конца файла: она возвращает пустую строку, если в файле есть символы, и *nil*, если достигнут конец файла.

22.2. Полная модель ввода/вывода

Для большего контроля за вводом/выводом вы можете использовать полную модель. Ключевым понятием в этой модели является *указатель файла* (file handle), который аналогичен `FILE *` в C: он представляет открытый файл с текущим положением.

Для того чтобы открыть файл, используется функция `io.open`, которая аналогична функции `fopen` в C. В качестве аргументов она принимает имя файла и строку, задающую режим. Эта строка может содержать `'r'` для чтения, `'w'` для записи (запись стирает предыдущее содержимое файла) или `'a'` для добавления к файлу, также она может содержать `'b'` для работы с двоичными файлами. Функция `open` возвращает новый указатель на файл. В случае ошибки `open` возвращает *nil*, а также сообщение об ошибке и код ошибки:

```
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 ошибки зависит от системы.

Типичным способом проверки ошибок является следующий:

```
local f = assert(io.open(filename, mode))
```

Если происходит ошибка, то сообщение об ошибке выступает вторым аргументом `assert`, которое печатает это сообщение.

После того как вы откроете файл, вы можете читать из него и писать в него при помощи методов `read/write`. Они аналогичны функциям `read/write`, но вы вызываете их как методы указателя на файл, используя двоеточие. Например, для того чтобы открыть файл и прочесть все из него, вы можете использовать следующий код:

```
local f = assert(io.open(filename, "r"))
local t = f:read("*a")
f:close()
```

Библиотека ввода/вывода предоставляет три предопределенных указателя на стандартные файлы в C: `io.stdin`, `io.stdout` и

`io.stderr`. Поэтому вы можете послать сообщение об ошибке прямо в соответствующий стандартный файл:

```
io.stderr:write(message)
```

Можно использовать полную модель вместе с простой моделью. Для того чтобы получить указатель на текущий входной файл, следует вызвать `io.input()` без аргументов. Для того чтобы задать указатель файла в качестве текущего входного файла, следует вызвать `io.input(handle)` (аналогичные вызовы работают и для `io.output`). Например, если вы хотите временно изменить текущий входной файл, то вы можете написать что-то вроде следующего:

```
local temp = io.input()           -- сохранить текущий файл
io.input("newinput")              -- открыть новый текущий файл
<обработать ввод>
io.input():close()                -- закрыть текущий файл
io.input(temp)                    -- восстановить предыдущий 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 указатель на файл. Если было передано имя файла, то `io.lines` откроет файл в режиме для чтения и закроет файл после достижения конца файла. Если был передан указатель на файл, 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, функция `io.lines` также принимает те же самые опции, что и `io.read` (после первого аргумента). В качестве примера следующий код копирует файл, 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 блоками (например, по 8К). Для того чтобы избежать 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 блока. Затем мы объединяем блок и полученный остаток. Таким образом блок 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 `"*L"` 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]) -- открыть входной файл
local cc, lc, wc = 0, 0, 0 -- счетчики
for lines, rest in io.lines(arg[1], BUFSIZE, "*L") do
    if rest then lines = lines .. rest end
    cc = cc + #lines
-- считаем слова в блоке
    local _, t = string.gsub(lines, "%S+", "")
    wc = wc + t
-- считаем '\n'
    _, 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 открывают файл в текстовом режиме. В UNIX нет никакой разницы между бинарными и текстовыми файлами. Но в некоторых других системах, в частности в Windows, бинарные файлы нужно открывать со специальным флагом. Для работы с такими файлами используйте `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 библиоте-

ках могут обрабатывать любые байты. Вы даже можете использовать шаблоны для работы с бинарными данными до тех пор, пока шаблон не содержит нулевого байта. Если вам нужно отлавливать этот байт, то используйте для этого специальный класс `%z`.

Обычно бинарные данные читают либо при помощи шаблона `*a`, который читает весь файл, либо при помощи шаблона `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 использует стандартных файлов (`stdin-stdout`), поскольку они открыты в текстовом режиме. Вместо этого считается, 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 есть завершенная нулем последовательность шести и более допустимых символов, где символ является допустимым, если он удовлетворяет шаблону `validchars`. В нашем примере этот шаблон состоит из всех печатных символов. Мы используем `string.rep` и конкатенацию для создания шаблона, которому удовлетворяют последовательности из шести и более допустимых символов, за которыми следует нулевой байт. Круглые скобки в шаблоне используются для захвата самой строки (но не 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. Пример дампа, сделанного программой 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 ранее, первым аргументом программы является имя файла; вывод идет на стандартный вывод. Программа читает файл блоками по 16 байт. Для каждого блока выводится шестнадцатеричное представление каждого байта, и затем блок выводится как текст, заменяя управляющие символы точками.

Листинг 22.2 показывает результат применения этой программы самой к себе (на UNIX-машине).

22.3. Другие операции над файлами

Функция `tmpfile` возвращает указатель на временный файл, открытый для чтения/записи. Этот файл будет автоматически удален по завершении программы.

Функция `flush` сбрасывает все буферы для файлов. Подобно `write`, вы вызываете ее как функцию `io.flush()` для сброса буферов для текущего выходного файла или же как метод `f:flush()` для сброса буферов конкретного файла `f`.

Метод `setvbuf` устанавливает режим буферизации для файла. Его первым аргументом является строка: `"no"` означает никакой буферизации; `"full"` означает, что запись осуществляется, только когда буфер заполнен или вы явно сбрасываете буферы; `"line"` означает, что вывод буферизуется до поступления символа перевода строки. Для последних двух аргументов `setvbuf` допускает второй параметр, задающий размер буфера.

В большинстве систем стандартный файл для ошибок (`io.stderr`) не буферизуется, в то время как стандартный выходной файл (`io.stdout`) буферизуется в режиме строки. Поэтому если вы записываете незавершенные строки в стандартный вывод (например, индикатор прогресса операции), то для того, чтобы увидеть вывод, вам может понадобиться явный сброс буфера.

Метод `seek` может и возвращать, и задавать текущую позицию внутри файла. Его общей формой является `f:seek(whence, offset)`, где параметр `whence` – это строка, задающая, как надо интерпретировать параметр `offset`. Ее допустимыми значениями являются `"set"`, когда смещение трактуется от начала файла, `"cur"`, когда смещение трактуется от текущего положения, и `"end"`, когда смещение трактуется относительно конца файла. Независимо от значения параметра `whence` вызов возвращает значение текущего смещения относительно начала файла.

Значениями по умолчанию являются `"cur"` для `whence` и `0` для `offset`. Поэтому вызов `file:seek()` просто возвращает текущее положение внутри файла, не меняя его; вызов `file:seek("set")` переставляет указатель на начало файла и возвращает ноль; и вызов `file:seek("end")` переставляет указатель на конец файла и возвращает его длину. Следующая функция возвращает размер файла, не меняя текущую позицию внутри файла:

```
function fsize (file)
  local current = file:seek()      -- получить текущее положение
  local size = file:seek("end")    -- получить размер файла
  file:seek("set", current)        -- восстановить положение
  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 выводит его строки, отсортированные в алфавитном порядке. Если программа была вызвана без аргументов, то она должна брать данные из стандартного ввода и выводить данные в стандартный вывод. Если программа была вызвана с одним аргументом – именем файла, то она должна прочитать все данные из файла и записать вывод в стандартный вывод. При вызове с двумя аргументами она должна читать из первого файла и писать во второй файл.

Упражнение 22.2. Измените предыдущую программу так, чтобы она запрашивала подтверждение, если для вывода задано имя существующего файла.

Упражнение 22.3. Сравните быстродействие программы на Lua, которая копирует стандартный ввод на стандартный вывод для следующих случаев:

- Копирование осуществляется побайтно.
- Копирование осуществляется построчно.
- Копирование осуществляется блоками по 8К.
- Копируется сразу весь файл.

Для последнего варианта насколько большим может быть входной файл?

Упражнение 22.4. Напишите программу, которая печатает последнюю строку текстового файла. Постарайтесь избежать чтения всего файла, когда файл большой и по нему можно перемещаться.

Упражнение 22.5. Обобщите предыдущую программу так, чтобы она печатала последние n строк текстового файла. Опять постарайтесь избежать чтения всего файла, когда он большой.



ГЛАВА 23

Библиотека функций операционной системы

Библиотека функций операционной системы включает в себя функции для работы с файлами (не чтения и записи), получения текущих даты и времени и ряда других возможностей 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 ту функциональность, которая предоставляется ANSI C. Многие возможности операционной системы, такие как работа с каталогами и сокетами, не входят в этот стандарт, и библиотека их не предоставляет. Существуют другие библиотеки, не включенные в основную поставку, которые предоставляют расширенный доступ к операционной системе. Примерами таких библиотек являются `posix`, предоставляющая функциональность POSIX.1, `luasocket` для работы с сетью и `LuaFileSystem` для работы с каталогами и атрибутами файлов.

Все, что предоставляет 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 предоставляется 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 — число секунд с какого-то определенного момента.) Когда в качестве аргумента была передана таблица, то функция возвращает число, задаваемое этой таблицей. У подобных таблиц могут быть следующие поля:

year	Год
month	01–12
day	01–31
hour	00–23
min	00–59
sec	00–59
isdst	Логическое значение, равное <i>true</i> , если задействован перевод времени

Первые три поля из этой таблицы обязательны; если остальные не заданы, то значением по умолчанию является полдень (12:00:00). В UNIX-системе (где время отсчитывается, начиная с 00:00:00 1 января 1970 г.) в Рио де Жанейро мы получаем следующие значения:

```
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, выраженные в секундах, 54 000 – это 10 800 плюс 12 часов, выраженные в секундах.)

Функция `date`, несмотря на свое имя, является обратной функцией к `time`: она переводит число, представляющее дату и время, обратно к высокоуровневому представлению. Ее первый параметр это *строка формата*, описывающая, какое именно представление мы хотим. Второй параметр – это число, описывающее дату и время, если второй параметр не задан, то он считается равным текущему времени.

Для того чтобы получить дату как таблицу, мы используем формат `"*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 функцией `os.time`, в таблице, созданной `os.date`, также задаются день недели (`wday`, 1 – это воскресенье) и день в году (`yday`, 1 – это 1 января).

Для других значений строки формата `os.date` возвращает дату как копию строки формата, где определенные теги заменены информацией о дате и времени. Тег состоит из символа `'%`, 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 локали `Brazil-Portuguese` `%B` даст `"setembro"` и `%x` даст `"16/09/98"`.

Следующая таблица показывает каждый тег, объясняет его значение, и дает его значение для 16 сентября 1998 года, 23:48:10.

<code>%a</code>	Сокращенное название дня недели (например, <code>Wed</code>)
<code>%A</code>	Полное название дня недели (например, <code>Wednesday</code>)
<code>%b</code>	Сокращенное название месяца (например, <code>Sep</code>)
<code>%B</code>	Полное название месяца (например, <code>September</code>)
<code>%c</code>	Дата и время (например, <code>09/16/98 23:48:10</code>)
<code>%d</code>	День месяца (например, <code>16</code>) [01–31]
<code>%H</code>	Час в 24-часовой системе (например, <code>23</code>) [00–23]
<code>%I</code>	Час в 12-часовой системе (например, <code>11</code>) [00–11]
<code>%j</code>	День года (например, <code>259</code>) [001–366]
<code>%M</code>	Минуты (например, <code>48</code>) [00–59]
<code>%m</code>	Месяц (например, <code>09</code>) [01–12]
<code>%p</code>	"am" или "pm"
<code>%S</code>	Секунды (10) [00–60]
<code>%w</code>	День недели (3) [0–6=Воскресенье–суббота]
<code>%x</code>	Дата (например, <code>09/16/98</code>)
<code>%X</code>	Время (например, <code>23:48:10</code>)
<code>%y</code>	Год как две цифры (например, <code>98</code>)
<code>%Y</code>	Полный год (например, <code>1998</code>)
<code>%%</code>	Знак %

Если вы вызовете `date` вообще без 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 `yy/mm/yyuu`, то используйте явную строку формата вроде `"%m/%d/%Y"`.

Функция `os.clock` возвращает число секунд CPU, потраченных на выполнение программы. Обычно она используется для замера производительности:

```
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. Другие вызовы системы

Функция `os.exit` прерывает выполнение программы. Ее необязательный первый аргумент – это код, который вернет программа при завершении. Он может быть числом (ноль соответствует успешному завершению) или логическим значением (*true* означает успешное завершение). Если необязательный второй аргумент равен *true*, то закрывается состояние Lua; при этом вызываются финализаторы и освобождается память, занятая состоянием Lua. (Обычно эта финализация не является необходимой, так как большинство операционных систем освобождает ресурсы при завершении процесса.)

Функция `os.getenv` возвращает значение переменной окружения. Она берет на вход имя переменной и возвращает строку, содержащую ее значение:

```
print(os.getenv("HOME")) --> /home/lua
```

Для неопределенных переменных вызов возвращает *nil*.

Функция `os.execute` выполняет команду операционной системы; она эквивалентна функции `system` в языке C. На вход она получает строку с командой и возвращает информацию о том, как команда была завершена. Первое возвращаемое значение логическое, *true* означает успешное завершение без ошибок. Второе возвращаемое значение – это строка, "exit", если программа завершилась нормально, и "signal", если она была завершена по сигналу. 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 каталогов:

```
function createDir (dirname)
    os.execute("mkdir " .. dirname)
end
```

Функция `os.execute` – очень мощный инструмент, но она сильно зависит от используемой системы.

Функция `os.setlocale` задает текущую локаль, которая будет использоваться программой на Lua. Локаль определяет поведение, которое зависит от культурных и языковых различий. Функция `os.setlocale` получает на вход два строковых аргумента: имя локали и категорию, которая определяет, что именно будет затронуто этой локалью. У локалей есть шесть возможных категорий:

- “collate” управляет упорядочением строк;
- “ctype” управляет типами отдельных символов (то есть что именно является буквами) 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 (то есть функцией `os.date`);
- “all” управляет всеми описанными категориями.

По умолчанию категория – это “all”, то есть если вы вызвали `setlocale` только с именем локали, то это повлияет на все категории. Функция `setlocale` возвращает имя локали и ***nil*** в случае ошибки (обычно, так как система не поддерживает данную локаль).

```
print(os.setlocale("ISO-8859-1", "collate")) --> ISO-8859-1
```

Категория “numeric” несет в себе некоторые тонкости. Поскольку португальский и ряд других языков используют запятую вместо точки для представления десятичных чисел, то локаль меняет то, как Lua читает и выводит числа. Но локаль не меняет то, как Lua разбирает числа внутри программы. Если вы используете Lua для создания кода на Lua, то в следующем примере 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 `,'
```

Для того чтобы избегать подобных проблем, убедитесь, что Lua использует стандартную локаль C для создания фрагментов кода.

Упражнения

Упражнение 23.1. Напишите функцию, которая возвращает дату и время спустя ровно месяц от текущей даты (предполагая стандартное кодирование даты как числа).

Упражнение 23.2. Напишите функцию, которая получает дату и время, представленные как число, и возвращает число секунд, прошедших с начала дня.

Упражнение 23.3. Можете ли вы использовать `os.execute` для того, чтобы изменить текущий каталог вашей программы на Lua? Почему?



ГЛАВА 24

Отладочная библиотека

Отладочная библиотека не даст вам отладчик для Lua, но она предложит все те функции, которые необходимы для написания вашего собственного отладчика. Из соображения быстродействия официальный интерфейс к этим функциям – это C API. Отладочная библиотека в Lua – это способ получить доступ к этим функциям прямо из Lua.

В отличие от других библиотек, вы должны пользоваться данными функциями очень осторожно. Во-первых, часть этой функциональности не отличается хорошим быстродействием. Во-вторых, она нарушает некоторые правила языка, например то, что вы не можете обратиться к локальной переменной вне области ее действия. Довольно часто вам не будет хотеться использовать данную библиотеку в окончательной версии вашей программы.

Отладочная библиотека состоит из функций двух типов: *функции для доступа к объектам* (introspective functions) и *ловушки*. Функции для доступа к объектам позволяют изучать различные стороны выполняемой программы, такие как стек вызовов, текущая выполняемая строка, значения и имена локальных переменных. Ловушки позволяют нам отслеживать выполнение программы.

Важным понятием в отладочной библиотеке является *уровень в стеке*. Уровень в стеке – это число, которое относится к конкретной функции, активной в данный момент: у функции, вызвавшей отладочную библиотеку, уровень 1, функция, которая вызвала эту функцию, имеет уровень 2 и т. д.

24.1. Возможности по доступу (интроспекции)

Главной функцией для доступа (интроспекции) является `debug.getinfo`. Ее первый параметр может быть функцией или уровнем в стеке. Когда вы вызываете `debug.getinfo(foo)` для какой-то функ-

ции `foo`, то вы получите таблицу с данными об этой функции. Таблица может иметь следующие поля:

- `source`: где функция была определена. Если эта функция задана строкой (при помощи `loadstring`), то `source` равен этой строке. Если функция была определена в файле, то `source` – это имя файла, в начало которого вставлен символ `'@'`;
- `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 на Lua, `"C"`, если это функция на C, или `"main"`, если это главная часть блока;
- `name`: подходящее для функции имя;
- `namewhat`: что обозначает предыдущее поле. Возможные значения: `"global"`, `"local"`, `"method"`, `"field"` и `""` (пустая строка). Пустая строка означает, что Lua не нашел имени для функции;
- `nups`: количество `upvalue` для этой функции;
- `activelines`: таблица, представляющая множество активных строк функции. *Активная строка* – это строка с каким-то кодом, в отличие от пустых строк и строк, состоящих только из комментариев (типичное использование этой функции – это установка точек прерывания (`breakpoint`)). Большинство отладчиков не позволяет ставить точки прерывания не на активных строках);
- `func`: сама функция.

Когда `foo` – это функция на C, у Lua нет почти никаких данных о ней. В этом случае только поля `what`, `name` и `namewhat` содержат полезную информацию.

Когда вы вызываете `debug.getinfo(n)` для какого-то числа *n*, вы получаете данные о функции, находящейся на соответствующем месте в стеке вызовов. Например, если *n* равно 1, то вы получаете данные о текущей функции. (Когда *n* равно 0, вы получаете данные о самой функции `getinfo`.) Если *n* больше, чем число активных функций в стеке, то `debug.getinfo` возвращает *nil*. Когда вы запрашиваете информацию об активной функции, передавая в качестве аргумента число, возвращаемая таблица будет иметь одно дополнительное поле `currentline`, содержащее номер строки внутри функции в данный момент. А поле `func` содержит соответствующую функцию.

Поле `name` не так просто, как это кажется. Поскольку функции – это объекты в Lua, то функция может вообще не иметь имени или иметь несколько имен. Lua пытается найти имя функции, обратившись к коду, который ее вызвал. Этот метод работает, только если мы вызываем `getinfo` с числом в качестве аргумента, то есть просим информацию о конкретном вызове.

Функция `getinfo` не очень эффективна. Lua содержит отладочную информацию в форме, которая не оказывает отрицательного влияния на быстродействие программы; эффективное получение этой информации не является здесь главной целью. Для того чтобы получить большее быстродействие, у `getinfo` есть необязательный второй параметр, который сообщает, какую именно информацию нужно вернуть. Таким образом, функция не тратит лишнее время на получение ненужной информации. Этот параметр является строкой, где каждая буква соответствует группе полей согласно следующей таблице:

<code>'n'</code>	<code>name, namewhat</code>
<code>'f'</code>	<code>func</code>
<code>'S'</code>	<code>source, short_src, what, linedefined, lastlinedefined</code>
<code>'l'</code>	<code>currentline</code>
<code>'L'</code>	<code>activelines</code>
<code>'u'</code>	<code>nup</code>

Следующая функция иллюстрирует использование `debug.getinfo`. Она распечатывает текущий стек вызовов:

```
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      -- это функция на C?
      print(level, "C function")
    else                          -- функции на Lua
      print(string.format("[%s]:%d", info.short_src,
                           info.currentline))
    end
  end
end
```

Эту функцию легко можно улучшить, используя дополнительные данные из `getinfo`. В действительности в отладочной библиотеке уже есть такая функция – `traceback`. В отличие от нашей функции, `debug.traceback` не печатает стек вызовов, а возвращает (обычно длинную) строку со стеком вызовов.

Доступ к локальным переменным

Мы можем изучать локальные переменные любой активной функции при помощи функции `debug.getlocal`. У этой функции два параметра: уровень в стеке интересующей нас функции и номер переменной. Она возвращает два значения: имя переменной и ее значение. Если переданный номер переменной больше, чем число локальных переменных, то `getlocal` возвращает *nil*. Если переданный уровень в стеке не является допустимым, то функция вызывает ошибку. (Для проверки допустимости уровня в стеке мы можем использовать `debug.getinfo`.)

Lua нумерует локальные переменные в том порядке, в котором появляются в функции, считая только активные переменные в данной области видимости. 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), переменная с индексом 2 – это `b`, 3 – `x`, переменная с индексом 4 – это другая переменная `a`. В момент вызова `getlocal` переменная `c` уже вышла из своей области видимости, в то время как `name` и `value` еще не вошли в свою область видимости. (Вспомните, что локальная переменная видна только 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 имя переменной – это всегда `"(*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* в случае недопустимого индекса.

Доступ к нелокальным переменным

Отладочная библиотека также позволяет обращаться к нелокальным переменным, используемым в функции на Lua, при помощи функции `getupvalue`. В отличие от локальных переменных, нелокальные переменные, используемые в функции, существуют, даже если функция не активна (собственно, это именно о замыканиях). 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 изменять значения нелокальных переменных при помощи функции `debug.setupvalue`. Как вы можете ожидать, у нее три параметра: замыкание, имя переменной и новое значение. Как и `setlocal`, она возвращает имя переменной или *nil* в случае неверного индекса.

Листинг 24.1 показывает, как мы можем получить доступ к значению любой переменной по ее имени. Параметр `level` сообщает, к какой функции на стеке нам нужно обратиться, плюс один нужен для того, чтобы не включать саму функцию `getvarvalue`. Функция `getvarvalue` сначала пытается найти локальную переменную с заданным именем. Если таких переменных несколько, то она использует переменную с наибольшим индексом. Если найти локальную переменную с заданным именем не удастся, то делается попытка найти нелокальную переменную с этим именем. Для этого при помощи `debug.getinfo` получается вызывающее замыкание, и затем все его нелокальные переменные перебираются. Наконец, если не удастся найти нелокальную переменную с заданным именем, то ищется глобальная переменная с заданным именем: функция вызывает себя рекурсивно для доступа к надлежащей переменной `_ENV`, в которой ищется соответствующая переменная.

Листинг 24.1. Получение значения переменной по имени

```
function getvarvalue (name, level)
  local value
  local found = false
  level = (level or 1) + 1
```

```
-- пробуем 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
-- пробуем 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
-- пробуем получить значение из окружения
local env = getvarvalue("_ENV", level)
return env[name]
end
```

Доступ к другим сопрограммам

Все интроспективные функции из отладочной библиотеки принимают в качестве необязательного (и первого) аргумента сопрограмму, таким образом, мы можем изучать сопрограмму со стороны. 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` выполнится для сопрограммы `co`, давая в результате что-то вроде:

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

Этот стек не включает в себя вызов `resume`, поскольку сопрограмма и главная программа выполняются на разных стеках.

Когда сопрограмма вызывает ошибку, то ее стек не раскрывается. Это значит, что в случае ошибки мы можем изучать его. Продолжая наш пример, попробуем продолжить нашу сопрограмму после ошибки:

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

Теперь если мы распечатаем стек, то мы получим что-то вроде:

```
stack traceback:
[C]: in function 'error'
temp:4: in function <temp:1>
```

Мы также можем изучать локальные переменные сопрограммы даже после ошибки:

```
print(debug.getlocal(co, 1, 1)) --> x 10
```

24.2. Ловушки (hooks)

Механизм ловушек позволяет нам зарегистрировать функцию, которая будет вызвана при наступлении определенных событий во время выполнения программы. Существует четыре типа событий, которые могут вызвать ловушки:

- событие *вызова* происходит, когда Lua вызывает функцию;
- событие *возврата* происходит при возврате из функции;
- событие *строки* происходит, когда Lua начинает выполнение следующей строки;
- 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`.

Для того чтобы зарегистрировать ловушку, мы вызываем функцию `debug.sethook` с двумя или тремя аргументами: первый аргумент — это соответствующая функция; второй аргумент — это строка-маска, описывающая, какие именно события мы хотим отслеживать, и необязательный третий аргумент — это число, задающее с какой частотой мы хотим получить события счетчика. Для того чтобы отслеживать события вызова, возврата и строки, мы добавляем буквы 'c', 'r' и 'l' к строке-маске. Для отслеживания событий счетчика мы просто передаем счетчик как третий аргумент. Для убирания всех ловушек просто вызовите `sethook` без параметров.

В качестве примера следующий код устанавливает примитивную ловушку, которая для каждой очередной выполняемой строки печатает ее номер:

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

Этот вызов устанавливает `print` как функцию-ловушку и задает ее вызов только для событий строки. Более сложная функция-ловушка может использовать `getinfo` для того, чтобы добавить к выдаче имя текущего файла:

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

Полезной функцией для использования в ловушках является `debug.debug`. Эта простая функция печатает приглашение, читает с ввода и затем выполняет заданные команды. 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 в ответ на приглашение вводит `"cont"`, то эта функция завершается. Стандартная реализация очень простая и выполняет команды в глобальном окружении вне отлаживаемого кода. Упражнение 24.5 обсуждает более удачную реализацию.

24.3. Профилирование

Несмотря на свое имя, отладочная библиотека также полезна и для не только отладочных задач. Типичной подобной задачей является профилирование (получение информации о времени, затрачиваемом на выполнении того или иного фрагмента кода). Для профилирования с учетом времени лучше использовать С-интерфейс: цена вызова каждой Lua-ловушки довольно высока и может сильно исказить результаты. Однако для простого профилирования, считающего количество раз, код на Lua вполне подходит. В этом разделе мы напишем простейший профилировщик, который для каждой вызываемой функции сообщит, сколько раз она была вызвана за время выполнения программы.

Главной структурой данных в нашей программе будут две таблицы: одна сопоставляет функциям их счетчики, а вторая сопоставляет функциям их имена. В качестве индексов для обеих этих таблиц будут выступать сами функции.

```
local Counters = {}  
local Names = {}
```

Мы можем извлечь имена функций и после профилировки, но мы получим лучшие результаты, если будем получать имена функций, пока они активны, поскольку в этом случае Lua может посмотреть в поисках имени функции вызывающий ее код.

Теперь давайте определим функцию-ловушку. Ее задачей является получить вызванную функцию и увеличить соответствующий счетчик, также она собирает имена функций:

```
local function hook ()  
    local f = debug.getinfo(2, "f").func  
    local count = Counters[f]  
    if count == nil then -- функция 'f' вызвана первый раз?  
        Counters[f] = 1  
        Names[f] = debug.getinfo(2, "Sn")  
    else -- только увеличить значение счетчика  
        Counters[f] = count + 1  
    end  
end
```

Следующим шагом является запуск программы с этой ловушкой. Мы будем считать, что главный блок программы находится в файле и имя этого файла передается как аргумент программе-профилировщику:

```
% lua profiler main-prog
```

Тогда профилировщик может взять имя файла из `arg [1]`, установить ловушку и выполнить файл:

```
local f = assert(loadfile(arg[1]))  
debug.sethook(hook, "c") -- установить ловушку  
f() -- выполнить профилируемую программу  
debug.sethook() -- отключить ловушку
```

Последним шагом является собственно показ результатов. Функция `getname` из листинга 24.2 выдает для каждой функции соответствующее имя. Для того, чтобы избежать путаницы, к каждому имени добавляем место соответствующей функции в виде *файл: строка*. Если у функции нет имени, то мы печатаем только место. Если функ-

ция является функцией на С, то мы используем только ее имя (так как у нее нет места). С учетом этого ниже приводится код, печатающий информацию о вызовах:

```
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 из раздела 10.3, то мы получим что-то вроде:

```
[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
```

Это показывает, 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.

Этот профилировщик можно улучшить, например добавить сортировку вывода, улучшенную печать имени функции и т. п. Тем не менее даже этот профилировщик уже полезен и может быть использован как основа для написания более продвинутых инструментов.

Упражнения

Упражнение 24.1. Почему рекурсия в функции `getvarvalue` (листинг 24.1) обязательно остановится?

Упражнение 24.2. Измените функцию `getvarvalue` (листинг 24.1) для работы с различными сопрограммами (подобно другим функциям из отладочной библиотеки).

Упражнение 24.3. Напишите функцию `setvarvalue`.

Упражнение 24.4. На основе функции `getvarvalue` напишите функцию `getallvars`, которая возвращает таблицу со всеми переменными, которые видны в заданном месте (возвращаемая 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 в качестве метаметода `__index` функцию `getvarvalue`.)

Упражнение 24.6. Измените предыдущий пример для того, чтобы можно было менять переменные.

Упражнение 24.7. Реализуйте некоторые из предложенных улучшений для профилировщика из раздела 24.3.

Упражнение 24.8. Напишите библиотеку для работы с точками останова (`breakpoint`). Она должна предлагать как минимум две функции:

```
setbreakpoint(function, line) --> возвращает handle  
removebreakpoint(handle)
```

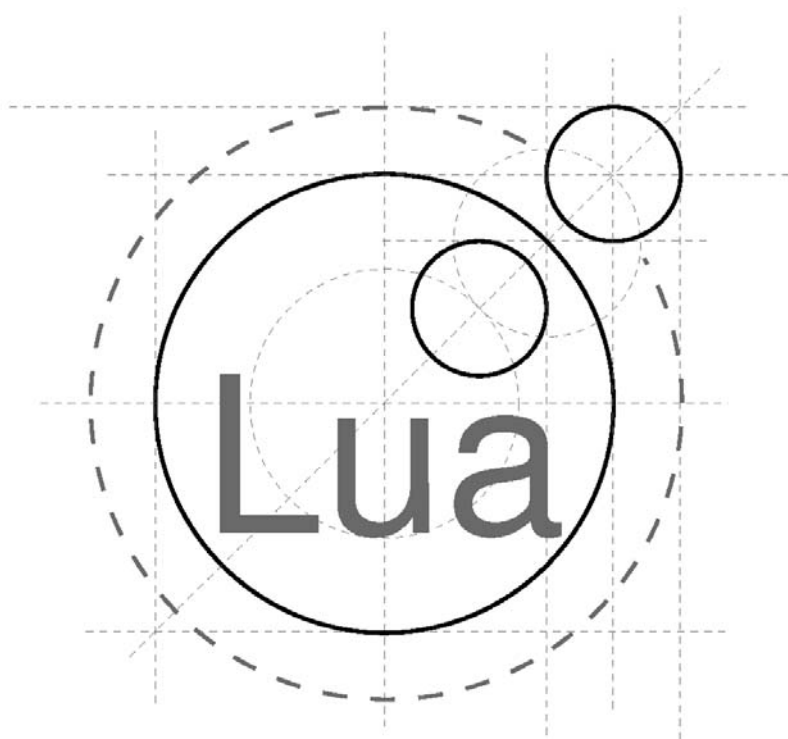
Точка останова задается функцией и строкой внутри функции. Когда выполнение доходит до точки останова, то следует вызывать `debug.debug`.

(Подсказка: для простейшей реализации используйте ловушку строки и функцию ловушки, которая проверяет, попали ли мы в точку останова; для улучшения быстродействия мы можем включать эту ловушку только когда мы находимся внутри интересующей нас функции.)



Часть IV

С API





ГЛАВА 25

Обзор C API

Lua – это *встраиваемый язык*. Это значит, что Lua – это не отдельный пакет, а библиотека, которую мы можем прилинковать к другим приложениям для добавления в них возможностей Lua.

Вы можете удивиться: если Lua – это не отдельная программа, но до сих пор в книге мы использовали Lua именно как отдельную программу. Решением этого вопроса является интерпретатор Lua (выполнимый файл `lua`). Данный интерпретатор – это маленькое приложение (насчитывающее меньше пятисот строк кода), которое использует библиотеку Lua для того, чтобы реализовать отдельный интерпретатор Lua. Эта программа занимается взаимодействием с пользователем, беря файлы и строки и передавая их библиотеке Lua, которая и выполняет основную работу (такую как запуск кода на Lua).

Эта способность использовать библиотеку для того, чтобы расширить возможности приложения, – это именно то, что делает Lua *расширяющим языком*. В то же самое время программа, которая использует Lua, может зарегистрировать новые функции в окружении Lua, добавляя тем самым возможности, которые не могли бы быть написаны на самой Lua. Это то, что делает Lua *расширяемым языком*.

Эти два взгляда на Lua (как на расширяющий язык и как на расширяемый язык) соответствуют двум типам взаимодействия между C и Lua. В первом случае C управляет, а Lua – это просто библиотека. Код на C, соответствующий этому типу взаимодействия, мы называем *кодом приложения* (application code). Во втором случае управление у Lua, а C – это библиотека. В этом случае код на C называется *библиотечным кодом*. Оба этих типа кода используют один и тот же API для взаимодействия с Lua, так называемый C API.

C API – это просто набор функций, которые позволяют коду на C взаимодействовать с Lua¹. Он включает в себя функции для чтения

¹ Далее термин функция на самом деле означает «функция» или «макрос». API реализует некоторые возможности как макросы.

и записи глобальных переменных Lua, для вызова функций на Lua, для выполнения фрагментов кода на Lua, для регистрации функций на C, так что они потом могут вызываться из кода на Lua и т. п. Практически все, что код на Lua может сделать, может быть сделано на C при помощи C API.

C API следует стилю языка C, который заметно отличается от стиля языка Lua. Когда мы программируем на C, то мы следим за типами данных, обработкой ошибок, ошибками выделения памяти и рядом других сложных мест. Большинство функций API не проверяет правильность своих аргументов; это ваша задача – убедиться в том, что аргументы корректны, перед вызовом функции². Если вы допустите ошибку, то, скорее всего, получите ошибку вроде «segmentation fault» или что-то вроде нее вместо красивого сообщения об ошибке. Более того, C API делает упор на гибкости и простоте, зачастую за счет легкости использования. Типичные задачи могут потребовать нескольких вызовов API. Это может быть утомительным, но зато дает вам полный контроль над происходящим.

В соответствии с названием целью этой главы является то что необходимо при использовании Lua из C. Не пытайтесь сейчас понять все детали происходящего. Позже мы на этом остановимся. Однако не забывайте, что вы всегда можете найти дополнительную информацию в справочном руководстве по Lua. Более того, вы можете найти некоторые примеры использования API в самой поставке Lua. Отдельный интерпретатор Lua (`lua.c`) дает примеры кода приложения, в то время как стандартные библиотеки (`lmathlib.c`, `lstrlib.c` и т. п.) дают примеры библиотечного кода.

С настоящего момента мы выступаем в роли программистов на C. Когда я говорю о «вас», то я имею в виду именно вас, программирующего на C.

Важной компонентой во взаимодействии между Lua и C является постоянно присутствующий виртуальный *стек*. Почти все функции API работают со значениями на этом стеке. Весь обмен данными между Lua и C происходит через этот стек. Более того, вы также можете использовать этот стек для хранения промежуточных результатов. Этот стек позволяет решить проблемы с принципиальным отличием между Lua и C: первое отличие заключается в том, что в Lua есть сборка мусора, в то время как C – явное управление памятью; второе

² Вы можете компилировать код с макросом `LUA_USE_APICHECK`, который включает некоторые проверки; эта опция особенно полезна при отладке кода на C. Тем не менее на C некоторые ошибки вроде недопустимых указателей просто не могут быть обнаружены.

отличие заключается в разнице 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 примера простого приложения: отдельного интерпретатора Lua. Мы можем написать примитивный интерпретатор Lua, как показано 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 на Lua (таких как `lua_pcall`), для чтения и записи глобальных переменных в окружении Lua, для регистрации новых функций, которые могут вызываться из Lua, и т. п. Все, что определено в файле `lua.h`, имеет префикс `_lua`.

Листинг 25.1. Простой отдельный 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); /* снять сообщение об ошибке со стека */
        }
    }
    lua_close(L);
    return 0;
}
```

Заголовочный файл `lauxlib.h` определяет функции, предоставленные *дополнительной библиотекой* (auxiliary library). Все определения из этого файла начинаются с `luaL_` (например, `luaL_loadstring`). Дополнительная библиотека использует базовый API, предоставленный `lua.h` для предоставления более высокого уровня абстракции,

в частности абстракций, используемых стандартными библиотеками. Базовый API стремится к экономичности и ортогональности, в то время как дополнительная библиотека стремится к практичности для распространенных задач. Конечно, это легко и для вашей программы может создавать необходимые абстракции. Имейте в виду, что у дополнительной библиотеки нет доступа к внутренностям Lua. Все, что она делает, она делает через стандартный API. Что делает она, может сделать 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 реентерабельной и готовой для использования в многонитевых приложениях.

Как и следует из ее имени, функция `luaL_newstate` создает новое состояние Lua. Когда `luaL_newstate` создает новое состояние, то не содержит никаких встроенных функций, даже `print`. Для того чтобы сохранить Lua маленькой, все стандартные библиотеки представлены как отдельные пакеты, поэтому вы не обязаны их использовать, если они вам не нужны. Заголовочный файл `lua.h` определяет функции для открытия библиотек. Функция `luaL_openlibs` открывает все стандартные библиотеки.

После создания состояния и заполнения его стандартными библиотеками пора начать выполнять ввод от пользователя. Для каждой строки, которую вводит пользователь, программа сначала вызывает `luaL_loadstring` для компиляции введенного кода. Если ошибок нет, то этот вызов возвращает ноль и помещает получившуюся функцию на стек. (Помните, что мы обсудим стек в деталях в следующем разделе.) После этого программа вызывает `lua_pcall`, которая забирает функцию со стека и выполняет ее в защищенном режиме. Как и `luaL_loadstring`, функция `lua_pcall` возвращает ноль, если нет ошибок. В случае ошибки обе функции помещают сообщение об ошибке на стек; мы получим это сообщение при помощи функции `lua_tostring`, и после того, как мы его напечатаем, мы удалим его со стека при помощи функции `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, часто зависит от типа вашего приложения. Ядро Lua само ничего не печатает ни в какой поток (файл); оно в случае ошибки просто возвращает сообщение об ошибке. Каждое приложе-

ние может обрабатывать эти сообщения наиболее подходящим для него способом. Для простоты мы будем использовать следующий обработчик ошибок, который в случае ошибки печатает сообщение об ошибке, закрывает состояние Lua и завершает работу 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 коде приложения.

Поскольку вы можете компилировать Lua и как код на C, и как код на C++, `lua.h` не включает в себя следующую стандартную поправку, часто встречающуюся в коде на C:

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

Если вы компилируете Lua как код на C (наиболее часто встречающийся случай) и используете его в C++, вы можете включать `lua.hrp` вместо `lua.h`. Он определен следующим образом:

```
extern "C" {
#include "lua.h"
}
```

25.2. Стек

При передаче значений 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 могут иметь самые разные типы, даже `a` может иметь другой тип (в связи с использованием

метаблиц). Однако если мы хотим предоставить эту операцию в С, то любая функция `settable` должна иметь фиксированный тип. Нам понадобятся десятки функций для этой простой операции (по одной функции на каждую комбинацию из типов трех аргументов).

Мы можем решить данную проблему, введя новый тип – `union` основных типов, назовем его `lua_Value`, который может представлять все значения в Lua. Тогда мы можем объявить `settable` следующим образом:

```
void lua_settable (lua_Value a, lua_Value k, lua_Value v);
```

Однако у этого решения есть два недостатка. Во-первых, может быть очень непросто отобразить сложный тип данных для других языков; мы разрабатывали Lua так, чтобы он легко взаимодействовал не только с C/C++, но также и с Java, Fortran, C# и др. Во-вторых, Lua осуществляет сборку мусора: если мы храним таблицу на Lua в переменной C, то сам Lua об этом никак знать не может и может (ошибочно) решить, что эта таблица больше не нужна, и удалить ее.

Поэтому Lua API не определяет ничего вроде типа `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. Когда мы хотим получить значение от Lua (например, значение глобальной переменной), мы вызываеме Lua и он помещает значение на стек. Когда вы хотите передать значение Lua, то вы сперва помещаете значение на стек и затем вызываете Lua (который снимет это значение со стека). Нам по-прежнему нужны различные функции для того, чтобы каждый тип C поместить на стек и каждый тип на C снять со стека, но у нас уже не возникает комбинаторного увеличения числа функций, как ранее. 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, функция `luaL_loadstring` оставляет свой результат на стеке (либо как откомпилированный блок, либо как сообщение об ошибке); `lua_pcall` берет вызываемую функцию со стека и оставляет любое сообщение об ошибке на стеке.

Lua работает со стеком строго в соответствии с принципом LIFO (Last In, First Out). Когда вы вызываете Lua, то она меняет только верхушку стека. У кода на C больше свободы; в частности, он может просматривать любой элемент на стеке, а также вставлять и удалять элементы с любого места в стеке.

Помещение элементов на стек

В API содержится по одной функции для помещения на стек каждого типа C, который может быть представлен в Lua: `lua_pushnil` для константы *nil*, `lua_pushboolean` для логических значений (целых чисел в C), `lua_pushnumber` (для `double`), `lua_pushinteger` для целых чисел, `lua_pushunsigned` для целых беззнаковых чисел, `lua_pushlstring` для произвольных строк (указатель на `char` и длина) и `lua_pushstring` для обычных ASCII-строк:

```
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 того, чтобы помещать на стек функции на C и объекты типа *userdata*, но мы их рассмотрим позже.

Тип `lua_Number` – это числовой тип в Lua. По умолчанию это тип `double`, но его можно изменить на `float` или даже `long int` для различных архитектур. Тип `lua_Integer` – это целочисленный тип со знаком, достаточно большой, чтобы хранить в себе размер больших строк. Обычно он определен как `ptrdiff_t`. Тип `lua_Unsigned` – это 32-битовый беззнаковый целый тип в C; используется библиотекой для побитовых операций и различными функциями.

Строки в Lua не завершаются нулевым байтом, они могут содержать произвольные бинарные данные. Соответственно, для них должна явно задаваться длина. Основной функцией для помещения строки на стек является `lua_pushlstring`, требующая явного задания длины строки. Для строк, завершенных нулевым байтом, вы можете использовать функцию `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 на C). Для любой строки, которую необходимо хранить, Lua или делает копию, или переиспользует уже существующую. Соответственно, вы можете освободить или изменить свой буфер, как только управление вернется из этих функций.

Когда вы помещаете элемент на стек, то это ваша обязанность – следить за тем, чтобы для него на стеке было достаточно места. Помните о том, что сейчас вы программист на C. Когда Lua начинает выполняться и в любой момент, когда Lua вызывает C, на стеке есть

как минимум 20 свободных слотов. (Заголовочный файл `lua.h` определяет эту константу как `LUA_MINSTACK`.) Обычно этого более чем достаточно, поэтому, как правило, об этом можно не думать. Однако для некоторых задач требуется больше места на стеке, в частности если вы в цикле помещаете элементы на стек. В этих случаях вы можете вызвать функцию `lua_checkstack`, которая проверяет, есть ли на стеке необходимое свободное место:

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

Обращение к элементам

Для обращения к элементам на стеке API использует *индексы*. Первый элемент, помещенный на стек, имеет индекс 1, следующий – индекс 2 и т. д. Мы также можем обращаться к элементам стека, используя вершину стека как отправную точку, в этом случае мы используем отрицательные индексы. В этом случае `-1` соответствует элементу на вершине стека (то есть последнему помещенному на стек элементу), `-2` соответствует предыдущему элементу и т. д. Например, вызов `lua_tostring(L, -1)` возвращает значение на вершине стека как строку. Как мы увидим далее, есть случаи, когда естественно обращаться к стеку, начиная с низа стека (то есть используя положительные индексы), и есть случаи, когда естественно использовать отрицательные индексы.

Для проверки того, является ли элемент значением заданного типа, API предлагает набор функций `lua_is*`, где `*` может быть любым типом в Lua. Соответственно, есть функции `lua_isnumber`, `lua_isstring`, `lua_istable` и т. д. Все эти функции имеют один и тот же общий прототип:

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

В действительности `lua_isnumber` не проверяет, является ли значение числом, а проверяет, может ли быть значение преобразовано к числу; `lua_isstring` ведет себя аналогично: в частности, любое число удовлетворяет `lua_isstring`.

Также существует функция `lua_type`, которая возвращает тип элемента на 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`. Мы обычно используем эту функцию совместно с оператором `switch`. Также она оказывается полезной, когда нам нужно

проверить, является ли значение числом или строкой без приведения типов.

Для получения значений со стека существуют 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 в логическое значение в C (0 или 1), используя при этом правила Lua для условных конструкций: *nil* и *false* дают 0, все остальные – значение 1.

Можно вызывать любую из `lua_to*` функций, даже когда значение имеет не тот тип. Функция `lua_toboolean` работает для значений любого типа; `lua_tolstring` возвращает NULL для нестроковых значений. У числовых функций нет никакого способа сообщить об ошибке, поэтому в случае ошибки они просто возвращают ноль. Обычно следует позвать `lua_isnumber` для проверки типа, но в Lua 5.2 ввели следующие 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);
```

Через параметр `isnum` возвращается булево значение, сообщающее о том, было ли соответствующее 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 запоминает длину строки через параметр `len`. Вы не можете менять эту внутреннюю копию (спецификатор `const` напоминает вам об этом). Lua гарантирует, что этот указатель валиден до тех пор, пока соответствующая строка находится в стеке. Когда функция на C, вызванная из Lua, возвращает управление, то Lua очищает стек; поэтому никогда не заводите указателей на строки Lua вне функции, получившей их.

Любая строка, которую возвращает `lua_tolstring`, всегда имеет нулевой байт в конце, но она также может содержать нулевые байты внутри себя. Настоящий размер строки возвращается через третий аргумент `len`. В частности, считая, что значение на вершине стека – это строка, то следующие `assert`'ы всегда верны:

```
size_t l;  
const char *s = lua_tolstring(L, -1, &l); /* any Lua string */  
assert(s[l] == '\\0');  
assert(strlen(s) <= l);
```

Вы можете вызвать `lua_tolstring` с третьим параметром, равным `NULL`, если вам не нужна длина строки. Или же вы можете использовать макрос `lua_tostring`, который на самом деле – это `lua_tolstring` с третьим параметром, равным `NULL`.

Для того, чтобы проиллюстрировать использование этих функций Листинг 25.2 показывает полезную вспомогательную функцию, которая печатает содержимое стека. Эта функция обходит весь стек снизу доверху, печатая каждый элемент с соответствии с его типом. Стоки печатаются в кавычках, для чисел используется формат `'%g'`, для других значений (функции, таблицы и т. п.) печатается только тип. (Функция `lua_typename` переводит числовое значение, идентифицирующее тип, в строку.)

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++) { /* повторить для каждого уровня */  
        int t = lua_type(L, i);  
        switch (t) {  
            case LUA_TSTRING: { /* strings */  
                printf("' %s'", lua_tostring(L, i));  
                break;  
            }  
            case LUA_TBOOLEAN: { /* booleans */  
                printf(lua_toboolean(L, i) ? "true" : "false");  
                break;  
            }  
            case LUA_TNUMBER: { /* numbers */  
                printf("%g", lua_tonumber(L, i));  
                break;  
            }  
            default: { /* other values */  
                printf("%s", lua_typename(L, t));  
                break;  
            }  
        }  
        printf(" "); /* put a separator */  
    }  
    printf("\n"); /* end the listing */  
}
```

Другие операции со стеком

Кроме предыдущих функций, служащих для обмена данными между C и Lua, API также предоставляет следующие функции для 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, что также равно индексу элемента на вершине стека. Функция `lua_settop` устанавливает количество элементов в стеке. Если предыдущее значение вершины стека было больше, то лишние значения выбрасываются. В противном случае в качестве недостающих значений используется *nil*. В частности, `lua_settop(L, 0)` очищает весь стек. В функции `lua_settop` вы также можете использовать отрицательные индексы. В частности, API предоставляет следующий макрос, который снимает со стека `n` элементов:

```
#define lua_pop(L,n) lua_settop(L, -(n) - 1)
```

Функция `lua_pushvalue` помещает на стек копию элемента с заданным индексом; функция `lua_remove` удаляет элемент с заданным индексом, сдвигая при этом все остальные элементы; `lua_insert` перемещает элемент с вершины стека в заданную позицию, сдвигая при этом элементы, чтобы освободить место; `lua_replace` снимает значение с вершины стека и устанавливает его как значение элемента с заданным индексом; наконец, `lua_copy` копирует значение с одним индексом в значение с другим индексом, не изменяя исходного значения. Обратите внимание, что следующие операции не влияют на непустой стек:

```
lua_settop(L, -1); /* установить вершину в текущее значение */
lua_insert(L, -1); /* переместить элемент с вершины на вершину */
lua_copy(L, x, x); /* скопировать элемент на его позицию */
```

Программа в листинге 25.3 использует функцию `stackDump` (определенную в листинге 25.2) для иллюстрации этих операций со стеком.

Листинг 25.3. Пример работы со стеком

```
#include <stdio.h>
#include "lua.h"
```

```
#include "luaolib.h"

static void stackDump (lua_State *L) {
    <то же, что и 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 возможно. Это означает, что в Lua мы постоянно сталкиваемся с возможной ошибкой при выделении памяти. Практически любая операция со временем может к этому привести. Более того, многие операции могут вызывать и другие ошибки; например, доступ к глобальной переменной может привести к вызову метаметода `__index`, и этот метаметод может вызвать ошибку. Наконец, операции, которые выделяют память, со временем вызывают сборщик мусора, который вызывает финализаторы, которые также могут привести к ошибкам. Говоря короче, подавляющее большинство функций в Lua может привести к ошибкам.

Вместо использования кодов ошибок в своем API Lua использует исключения для сообщения об ошибках. В отличие от C++ или Java, язык C не содержит механизм для работы с исключениями. Для того чтобы обойти эту проблему, Lua использует функцию `setjmp` из C,

которая дает механизм, похожий на обработку исключений. Поэтому большинство функций из API могут выкинуть ошибку (то есть вызвать `longjmp`) вместо возвращения значения.

Когда мы пишем библиотечный код (то есть функции, которые будут вызываться из Lua), использование функции `longjmp` почти так же удобно, как и использование настоящих исключений, поскольку Lua поймает любую возникающую ошибку. Когда мы пишем код приложения (то есть код на C, который вызывает Lua), то мы должны обеспечить способ для перехвата подобных ошибок.

Обработка ошибок в коде приложения

Когда ваше приложение вызывает функции из Lua API, то оно подвержено ошибкам. Как мы уже обсуждали, Lua обычно сообщает об этих ошибках при помощи 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 *специальную функцию* (panic function), и если управление из этой функции возвращается, то выполнение приложения прерывается. Вы можете задать свою подобную функцию при помощи `lua_atpanic`, но эта функция мало что может сделать.

Для того чтобы правильно обрабатывать ошибки в коде вашего приложения, вы должны позвать ваш код через Lua, так что он установит соответствующий контекст для перехвата ошибок (то есть он выполнит ваш код в контексте `setjmp`). Точно так же, как мы можем запускать код на Lua в защищенном режиме, используя `pcall`, мы можем выполнять код на C, используя `lua_pcall`. Более точно, мы помещаем код на C в функцию и вызываем эту функцию через Lua, используя `lua_pcall`. (Мы подробно обсудим, как вызывать функции на C из Lua, в главе 27.) Тогда наш код на C выполнится в защищенном режиме. Даже в случае ошибки при выделении памяти `lua_pcall` возвращает соответствующий код ошибки, оставляя интерпретатор в рабочем состоянии.

Обработка ошибок в коде библиотек

Lua – это *безопасный* язык. Это значит, что не важно, что вы напишете на Lua, не важно, насколько это неверно, вы всегда можете понять поведение программы в терминах самого Lua. Более того, ошибки также обнаруживаются и объясняются в терминах Lua. Вы можете сравнить это с C, где поведение многих неправильно написанных программ мо-

жет быть объяснено только в терминах используемого оборудования (например, места ошибок задаются как адреса команд).

Когда вы добавляете функцию на С к Lua, вы нарушаете эту безопасность. Например, такая функция, как `poke`, которая записывает произвольный байт по произвольному адресу памяти, может привести к большому числу ошибок при работе с памятью. Вам надо стремиться к тому, чтобы ваши добавления были безопасны для Lua и обеспечивали хорошую обработку ошибок.

Как мы уже обсуждали ранее, программы на С должны задавать свою обработку ошибок при помощи `lua_pcall`. Однако когда вы пишете произвольные функции на Lua, обычно вам не нужно обрабатывать ошибки. Ошибки, выброшенные библиотечной функцией, будут пойманы либо при помощи `pcall` в Lua, либо при помощи `lua_pcall` в коде приложения. Поэтому, когда функция в библиотеке на С обнаруживает ошибку, она может просто позвать `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, передавая сообщение об ошибке.

Упражнения

Упражнение 25.1. Откомпилируйте и запустите простой отдельный интерпретатор Lua (листинг 25.1).

Упражнение 25.2. Предположим, что стек пустой. Что будет находиться в стеке 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 интерпретатор Lua из листинга 25.1 и функцию `stackDump` (листинг 25.2) для того, чтобы проверить ваш ответ к предыдущему упражнению.



ГЛАВА 26

Расширение вашего приложения

Важным использованием Lua является использование его как *конфигурационного* языка. В этой главе мы покажем, как мы можем использовать Lua для конфигурации программы, начиная с простого примера, и будем развивать его для выполнения все более сложных задач.

26.1. Основы

В качестве первой задачи давайте рассмотрим простой конфигурационный сценарий: у вашей программы на C есть окно, и вы хотите иметь возможность задавать начальный размер окна. Ясно, что для такой простой задачи существуют и более простые решения, чем использование Lua, такие как 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 на Lua (то есть текстовый файл, который является программой на Lua). В простейшей форме этот текстовый файл может содержать следующие строки:

```
-- define window size
width = 200
height = 300
```

Теперь вы должны использовать Lua API для того, чтобы Lua разобрал 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, подобно тому, что мы видели в предыдущей главе. Она вызывает функцию `luaL_loadfile` для загрузки блока из файла `fname` и затем вызывает `lua_pcall` для запуска откомпилированного блока. В случае

ошибок (например, синтаксических ошибок в вашем конфигурационном файле) эти функции помещают сообщение об ошибке на стек и возвращают ненулевой код ошибки; наша программа тогда использует `lua_tostring` с индексом `-1`, для того чтобы получить сообщение с вершины стека. (Мы обсудили 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 блока кода программе нужно получить значения глобальных переменных. Для этого она дважды вызывает функцию `lua_getglobal`, параметром которой (кроме постоянно присутствующего `lua_State`) является имя переменной. Каждый такой вызов помещает соответствующее значение на стек, поэтому ширина окна будет находиться на позиции с индексом `-2` и высота на позиции с индексом `-1` (на вершине). (Поскольку стек был изначально пуст, что мы можем также индексировать, начиная с основания стека, то есть использовать 1 для первого значения и 2 для второго. Однако, индексируя с вершины, нам не нужно делать каких-либо предположений о пустоте стека.) Далее наш пример использует функцию `lua_isnumber` для того, чтобы проверить, является ли каждое значение числом. Затем вызывается `lua_tointeger`, и соответствующие значения присваиваются.

Стоило ли использовать Lua для подобной задачи? Как я сказал ранее, для такой простой задачи простой текстовый файл с двумя числами будет гораздо легче, чем Lua. Даже так, использование Lua дает нам некоторые преимущества. Во-первых, Lua полностью занимается синтаксисом за вас; ваш конфигурационный файл может даже содержать комментарии! Во-вторых, пользователь уже может выполнить сложное конфигурирование с тем, что у нас есть. Например, скрипт может запросить у пользователя какую-то информацию или взять

значение из переменной окружения для выбора подходящего размера:

```
-- configuration file
if getenv("DISPLAY") == ":0.0" then
    width = 300; height = 300
else
    width = 200; height = 200
end
```

Даже в таких простых сценариях конфигурации трудно заранее предвидеть, что могут захотеть пользователи; но до тех пор, пока скрипт определяет эти две переменные, ваша программа на С будет работать без изменений.

Окончательным доводом для использования Lua является то, что теперь стало легко добавлять новые конфигурационные возможности к вашей программе; эта легкость создает подход, который приводит к гораздо более гибким программам.

26.2. Работа с таблицами

Давайте примем этот подход: теперь мы хотим задавать цвет фона для окна. Мы будем считать, что заданный цвет состоит из трех чисел, каждое из чисел является компонентой RGB. Обычно на С эти числа являются целыми в некотором диапазоне например [0, 255]. В Lua, поскольку все числа являются с плавающей точкой, нам будет более естественно использовать диапазон [0, 1].

Наивным подходом было бы попросить пользователя задавать каждую компоненту в отдельной глобальной переменной:

```
-- configuration file
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

У такого похода два недостатка: во-первых, он слишком избыточен и громоздок (настоящим программам могут понадобиться десятки цветов для фона в окне, для цвета в окне, для фона меню и т. п.); и нет способа заранее определить распространенные цвета, так чтобы пользователь мог потом просто написать `background=WHITE`. Чтобы избежать этих недостатков, мы можем представлять цвета при помощи таблиц:

```
background = {r=0.30, g=0.10, b=0}
```

Использование таблиц придает структуру вашему скрипту; теперь легко для пользователя (или приложения) определить цвета для дальнейшего использования в конфигурационном файле:

```
BLUE = {r=0, g=0, b=1.0}
```

<other color definitions>

```
background = BLUE
```

Для получения этих значений на С 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");
```

Листинг 26.2. Пример реализации функции `getcolorfield`

```
#define MAX_COLOR 255
/* будем считать, что таблица находится на вершине стека */
int getcolorfield (lua_State *L, const char *key) {
    int result;
    lua_pushstring(L, key); /* поместить ключ на стек */
    lua_gettable(L, -2);    /* получить 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); /* удалить число */
    return result;
}
```

Мы сначала получаем значение глобальной переменной `background` и убеждаемся в том, что это таблица, и затем используем `getcolorfield` для получения каждой компоненты.

Конечно, функция `getcolorfield` – это не часть API, мы должны ее определить. Опять мы сталкиваемся с проблемой полиморфизма: может быть много версий функции `getcolorfield`, отличающихся типом ключа, типом значения, обработкой ошибок и т. п. Lua API предлагает всего одну функцию `lua_gettable`, которая работает для всех типов. Она берет позицию таблицы в стеке, снимает ключ со стека и помещает на стек соответствующее значение. Наша функция `getcolorfield`, определенная в листинге 26.2, считает, что таблица находится на вершине стека, поэтому после помещения ключа на стек

при помощи функции `lua_pushstring` таблица будет находиться по индексу `-2`. Перед возвратом `getcolorfield` снимает со стека полученное значение, оставляя стек в том же состоянии, в котором он был перед этим вызовом.

Поскольку индексирование таблицы при помощи строкового ключа очень распространено, Lua 5.1 ввел специализированную версию `lua_gettable` именно для этого случая: `lua_getfield`. Используя эту функцию, мы можем переписать следующие две строки:

```
lua_pushstring(L, key);
lua_gettable(L, -2); /* получить background[key] */
```

как

```
lua_getfield(L, -1, key);
```

(Поскольку мы не помещаем строку на стек, то у таблицы индекс по-прежнему `-1` в момент вызова `lua_getfield`.)

Мы немного расширим наш пример и введем имена цветов для пользователя. Пользователь может по-прежнему использовать таблицы для покомпонентного задания цветов, но также можно использовать заранее определенные имена цветов. Для реализации этого нам понадобится таблица цветов в нашей программе на 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},
    <other colors>
    {NULL, 0, 0, 0} /* терминатор */
};
```

Наша реализация создаст глобальные переменные с именами цветов и инициализирует эти переменные при помощи таблиц цветов. Результат будет тем же самым, 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}
<other colors>
```

Для задания полей таблицы мы введем вспомогательную функцию `setcolorfield`; она помещает индекс и значение поля на стек и затем вызывает `lua_settable`:

```
/* считаем, 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 многих различных типов, поэтому она берет все свои операнды со стека. Она берет индекс таблицы в качестве аргумента и снимает ключ и значение со стека. Функция `setcolorfield` предполагает, что перед вызовом таблица находится на вершине стека (индекс `-1`); после помещения индекса и значения на стек таблица будет находиться по индексу `-3`.

Lua 5.1 также ввел специализированную версию `lua_settable` для строковых ключей, она называется `lua_setfield`. Используя эту новую функцию, мы можем переписать `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 значения соответствующих полей и присваивает эту таблицу соответствующей глобальной переменной:

```
void setcolor (lua_State *L, struct ColorTable *ct) {
    lua_newtable(L); /* creates a table */
    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 помещает ее на стек; вызовы `setcolorfield` задают поля этой таблицы; наконец, `lua_setglobal` снимает таблицу со стека и использует ее как значение глобальной переменной с заданным именем.

Используя эти 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 перед выполнением скрипта.

Листинг 26.3. Цвета как строки или таблицы

```
lua_getglobal(L, "background");
if (lua_isstring(L, -1)) { /* значение - это строка? */
    const char *name = lua_tostring(L, -1); /* получить строку */
    int i; /* смотрим в таблице */
    for (i = 0; colortable[i].name != NULL; i++) {
        if (strcmp(colortable[i].name, name) == 0)
            break;
    }
    if (colortable[i].name == NULL) /* строка не найдена? */
        error(L, "invalid color name (%s)", name);
    else { /* используем 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, "недопустимое значение для 'background'");
```

Листинг 26.3 показывает другой вариант для реализации именованных цветов. Вместо глобальных переменных пользователь может обозначать цвета при помощи строк, записывая настройки как `background="BLUE"`. Таким образом, `background` может быть или таблицей, или строкой. При подобном подходе приложению не нужно что-либо делать перед запуском скрипта. Вместо этого для получения цвета нужно выполнить немного больше работы. Когда программа получает значение переменной `background`, то необходимо проверить, является ли это значение строкой, и в этом случае поискать цвет в таблице цветов.

Что является лучшей опцией? В программах на С использование строк для обозначения опций не является хорошей практикой, поскольку компилятор не может обнаружить опечатки. Однако в Lua сообщение об опечатке в имени цвета пойдет тому, для кого пишется эта конфигурация. Различие между программистом и пользователем несколько размыто; разница между ошибкой компиляции и ошибкой времени выполнения не столь велика.

Со строками значение переменной `background` может 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 скрипт небольшой и найдено много ошибок, то это может быть не очень удачным – добавлять сотни цветов (и создавать сотни таблиц и глобальных переменных) только для того, чтобы пользователь выбрал несколько цветов. Со строками вы избегаете этого.

26.3. Вызовы функций на Lua

Сильной стороной Lua является то, что конфигурационный файл может определить функции, которые потом могут быть вызваны приложением. Например, вы можете написать приложение для того, чтобы построить график функции, и использовать Lua для того, чтобы задать функцию, чей график будет строиться.

Предоставленный API способ вызова функций довольно прост: во-первых, вы помещаете функцию, которую нужно вызвать на стек; во-вторых, помещаете аргументы для вызова также на стек; после этого используете `lua_pcall` для вызова функции и, наконец, снимаете результаты со стека.

Пусть, в качестве примера, наш конфигурационный файл содержит функцию вроде приведенной ниже:

```
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 . Считая, что вы уже открыли библиотеку Lua и выполнили конфигурационный файл, функция `f` в листинге 26.4 реализует этот вызов.

Листинг 26.4. Вызов функции на Lua из C

```
/* вызов функции 'f', определенной на Lua */
double f (lua_State *L, double x, double y) {
    int isnum;
    double z;
    /* поместить на стек функцию и аргументы */
    lua_getglobal(L, "f"); /* вызываемая функция */
    lua_pushnumber(L, x); /* поместить на стек 1-й аргумент */
    lua_pushnumber(L, y); /* поместить на стек 2-й аргумент */
    /* вызвать функцию (2 аргумента, 1 результат) */
    if (lua_pcall(L, 2, 1, 0) != LUA_OK)
        error(L, "error running function 'f': %s",
              lua_tostring(L, -1));
    /* получить результат */
    z = lua_tonumberx(L, -1, &isnum);
    if (!isnum)
```

```
error(L, "function 'f' must return a number");  
lua_pop(L, 1); /* поднять со стека результат */  
return z;  
}
```

Второй и третий аргументы `lua_pcall` – это соответственно число аргументов, которые вы передаете, и число результатов, которое вы хотите получить. 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 результирующих значений к заданному вами числу; при необходимости помещая на стек *nil*ы или убирая лишние значения. Перед помещением результатов на стек `lua_pcall` удаляет со стека функцию и ее аргументы. Когда функция возвращает несколько значений, то первое значение помещается на стек первым; например, если возвращаются три значения, то первое из них будет иметь индекс `-3`, а последнее `-1`.

В случае возникновения ошибки при выполнении `lua_pcall` функция `lua_pcall` возвращает код ошибки; кроме того, она помещает сообщение об ошибке на стек (но по-прежнему снимает со стека функцию и ее аргументы). Однако перед помещением сообщения на стек `lua_pcall` вызывает функцию обработки сообщения, если она была задана. Для задания функции обработки сообщения используйте последний аргумент функции `lua_pcall`. Ноль означает, что нет никакой функции обработки и окончательное сообщение – это исходное сообщение об ошибке. В противном случае этот аргумент должен быть индексом в стеке, где размещена функция обработки сообщения. В подобном случае функция обработки должна быть помещена на стек до вызываемой функции и ее аргументов.

Для нормальных ошибок функция `lua_pcall` возвращает код ошибки `LUA_ERRRUN`. Два специальных типа ошибок заслуживают отдельных кодов, поскольку для них никогда не вызывается функция обработки. Первый тип – это ошибки по выделению памяти. Для подобных ошибок `lua_pcall` всегда возвращает `LUA_ERRMEM`. Второй тип ошибок – 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 тип ошибок: когда финализатор выкидывает ошибку, то `lua_pcall` возвращает код `LUA_ERRGCM`. Этот код обозначает, что ошибка не связана непосредственно с самим вызовом.

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 nargs, nres; /* число аргументов и результатов */
    va_start(vl, sig);
    lua_getglobal(L, func); /* поместить функцию на стек */
    <поместить аргументы на стек (листинг 26.6)>
    nres = strlen(sig); /* число ожидаемых результатов */
    if (lua_pcall(L, nargs, nres, 0) != 0) /* выполнить вызов */
        error(L, "error calling '%s': %s", func,
              lua_tostring(L, -1));
    <получить результаты (листинг 26.7)>
    va_end(vl);
}
```

26.4. Обобщенный вызов функции

В качестве более сложного примера мы построим универсальную функцию для вызова функций на Lua, используя `vararg` в C. Наша функция, давайте назовем ее `call_va`, берет имя функции, которую нужно вызывать, строку, описывающую типы аргументов и результатов, затем список аргументов и, наконец, список указателей на переменные, в которых мы хотим получить результаты вызова. 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 описатель использует `'d'` для `double`, `'i'` для целых чисел и `'s'` для строк; символ `'>'` отделяет аргументы от результатов. Если функция ничего не возвращает, то символ `'>'` необязателен.

Листинг 26.5 показывает реализацию функции `call_va`. Несмотря на общий вид этой функции, она идет тем же путем, что и наш первый пример: помещает функцию на стек, помещает аргументы на стек (листинг 26.6), выполняет вызов и получает результаты (листинг 26.7). Большая часть кода довольно проста, но есть некоторые тонкости. Во-первых, она не проверяет, что `func` является функцией; если это не так, то `lua_pcall` вызовет ошибку. Во-вторых, поскольку она

помещает на стек произвольное число аргументов, она должна проверять наличие свободного места на стеке. В-третьих, поскольку функция может вернуть строки, то `call_va` не может снять результаты со стека. Это должна сделать вызывающая сторона после использования строк-результатов (или копирования их в другое место).

Листинг 26.6. Помещение аргументов на стек в общем случае

```
for (narg = 0; *sig; narg++) { /* выполнить для каждого аргумента */
/* проверяем место на стеке */
luaL_checkstack(L, 1, "слишком много аргументов");
switch (*sig++) {
case 'd': /* double argument */
lua_pushnumber(L, va_arg(vl, double));
break;
case 'i': /* int argument */
lua_pushinteger(L, va_arg(vl, int));
break;
case 's': /* string argument */
lua_pushstring(L, va_arg(vl, char *));
break;
case '>': /* end of arguments */
goto endargs;
default:
error(L, "invalid option (%c)", *(sig - 1));
}
}
endargs;
```

Листинг 26.7. Получение результатов вызова

```
nres = -nres; /* индекс певрого результата на стеке */
while (*sig) { /* повторить для каждого результата */
switch (*sig++) {
case 'd': { /* double result */
int isnum;
double n = lua_tonumberx(L, nres, &isnum);
if (!isnum)
error(L, "wrong result type");
*va_arg(vl, double *) = n;
break;
}
case 'i': { /* int result */
int isnum;
int n = lua_tointegerx(L, nres, &isnum);
if (!isnum)
error(L, "wrong result type");
*va_arg(vl, int *) = n;
break;
}
```

```
    }  
    case 's': { /* string result */  
        const char *s = lua_toststring(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 на C, которая читает файл на Lua, определяющий функцию `f`, которая берет на вход число и возвращает значение функции от этого числа. Ваша программа должна построить график этой функции. (Вам не обязательно использовать графику, вполне подойдет обычный текстовый вид, использующий `''` для графика.)

Упражнение 26.2. Измените функцию `call_va` (листинг 26.5) для обработки булевых значений.

Упражнение 26.3. Пусть есть программа, которой необходимо следить за несколькими погодными станциями. Внутри, для представления каждой станции, она использует 4-байтовую строку, и есть конфигурационный файл, который сопоставляет каждой такой строке URL соответствующей станции. Конфигурационный файл на Lua должен выполнять это сопоставление несколькими разными способами:

- набор глобальных переменных, по одной для каждой станции;
- одна таблица, отображающая строки на URL;
- одна функция, для каждой строки возвращающая URL.

Обсудите плюсы и минусы каждого варианта, принимая во внимание общее число станций, типы пользователей, наличие структуры в URL и т. п.



ГЛАВА 27

Вызываем C из Lua

Когда мы говорим, что Lua может вызывать C, это не значит, что Lua может вызвать любую функцию на C¹. Как мы видели в предыдущей главе, когда C вызывает функцию на Lua, необходимо следовать определенному протоколу передачи аргументов и получения результата. Аналогично, чтобы Lua мог вызвать функцию на C, эта функция должна следовать определенному протоколу для получения своих аргументов и возвращению результатов. Более того, чтобы Lua мог вызвать функцию на C, мы должны зарегистрировать эту функцию, то есть должны передать Lua ее адрес определенным способом.

Когда Lua вызывает функцию на C, он использует такой же самый стек, какой C использует для вызова кода на Lua. Функция на C получает свои аргументы со стека и помещает свои результаты на стек.

Важным понятием здесь является то, что стек – это некоторая структура; у каждой функции есть свой собственный локальный стек. Когда Lua вызывает функцию на C, то первый аргумент всегда будет иметь индекс 1 в этом локальном стеке. Даже когда код на C вызывает код на Lua, который вызывает эту же (или другую) функцию, каждый из этих вызовов будет видеть только свой личный стек с первым аргументом по индексу 1.

27.1. Функции на 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); /* получить аргумент */  
    lua_pushnumber(L, sin(d)); /* поместить результат на стек */  
}
```

¹ Есть пакеты, которые позволяют Lua вызывать любую функцию на C, но они не переносимы и не безопасны.

```
    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 на C получает в качестве своего единственного аргумента состояние Lua и возвращает целое число, равное числу значений, возвращаемых через стек. Поэтому функции не нужно очищать стек перед помещением на него своих результатов. После возвращения функции Lua сам сохраняет результаты и очищает стек.

Перед тем как мы сможем использовать эту функцию, мы должны сперва зарегистрировать ее. Мы делаем это при помощи `lua_pushcfunction`: она получает указатель на функцию на C и создает значение типа "function", которое представляет эту функцию внутри Lua. После регистрации функция на C ведет себя, как любая другая функция внутри Lua.

Быстрый- и -грязный способ проверить функцию `l_sin` – это 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 помещает на стек значение типа функции, а вторая присваивает его значение глобальной переменной `mysin`. После этих изменений вы можете использовать эту новую функцию прямо в своих скриптах на Lua. (В следующем разделе мы рассмотрим более правильные способы подключения функций на C к Lua.)

Для более серьезной функции, вычисляющей синус, мы должны проверить тип ее аргумента. Здесь нам помогает вспомогательная библиотека. Функция `luaL_checknumber` проверяет, действительно ли заданный аргумент – это число: в случае ошибки она выбрасывает осмысленное сообщение об ошибке, иначе 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 */  
}
```


При таком определении функции, если мы вызовем `mysin('a')`, то мы получим 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 реальным типом параметра (`string`).

В качестве более сложного примера давайте напишем функцию, которая вернет содержимое заданного каталога. Lua не предоставляет такой функции в своих стандартных библиотеках, поскольку в ANSI C нет подходящей для этого функции. Здесь мы будем считать, что наша функция поддерживает POSIX. Наша функция – назовем ее `dir` в Lua и `l_dir` в C – получает в качестве аргумента строку с путем на каталог и возвращает массив с содержимым этого каталога. Например, вызов `dir("/home/lua")` может вернуть следующую таблицу `{".", "..", "src", "bin", "lib"}`. В случае ошибки функция возвращает *nil* и строку с сообщением об ошибке. Полный код этой функции приведен в листинге 27.1. Обратите внимание на использование функции `luaL_checkstring`, которая ведет себя аналогично `luaL_checknumber`, но только для строк.

(В крайних случаях использование этой функции может привести к небольшой утечке памяти. Три из Lua-функций, которые она вызывает, могут завершиться неудачно из-за недостаточного объема памяти: `lua_newtable`, `lua_pushstring` и `lua_settable`. Если любая из этих функций завершится неудачей, то будет вызвана ошибка и выполнение `l_dir` будет прервано, соответственно, `closedir` не будет вызвана. Как мы обсудили ранее, для большинства программ это не является большой проблемой: если заканчивается память, то лучшее, что можно сделать, – это завершить выполнение программы. Тем не менее в главе 30 мы увидим другую реализацию функции получения содержимого каталога, которая уже не содержит этой ошибки.)

Листинг 27.1. Функция для чтения содержимого каталога

```
#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);
/* открыть каталог */
dir = opendir(path);
if (dir == NULL) { /* ошибка при открытии каталога? */
    lua_pushnil(L); /* вернуть nil... */
    luaL_pushstring(L, strerror(errno)); /* и сообщение */
    return 2; /* число результатов */
}
/* создать таблицу с результатом */
lua_newtable(L);
i = 1;
while ((entry = readdir(dir)) != NULL) {
    lua_pushnumber(L, i++); /* push key */
    luaL_pushstring(L, entry->d_name); /* push value */
    lua_settable(L, -3);
}
closedir(dir);
return 1; /* таблица уже наверху стека */
}
```

27.2. Продолжения

При помощи `lua_pcall` и `lua_call` функция на С, вызванная из Lua, может, в свою очередь, вызвать Lua. Некоторые функции из стандартной библиотеки так и делают: `table.sort` может вызвать функцию сравнения; `string.gsub` может вызвать функцию замены; `pcall` и `xpcall` могут вызвать функции в защищенном режиме. Если мы помним, что главный код на Lua был сам, в свою очередь, вызван из С (основной программы), то мы получаем последовательность вроде следующей: С (приложение) вызывает Lua (скрипт), который вызывает функцию на С (библиотека), которая вызывает Lua.

Обычно Lua обрабатывает эти последовательности вызовов без проблем; в конце концов, ее главной задачей является интеграция с С. Однако есть ситуация, в которой подобная цепочка вызовов может вызвать проблемы: сопрограммы.

Каждая сопрограмма в Lua имеет свой собственный стек, который содержит информацию об ожидающих вызовах сопрограммы. Точнее, стек запоминает адрес возврата, параметры и локальные переменные каждого вызова. Для вызовов функций на Lua интерпретатор использует подходящую структуру данных для реализации стека, которая называется *гибкий стек* (soft stack). Однако для вызовов функций на С интерпретатор также должен использовать стек С. В конце концов, адрес возврата и локальные переменные функции на С живут на стеке С.

Интерпретатор легко может иметь много гибких стеков, но код на C имеет только один стек. Поэтому сопрограммы в Lua не могут приостановить выполнение внутри функции на C: если в цепочке вызовов от `resume` до соответствующего `yield` есть функция на C, то Lua не может сохранить состояние этой функции для того, чтобы восстановить его при следующем `resume`. Давайте 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
```

Вызов `pcall` – это функция на C; поэтому Lua не может «заморозить» ее, поскольку в ANSI C нет способа приостановить выполнение функции на C и потом продолжить ее выполнение.

Lua 5.2 справился с этой сложностью при помощи *продолжений* (continuation). Lua 5.2 реализует `yield` при помощи `longjmp`, то есть так же, как она реализует ошибки. Такой вызов (`longjmp`) просто отбрасывает всю информацию о функциях на C-стеке, поэтому невозможно продолжить функцию на C. Однако функция на C `foo` может указать функцию – продолжение `foo-c`, которая является другой функцией на C, которая должна быть вызвана, когда настанет время «продолжить» функцию `foo`. То есть интерпретатор обнаруживает, что он должен продолжить выполнение `foo`, но поскольку вся информация о `foo` была уничтожена со стека C, то вместо этого он вызывает `foo-c`.

Для того чтобы стало более понятно, давайте рассмотрим пример: реализацию функции `pcall`. В Lua 5.1. у этой функции был следующий код:

```
static int luaB_pcall (lua_State *L) {
    int status;
    luaL_checkany(L, 1); /* как минимум один параметр */
    status = lua_pcall(L, lua_gettop(L) - 1, LUA_MULTRET, 0);
    lua_pushboolean(L, (status == 0)); /* status */
    lua_insert(L, 1); /* status – это первый результат */
    return lua_gettop(L); /* вернуть status + все результаты */
}
```

Если функция, которую позвали через `lua_pcall`, приостановила свое выполнение (через `yield`), то будет невозможно продол-

жить позже выполнение `luaB_pcall`. Поэтому интерпретатор выдаст ошибку каждый раз, когда мы попытаемся вызвать `yield` внутри защищенного вызова. Lua 5.2 реализует `pcall` примерно так, как показано в листинге 27.2². Здесь есть три отличия от версии на Lua 5.1: во-первых, новая версия заменила вызов `lua_pcall` на `lua_pcallk`; во-вторых, она сгруппировала все, что делается после этого вызова в новой вспомогательной функции `finishpcall`; третье отличие – это функция `pcallcont`, последний аргумент `lua_pcallk`, который и является функцией-продолжением.

Если нет никаких вызовов `yield`, то `lua_pcallk` работает точно как `lua_pcall`. Если есть вызов `yield`, то все совершенно иначе. Если функция, вызванная `lua_pcall`, пытается вызвать `yield`, то Lua 5.2 вызывает ошибку, как и Lua 5.1. Однако когда функция, вызванная `lua_pcallk`, вызывает `yield`, то нет никаких ошибок: Lua вызывает `longjmp` и отбрасывает запись для `luaB_pcall` со стека C, но сохраняет в гибком стеке ссылку на функцию-продолжение `pcallcont`. Позже, когда интерпретатор обнаруживает, что он должен вернуться к `luaB_pcall` (что невозможно), он вместо этого вызывает функцию-продолжение `pcallcont`.

В отличие от `luaB_pcall`, функция-продолжение `pcallcont` не может получить значение, которое вернуло вызов `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 функции-продолжения, она возвращает `LUA_YIELD`. Функция-продолжение также может быть вызвана в случае некоторых ошибок; в этом случае `lua_getctx` возвращает код ошибки, то есть то самое значение, которое в этом случае вернуло бы `lua_callk`.

Листинг 27.2. Реализация `pcall` с продолжениями

```
static int finishpcall (lua_State *L, int status) {
    lua_pushboolean(L, status); /* первый результат (статус) */
    lua_insert(L, 1); /* поместить 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));
}
```

² Настоящий код сложнее, чем показано здесь, поскольку у него есть некоторые общие части с `xpcall` и проверка на переполнение стека перед помещением на него булевого значения.

```
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, функция `lua_getctx` также может вернуть информацию о контексте. Пятый параметр в `lua_pcallk` – это произвольное целое число, которое можно получить через второй параметр `lua_getctx`, который является указателем на целое значение. Это целое значение позволяет исходной функции передать произвольную информацию прямо своему продолжению. Она также может передать дополнительную информацию через стек Lua. (Наш пример не использует эту возможность.)

Механизм продолжений в Lua 5.2 – это потрясающий механизм для поддержки `yield`, но это не панацея. Некоторым функция на С может понадобиться передать слишком много контекста своим продолжениям. В качестве такого примера можно использовать `table.sort`, которая использует стек С для рекурсии, и `string.gsub`, которая должна отслеживать найденные подстроки и буфер для частичных результатов. Хотя их можно переписать способом, поддерживающим `yield`, выигрыш этого не стоит вводимой сложности.

27.3. Модули на С

Модуль в Lua – это блок кода, который определяет различные функции на Lua и запоминает их в подходящих местах, обычно полях таблицы. Модуль на С для Lua ведет себя так же. Кроме определения своих функций на С, он должен еще определить специальную функцию, которая исполняет роль главного блока в библиотеке Lua. Эта функция должна зарегистрировать все функции на С из модуля и запомнить их в подходящих для этого местах, обычно в полях таблицы. Подобно главному блоку на Lua, эта функция также должна проинициализировать все, что требует инициализации.

Lua получает функции на С через механизм регистрации. После того как функция на С представлена и запомнена в Lua, Lua вызывает ее через непосредственную 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, для того чтобы вызвать эту функцию, когда она зарегистрирована.

Обычно модуль на С имеет всего одну открытую (*extern*) функцию, которая является функцией, открывающей библиотеку. Все остальные функции могут быть закрыты, например объявлением их как *static*.

Когда вы расширяете Lua при помощи функций на С, то хорошей идеей является организовать ваш код как модуль на С, даже если вы хотите зарегистрировать всего одну функцию: рано или поздно (обычно рано) вам понадобятся другие функции. Как обычно, вспомогательная библиотека предлагает вспомогательную функцию для этого. Макро `luaL_newlib` берет список функций на С вместе с их соответствующими именами и регистрирует их все внутри новой таблицы. В качестве примера пусть 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 ранее>  
}
```

Дальше мы определяем массив со всеми функциями вместе с их 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} /* терминатор */  
};
```

В нашем примере есть только одна функция (`l_dir`), которую мы хотим зарегистрировать. Последней парой в массиве всегда является `{NULL, NULL}`, обозначающая конец массива. Наконец, мы определяем 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`. По возвращении `luaL_newlib` оставляет на стеке новую таблицу. Функция `luaopen_mylib` возвращает 1 для того, чтобы вернуть эту таблицу в Lua.

После завершения библиотеки мы должны прилинковать ее к интерпретатору. Наиболее простым способом сделать это является

использование динамических библиотек, если интерпретатор Lua поддерживает их. В этом случае вы должны создать динамическую библиотеку с вашим кодом (`mylib.dll` в Windows и `mylib.so` в Linux) и поместить ее вдоль С-пути. После этих шагов вы можете загрузить эту библиотеку непосредственно из Lua при помощи `require`:

```
local mylib = require "mylib"
```

Этот вызов загружает динамическую библиотеку в Lua, находит функцию `luaopen_mylib`, регистрирует ее как функцию на С и вызывает ее, открывая тем самым модуль. (Это объясняет, почему `luaopen_mylib` должна иметь тот же самый прототип, что и любая другая функция на С.)

При загрузке динамической библиотеки мы должны знать имя функции `luaopen_mylib` для того, чтобы ее найти. Всегда будет исаться функция с именем `luaopen_`, к которому присоединено название модуля. Поэтому если ваш модуль называется `mylib`, то функция должна называться `luaopen_mylib`.

Если ваш интерпретатор не поддерживает динамическую линковку, то вам нужно пересобрать Lua вместе с вашей новой библиотекой. Кроме этой пересборки, вам также нужен некоторый способ сказать интерпретатору, что он должен открыть эту библиотеку при создании нового состояния. Обычно это делают добавлением `luaopen_mylib` в список стандартных библиотек, которые открывает `luaL_openlib` в файле `linit.c`.

Упражнения

Упражнение 27.1. Напишите на С функцию `summation`, которая считает сумму переменного числа своих 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 получает на вход таблицу и функцию и вызывает эту функцию для каждой пары ключ–значение в таблице:

```
foreach({x = 10, y = 20}, print)
--> x 10
--> y 20
```

Упражнение 27.5. Перепишите функцию `foreach` из предыдущего упражнения так, чтобы вызываемая функция могла вызвать `yield`.

Упражнение 27.6. Создайте модуль на С со всеми функциями из предыдущих упражнений.



ГЛАВА 28

Приемы написания функций на С

И официальный API, и *вспомогательная* (auxiliary) библиотека предоставляют несколько механизмов, помогающих писать функции на С. В этой главе мы рассмотрим механизмы для работы с массивами, строками и сохранения значений Lua в С.

28.1. Работа с массивами

В Lua «массив» – это просто имя для таблицы, используемой специальным образом. Мы можем работать с массивами, используя те же самые функции, которые мы использовали для работы с таблицами, то есть `lua_settable` и `lua_gettable`. Однако API предоставляет несколько специальных функций для работы с массивами. Одним из плюсов использования этих функций является быстроедействие: часто у нас встречается обращение к массиву внутри цикла алгоритма (например, сортировки), так что любое увеличение быстрогодействия, которое есть в этих операциях, может иметь большое влияние на итоговое быстроедействие алгоритма. Еще одним плюсом является удобство, целочисленные ключи достаточно часто встречаются, чтобы заслужить специального обращения.

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 сразу два индекса: `index` описывает, где таблица находится на стеке; `key` задает элемент в самой таблице. Вызов `lua_rawgeti(L, t, key)` эквивалентен следующей последовательности, когда `t` больше нуля (в противном случае необходимо компенсировать появление нового элемента на стеке):

```
lua_pushnumber(L, key);
lua_rawget(L, t);
```

Вызов `lua_rawseti(L, t, key)` (опять для положительного `t`) эквивалентен следующей последовательности:

```
lua_pushnumber(L, key);
lua_insert(L, -2); /* поместить 'key' ниже предыдущего значения */
lua_rawset(L, t);
```

Обратите внимание, что обе функции используют прямое обращение к таблице. 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); /* получить размер таблицы */
    for (i = 1; i <= n; i++) {
        lua_pushvalue(L, 2); /* поместить на стек f */
        lua_rawgeti(L, 1, i); /* поместить на стек t[i] */
        lua_call(L, 1, 1); /* вызвать f(t[i]) */
        lua_rawseti(L, 1, i); /* t[i] = result */
    }
    return 0; /* нет результатов */
}
```

В качестве примера использования этих функций листинг 28.1 реализует функцию `map`: 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.

Примитив `lua_len` (не используемый в примере выше) эквивалентен оператору `‘#’`. Из-за метаметодов этот оператор может вернуть объект любого типа, а не только числа; поэтому `lua_len` возвращает свой результат на стеке. Функция `luaL_len` (используемая в примере) вызывает ошибку, если длина не является числом, в противном случае она возвращает длину как обычное значение целого типа.

Функция `lua_call` выполняет незащищенный вызов. Он аналогичен `lua_pcall`, но он передает ошибки выше, а не возвращает код ошибки. Когда вы пишете главный код приложения, то вам лучше не использовать `lua_call`, поскольку вы хотите ловить любые ошибки. Однако когда вы пишете функции, то лучше использовать именно `lua_call`; если возникнет ошибка, то мы оставим ее для кого-нибудь, кому это важно.

28.2. Работа со строками

Когда функция на C получает строковый аргумент из Lua, то существуют только два правила, которые нужно выполнить: не снимать строку со стека при работе с ней и никогда не изменять строку.

Ситуация становится сложнее, когда функции на C нужно создать строку, для того чтобы вернуть ее Lua. Код на C должен беспокоиться о выделении/освобождении буфера, переполнении буфера и т. п. Тем не менее Lua API предоставляет для этого несколько функций.

Стандартный API предоставляет помощь в двух наиболее часто встречающихся операциях: выделение подстроки и конкатенация строк. При выделении подстроки помните, что `lua_pushlstring` получает длину строки как дополнительный аргумент. Поэтому если вы хотите передать Lua подстроку строки `s` с символами в позициях от `i` до `j` (включительно), то все что, вам нужно сделать, – это следующее:

```
lua_pushlstring(L, s + i, j - i + 1);
```

В качестве примера пусть 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 сам Lua.

Листинг 28.2. Разбиение строки

```
static int l_split (lua_State *L) {
    const char *s = luaL_checkstring(L, 1); /* строка */
    const char *sep = luaL_checkstring(L, 2); /* разделитель */
    const char *e;
    int i = 1;
    lua_newtable(L); /* таблица с результатом */
    /* повторить для каждого разделителя */
```

```
while ((e = strchr(s, *sep)) != NULL) {
    lua_pushlstring(L, s, e-s); /* подстроку на стек */
    lua_rawseti(L, -2, i++);    /* вставить в таблицу */
    s = e + 1; /* пройти за разделитель */
}
/* вставить последнюю подстроку */
lua_pushstring(L, s);
lua_rawseti(L, -2, i);
return 1; /* вернуть таблицу */
}
```

Для конкатенации строк Lua предоставляет специальную функцию в своем API, называемую `lua_concat`. Она эквивалентна оператору конкатенации `..` в Lua; она преобразует числа в строки и при необходимости вызывает метаметоды. Более того, она может сразу объединить больше двух строк. Вызов `lua_concat(L, n)` конкатенирует (подняв со стека) `n` значений и поместит результат на вершину стека.

Другой полезной функцией является `lua_pushfstring`:

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

Она несколько похожа на функцию `sprintf` тем, что создает строку по строке формата и дополнительным аргументам. Однако, в отличие от `sprintf`, вам не нужно предоставлять буфер. Lua динамически создает строку для вас такой большой, как необходимо. Эта функция помещает получившуюся строку на стек и возвращает указатель на нее. Вам не нужно беспокоиться о переполнении буфера.

Сейчас эта функция поддерживает только следующие форматы¹:

%s	Вставить строку, завершенную нулевым байтом
%d	Вставить целое число
%f	Вставить число Lua, то есть <code>double</code>
%p	Вставить указатель
%c	Вставить целое как символ
%%	Вставить символ <code>'\%'</code>

Не поддерживаются никакие модификаторы, такие как ширина или точность.

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.

¹ Опция `%p` для указателей появилась только в Lua 5.2.

В этом случае мы можем использовать буферы, предоставленные вспомогательной библиотекой.

В простейшем случае буферы работают как две функции: одна дает вам буфер любого размера, куда вы можете писать свою строку; другая преобразует буфер в строку на Lua². Листинг 28.3 показывает использование этих функций при помощи реализации функции `string.upper` прямо из исходного файла `lstrlib.c`. Первым шагом использования буфера из вспомогательной библиотеки является объявление переменной типа `luaL_Buffer`. Следующим шагом является вызов `luaL_buffinitsize` для получения указателя на буфер с заданным размером; затем вы можете 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 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 у вас есть ее максимальный размер, то вы можете заказать 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

² Эти две функции появились в Lua 5.2.

в буфер элементы один за другим, в этом случае используя функцию `luaL_addvalue`. Наконец, `luaL_pushresult` освобождает буфер и помещает итоговую строку на вершине 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); /* получить строку из таблицы */
        luaL_addvalue(b);      /* добавить ее к буферу */
    }
    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 нулевым байтом, 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, обратите внимание на следующее. После инициализации буфера он хранит некоторые вспомогательные результаты на стеке Lua. Поэтому вы не можете предполагать, что вершина стека останется там, где она была перед тем, как вы начали использовать буфер. Вы можете использовать стек для других задач во время работы с буфером, главное, чтобы вызовы `push` и `pop` были сбалансированы каждый раз, когда вы используете буфер. Исключением к этому правилу является функция `luaL_addvalue`, которая предполагает, что строка, которую надо добавить к буферу, была помещена на вершину стека.

28.3. Сохранение состояния в функциях на C

Часто функциям на C нужно хранить какие-то нелокальные данные, то есть данные, которые переживут текущий вызов функции. В C мы обычно используем глобальные (extern) или статические переменные для этой цели. Однако когда вы пишете библиотечные функции для Lua, то использование глобальных или статических переменных не является хорошим решением. Во-первых, вы не можете сохранить произвольное значение Lua в переменной C. Во-вторых, библиотека, которая использует такие переменные, не будет корректно работать с несколькими состояниями Lua.

У функции на Lua есть два базовых места, где можно хранить нелокальные данные: глобальные переменные и нелокальные переменные. C API также предоставляет два базовых места для хранения нелокальных данных: реестр и значения, связанные с функцией (upvalue).

Реестр – это глобальная таблица, к которой можно обратиться только при помощи кода на C³. 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 значения, связанные с функцией.

Реестр

Реестр обычно расположен по *псевдоиндексу*, значение которого определено как `LUA_REGISTRYINDEX`. Псевдоиндекс выглядит как индекс на стеке, за исключением того, что связанные с ним значения не находятся на стеке. Большинство функций в Lua API, которые принимают индексы в качестве аргументов, также принимают и псевдоиндексы – за исключением тех функций, которые изменяют стек, такие как `lua_remove` и `lua_insert`. Например, для того чтобы получить значение, связанное с ключом “Key” в реестре, мы можем использовать следующий вызов:

```
lua_getfield(L, LUA_REGISTRYINDEX, "Key");
```

Реестр – это обычная таблица Lua. Соответственно, вы можете для обращения к ней использовать любое значение Lua, кроме *nil*. Одна-

³ На самом деле к реестру можно обратиться и из Lua при помощи функции из отладочной библиотеки `debug.getregistry`.

ко поскольку все модули на С разделяют один и тот же реестр, то вы должны очень аккуратно выбирать значения, которые вы будете использовать как ключи, для того чтобы избежать возможных конфликтов. Строковые ключи особенно удобны, когда вы хотите позволить другим модулям обращаться к вашим данным, поскольку все, что им нужно, – это имя. Не существует полностью надежного метода выбора ключей, но есть некоторые зарекомендовавшие себя подходы, такие как не использовать распространенные имена и начинать ваши имена с имени библиотеки или чего-то вроде него. (Префиксы вроде `lua` и `luaLib` не являются хорошими вариантами.)

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 сохранять значения в таблице, не беспокоясь об уникальности ключей. Функция `luaL_ref` создает новые ссылки:

```
int r = luaL_ref(L, LUA_REGISTRYINDEX);
```

Этот вызов снимет значение с вершины стека, сохранит его в таблице по новому целочисленному индексу и вернет этот индекс. Подобные индексы называются *ссылками* (reference).

Как следует из имени, мы будем использовать ссылки в основном для тех случаев, когда нам нужно сохранить значение Lua внутри структуры С. Как мы уже видели, мы никогда не должны запоминать указатели на строки Lua вне той функции на С, которая их получила. Более того, Lua даже не предлагает указателей на другие объекты, такие как таблицы или функции. Поэтому мы не можем ссылаться на объекты Lua при помощи указателей. Вместо этого, когда нам понадобятся такие указатели, мы будем создавать ссылки и запоминать их в С.

Для того чтобы поместить значение, связанное со ссылкой `r` на стек, мы просто используем следующий фрагмент кода:

```
lua_rawgeti(L, LUA_REGISTRYINDEX, r);
```

Наконец, для того, чтобы освободить и значение и ссылку, мы вызываем `luaL_unref`:

```
luaL_unref(L, LUA_REGISTRYINDEX, r);
```

После этого новый вызов `luaL_ref` может вернуть снова эту же ссылку.

Система ссылок трактует *nil* как особый случай. Когда мы вызываем `luaL_ref` для значения *nil*, новая ссылка не создается, и вместо

этого возвращается константа `LUA_REFNIL`. Следующий вызов на самом деле ничего не делает:

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

Следующий вызов помещает на стек *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 ссылки. Она полезна для того, чтобы пометить ссылки как уничтоженные/неинициализированные.

Другим надежным методом для создания ключей в реестр является использование в качестве ключа адреса статической переменной в вашем коде: линковщик гарантирует, что этот адрес будет уникальным. Для того чтобы использовать этот вариант, вам понадобится функция `lua_pushlightuserdata`, которая помещает на стек Lua значение, представляющее собой указатель на C. Следующий код показывает, как сохранить и получить строку из реестра при помощи этого метода:

```
/* переменная с уникальным адресом */
static char Key = 'k';
/* запомнить строку */
lua_pushlightuserdata(L, (void *)&Key); /* push address */
lua_pushstring(L, myStr); /* push value */
lua_settable(L, LUA_REGISTRYINDEX); /* registry[&Key] = myStr */
/* получить строку */
lua_pushlightuserdata(L, (void *)&Key); /* push address */
lua_gettable(L, LUA_REGISTRYINDEX); /* получить значение */
myStr = lua_tostring(L, -1); /* преобразовать его в строку */
```

Более подробно мы обсудим использование типа `userdata` в разделе 29.5.

Для того 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`/`lua_rawseti`, но вместо целых чисел они используют указатели (переведенные в `userdata`) в качестве ключей. Используя их, мы можем переписать предыдущий код следующим образом:

```
static char Key = 'k';
/* запомнить строку */
lua_pushstring(L, myStr);
lua_rawsetp(L, LUA_REGISTRYINDEX, (void *)&Key);
/* получить строку */
```

```
lua_rawgetp(L, LUA_REGISTRYINDEX, (void *)&Key);  
myStr = lua_tostring(L, -1);
```

Значения, связанные с функцией

В то время как реестр предлагает глобальные переменные, механизм значений, связанных с функциями, предлагает аналог *static*-переменных в С, которые видны только внутри отдельной функции. Каждый раз, когда вы создаете новую функцию на С в Lua, вы можете связать с ней любое количество подобных значений; каждое такое значение является значением на Lua. Потом, когда функция будет вызвана, она получает свободный доступ к любому из этих значений, используя псевдоиндексы.

Мы называем эту связь функции на С со своими значениями *замыканием* (closure). Замыкание на С – это аналог замыкания на Lua для языка С. В частности, вы можете создавать различные замыкания, используя один и тот же код функции, но разные связанные значения.

В качестве простого примера давайте напишем функцию `newCounter` на С⁴. Эта функция является фабрикой: она возвращает новую считающую функцию при каждом вызове. Хотя все такие функции имеют один и тот же код на С, каждая из них хранит свой собственный счетчик. Эта функция-фабрика выглядит следующим образом:

```
static int counter (lua_State *L); /* упреждающее 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`), и третьим аргументом является число связанных значений (в примере это 1). Перед созданием нового замыкания мы должны поместить начальные значения для связанных значений на стек. В нашем примере мы помещаем 0 как начальное значение для единственного связанного значения. Как и ожидалось, `lua_pushcclosure` оставляет новое замыкание на стеке, поэтому замыкание уже готово к возвращению как результат `newCounter`.

Давайте теперь посмотрим на определение функции `counter`:

⁴ Мы ввели эту функцию на Lua в разделе 6.1.

```
static int counter (lua_State *L) {
    int val = lua_tointeger(L, lua_upvalueindex(1));
    lua_pushinteger(L, ++val); /* новое значение */
    lua_pushvalue(L, -1); /* дублировать его */
    lua_replace(L, lua_upvalueindex(1)); /* обновить значение */
    return 1; /* вернуть новое значение */
}
```

Ключевым элементом здесь является макрос `lua_upvalueindex`, который возвращает псевдо-индекс связанного значения. В частности, выражение `lua_upvalueindex(1)` возвращает псевдоиндекс первого связанного значения текущей функции. Этот псевдоиндекс выглядит как и любой индекс в стеке, только он не находится в стеке. Поэтому вызов `lua_tointeger` возвращает текущее значение первого (и единственного) связанного значения как число. Затем функция помещает на стек новое значение `++val` на стек, делает его копию и использует одну из копий, для того чтобы заменить связанное значение. Наконец, она возвращает другую копию как свое значение.

В качестве более сложного примера мы реализуем кортежи (tuple) при помощи связанных значений. *Кортеж* – это что-то вроде постоянной записи с анонимными полями; вы можете получить конкретное поле по индексу или можете получить сразу все поля. В нашей реализации мы будем представлять кортежи как функции, которые запоминают свои значения в связанных значениях. Когда функция вызвана с числовым аргументом, то она возвращает конкретное поле. Когда она вызвана без аргументов, то она возвращает все поля. Следующий код демонстрирует использование кортежей:

```
x = tuple.new(10, "hi", {}, 3)
print(x(1)) --> 10
print(x(2)) --> hi
print(x()) --> 10 hi table: 0x8087878 3
```

На C мы представляем все кортежи при помощи одной и той же функции `t_tuple`, показанной в листинге 28.5. Поскольку мы можем вызвать кортеж как с числовым аргументом, так и вообще без аргументов, то функция `t_tuple` использует `luaL_optint` для получения необязательного аргумента. Функция `luaL_optint` похожа на `luaL_checkint`, но в случае если аргумент отсутствует, то она просто 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) { /* аргументов нет? */
    int i;
    /* поместить каждое связанное значение на стек */
    for (i = 1; !lua_isnone(L, lua_upvalueindex(i)); i++)
        lua_pushvalue(L, lua_upvalueindex(i));
    return i - 1; /* число значений на стеке */
}
else { /* получить поле 'op' */
    luaL_argcheck(L, 0 < op, 1, "index out of range");
    if (lua_isnone(L, lua_upvalueindex(op)))
        return 0; /* 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 обращаемся к несуществующему связанному значению, то результатом является псевдозначение типа `LUA_TNONE`. (Когда мы обращаемся к значению выше вершины стека, то мы также получаем псевдозначение типа `LUA_TNONE`.) Поэтому наша функция `t_tuple` использует `lua_isnone` для проверки, есть ли соответствующее значение. Однако мы должны никогда не вызывать `lua_upvalueindex` с отрицательным индексом, поэтому мы должны проверять это, когда индекс предоставляет пользователь. Функция `luaL_argcheck` проверяет любое заданное значение, вызывая ошибку при необходимости.

Функция для создания кортежей `t_new` (тоже в листинге 28.5) тривиально: поскольку все ее аргументы уже на стеке, она просто вызывает `lua_pushcclosure` для создания замыкания используя свои аргументы как связанные значения. Наконец, массив `tuplelib` и функция `luaopen_tuple` (тоже в листинге 28.5) являются стандартным кодом для создания библиотеки `tuple` с единственной функцией `new`.

Значения, связанные с функцией, используемые несколькими функциями

Довольно часто нам нужно дать доступ к нескольким значениям или переменным всем функциям данного модуля. Хотя мы можем использовать реестр для этой цели, мы также можем использовать значения, связанные с функциями.

В отличие от замыканий Lua, C-замыкания не могут иметь общие связанные значения. У каждого замыкания имеются свои независимые связанные значения. Однако мы можем сделать так, что связанные значения нескольких функций будут указывать на одну и ту же таблицу, таким образом эта таблица становится окружением где все эти функции могут хранить общие данные.

В Lua 5.2 есть функция, которая облегчает задачу разделения связанного значения между всеми функциями библиотеки. Мы открывали библиотеки на C при помощи `luaL_newlib`. Lua реализует эту функцию при помощи следующего макроса:

```
#define luaL_newlib(L,l) \
(luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))
```

Макрос `luaL_newlibtable` просто создает таблицу для библиотеки. (Мы также могли использовать `lua_newtable`, но этот макрос использует `lua_createtable` для создания таблицы с заранее выделенным размером, оптимальным для количества функций в этой библиотеке.) Функция `luaL_setfuncs` добавляет функции из списка `l` к новой таблице, которая находится на вершине стека.

Нас здесь интересует третий параметр функции `luaL_setfuncs`. Он сообщает, как много связанных значений будут иметь функции библиотеки. Начальные значения для этих связанных значений должны находиться на стеке, как и в случае с функцией `lua_pushcclosure`. Таким образом, для создания библиотеки, где функции будут иметь общую таблицу как единственное связанное значение, мы можем использовать следующий код:

```
/* создать таблицу для библиотеки ('lib' - это список ее функций) */
luaL_newlibtable(L, lib);
/* создать разделяемое значение */
lua_newtable(L);
/* добавить функции из списка 'lib' к новой библиотеке, так что */
/* они все будут иметь в качестве связанного значения эту таблицу */
luaL_setfuncs(L, lib, 1);
```

Последний вызов также удаляет таблицу со стека, оставляя там только новую библиотеку.

Упражнения

Упражнение 28.1. Напишите на С функцию `filter`. Она получает на вход список и функцию и возвращает все элементы из заданного списка, для которых функция возвращает истинное значение:

```
t = filter({1, 3, 20, -4, 5}, function (x) return x < 5 end)
-- t = {1, 3, -4}
```

Упражнение 28.2. Измените функцию `l_split` (из листинга 28.2) так, чтобы она могла работать со строками, содержащими нулевой байт. (Кроме остальных изменений, она также должна использовать `memchr` вместо `strchr`.)

Упражнение 28.3. Реализуйте функцию `transliterate` (упражнение 21.3) на С.

Упражнение 28.4. Реализуйте библиотеку с измененной функцией `transliterate` так, что таблица замены не передается как аргумент, а хранится самой библиотекой. Ваша библиотека должна предоставить следующие функции:

```
lib.settrans (table)  -- задать таблицу замены
lib.gettrans ()       -- вернуть таблицу замены
lib.transliterate(s)  -- перевести 's', используя текущую таблицу
```

Используйте реестр для хранения таблицы замены.

Упражнение 28.5. Повторите предыдущее упражнение, используя для хранения таблицы связанное значение.

Упражнение 28.6. Считаете ли вы хорошим дизайном хранить таблицу замены как часть состояния библиотеки, а не передавать ее как параметр?



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, заданных на C. Мы начнем с небольшого примера; на протяжении всей главы мы будем расширять его при помощи метаметодов и других возможностей.

Наш пример будет довольно простым: массив логических (булевых) значений. Такая простая структура выбрана потому, что с ней не связано каких-либо сложных алгоритмов и мы можем полностью сконцентрироваться на API. Тем не менее этот пример все равно является полезным. Конечно, в Lua мы можем использовать таблицы для реализации массивов логических значений. Но в реализации на C мы будем использовать по одному биту на каждый элемент, то есть нам потребуется всего около 3% от той памяти, которая бы понадобилась для соответствующей 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))
```

Константа `BITS_PER_WORD` — это количество бит в беззнаковом целом числе. Макрос `I_WORD` вычисляет слово, которое содержит бит по заданному индексу, а макрос `I_BIT` вычисляет битовую маску для соответствующего бита.

Мы будем представлять наши массивы при помощи следующей структуры:

```
typedef struct NumArray {
    int size;
    unsigned int values[1]; /* изменяемая часть */
} NumArray;
```

Мы объявляем массив `values` с размером в 1, поскольку C89 не позволяет объявлять массивы с размером 0; на самом деле мы будем выделять необходимое число элементов при выделении памяти под наш массив. Следующее выражение вычисляет итоговый размер для нашего битового массива с `n` элементами:

```
sizeof(NumArray) + I_WORD(n - 1)*sizeof(unsigned int)
```

(Мы вычитаем единицу из `n`, поскольку в нашей структуре мы уже выделили место под один элемент.)

29.1. Пользовательские данные (userdata)

Нашей первой задачей является представление структуры `NumArray` в Lua. Lua предоставляет специальный базовый тип для этого: *userdata*. Этот тип соответствует просто области, памяти в которой мы можем хранить что угодно без каких-либо определенных операций.

Функция `lua_newuserdata` выделяет блок памяти заданного размера, помещает соответствующее значение Lua на стек и возвращает адрес выделенного блока:

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

Если по каким-либо причинам вам нужно выделить блок памяти другим образом, то легко можно создать соответствующий объект Lua с размером указателя и запомнить там указатель на выделенный блок. Мы обсудим этот прием в главе 30.

Используя функцию `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; /* инициализируем массив */
    return 1; /* новый объект уже находится на стеке */
}
```


Как только функция `newarray` зарегистрирована в Lua, мы можем создавать новые массивы при помощи выражений вида: `a=array.new(1000)`.

Для того чтобы записать значение в наш массив, мы будем использовать выражения вида: `array.set(a, index, value)`. Позже мы увидим, как можно использовать метатаблицы для поддержки более традиционного синтаксиса вида `a[index]=value`. В обоих случаях функция, осуществляющая запись элемента в массив, одна и та же. Мы предполагаем, что, как это принято в Lua, индексы начинаются с 1:

```
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); /* установить бит */
    else
        a->values[I_WORD(index)] &= ~I_BIT(index); /* снять бит */
    return 0;
}
```

Поскольку Lua в качестве логического значения может использовать любое значение, то мы используем `luaL_checkany` для того, чтобы убедиться, что для этого параметра есть какое-либо значение. Если мы вызовем `setarray` с некорректными аргументами, то мы получим соответствующие 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 значение по заданному индексу:

```
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 отдельную функцию для того, чтобы возвращать размер 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 понадобится дополнительный код 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;
}
```

Снова мы используем функцию `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 большая проблема с безопасностью. Допустим, пользователь напишет что-то вроде `array.set(io.stdin, 1, false)`. Значение `io.stdin` — это объект типа `userdata` с указателем на `FILE`. Из-за того, что это тоже значение типа `userdata`, то `array.set` воспримет это как допустимый аргумент; в результате мы, скорее всего, получим запись в произвольное место памяти (если

нам очень повезет, то мы получим всего лишь сообщение о записи по недопустимому индексу). Подобное поведение недопустимо для любой библиотеки Lua. Независимо от того, как вы используете библиотеку, мы не должны писать что-то по произвольному адресу памяти (или вызывать падение всего приложения).

Обычным способом отличать один тип объекта `userdata` от другого является задание уникальной метатаблицы для этого типа. Каждый раз, когда мы создаем объект типа `userdata`, мы выставляем ему соответствующую метатаблицу; каждый раз, когда мы получаем объект типа `userdata`, мы проверяем, что у него правильная метатаблица. Поскольку код на Lua не может изменить метатаблицу для объектов типа `userdata`, у нас гарантированно будет все в порядке.

Нам также нужно место для того, чтобы хранить эту метатаблицу так, что мы можем обратиться к ней при создании новых объектов и проверок, имеет ли объект типа `userdata` нужный нам тип. Как мы сказали ранее, есть два варианта для хранения метатаблицы: в реестре или как связанное значение для функций в библиотеке. В Lua принято регистрировать каждый новый тип на C в реестре, используя имя типа как индекс и метатаблицу в качестве соответствующего ему значения. Как и в случае с любыми другими индексами реестра мы должны аккуратно выбирать имя типа, чтобы избежать возможных конфликтов. В нашем примере мы будем использовать для этого имя `"LuaBook.array"`.

Как обычно, вспомогательная библиотека предоставляет нам функции. Этими новыми вспомогательными функциями являются следующие функции:

```
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_newmetatable` создает новую таблицу (которая будет нашей метатаблицей), помещает ее на вершине стека и связывает таблицу с заданным именем в реестре. Функция `luaL_getmetatable` возвращает метатаблицу, связанную с именем `tname` в реестре. Наконец, функция `luaL_checkudata` проверяет, что объект на заданном месте в стеке является объектом типа `userdata` с метатаблицей, соответствующей заданному имени. Он вызывает ошибку, если у объекта другая метатаблица (или ее нет) или это не объект типа `userdata`; в противном случае он возвращает адрес объекта.

Теперь мы можем начать нашу реализацию. Первым шагом будет изменение функции, которая открывает нашу библиотеку. Новая версия должна создать метатаблицу для наших массивов:

```
int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array");
    luaL_newlib(L, arraylib);
    return 1;
}
```

Следующим шагом будет изменить функцию `newarray` таким образом, чтобы она устанавливала метатаблицу для создаваемых массивов:

```
static int newarray (lua_State *L) {
    <как ранее>
    luaL_getmetatable(L, "LuaBook.array");
    lua_setmetatable(L, -2);
    return 1; /* новый объект уже на стеке */
}
```

Функция `lua_setmetatable` снимает таблицу со стека и устанавливает ее как метатаблицу для объекта, находящегося в стеке по заданному индексу. В нашем случае этим объектом является созданный объект типа `userdata`.

Наконец, функции `setarray`, `getarray` и `getsize` должны знать действительно ли они получили допустимый массив в качестве своего первого аргумента. Для упрощения этого задания мы определим следующий макрос:

```
#define checkarray(L) \
    (NumArray *)luaL_checkudata(L, 1, "LuaBook.array")
```

Используя этот макрос, новая реализация `getsize` становится очень простой:

```
static int getsize (lua_State *L) {
    NumArray *a = checkarray(L);
    lua_pushinteger(L, a->size);
    return 1;
}
```

Поскольку `setarray` и `getarray` имеют общий код для проверки индекса как своего второго аргумента, мы вынесем 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");
/* вернуть адрес элемента */
*mask = I_BIT(index);
return &a->values[I_WORD(index)];
}

```

Ниже приводятся получившиеся реализации `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 вы получите соответствующее сообщение об ошибке:

```
error: bad argument #1 to 'get' ('array' expected)
```

29.3. Объектно-ориентированный доступ

Нашим следующим шагом будет преобразовать наш новый тип в объект так, чтобы мы могли работать с ним, используя объектно-ориентированный синтаксис, как показано ниже:

```

a = array.new(1000)
print(a:size())           --> 1000
a:set(10, true)
print(a:get(10))          --> true

```

Вспомним, что `a:size()` это то же самое, что `a.size(a)`. Поэтому мы должны сделать так, чтобы `a.size` возвращало нашу функцию `getsize`. Ключевым механизмом здесь является метаметод `__index`.

Для таблиц Lua вызывает этот метаметод, когда он не может найти значения для заданного ключа. Для объектов типа `userdata` Lua вызывает его при каждом обращении, поскольку у таких объектов 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 мы создаем массив только для того, чтобы получить его метатаблицу, которую мы записываем в `metaarray`. (Мы не можем установить метатаблицу объекта типа `userdata` из Lua, но мы можем получить ее.) Затем мы устанавливаем `metaarray.__index` равной `metaarray`. Тогда когда мы выполняем `a.size`, Lua не может найти ключ `size` в объекте `a`, поскольку это объект типа `userdata`. 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)`, чего мы и хотели.

Конечно, мы можем то же самое сделать и на C. Мы можем сделать даже лучше: теперь, когда массивы являются объектами со своими собственными операциями, нам больше не нужно иметь эти операции в таблице `array`. Единственной функцией из нашей библиотеки, которую мы должны отдать наружу, является функция `new` для создания новых массивов. Все остальные операции будут доступны только как методы. Код на C может их сам зарегистрировать.

Операции `getsize`, `getarray` и `setarray` не изменяются по сравнению с нашим предыдущим подходом. Все, что изменится, — это то как мы их зарегистрируем. Для этого нам нужно изменить код, который открывает библиотеку. Во-первых, нам нужно два отдельных списка функций: один — для обычных функций и один — для 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 собственному полю `__index`, там зарегистрировать все методы и создать и заполнить таблицу `array`:

```
int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array");
    /* metatable.__index = metatable */
    lua_pushvalue(L, -1); /* создаст копию метатаблицы */
    lua_setfield(L, -2, "__index");
    luaL_setfuncs(L, arraylib_m, 0);
    luaL_newlib(L, arraylib_f);
    return 1;
}
```

Здесь мы опять используем `luaL_setfuncs` для того, чтобы записать функции из `arraylib_m` в метатаблицу, которая находится на вершине стека. Затем мы используем `luaL_newlib` для того, чтобы создать новую таблицу и зарегистрировать функции из `arraylib_f` (на самом деле только функцию `new`).

В качестве завершающего штриха мы добавим метод `__tostring` к нашему типу так, что `print(a)` напечатает `"array"` и размер массива в круглых скобках, что-то вроде `"array(1000)"`. Соответствующая функция показана ниже:

```
int array2string (lua_State *L) {
    NumArray *a = checkarray(L);
    lua_pushfstring(L, "array(%d)", a->size);
    return 1;
}
```

Вызов `lua_pushfstring` строит строку и оставляет ее на вершине стека. Мы также должны добавить `array2string` к списку `arraylib_m`, для того чтобы включить ее в соответствующую метатаблицу:

```
static const struct luaL_Reg arraylib_m [] = {
    {"__tostring", array2string},
    <другие методы>
};
```

29.4. Доступ как к обычному массиву

Альтернативой объектно-ориентированному способу записи является обычный способ работы с массивами. Вместо записи `a:get(i)` мы

можем просто записать `a[i]`. В нашем примере это довольно легко сделать, поскольку наши функции `setarray` и `getarray` уже получают свои аргументы в том порядке, в котором они должны передаваться соответствующим метаметодам. Быстрым решением было бы определить эти метаметоды прямо в коде на Lua:

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
metaarray.__len = array.size
```

(Мы должны выполнить этот код для нашей исходной реализации массивов, без модификаций для объектно-ориентированного синтаксиса.) 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 мы этого хотим, то мы можем зарегистрировать эти метаметоды прямо в коде на C. Для этого мы опять 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 версии у нас снова только одна видимая всем функция `new`. Все остальные функции доступны только как метаметоды для соответствующих операций.

29.5. Легкие объекты типа userdata (light userdata)

Тип объектов, которые мы до сих пор использовали, называется *full userdata*. Lua предлагает и другой тип объекта *userdata*, называемый легким, – *light userdata*.

Такие объекты представляют собой просто указатель в C (то есть значение типа `void*`). Это значение, а не объект; мы не создаем их (так же, как мы не создаем чисел). Для того чтобы поместить такой объект на стек, мы вызываем `lua_pushlightuserdata`:

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

Несмотря на общее имя, полноценные и «легкие» объекты типа *userdata* на самом деле сильно отличаются. Легкие объекты – это не буферы, а просто указатели. У них нет метаблиц. Как и числа, они не управляются сборщиком мусора.

Иногда мы используем легкий вариант как дешевую альтернативу полноценным объектам типа *userdata*. Однако это не их типичное использование. Во-первых, у легких объектов нет метаблиц, поэтому мы не можем узнать их тип. Во-вторых, несмотря на свое название полноценные объекты типа *userdata* довольно дешевы. Они добавляют совсем немного расходов, по сравнению с вызовом `malloc`.

Настоящее использование легких объектов идет из равенства. Полноценный объект типа *userdata* равен только самому себе. Легкий объект представляет собой просто указатель. И как таковой он равен любому другому объекту типа *userdata*, представляющему тот же самый указатель. Таким образом мы можем использовать легкие объекты типа *userdata* для того, чтобы объекты C были внутри Lua.

Мы уже видели типичное использование легких объектов как ключей в реестре (см. раздел 28.3). Там равенство легких объектов было крайне важно. Каждый раз, когда мы помещаем легкий объект на стек при помощи `lua_pushlightuserdata`, мы получаем то же значение Lua и, соответственно, тот же элемент в реестре.

Другим типичным сценарием является необходимость получить полноценный объект типа *userdata* по его адресу в C. Допустим, мы организуем связь между Lua и оконной системой. Тогда мы можем использовать полноценные объекты типа *userdata* для представления окон. Каждый такой объект содержит или всю структуру, представляющую окно, или только указатель на структуру, созданную системой. Когда случается событие внутри окна (например, нажатие кнопки

мышь), система вызывает соответствующий обработчик, идентифицируя окно по его адресу. Для того чтобы передать обработчик Lua, мы должны найти объект типа `userdata`, представляющий данное окно. Для того чтобы его найти, мы можем использовать таблицу, где индексами являются легкие объекты, содержащие адреса окон, а значениями являются полноценные объекты типа `userdata`, представляющие соответствующие окна. Если у нас есть адрес окна, мы помещаем его на стек как легкий объект типа `userdata` и используем его как индекс в таблице. (Скорее всего, эта таблица должна иметь слабые значения. Иначе они никогда не будут собраны сборщиком мусора.)

Упражнения

Упражнение 29.1. Измените реализацию `setarray` так, чтобы она принимала на вход только булевы значения.

Упражнение 29.2. Мы можем рассматривать булевский массив как множество целых чисел (индексы, которым соответствуют истинные значения в массиве). Добавьте к реализации булевых массивов функции, которые вычисляют объединение и пересечение двух массивов. Эти функции должны получать на вход два массива и возвращать новый массив, не изменяя входных массивов.

Упражнение 29.3. Измените реализацию метаметода `__tostring` так, чтобы он показывал полное содержимое массива каким-либо образом. Используйте буферы (см. раздел 28.2) для создания итоговой строки.

Упражнение 29.4. На основе примера с булевыми массивами реализуйте библиотеку на C для работы с массивами целых чисел.



ГЛАВА 30

Управление ресурсами

В нашей реализации булевых массивов из предыдущей главы мы не беспокоились об управлении ресурсами. Эти массивы используют только память. Каждый объект типа `userdata`, представляющий массив, имеет свой блок памяти, которым управляет Lua. Когда массив становится мусором (то есть никто не хранит на него ссылок), Lua со временем соберет его и освободит занимаемую память.

Однако жизнь не всегда так легка. Иногда объекту нужны другие ресурсы, кроме памяти, такие как дескрипторы файлов, указатели на окна и т. п. (часто эти ресурсы также являются памятью, но ей управляет другая часть системы). В подобных случаях, когда объект становится мусором, необходимо как-то освободить эти ресурсы.

Как мы уже видели в разделе 17.6, Lua предоставляет финализаторы в виде метаметода `__gc`. Для того чтобы показать использование этого метаметода в C, мы реализуем две библиотеки на C, предоставляющие доступ к внешним ресурсам. Первый пример – это другая реализация функции для обхода содержимого каталога. Второй (и более сложный) пример – это использование библиотеки *Expat* для разбора XML-файлов.

30.1. Итератор по каталогу

В разделе 27.1 мы реализовали функцию `dir`, которая возвращала таблицу со всеми файлами из заданного каталога. Наша новая реализация вернет итератор, который возвращает новый файл при каждом вызове. Используя эту реализацию, мы можем перебрать содержимое каталога при помощи цикла, как показано ниже:

```
for fname in dir.open(".") do
    print(fname)
end
```

Для того чтобы в C перебрать содержимое каталога, нам нужна структура `DIR`. Эти структуры создаются при помощи вызова `opendir`

и уничтожаются при помощи вызова `closedir`. Наша предыдущая реализация функции `dir` хранила эту структуру как локальную переменную и освобождала при получении имени последнего файла. Наша новая реализация не может хранить `DIR` в локальной переменной, поскольку эта структура будет нужна на протяжении целого ряда вызовов. Более того, мы не можем уничтожить ее при получении имени последнего файла, поскольку программа может преждевременно выйти из цикла, и в этом случае мы никогда не дойдем до последнего файла. Поэтому для того, чтобы гарантировать, что эта структура будет всегда освобождена, нам нужно хранить ее адрес в объекте типа `userdata` и использовать метаметод `__gc` для освобождения этой структуры.

Несмотря на свою центральную роль в нашей реализации, этот объект, представляющий каталог, не обязательно должен быть виден из Lua. Функция `dir` возвращает итерирующую функцию; это все, что Lua видит. Каталог может быть связанным значением этой итерирующей функции. В этом случае итерирующая функция будет иметь непосредственный доступ к этой структуре, но код на Lua к ней доступа не имеет (и ему это не нужно).

Итого нам нужны три функции на C. Во-первых, нам нужна `dir.open` – функция, которую Lua вызывает для создания итераторов; она должна создать структуру `DIR` и замыкание итерирующей функции с этой структурой (как связанного значения). Во-вторых, нам нужна итерирующая функция. В-третьих, нам нужен метаметод `__gc`, который освобождает созданную структуру `DIR`. Как обычно, нам также понадобится функция для начальной настройки, например создания и инициализации метатаблицы для каталога.

Давайте начнем наш код с функции `dir.open`, показанной в листинге 30.1. Важным моментом является то, что эта функция должна создать объект `userdata` перед открытием каталога. Иначе если он сперва откроет каталог, а затем вызов `lua_newuserdata` приведет к ошибке при работе с памятью, то происходит утечка памяти, так как созданную структуру никто не освободит. При правильном порядке структура `DIR`, как только она создана, сразу же привязывается к объекту `userdata`; что бы не случилось после, метаметод `__gc` со временем освободит эту структуру.

Следующая функция – это `dir.iter` (листинг 30.2), сам итератор. Ее код довольно прост. Она получает адрес структуры `DIR` из связанного с ней значения и вызывает `readdir` для получения следующего значения.

Функция `dir_gc` (также в листинге 30.2) – это метаметод `__gc`. Он освобождает созданную структуру `DIR`, но нужно быть аккуратным: поскольку объект `userdata` создается перед открытием каталога, но даже если функция `opendir` вернет ошибку, то объект `userdata` все равно будет создан. Поэтому нам необходимо проверить, что есть, что закрывать.

Последняя функция в листинге 30.2 – это `luaopen_dir`, функция, которая открывает нашу библиотеку.

В полном примере есть одна тонкость. Сперва может показаться, что функция `dir_gc` должна знать действительно ли ее аргумент является каталогом. В противном случае пользователь может вызвать ее с другим типом объекта `userdata` (например, файл), что приведет к серьезной ошибке. Однако у программы на Lua нет способа обратиться к этой функции: она хранится как метатаблица каталогов, которые, в свою очередь, хранятся как связанные с итерирующей функцией значения. Программы на Lua никак не могут обратиться к объектам этого типа.

Листинг 30.1. Функция `dir.open`

```
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include "lua.h"
#include "luaolib.h"
/* объявление итерирующей функции */
static int dir_iter (lua_State *L);
static int l_dir (lua_State *L) {
    const char *path = luaL_checkstring(L, 1);
    /* создать объект userdata для хранения адреса структуры DIR */
    DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));
    /* задать метатаблицу */
    luaL_getmetatable(L, "LuaBook.dir");
    lua_setmetatable(L, -2);
    /* пытаемся открыть каталог */
    *d = opendir(path);
    if (*d == NULL) /* ошибка при открытии каталога? */
        luaL_error(L, "cannot open %s: %s", path, strerror(errno));
    /* создаем и возвращаем итерирующую функцию;
    ее связанным значением является объект userdata,
    уже находящийся на стеке */
    lua_pushcclosure(L, dir_iter, 1);
    return 1;
}
```

Листинг 30.2. Другие функции в библиотеке `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 нет значений */
}

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");
    /* задать поле __gc */
    lua_pushcfunction(L, dir_gc);
    lua_setfield(L, -2, "__gc");
    /* создать библиотеку */
    luaL_newlib(L, dirlib);
    return 1;
}
```

30.2. Парсер XML

Теперь мы обратимся к упрощенной реализации библиотеки для связи между Lua и библиотекой Expat, которую мы назовем `lxp`. Expat – это открытый парсер XML 1.0, написанный на C. Он реализует SAX, то есть *простой API для XML* (simple API for XML). SAX – это событийно-ориентированный API. Это значит, что SAX-парсер читает XML-документ и по мере чтения сообщает приложению, что он находит при помощи задаваемых пользователем *функций-обработчиков* (callback). Например, если мы хотим, чтобы Expat разобрал строку вида `<tag cap="5">hi</tag>`, то он создаст три события: событие начала, когда он читает строку `<tag cap="5">`; событие текста, когда он читает `hi`, и событие конца элемента, когда он читает `</tag>`. Каждое из этих событий вызывает соответствующий обработчик в приложении.

Здесь мы не будем покрывать всю библиотеку Expat. Мы сосредоточимся только на тех частях, которые показывают новые приемы взаимодействия с Lua. Хотя Expat обрабатывает более дюжины различных событий, мы рассмотрим только те три события, которые мы увидели в предыдущем примере (начало элемента, конец элемента и текст)¹.

Та часть API Expat, которая нам нужна, довольно невелика. Во-первых, нам нужны функции для создания и уничтожения парсера:

```
XML_Parser XML_ParserCreate (const char *encoding);  
void XML_ParserFree (XML_Parser p);
```

Аргумент `encoding` не обязателен, мы будем вместо него передавать `NULL`.

После того как у нас есть парсер, мы должны зарегистрировать наши функции-обработчики:

```
void XML_SetElementHandler(XML_Parser p,  
                           XML_StartElementHandler start,  
                           XML_EndElementHandler end);  
void XML_SetCharacterDataHandler(XML_Parser p,  
                                 XML_CharacterDataHandler hnd1);
```

Первая функция задает обработчики для событий начала и конца элемента. Вторая функция задает обработчик для текста.

Все обработчики получают в качестве своего первого аргумента некоторый указатель. Обработчик начала 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, завершенный `NULL`, где каждая пара последовательных строк содержит атрибут и его значение. Обработчик конца элемента получает только один дополнительный элемент – имя тега:

```
typedef void (*XML_EndElementHandler) (void *uData,  
                                       const char *name);
```

Наконец, обработчик текста получает в качестве дополнительного параметра сам текст. Строка текста не завершена нулевым байтом, и для нее явно передается длина:

```
typedef void (*XML_CharacterDataHandler) (void *uData,  
                                          const char *s,  
                                          int len);
```

¹ Пакет LuaExpat предоставляет почти полный интерфейс к Expat.

Для того чтобы передать текст в Expat для разбора, мы используем следующую функцию:

```
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 такого вызова isLast сообщает Expat, был ли переданный фрагмент последним в документе. Обратите внимание, что каждый фрагмент текста не обязательно должен завершаться нулевым байтом, мы явно передаем его длину. Функция XML_Parse возвращает нуль в случае ошибки. (Expat также предоставляет функции для получения информации об ошибке, но для простоты мы не будем их здесь рассматривать.)

Последней функцией, которая нам нужна от Expat, является функция, которая позволяет задать указатель, который будет передаваться обработчикам:

```
void XML_SetUserData (XML_Parser p, void *uData);
```

Теперь давайте посмотрим, как мы можем использовать эту библиотеку в Lua. Первый подход самый простой: давайте просто дадим доступ ко всем этим функциям из Lua. Более удачным будет адаптировать эту функциональность для Lua. Например, поскольку Lua – нетиповой язык (точнее, язык без строгой типизации), то нам не нужны различные функции для каждого типа обработчика. Более того, мы можем вообще избежать регистрации обработчиков. Вместо этого мы создадим парсер, передадим ему таблицу обработчиков, каждый с подходящим ключом. Например, если мы хотим напечатать структуру документа, то мы можем использовать следующую таблицу обработчиков:

```
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,
}
```

Если мы на вход дадим строку "<to> <yes/> </to>", то эти обработчики сгенерируют следующий вывод:


```

+ to
+ yes
- yes
- to

```

При таком API нам не нужны функции для работы с обработчиками. Мы работаем с ними непосредственно в таблице обработчиков. Таким образом, весь API будет состоять всего из трех функций: одна для создания парсеров, одна для обработки фрагмента текста и одна для уничтожения парсера. На самом деле мы реализуем две последние функции как методы парсера. В результате мы приходим к следующему типичному использованию нашего API:

```

local lxp = require"lxp"
p = lxp.new(callbacks)
for l in io.lines() do
    assert(p:parse(l))
    assert(p:parse("\n"))
end
assert(p:parse())
p:close()

```

-- создать новый парсер
-- обрабатываем входные строки
-- разобрать строку
-- добавить '\n'

-- завершить документ

Давайте теперь обратимся к реализации. Нашим первым решением будет то, как мы будем представлять наш парсер в Lua. Вполне естественно использовать для этого объект типа `userdata`, но что нам нужно поместить внутрь него? Как минимум нам понадобятся сам парсер и таблица обработчиков. Мы не можем запомнить таблицу внутри объекта типа `userdata` (или внутри структуры C), но Lua позволяет каждому объекту типа `userdata` иметь *пользовательское значение* (user value), которое может быть любой таблицей Lua². Мы должны также запомнить состояние Lua в объект парсера, поскольку все, что получает обработчик Expat, – это сам парсер, а для того чтобы вызвать Lua, нам нужно это состояние. Поэтому мы будем использовать следующее определение парсера:

```

#include <stdlib.h>
#include "expat.h"
#include "lua.h"
#include "lauxlib.h"
typedef struct lxp_userdata {
    XML_Parser parser; /* соответствующий парсер Expat */
    lua_State *L;
} lxp_userdata;

```

² В Lua 5.1 окружение объекта `userdata` выполняет роль пользовательского значения.

Следующим нашим шагом будет создание функции, которая создаст парсеры, `lxp_make_parser`. Ее код приведен в листинге 30.3. Эта функция состоит из четырех важных шагов:

- Ее первый шаг следует стандартному шаблону: сначала создается объект `userdata`; затем он инициализируется соответствующими значениями, и, наконец, для него задается метатаблица. Причина подобной инициализации следующая: если в ходе инициализации возникает какая-либо ошибка, то необходимо, чтобы финализатор (метаметод `__gc`) нашел наши данные целостными.
- На шаге 2 функция создает Expat-парсер, запоминает его в `userdata`-объекте и проверяет ошибки.
- Шаг 3 проверяет, что первым аргументом функции действительно является таблица (таблица обработчиков), и запоминает ее как пользовательское значение для `userdata`-объекта.
- Последний шаг инициализирует парсер Expat. Наш `userdata`-объект задается в качестве указателя, который будет передаваться в обработчики, также задаются функции-обработчики. Обратите внимание, что эти обработчики одни и те же для всех парсеров; в конце концов, на C нельзя динамически построить функцию. Вместо этого фиксированные функции, используя таблицу обработчиков, решают, какие функции на Lua следует вызвать.

Листинг 30.3. Функция для создания парсеров XML

```
/* описания функций-обработчиков */
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) создать объект-парсер */
    lxp_userdata *xpu = (lxp_userdata *)
        lua_newuserdata(L,
                       sizeof(lxp_userdata));
    /* инициализировать его на случай ошибки */
    xpu->parser = NULL;
    /* задать для него метатаблицу */
    luaL_getmetatable(L, "Expat");
    luaL_setmetatable(L, -2);
```

```

/* (2) создать парсер Expat */
p = xpu->parser = XML_ParserCreate(NULL);
if (!p)
    luaL_error(L, "XML_ParserCreate failed");
/* (3) проверить и сохранить таблицу обработчиков */
luaL_checktype(L, 1, LUA_TTABLE);
lua_pushvalue(L, 1); /* поместить таблицу на стек */
lua_setuservalue(L, -2);
/* (4) сконфигурировать парсер Expat */
XML_SetUserData(p, xpu);
XML_SetElementHandler(p, f_StartElement,
                      f_EndElement);
XML_SetCharacterDataHandler(p, f_CharData);
return 1;
}

```

Листинг 30.4. Функция для разбора фрагмента текста

```

static int lxp_parse (lua_State *L) {
    int status;
    size_t len;
    const char *s;
    lxp_userdata *xpu;
    /* получить и проверить первый аргумент */
    xpu = (lxp_userdata *)luaL_checkudata(L, 1, "Expat");
    /* проверить, что он не закрыт */
    luaL_argcheck(L, xpu->parser != NULL, 1, "parser is closed");
    /* получить второй аргумент (строку) */
    s = luaL_optlstring(L, 2, NULL, &len);
    /* поместить таблицу обработчиков по индексу 3 в стеке */
    lua_settop(L, 2);
    lua_getuservalue(L, 1);
    xpu->L = L; /* задать состояние Lua */
    /* вызывать Expat для разбора строки */
    status = XML_Parse(xpu->parser, s, (int)len, s == NULL);
    /* вернуть код ошибки */
    lua_pushboolean(L, status);
    return 1;
}

```

Следующим шагом является метод для разбора текста `lxp_parse` (листинг 30.4), который разбирает фрагмент данных XML. Он получает два аргумента: парсер (*self* в методе) и необязательный фрагмент XML. Когда он вызывается без данных, он сообщает Expat, что больше частей нет.

Когда `lxp_parse` вызывает `XML_Parse`, то она вызовет обработчики для тех элементов, которые она найдет в переданном фрагменте текс-

та. Этим обработчикам понадобится доступ к таблице обработчиков, поэтому `lxp_parse` помещает 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 переданный фрагмент текста последним. Когда мы вызываем `parse` без аргументов, `s` будет равно `NULL`, и этот последний аргумент примет истинное значение.

Теперь давайте обратим наше внимание на функции-обработчики `f_StartElement`, `f_EndElement` и `f_CharData`. Все эти функции обладают одинаковой структурой: каждая из них проверяет, есть ли в таблице обработчиков обработчик для данного события, и если такой обработчик присутствует, то подготавливает аргументы и затем вызывает этот обработчик.

Давайте посмотрим на 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`, когда создавали наш парсер. После получения состояния Lua обработчик может обратиться к таблице обработчиков на стеке по индексу 3, заданному `lxp_parse`, и 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;
    /* получить обработчик */
    lua_getfield(L, 3, "CharacterData");
    if (lua_isnil(L, -1)) { /* обработчика нет? */
        lua_pop(L, 1);
        return;
    }
    lua_pushvalue(L, 1); /* поместить парсер ('self') на стек */
    lua_pushlstring(L, s, len); /* поместить строку на стек */
    lua_call(L, 2, 0); /* вызвать обработчик */
}
```

Обработчик `f_EndElement` довольно похож на `f_CharData`; обратитесь к листингу 30.6. Он также вызывает соответствующий обработчик на Lua с двумя аргументами – парсером и именем тега (опять строкой, на этот раз завершенной нулевым байтом).

Листинг 30.6. Обработчик конца элемента

```
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); /* поместить парсер ('self') на стек */
    lua_pushstring(L, name); /* поместить тег на стек */
    lua_call(L, 2, 0); /* вызвать 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 необходимо перевести список атрибутов в Lua. Он использует вполне естественный перевод, строя таблицу, которая сопоставляет именам атрибутов их значения. Например, для короткого тега, показанного ниже

```
<to method="post" priority="high">
```

генерируется следующая таблица атрибутов:

```
{method = "post", priority = "high"}
```

Листинг 30.7. Обработчик начала элемента

```
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); /* поместить парсер ('self') на стек */
    lua_pushstring(L, name); /* поместить имя тега на стек */
    /* создать и заполнить 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); /* вызвать обработчик */  
}
```

Последний метод для парсеров – это `close`, показанный в листинге 30.8. Когда мы закрываем парсер, мы должны освободить все его ресурсы, а именно структуру `Expat`. Помните, что из-за ошибок при создании у парсера этой структуры может и не быть. Обратите внимание, как мы поддерживаем парсер в последовательном состоянии по мере того, как мы его закрываем, так что не будет никаких проблем, если мы попытаемся снова его закрыть или когда сборщик мусора его финализирует. Это гарантирует, что каждый парсер со временем освободит свои ресурсы, даже если программист не закрыл его.

Листинг 30.8. Метод для закрытия XML-парсера

```
static int lxp_close (lua_State *L) {  
    lxp_userdata *xpu =  
        (lxp_userdata *)luaL_checkudata(L, 1, "Expat");  
    /* освободить парсер Expat (если он есть) */  
    if (xpu->parser)  
        XML_ParserFree(xpu->parser);  
    xpu->parser = NULL; /* если мы снова его закроем */  
    return 0;  
}
```

Завершающий шаг показан в листинге 30.9: он показывает функцию `luaopen_lxp`, которая открывает библиотеку, совмещая вместе все ранее рассмотренные функции. Мы здесь используем ту же схему, которую использовали для объектно-ориентированного булевого массива в разделе 29.3: мы создаем метаблицу, устанавливаем ее поле `__index` на нее саму и помещаем все методы внутрь нее. Для этого нам понадобится список со всеми 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. Инициализирующий код 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) {  
    /* создать метатаблицу */  
    luaL_newmetatable(L, "Expatriate");  
    /* metatable.__index = metatable */  
    lua_pushvalue(L, -1);  
    lua_setfield(L, -2, "__index");  
    /* зарегистрировать методы */  
    luaL_setfuncs(L, lxp_meths, 0);  
    /* зарегистрировать функции (только lxp.new) */  
    luaL_newlib(L, lxp_funcs);  
    return 1;  
}
```

Упражнения

Упражнение 30.1. Модифицируйте функцию `dir_iter` так, чтобы она закрывала структуру `DIR`, когда она доходит до конца каталога. При этом изменении программе не нужно ждать сборки мусора для освобождения ресурса, который ей больше не нужен.

(Когда вы закрываете каталог, то вы должны установить адрес, записанный в объекте `userdata` в `NULL`, чтобы сообщить финализатору, что каталог уже закрыт. Также функция `dir_iter` перед использованием каталога должна проверять, что он не закрыт.)

Упражнение 30.2. В примере с библиотекой `lxp` обработчик начала элемента получает таблицу с атрибутами элемента. В этой таблице порядок, в котором элементы были заданы внутри элемента, уже потерян. Как вы можете передать эту информацию в обработчик?

Упражнение 30.3. В примере с библиотекой `lxp` мы использовали пользовательское значение для того, чтобы связать таблицу обработчиков с соответствующим объектом `userdata`, представляющим парсер. Этот выбор создал небольшую проблему, поскольку то, что получают обработчики на C, — это структура `lxp_userdata`, и эта структура не предоставляет прямого доступа к данной таблице. Мы решили эту проблему путем сохранения таблицы обработчиков на заданном месте в стеке во время разбора каждого фрагмента.

Другим решением может быть связать таблицу обработчиков с объектом `userdata` при помощи ссылок (раздел 28.3): мы

создаем ссылку на таблицу обработчиков и запоминаем эту ссылку (целое число) в структуре `lxp_userdata`. Реализуйте этот вариант. Не забудьте освободить ссылку при закрытии парсера.



ГЛАВА 31

Нити и состояния

Lua не поддерживает настоящую многонитевость, то есть вытесняющие нити, разделяющие общую память. Для этого есть две причины. Первой причиной является то, что такую поддержку не предоставляет ANSI C, и поэтому нет переносимого способа реализовать эту поддержку в Lua. Второй и более серьезной причиной является то, что мы не думаем, что многонитевость – это хорошая идея для Lua.

Многонитевость была разработана для низкоуровневого программирования. Механизмы синхронизации вроде семафоров и мониторов была предложены для операционных систем (и опытных программистов), а не для приложений. Крайне сложно находить и исправлять ошибки, связанные с многонитевостью, и некоторые из них могут привести к дырам в безопасности. Также многонитевость может вести к серьезным проблемам с быстродействием из-за необходимости синхронизации в ряде критических мест программы, таких как выделение памяти.

Проблемы с многонитевостью возникают из-за комбинации вытесняющих нитей и разделяемой памяти, поэтому мы можем избежать их, либо не используя вытесняющих нитей, либо не используя разделяемую память. Lua предлагает поддержку и для того, и для другого. Нити Lua (также известные как сопрограммы) не являются вытесняющими и поэтому избегают проблем, связанных с непредсказуемым переключением нитей. Состояния Lua не имеют общей памяти, поэтому образуют хорошую базу для параллельных вычислений. В этой главе мы рассмотрим оба этих варианта.

31.1. Многочисленные нити

Нить – это суть сопрограммы в Lua. Мы рассматриваем сопрограмму как нить и удобный интерфейс, или мы можем рассматривать нить как сопрограмму с низкоуровневым API.

С точки зрения C, может быть полезно думать о нити как о стеке — чем на самом деле и является нить с точки зрения реализации. Каждый стек хранит информацию о текущих вызовах нити, а также параметрах и локальных переменных каждого вызова. Другими словами стек, содержит всю информацию, которая нужна нити для продолжения ее выполнения. Поэтому много нитей означает много независимых стеков.

Когда мы вызываем большинство функций из Lua-C API, то эти функции работают с определенным стеком. Например, `lua_pushnumber` должна поместить число на определенный стек; также `lua_pcall` нужен стек для вызова. Как Lua знает, какой стек следует использовать? Секрет заключается в том, что тип `lua_State`, первый аргумент всех этих функций, представляет собой не только состояние Lua, но и нить внутри этого состояния. (Многие считают, что этот тип должен называться `lua_Thread`.)

Когда вы создаете новое состояние, Lua автоматически создает нить внутри этого состояния и возвращает `lua_State`, представляющее эту нить. Эта *главная нить* уничтожается вместе с состоянием, когда вы вызываете `lua_close`.

Вы можете создавать другие нити внутри состояния при помощи `lua_newthread`:

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

Эта функция возвращает указатель на `lua_State`, представляющий новую нить, и также помещает новую нить на стек как значение типа `thread`. Например, после выполнения оператора

```
L1 = lua_newthread(L);
```

у нас будет две нити, `L1` и `L`, обе ссылающиеся внутри на одно и то же состояние Lua. Каждая нить обладает собственным стеком. Новая нить `L1` начинает с пустого стека; старая нить `L` имеет новую нить на вершине стека:

```
printf("%d\n", lua_gettop(L1)); --> 0  
printf("%s\n", luaL_typename(L, -1)); --> thread
```

За исключением главной нити, нити могут быть собраны сборщиком мусора, как и любой другой объект Lua. Когда вы создаете новую нить, то она помещается на стек, что гарантирует, что эта нить не является мусором. Вы никогда не должны использовать нить, которая не привязана к состоянию. (Главная нить 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, давайте рассмотрим следующий фрагмент:

```
lua_State *L1 = lua_newthread (L);  
lua_pop(L, 1); /* L1 теперь является для Lua мусором */  
lua_pushstring(L1, "hello");
```

Вызов `lua_pushstring` может вызвать сборщик мусора и собрать `L1` (приводя к ошибке приложения), несмотря на то что `L1` все еще используется. Для того чтобы избежать этого, всегда храните ссылки на нити, которые вы используете, например на стеке привязанной нити или в реестре.

Как только у нас появляется новая нить, мы сразу же можем начать ее использовать, как и главную нить. Мы можем помещать значения на ее стек и снимать значения с ее стека, можем использовать ее для вызова функций и т. п. Например, следующий код выполняет вызов `f(5)` на новой нити и затем помещает результат в старую нить:

```
lua_getglobal(L1, "f"); /* считаем, что есть глобальная 'f' */  
lua_pushinteger(L1, 5);  
lua_call(L1, 1, 1);  
lua_xmove(L1, L, 1);
```

Функция `lua_xmove` перемещает значение Lua между двумя стеками одного и того же состояния. Вызов типа `lua_xmove(F, T, n)` снимет `n` элементов со стека `F` и поместит их на стек `T`.

Однако для этих целей нам не нужна новая нить; мы можем легко использовать главную нить. Основной целью использования нескольких нитей является реализация сопрограмм так, что мы можем приостанавливать их выполнение и продолжать его снова. Для этого нам нужна функция `lua_resume`:

```
int lua_resume (lua_State *L, lua_State *from, int nargs);
```

Для начала выполнения сопрограммы мы используем `lua_resume` так же, как мы используем `lua_pcall`: мы помещаем функцию на стек, помещаем на стек ее аргументы и вызываем `lua_resume`, передавая в `nargs` число аргументов. (Параметр `from` — это нить, которая выполняет вызов.) Это очень похоже на `lua_pcall`, однако есть три отличия. Во-первых, `lua_resume` не содержит параметра для числа желаемых результатов; она всегда возвращает все результаты вызванной функции. Во-вторых, у нее нет параметра для обработчика ошибок; ошибка не раскручивает стек, поэтому вы можете потом его изучить. В-третьих, если функция приостанавливает свое выполнение (при помощи `yield`), то `lua_resume` возвращает специальный

код `LUA_YIELD` и оставляет нить в таком состоянии, что мы можем возобновить ее выполнение позже.

Когда `lua_resume` возвращает `LUA_YIELD`, видимая часть стека нити содержит только значения, переданные `yield`. Вызов `lua_gettop` вернет число этих значений. Для того чтобы перенести эти значения на другую нить, мы можем использовать `lua_xmove`.

Для того чтобы продолжить выполнение приостановленной нити, мы снова зовем `lua_resume`. В этом случае Lua считает, что все значения, находящиеся на стеке, являются значениями, возвращенными `yield`. Например, если вы не трогаете нить стека между возвращением от предыдущего `lua_resume` и следующим `lua_resume`, то `yield` вернет именно те значения, с которыми он был вызван.

Обычно мы запускаем сопрограмму с функцией на Lua в качестве тела. Эта функция на Lua может вызывать другие функции, и любая из этих функций может вызывать `yield`, завершая вызов `lua_resume`. Например, рассмотрим следующие определения:

```
function foo (x) coroutine.yield(10, x) end
function fool (x) foo(x + 1); return 3 end
```

Теперь мы выполним следующий код на C:

```
lua_State *L1 = lua_newthread(L);
lua_getglobal(L1, "fool");
lua_pushinteger(L1, 20);
lua_resume(L1, L, 1);
```

Вызов `lua_resume` вернет `LUA_YIELD`, для того чтобы сообщить, что нить приостановлена. В этот момент стек `L1` содержит значения, переданные `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 снова вызовем `lua_resume`, нить продолжит выполнение оттуда, где она остановилась (вызов `yield`). Оттуда `foo` вернет управление `fool`, 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 нормальный возврат.

Сопрограммы также могут вызывать функции на C, которые могут вызывать другие функции на Lua. Мы уже обсуждали, как использовать продолжения, для того чтобы позволить этим функциям на Lua

вызвать `yield` (раздел 27.2). Функция на С сама также может вызвать `yield`. В этом случае вы должны представить функцию-продолжение, которая будет вызвана при продолжении выполнения. На С следующая функция выполняет роль `yield`:

```
int lua_yieldk (lua_State *L, int nresults, int ctx,
               lua_CFunction k);
```

Мы должны всегда использовать эту функцию в операторе возврата, как показано ниже:

```
static int myCfunction (lua_State *L) {
    ...
    return lua_yieldk(L, nresults, ctx, k);
}
```

Этот вызов немедленно приостанавливает выполняющуюся программу. Параметр `nresults` – это количество значений на стеке, которые нужно вернуть соответствующему `lua_resume`; `ctx` – это контекст, который будет передан продолжению, и `k` – это функция-продолжение. Когда сопрограмма продолжит выполнение, управление переходит к функции-продолжению `k`. После вызова `lua_yieldk` функция `myCfunction` не может больше ничего сделать; она должна делегировать всю дальнейшую работу своему продолжению.

Давайте рассмотрим гипотетический пример. Пусть мы хотим написать функцию, которая читает некоторые данные, вызывая `yield`, когда данные не готовы. Мы можем написать эту функцию на С 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, то она читает и возвращает их. В противном случае она вызывает `lua_yieldk`. Когда нить продолжит выполнение, то она вызовет функцию-продолжение. В этом примере функция-продолжение – это сама `prim_read`, поэтому нить будет снова и снова вызывать ее для чтения данных. (Этот шаблон, когда вызывающая `lua_yieldk` функция сама является своим продолжением, не является редким.)

Если функции на С нечего делать после вызова `lua_yieldk`, то она может вызвать `lua_yieldk` без функции-продолжения или использовать макрос `lua_yield`:

```
return lua_yield(L, nres);
```

После этого вызова, когда нить продолжит свое выполнение, перейдет 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. Различные состояния Lua никак не зависят друг от друга. И у них нет никаких общих данных. Это значит, что независимо от того, что происходит в одном состоянии Lua, оно никак не может «навредить» другому состоянию. Также это значит, что различные состояния Lua не могут между собой общаться; для этого мы должны использовать специальный код на C. Например, если у нас есть два состояния `L1` и `L2`, то следующая команда – поместить на вершину стека `L2` значение с вершины стека в `L1`:

```
lua_pushstring(L2, lua_tostring(L1, -1));
```

Поскольку данные должны проходить через C, то различные состояния в Lua могут обмениваться между собой только теми типами данных, которые представимы в C, вроде строк и чисел. Другие типы, например таблицы, для переноса должны быть сериализованы.

В системах, которые предлагают многопоточность, интересным архитектурным решением было бы создать по отдельному состоянию Lua на каждую нить. В результате мы получаем нити, которые ведут себя наподобие процессов в UNIX, то есть у нас есть параллельность без разделяемой (общей) памяти. В этом разделе мы построим прототип приложения, использующего данный подход. Для этой реализации я буду использовать нити POSIX (`pthread`). Поскольку я использую только самые базовые возможности, то будет несложно перенести этот код на другие системы с поддержкой многопоточности.

Система, которую мы хотим построить, очень проста. Ее целью является показать использование нескольких состояний Lua в контексте многопоточности. После того как она будет готова, вы сами можете добавлять к ней дополнительные возможности. Мы назовем нашу библиотеку `lproc`. Она предлагает всего четыре функции:

- `lproc.start(chunk)` начинает новый процесс для выполнения заданного блока кода (`chunk`). Библиотека реализует процесс в Lua как нить на C и связанное состояние Lua.

- `lproc.send(channel, val1, val2, ...)` посылает заданные значения (которые должны быть строками) в заданный канал, идентифицируемый своим именем (строкой).
- `lproc.receive(channel)` получает значения из заданного канала.
- `lproc.exit()` завершает процесс. Эта функция нужна только главному процессу. Если этот процесс завершается без вызова `lproc.exit`, то вся программа прекращает свое выполнение без ожидания окончания других процессов.

Библиотека идентифицирует каналы при помощи строк и использует их для того, чтобы сопоставить посылающему получателя. Операция отправки может послать любое количество строковых значений, которые возвращаются соответствующей операцией получения данных. Все взаимодействие синхронно: процесс, посылающий сообщение в канал, ожидает, пока не найдется процесс, читающий из этого канала, в то время, пока процесс, читающий из канала, также ожидает, пока не появится процесс, посылающий в него.

Библиотека `lproc`, как и ее интерфейс, довольно проста. Она использует два кольцевых двухсвязных списка, один для процессов, ожидающих для того, чтобы послать сообщение, а другой – для процессов, ожидающих получить сообщение. Также используется один мютекс для управления доступом к обоим этим спискам. У каждого процесса есть своя *условная переменная* (*condition variable*). Когда процесс хочет послать сообщение в канал, он ищет в списке ожидающих сообщения процесс, ожидающий именно этого канала. Если он находит такой процесс, то он удаляет его из списка ожидающих, переносит значения от себя найденному процессу и сигнализирует остальным процессам. В противном случае он вставляет себя в список посылающих и ожидает своей условной переменной. Получение сообщения ведет себя аналогично.

Главным элементом 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 собой состояние Lua, используемое процессом, и соответствующую C нить, выполняющую данный процесс. Другие поля используются, только когда процесс должен ждать соответствующего `send/receive`. Третье поле `cond` – это условная переменная, которую нить использует для того, чтобы ждать; четвертое поле – это канал, которого процесс ждет; и последние два поля, `previous` и `next`, используются для соединения процесса в двухсвязный список.

Следующий код объявляет два списка ожидающих процессов 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 соответствующая структура `Proc`, и ему нужен доступ к ней всегда, когда его тело вызывает `send` или `receive`. Единственным параметром, который эти функции получают, является соответствующее состояние Lua, поэтому каждый процесс должен запомнить свою структуру `Proc` внутри своего состояния Lua. В нашей реализации каждое состояние Lua хранит соответствующую структуру `Proc` как объект типа `userdata`, связанный с ключом `"_SELF"`. Вспомогательная функция `getself` возвращает состояние `Proc`, соответствующее заданному состоянию:

```
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`, переносит значения из посылающего процесса в получающий:

```
static void movevalues (lua_State *send, lua_State *rec) {
    int n = lua_gettop(send);
    int i;
    for (i = 2; i <= n; i++) /* перенести значения получающему */
        lua_pushstring(rec, lua_tostring(send, i));
}
```

Она переносит получающему все значения со стека посылающего, кроме первого значения, которым является канал.

Листинг 31.1 определяет функцию `searchmatch`, которая обходит список ожидающих в поисках процесса, ожидающего заданного кана-

ла. 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; /* список пуст? */
    do {
        if (strcmp(channel, node->channel) == 0) { /* нашли? */
            /* удалить узел из списка */
            if (*list == node) /* это первый элемент списка? */
                *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; /* не нашли */
}
```

Последняя вспомогательная функция, определенная в листинге 31.2, вызывается, когда процесс не может найти нужного ему другого процесса. В этом случае процесс подключает себя к концу соответствующего списка и ожидает, пока другой процесс не найдет и не разбудит его. (Цикл вокруг `pthread_cond_wait` защищает от случайных пробуждений, которые возможны в нитях POSIX.) Когда процесс будит другой процесс, то он устанавливает поле `channel` у разбуженного процесса в NULL. Поэтому если `p->channel` не равен NULL, то это значит, что никакой другой процесс не разбудил этот процесс, поэтому надо ждать дальше.

Листинг 31.2. Функция для добавления процесса в список ожидающих

```
static void waitonlist (lua_State *L, const char *channel,
    Proc **list) {
    Proc *p = getself(L);
    /* подключить себя к концу списка */
    if (*list == NULL) { /* список пуст? */
        *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 { /* ожидает условной переменной */
        pthread_cond_wait(&p->cond, &kernel_access);
    } while (p->channel);
}

```

Теперь при помощи этих вспомогательных функций мы можем написать `send` и `receive` (листинг 31.3). Функция `send` начинает с проверки канала. Затем она закрывает мьютекс и ищет соответствующего получателя. Если она его находит, то она переносит свои значения этому получателю, помечает получателя как готового к выполнению и будит его. В противном случае она ждет сама. По завершении этой операции она открывает мьютекс и возвращается 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) { /* нашли соответствующего получателя? */
        movevalues(L, p->L); /* перенести значения получателю */
        p->channel = NULL; /* пометить получателя */
        pthread_cond_signal(&p->cond); /* разбудить 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) { /* нашли соответствующего получателя? */
        movevalues(p->L, L); /* перенести значения получателю */
        p->channel = NULL; /* пометить получателя */
        pthread_cond_signal(&p->cond); /* разбудить его */
    }
    else
        waitonlist(L, channel, &waitreceive);
    pthread_mutex_unlock(&kernel_access);
    /* вернуть все значения со стека, кроме канала */
}

```

```
    return lua_gettop(L) - 1;
}
```

Теперь давайте посмотрим, как создавать новые процессы. Новому процессу нужна новая нить POSIX, и новой нити нужно тело для выполнения. Мы определим это тело позже; здесь приводится прототип:

```
static void *ll_thread (void *arg);
```

Листинг 31.4. Функция для создания нового процесса

```
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;
}
```

Для создания и запуска нового процесса системе нужно создать новое состояние Lua, начать новую нить, откомпилировать переданный блок, вызвать его и в конце освободить его ресурсы. Исходная нить выполняет первые три задачи, и новая нить делает остальное. (Для упрощения обработки ошибок система начинает новую нить после того, как она успешно откомпилировала переданный блок.)

Функция `ll_start` создает новый процесс (листинг 31.4). Эта функция создает новое состояние Lua `L1` и компилирует заданный блок в этом новом состоянии. В случае ошибки она сообщает 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` говорит системе, что мы не ожидаем окончательного ответа от этой нити.

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); /* открыть стандартные библиотеки */
}
```

```
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;
}
```

Телом каждой новой нити является функция `ll_thread` (листинг 31.5). Она получает свое состояние Lua (созданное `ll_start`) с уже откомпилированным блоком на стеке. Новая нить открывает стандартные библиотеки Lua, открывает библиотеку `lproc` и затем вызывает свой блок. В конце она освобождает свою условную переменную (которая была создана `luaopen_lproc`) и закрывает свое состояние Lua.

Обратите внимание на использование `luaL_require` для того, чтобы открыть `lproc`¹. Эта функция в чем-то эквивалентна `require`, но вместо поиска загрузчика использует заданную функцию (в нашем случае это `luaopen_lproc`) для того, чтобы открыть библиотеку. После вызова открывающей функции `luaL_requiref` регистрирует результат в таблице `package.loaded`. Если ее последний параметр – это `true`, то она также регистрирует библиотеку в соответствующей глобальной переменной (в нашем случае `lproc`).

Последняя функция в нашем модуле, `exit`, очень проста:

```
static int ll_exit (lua_State *L) {
    pthread_exit(NULL);
    return 0;
}
```

Только главному процессу нужно вызвать эту функцию, когда он завершит выполнение, для того чтобы не прервать немедленно выполнение всей программы.

Нашим последний шагом является определение открывающей функции для модуля `lproc`. Эта функция `luaopen_lproc` (листинг 31.6) должна зарегистрировать функции модуля, но также она должна создать и проинициализировать структуру `Proc` текущего процесса.

Как я сказал ранее, это определение процессов в Lua очень простое. Существует бесконечное число улучшений, которые вы можете реализовать. Здесь я хочу вкратце обсудить некоторые из них.

Первым очевидным улучшением будет заменить линейный поиск процесса, ожидающего заданного канала. Красивой альтернативой

¹ Эта функция появилась в Lua 5.2.

было бы использование хэш-таблицы для поиска канала и использование независимых списков ожидания для каждого канала.

Другое улучшение относится к эффективности создания процесса. Создание нового состояния в Lua – это очень быстрая операция. Однако открытие всех стандартных библиотек уже не так быстро, и большинству процессов, скорее всего, не понадобятся все стандартные библиотеки. Мы можем избежать цены, связанной с открытием библиотеки, при помощи пререгистрации библиотек, как мы обсудили в разделе 15.1. При использовании этого подхода вместо вызова `luaL_requiref` для каждой стандартной библиотеки мы просто помещаем функцию, открывающую библиотеку, в таблицу `package.preload`. Если процесс вызывает `require“lib”`, тогда и только тогда `require` вызовет соответствующую функцию, для того чтобы открыть библиотеку. Функция `registerlib` (листинг 31.7) выполняет эту регистрацию.

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) {
    /* создать собственный управляющий блок */
    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); /* открыть библиотеку */
    return 1;
}
```

Листинг 31.7. Регистрация библиотек по запросу

```
static void registerlib (lua_State *L, const char *name,
                        lua_CFunction f) {
    luaL_getglobal(L, "package");
    luaL_getfield(L, -1, "preload"); /* получить 'package.preload' */
    luaL_pushcfunction(L, f);
    luaL_setfield(L, -2, name); /* package.preload[name] = f */
    lua_pop(L, 2); /* поднять со стека '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); /* удалить результаты 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);  
}
```

Это всегда хорошая идея – открыть основную библиотеку. Вам также понадобится библиотека для работы с пакетами, иначе вы не сможете использовать `require` для загрузки других библиотек. (Вы даже не получите таблицу `package.preload`.) Все другие библиотеки могут быть необязательными. Поэтому вместо вызова `luaL_openlibs` мы подставим свою собственную функцию `openlibs` (также показанную в листинге 31.7) при создании новых состояний. Когда процессу понадобится любая из этих библиотек, он ее явно потребует, и `require` вызовет соответствующую функцию `luaopen_*`.

Другие улучшения включают в себя примитивы для коммуникации. Например, было бы полезным задавать пределы того, как долго `lproc.send` и `lproc.receive` могут ждать. В частности, предел ожидания, равный нулю, сделает эти функции неблокирующими. В нитях POSIX мы можем реализовать эту функциональность при помощи `pthread_cond_timedwait`.

Упражнения

Упражнение 31.1. Как мы уже видели, если функция вызывает `lua_yield` (версия без функции-продолжения), управление передается к функции, которая ее вызвала, когда нить снова продолжит свое выполнение. Какие значения вызывающая функция получит как результаты этого вызова?

Упражнение 31.2. Измените библиотеку `lproc` так, чтобы она могла посылать другие базовые типы, такие как логические значения и числа. (*Подсказка:* вам нужно только изменить функцию `movevalues`.)

Упражнение 31.3. Реализуйте в библиотеке `lproc` неблокирующую функцию `send`.



ГЛАВА 32

Управление памятью

Lua динамически выделяет все свои структуры данных. Все эти структуры растут динамически по мере надобности и со временем уменьшают свой размер 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 приложений может понадобиться адаптация, например для работы в условиях ограниченного объема памяти или для сведения задержек от работы сборщика мусора к минимуму. Lua позволяет осуществлять подобные адаптации сразу на двух уровнях. На нижнем уровне мы можем задать функцию, используемую для выделения памяти. На более высоком уровне мы можем задать некоторые параметры для управления сборщиком мусора или можем даже получить прямой контроль над сборщиком мусора. В этой главе мы рассмотрим оба этих варианта.

32.1. Функция для выделения памяти

Ядро Lua не предполагает ничего о выделении памяти. Для выделения памяти оно не вызывает ни `malloc`, ни `realloc`. Вместо этого оно осуществляет все свое выделение и освобождение памяти через единственную *выделяющую функцию* (allocation function), которую пользователь должен предоставить при создании состояния Lua.

Функция `luaL_newstate`, которую мы использовали для создания состояний Lua, является вспомогательной функцией, которая создает состояние Lua с выделяющей функцией по умолчанию. Эта

функция по умолчанию использует стандарт `malloc-realloc-free` из стандартной библиотеки C, чего должно быть достаточно для обычных приложений. Однако это очень легко – получить контроль над выделением памяти, создав свое состояние при помощи функции `lua_newstate`:

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

Эта функция получает два аргумента: выделяющую функцию и *пользовательские данные* (user data). 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_Alloc` определен следующим образом:

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

Первый параметр – это пользовательские данные, которые мы предоставили `lua_newstate`; второй параметр – это адрес блока, который мы хотим освободить или изменить его размер; 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, то функция должна освободить блок по адресу `ptr` и вернуть `NULL`, который соответствует запрашиваемому размеру блока. Когда `ptr` равен `NULL`, то 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` равно нулю, то функция ничего не делает и возвращает `NULL`.

Наконец, когда и `ptr` не равен `NULL`, и `nsize` не равно нулю, функция должна перевыделить этот блок (как и `realloc`) и вернуть новый адрес (который может совпадать с исходным адресом, а может и отличаться от него). Опять же, в случае ошибки функция должна вернуть `NULL`. Lua предполагает, что выделяющая функция всегда успешно обрабатывает, когда новый размер меньше или равен старому размеру. (Lua уменьшает некоторые структуры во время сборки мусора и не в состоянии корректно обрабатывать ошибки в это время.)

Стандартная выделяющая функция, используемая `luaL_newstate`, выглядит следующим образом (взято 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 считает, что `free(NULL)` не делает ничего и что вызов `realloc(NULL, size)` эквивалентен `malloc(size)`. Это гарантируется стандартом ANSI C.

Вы можете получить выделяющую функцию для заданного состояния Lua при помощи `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 новая выделяющая функция должна быть в состоянии освобождать блоки, выделенные старой функцией. Чаще всего новая выделяющая функция – это просто обертка над старой функцией, например для отслеживания выделений или синхронизации доступа к куче.

Внутри себя Lua не кэширует свободные блоки для их переиспользования. Она предполагает, что это делает выделяющая функция, многие хорошие функции для выделения памяти это делают. Lua не пытается минимизировать фрагментацию памяти. Исследования показывают, что фрагментация – это больше результат плохой схемы выделения памяти, чем поведения программы; хорошие функции для выделения памяти не создают сильной фрагментации.

Сделать хорошую выделяющую функцию довольно сложно, но иногда вы можете попробовать это сделать. Например, Lua дает вам старый размер любого блока при его освобождении или изменении его размера. Соответственно, специализированной выделяющей функции не нужно хранить где-то 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 выделения памяти, так как они используют глобальный ресурс (память). Однако доступ к состоянию Lua также должен быть синхронизован – 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 собственного пула памяти, то можно убрать требование явной синхронизации.

32.2. Сборщик мусора

До версии 5.0 Lua использовал простой сборщик мусора типа *mark-and-sweep*. Этот сборщик мусора иногда называется *сборщик-останови-мир*. Это значит, что время от времени Lua прекращает интерпретировать главную программу для выполнения полного цикла сборки мусора. Каждый такой цикл состоит из трех фаз: *пометить* (*mark*), *очистить* (*clean*) и *подмести* (*sweep*).

Lua начинает фазу пометить с того, что помечает как живой свой *корневой набор*, который включает в себя все объекты, к которым Lua имеет прямой доступ: реестр и главная нить. Любой объект, который хранится в живом объекте, достижим программой и поэтому тоже помечается как живой. Фаза пометки оканчивается, когда все достижимые объекты помечены как живые.

Перед началом фазы «подметания» Lua выполняет фазу очистки, 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 (воскрешенные) и помещаются в отдельный список для использования на стадии финализации. Во-вторых, Lua обходит свои слабые таблицы и удаляет из них все элементы, где либо ключ, либо само значение не помечено.

Фаза «подметания» обходит все объекты Lua. (Для того чтобы это было возможно, Lua хранит все создаваемые объекты с связным списком.) Если объект не помечен как живой, то он удаляется. В противном случае Lua снимает его пометку для подготовки к следующему циклу. Во время этой фазы Lua также вызывает финализаторы объектов, которые были собраны во время фазы очистки.

Начиная с версии 5.1, Lua использует инкрементальный сборщик мусора. Этот сборщик выполняет те же самые шаги, что и старый, но для этого ему не нужно «останавливать мир». Вместо этого он ра-

ботает вместе с интерпретатором. Каждый раз, когда интерпретатор выделяет некоторое количество памяти, сборщик мусора выполняет небольшой шаг. Это значит, что в то время, когда сборщик мусора работает, интерпретатор может изменить видимость объекта. Для того чтобы гарантировать правильность работы сборщика мусора, некоторые операции в интерпретаторе имеют специальные барьеры, которые обнаруживают опасные изменения и поправляют пометки соответствующих 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 надо сделать. Возможными значениями являются:

- `LUA_GCSTOP("stop")`: останавливает сборщик мусора до следующего вызова `collectgarbage/lua_gc` с опцией `"restart"`.
- `LUA_GCRESTART("restart")`: снова запускает сборщик мусора.
- `LUA_GCCOLLECT("collect")`: осуществляет полный цикл сборки мусора так, что все недостижимые 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(нет)`: возвращает дробную часть от количества килобайт памяти, используемой Lua. В C следующее выражение возвращает полный объем памяти в байтах (считая, 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 получено 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` равно 100, это соответствует параметру, установленному в единицу (100%).
- `LUA_GCSETSTEPMUL("setstepmul")`: задает множитель (`stepmul`) для сборщика мусора. Эта значение задается параметром `data` также в процентах.

Любой сборщик мусора балансирует между памятью и затратами времени CPU. С одной стороны, сборщик мусора может вообще не выполняться. Он вообще не будет тратить времени процессора, но ценой огромных трат памяти. С другой стороны, сборщик мусора может выполнять полный цикл сборки для каждого изменения графа достижимости. Программа будет использовать точный минимум требуемой памяти ценой громадных затрат времени процессора. Настоящие сборщики мусора пытаются найти хороший баланс между этими крайностями.

Как и ее выделяющая функция, сборщик мусора в Lua вполне хорош для большинства приложений. Однако в некоторых случаях имеет смысл попробовать его оптимизировать. Два параметра, `pause` и `stepmul`, предоставляют некоторый контроль над «характером» сборщика мусора.

Параметр `pause` управляет тем, как долго сборщик мусора ждет между завершением одной сборки мусора и началом следующей. Значение этого параметра, равное нулю, заставит сборщик мусора начать новый сбор мусора, как только он завершит предыдущий. Значение, равное 200%, ждет, когда потребление памяти вырастет в два раза перед началом сборки мусора. Вы можете установить меньшее значение, если вы хотите получить меньшее потребление памяти ценой больших затрат времени процессора. Обычно это значение выбирается 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 является 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 – это типичные клиенты для этого типа 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 моменты ожидания, вы можете держать сборщик мусора остановленным и вызывать `collectgarbage("step", n)` во время таких моментов ожидания. Для того чтобы определить, как много работы нужно выполнить во время такого момента ожидания, либо экспериментально подберите подходящее значение для `n`, либо в цикле вызывайте `collectgarbage c n`, равным нулю, до истечения такого момента ожидания.

Упражнения

Упражнение 32.1. Напишите библиотеку, которая позволит скрипту ограничить общий объем памяти, используемый состоянием в Lua. Она может предоставить единственную функцию `setlimit` для задания этого максимального объема.

Библиотека должна задать 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 `lua_gc` для инициализации своего счетчика памяти при старте. Она также может использовать пользовательские данные, для того чтобы хранить свое состояние: число байт, текущее максимальное ограничение и т. п. Не забудьте использовать исходный указатель на пользовательские данные при вызове исходной выделяющей функции.)

Упражнение 32.2. Для этого упражнения вам понадобится как минимум один скрипт на Lua, использующий очень много памяти. Если у вас такого нет, то напишите его. (Он может быть очень прост – например, цикл, создающий таблицы.)

- Запустите ваш скрипт с различными параметрами сборщика мусора. Насколько они влияют на быстродействие?
- Что случится, если вы установите параметр `pause` равный нулю? Что случится, если вы установите его равным 1000?
- Что случится, если вы установите параметр `stepmul`, равным нулю? Что случится, если вы установите его равным 1 000 000?
- Поправьте ваш скрипт, для того 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 таким образом?

Роберту Иерузалимски

Программирование на языке Lua

Третье издание

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Боресков А. В.*

Корректор *Синяева Г. И.*

Верстка *Паранская Н. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90 ¹/₁₆. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 23,52.

Тираж 200 экз.

Веб-сайт издательства: www.dmk.rf

Книга посвящена одному из самых популярных встраиваемых языков - Lua. Этот язык используется во многих играх и большом количестве различных приложений. Он сочетает небольшой объем занимаемой памяти, высокое быстродействие, простоту использования и большую гибкость. Книга рассматривает практически все аспекты использования Lua, начиная с основ языка и заканчивая тонкостями расширения языка и взаимодействия с C.

Важной особенностью книги является огромный спектр охватываемых тем - практически все, что может понадобиться при использовании Lua. Также к каждой главе дается несколько упражнений, позволяющих проверить свои знания.

Книга будет полезна широкому кругу программистов и разработчиков игр. Для понимания последних глав книги необходимо знание языка C, но для большинства остальных глав достаточно базовых знаний о программировании.



Роберту Иерузалимски - доцент католического университета Рио-де-Жанейро, где он занимается разработкой и реализацией языков программирования. Создатель и ведущий архитектор языка Lua. Обладатель докторской степени по информатике. Работал в качестве приглашенного исследователя в университете Ваатерло, International Computer Science Institute, институте Фраунгофера в Иллинойском университете Урабана-Шампейне, а также в качестве профессора в Центре латино-американских исследований (Center For Latin American Studies) Стэнфордского университета.

Internet-магазин

www.dmkpress.com

Книга-почтой:

orders@alians-kniga.ru

Оптовая продажа:

"Альянс-книга"

(499)782-3889

books@alians-kniga.ru

DMK
издательство
www.dmk.pф

ISBN 978-5-94074-767-3



9 785940 747673 >