Learn You Some Erlang for Great Good!
Изучай Erlang во имя добра!

Автор: Frederic Trottier-Hebert

v. 0.14.1

Глава 1

Введение

1.1 О книге

Именно здесь начинается изучение Erlang во имя добра! Это руководство, скорее всего, окажется одним из ваших первых шагов в изучении Erlang, поэтому скажу о нём пару слов.

Мысли об этой книге появились у меня после того, как я прочитал «Изучай Haskell во имя добра!» Miran Lipovača. Мне показалось, что ему удалось представить язык в привлекательном свете и сделать процесс обучения приятным. Я уже был знаком с Мираном, и поэтому поинтересовался, как он отнесётся к тому, что я напишу версию его книги, посвящённую Erlang. Идея пришлась ему по душе, так как он и сам немного интересовался языком.

Всё это привело к тому, что я теперь печатаю эти слова. Конечно, были и другие источники мотивации: я считаю что «порог вхождения» в язык довольно высок (в web маловато документации и вам, скорее всего, придётся покупать книги). Поэтому я посчитал, что сообществу пригодится руководство похожее на LYAH. Ещё я заметил, что кое-кто приписывал Erlang слишком много или слишком мало достоинств, основываясь при этом на поверхностных суждениях. Есть люди, абсолютно уверенные, что Erlang это просто разрекламированная пустышка. Даже если бы я хотел убедить их в обратном, знаю, что эти строки они вряд ли прочитают.

Книга позволяет изучить Erlang тем, кто обладает базовыми знаниями о программировании на императивных языках (таких как C/C++,

Java, Python, Ruby и т.д.) и имеет или не имеет представление о функциональном программировании (Haskell, Scala, Erlang, Clojure, OCaml...) Также я хочу, чтобы эта книга честно рассказывала об Erlang, правдиво освещая слабые и сильные стороны языка.

1.2 Так что же такое Erlang?

Сразу нужно сказать, что Erlang – функциональный язык программирования. Если вам приходилось когда-либо работать с императивными языками, то выражения вроде `i++;` для вас в порядке вещей. В функциональном программировании такие выражения не разрешаются. Более того, изменять значение любой переменной строго запрещено! Сначала это может прозвучать странно, но припомните уроки математики – вас обучали именно этому:
y = 2
x = y + 3
x = 2 + 3
x = 5
Если бы я добавил:
x = 5 + 1
x = x
$\Longrightarrow$ $5 = 6$

То привёл бы вас в замешательство. В функциональном программировании принято так: если я говорю, что x это 5, то, согласно логике, я не могу заявить, что x также и 6! Было бы нечестно. Поэтому функции с одинаковыми параметрами всегда должны возвращать тот же результат:
x = add_two_to(3) = 5
$\Longrightarrow$ $x = 5$

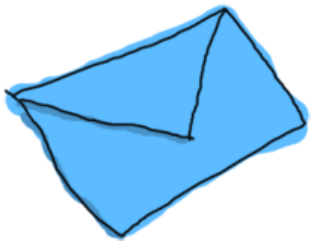
Принцип, согласно которому функции должны всегда возвращать тот же результат для одинаковых параметров, называется ссылочной прозрачностью. Опираясь на него, мы можем заменить `add_two_to(3)` на 5, так как результат операции `3+2` будет всегда равен 5. Мы можем решать более сложные проблемы, комбинируя десятки функций, при этом сохраняется уверенность, что ничего не сломается. Ясно и логично, не так ли? Правда, есть одна проблема:
x = today() = 2009/10/22
— ждём один день –
x = today() = 2009/10/23

$x = x$

$\Longrightarrow\ 2009/10/22 = 2009/10/23$

О, нет! Мои прекрасные равенства! Они вдруг стали неверными! Как же так вышло, что моя функция возвращает каждый день разные результаты?

Очевидно, существуют случаи, когда бывает полезно отказаться от ссылочной прозрачности. В Erlang очень прагматичный подход к функциональному программированию: следуй принципам чистоты (пиши функции без побочных эффектов, избегай изменяемых данных и т.д.), но уходи от них, когда решаешь задачи реального мира.



Итак, мы определили, что Erlang – функциональный язык программирования. Но ему также присущ сильный уклон в параллелизм и высокую надёжность. Для одновременного выполнения десятков задач, Erlang использует актор-модель, где каждый актор – это отдельный процесс виртуальной машины. В двух словах, если бы вы были актором в мире Erlang, быть вам одиноким человеком, который сидит в тёмной комнате без окон рядом с почтовым ящиком и ожидает сообщений. На полученное сообщение вы реагируете следующим образом: если вам пришёл счёт – вы его оплачиваете; если пришло поздравление с днём рождения – вы посылаете ответное письмо с благодарностью, а все письма, которые вам непонятны, вы игнорируете.

Актор-модель в Erlang можно представить как мир, где каждый сидит в своей комнатке и может исполнять несколько определённых задач. Любое общение происходит только посредством почтовой переписки. Скучноватая жизнь (и золотая эра для почтовых служб), но благодаря такому порядку вещей, вы можете попросить нескольких людей выполнить для вас строго определённый набор заданий, и никто не выполнит свою задачу неверно, не сделает ошибку, которая окажет воздействие на работу других людей. Они даже не будут предполагать о существовании кого-то ещё, кроме вас (и это прекрасно).

Но не будем заходить с аналогией слишком далеко. Можно сказать, что Erlang заставляет вас писать акторы (процессы), которые не разделяют информацию с другими частями кода, кроме случаев, когда они посылают друг другу сообщения. Каждая коммуникация происходит явно, она безопасна и её можно отследить. Мы дали Erlang характеристику на языковом уровне. Но это ещё не всё: в целом Erlang является ещё и средой разработки. Код компилируется в байт-код и исполняется виртуальной машиной. Благодаря этому программы на Erlang, как

и программы на Java, можно запускать где угодно. Стандартный дистрибутив включает (кроме прочего) средства разработки (компилятор, отладчик, профайлер, библиотека для unit-тестирования), фреймворк Open Telecom Platform (OTP), веб-сервер, парсер и базу данных mnesia – систему хранения пар ключ-значение, которая способна реплицироваться на несколько серверов, поддерживает вложенные транзакции и обеспечивает хранение любых данных, которые определены в Erlang.

Виртуальная машина и библиотеки также позволяют обновлять код на работающей системе, не прерывая исполнение, легко распределять код на несколько компьютеров и осуществлять простую, но эффективную обработку ошибок.

Позже мы увидим как использовать эти инструменты, но сейчас я расскажу ещё об одном основополагающем принципе Erlang: пусть процесс падает. Но пусть он падает не как самолёт в авиакатастрофе с десятками человеческих жертв, а как канатоходец, под которым натянута страховочная сеть. Избегать ошибок, конечно же, нужно, но делать проверки на каждый тип ошибки в большинстве случаев нет необходимости.

Итак, Erlang умеет восстанавливаться после ошибок, организовывать код при помощи акторов, производить распределённое масштабирование и обеспечивать параллелизм. Но это приводит нас к следующему разделу...

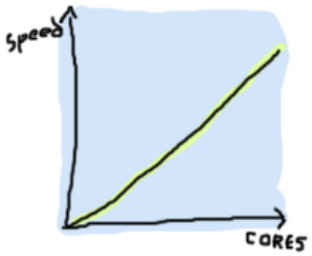
1.3 Не забывайтесь

В книге будет много маленьких жёлто-оранжевых разделов, с названиями похожими на это (их легко заметить). Erlang в данный момент набирает популярность во многом благодаря пылким речам, которые могут ввести людей в заблуждение. Могут убедить, что Erlang – нечто большее, чем есть на самом деле. Эти напоминания помогут вам не забываться, если вы переполнены энтузиазмом сверх меры.

Первый случай такого заблуждения относится к мощным возможностям масштабирования, которые заложены в Erlang, и осуществляются при помощи лёгких процессов. Да, действительно, процессы в Erlang очень легки: одновременно могут существовать сотни тысяч таких процессов, но это совсем не означает, что они должны существовать про-

сто потому, что есть такая возможность. К примеру, вы создаёте игру-стрелялку. Было бы безумием представлять все объекты в игре, включая пули, при помощи акторов. В такой игре можно только себе в ногу выстрелить. Пересылка сообщений от актора к актору всё-таки отнимает ресурсы, как бы малы они ни были. Если будете дробить задачи слишком сильно, *всё может сильно замедлиться*!

Подробнее я об этом расскажу, когда мы погрузимся в изучение достаточно глубоко, чтобы это нас начало беспокоить. Но пока имейте в виду, что бездумно использовать параллелизм при решении проблемы совсем недостаточно для того, чтобы решение стало быстрым. Не унывайте! Будут и случаи, когда использовать сотни процессов можно и нужно! Просто не каждый случай – тот самый.



Ещё об Erlang говорят, что он может масштабировать вычисления прямо пропорционально количеству вычислительных ядер, которые есть в вашем компьютере, но обычно это не так. Возможность такая имеется, но большинство задач не получится запустить так, чтобы всё исполнялось одновременно.

Стоит помнить ещё об одном: хотя Erlang и делает некоторые вещи очень хорошо, но всё-таки, формально говоря, можно получить те же результаты и при помощи других языков. Верно также и обратное. Тщательно оценивайте проблему и выбирайте правильный инструмент, исходя из той задачи, которую вы пытаетесь решить. Erlang – не панацея, и особенно плохо себя покажет в обработке изображений и сигналов, или в качестве языка для написания драйверов. Однако он будет блистать в области большого программного обеспечения для серверов (очереди, map-reduce), в вычислениях, производимых совместно с другими языками, в высокоуровневой реализации протоколов и т.д. Всё, что находится между этими полюсами, зависит лишь от вас. Нет смысла ограничивать себя лишь серверными вычислениями на Erlang. Были случаи, когда люди создавали с его помощью неожиданные и удивительные вещи. Один из примеров это IANO – робот, созданный командой UNICT. Они используют Erlang для реализации искусственного интеллекта и получили серебрянную медаль на соревновании eurobot в 2009 году. Ещё один пример – Wings 3D. Это платформонезависимая программа трёхмерного моделирования (без рендерера) с открытым исходным кодом, которая написана на Erlang.

1.4 Что нужно для изучения

Для начала вам понадобится лишь текстовый редактор и среда Erlang. Можно загрузить исходный код и сборки для Windows с официального сайта Erlang. Не буду углубляться в детали установки, но для Windows достаточно скачать и запустить исполняемый файл. Не забудьте добавить директорию Erlang в переменную окружения PATH, чтобы получить доступ к ней из командной строки.

В Debian-подобных Linux дистрибутивах необходимо установить пакет командой `# apt-get install erlang`. В Fedora (если установлен «yum»), можно проделать то же самое, набрав `# yum install erlang`. Однако, официальные репозитории обычно содержат устаревшие версии пакетов Erlang. Если вы будете использовать старую версию пакета, это может привести к расхождениям между вашим результатом и этим руководством. К тому же, в некоторых приложениях производительность будет снижена. Поэтому я рекомендую вам компилировать всё из исходного кода. Изучите содержимое файла README, который поставляется с пакетом, обратитесь к Google для получения подробностей установки, они справятся с этой задачей намного лучше, чем я.

В FreeBSD существует много способов установки. Если вы используете `postmaster`, можно исполнить команду
`postmaster lang/erlang`.
Для установки из портов воспользуйтесь командой
`cd /usr/ports/lang/erlang;make install clean`.
И, наконец, если хотите использовать систему пакетов, запустите
`pkg_add -rv erlang`.
Если вы пользователь OSX, то можете установить Erlang командой
`brew install erlang`
при помощи Homebrew, или воспользуйтесь
`port install erlang`, если предпочитаете MacPorts.

Замечание: на момент написания я использую Erlang версии R13B+, поэтому для наилучших результатов используйте такую же версию, либо более новую.

1.5 Где искать помощь

Есть несколько мест, где вам помогут. Хорошую техническую документацию можно найти в man страницах, если вы используете Linux. Например, в Erlang есть модуль lists (с которым мы скоро столкнёмся): чтобы получить документацию по этому модулю напишите в консоли

`$ erl -man lists`.

Инсталляция в Windows должна содержать документацию в формате HTML. Её всегда можно скачать с официального сайта Erlang или обратиться к одному из альтернативных сайтов.

Как только почувствуете, что пишете что-то не то, обратитесь к правилам оформления кода, которые можно найти здесь. Код в этой книге будет стараться придерживаться этих правил.

Бывают случаи, когда простого понимания технических деталей недостаточно. Когда наступает такой момент, я обращаюсь к двум источникам знаний: официальной почтовой рассылке (там есть чему поучиться) и irc каналу #erlang на irc.freenode.net.

А если вы любите готовые рецепты, то trapexit это то что вам нужно. Ещё они держат зеркало почтовой рассылки в виде форума и общую вики. Там всегда можно найти что-нибудь полезное.

Глава 2

Начинаем

2.1 Оболочка

В Erlang можно проверить большую часть кода в эмуляторе. В нём можно запускать компилированные скрипты, но также можно редактировать код вживую. Чтобы запустить оболочку в Linux, откройте терминал и наберите `$ erl`. Если вы всё правильно настроили, появится текст похожий на этот:

```
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2]
[rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
```

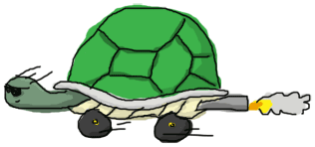
Поздравляю, вы запустили оболочку Erlang!

Пользователи Windows тоже могут запустить `erl.exe`, но лучше использовать `werl.exe`, который можно найти в стартовом меню (`programs > Erlang`). Werl это оболочка Erlang, которая создана специально для Windows. Она поддерживает оформление окон с полосами прокрутки и операциями редактирования (такими, например, как копирование-вставка, которых очень не хватает в стандартной оболочке `cmd.exe` для Windows). Правда, если вы захотите использовать перенаправление стандартного ввода-вывода или каналы, то erl вам всё-таки пригодится.

В оболочке мы сможем вводить и запускать код, но сначала посмотрим как с ней обращаться.

2.2 Команды оболочки

В оболочку Erlang встроено подмножество функций популярного текстового редактора Emacs, использование которого берёт начало в 70-х годах. Если вы знакомы с Emacs, то ваших знаний будет достаточно. У всех остальных и без этих знаний всё получится.



К примеру, если напечатать какой-либо текст и нажать `^A` (Ctrl+A), ваш курсор переместится в начало строки. `^E` (Ctrl+E) переносит курсор в конец строки. Чтобы перемещаться вперёд и назад, используйте клавиши со стрелками «влево» и «вправо». Клавиши «вверх» и «вниз» воспроизводят предыдущие или последующие строки, которые уже вводились ранее. Это позволяет не набирать их повторно.

Если написать, к примеру, `li` и нажать «tab», оболочка дополнит за вас слово до `lists:`. Если нажать tab ещё раз, то оболочка предложит функции, которые могут следовать за lists:. Так Erlang дополняет имя модуля lists и предлагает функции, которые тот содержит. Способ записи может показаться странным, но не переживайте, вы к нему быстро привыкнете. Думаю, мы достаточно ознакомились с функциональностью оболочки, чтобы понимать что к чему. Кроме одного момента: мы не знаем как из неё выйти! Ответ на этот вопрос получить очень просто. Напишите `help().` и вы получите описание команд, которые можно использовать в оболочке (не забудьте написать точку (`.`), она нужна для исполнения команды). Некоторыми из них мы воспользуемся чуть позже, но сейчас для выхода из оболочки нам нужна единственная строка: `q() – quit - shorthand for init:stop()`

Это один из способов выхода (если быть точным, то два способа). Но от этой команды мало толку, если оболочка зависла! Те из вас, кто внимательно следил за выводом оболочки при запуске, видели комментарий о том, что можно «прекратить исполнение при помощи `^G`». Нажмём это сочетание клавиш и напечатаем h, чтобы получить подсказку.

```
User switch command
--> h
c [nn]            - connect to job
i [nn]            - interrupt job
k [nn]            - kill job
j                 - list all jobs
s [shell]         - start local shell
r [node [shell]]  - start remote shell
```

```
q        - quit erlang
? | h              - this message
-->
```

Если нажать `i` и следом `c`, то Erlang остановит исполнение кода и возвратится в интерактивную оболочку. По нажатию `j` будет выведен список запущенных процессов (звёздочка следом за номером указывает, что задача исполняется в данный момент). Процесс можно прервать, написав `i` и следом номер задачи. Команда `k` не прерывает оболочку, а полностью завершает её исполнение. Команда `s` запускает новую оболочку.

```
Eshell V5.7.2  (abort with ^G)
1> "OH_NO_THIS_SHELL_IS_UNRESPONSIVE!!!_*hits_ctrl+G*"
User switch command
--> k
--> c
Unknown job
--> s
--> j
2* {shell,start,[]}
--> c 2
Eshell V5.7.2  (abort with ^G)
1> "YESS!"
```

Если вы полностью прочитали текст подсказки, то, возможно, заметили, что мы можем запустить удалённую оболочку. Не буду сейчас вдаваться в детали, но этот факт даёт представление о том, на что ещё способна виртуальная машина Erlang, кроме запуска кода. А теперь начнём (на этот раз по-настоящему).

Глава 3

Начинаем (по-настоящему)

Erlang довольно простой и компактный язык (в том же смысле, в котором C проще чем C++). В языке определены несколько фундаментальных типов данных. В этой главе мы рассмотрим их большую часть. Рекомендуется обязательно прочитать главу, так как в ней объясняются элементы, необходимые для построения всех программ, которые вы позже напишете на Erlang.

3.1 Числа

В оболочке Erlang выражения должны завершаться точкой, за которой следует пробел (возврат каретки, пробел и т.д.), иначе это выражение не будет выполнено. Можно разделять выражения запятыми, но лишь результат последнего выражения отобразится на экране (однако, все остальные выражения будут всё равно исполнены). Для большинства людей этот синтаксис покажется довольно непривычным. Он восходит к тем временам, когда Erlang был реализован на Prolog – языке логического программирования.

Откройте оболочку Erlang так, как мы это делали в предыдущей главе, и давайте-ка что-нибудь попечатаем!

```
1> 2 + 15.
17
2> 49 * 100.
4900
3> 1892 - 1472.
420
4> 5 / 2.
```

```
2.5
5> 5 div 2.
2
6> 5 rem 2.
1
```

Возможно вы заметили, что для Erlang не имеет значения, вводите вы дробные числа или целые: в арифметических операциях поддерживаются оба типа. Целые и дробные значения это, в общем-то, единственные типы данных, которые будут прозрачно обрабатываться математическими операторами Erlang. Однако, если вы хотите делить целое на целое – используйте `div`, а для операции взятия остатка по модулю используйте оператор `rem` (remainder, остаток).

Обратите внимание, что мы можем использовать несколько операторов в одном выражении, и порядок вычисления арифметических операций подчиняется обычным правилам.

```
7> (50 * 100) - 4999.
1
8> -(50 * 100 - 4999).
-1
9> -50 * (100 - 4999).
244950
```

Если хотите выразить целое число в системе счисления с основанием отличным от 10, то просто введите его в виде `Основание#Число` (основание может меняться в диапазоне 2...36):

```
10> 2#101010.
42
11> 8#0677.
447
12> 16#AE.
174
```

Прекрасно! Erlang обладает возможностями калькулятора, который пылится где-то в дальнем углу вашего стола. Вдобавок, у калькулятора весьма странный синтаксис! Просто великолепно!

3.2 Неизменные переменные

Арифметические вычисления это отлично, но без возможности сохранять промежуточные результаты далеко не уедешь. Для этого мы бу-

дем использовать переменные. Если вы прочитали введение к этой книге, то знаете, что переменные в функциональном программировании не могут меняться. Общее поведение переменных можно продемонстрировать в следующих 7 выражениях (обратите внимание, что имена переменных начинаются с заглавной буквы):

```
1> One.
* 1: variable 'One' is unbound
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
** exception error: no match of right hand side value 3
7> two = 2.
** exception error: no match of right hand side value 2
```

Первое, что можно заметить: значение можно присваивать переменной ровно один раз, после этого можно «притвориться», что вы присваиваете значение, только если это то же самое значение, которое ей уже присвоено. Erlang-у не понравится, если они будут различны. Объяснение кроется в свойствах оператора `=`. Оператор `=` (не переменные) выполняет функцию сравнения значений и сообщает, если они отличаются. Если они одинаковы, то оператор просто возвращает значение:

```
8> 47 = 45 + 2.
47
9> 47 = 45 + 3.
** exception error: no match of right hand side value
   48
```

Оператор в сочетании с переменными делает следующее: если выражение слева – свободная переменная (с ней не связано значение), Erlang автоматически свяжет значение, которое находится справа, с переменной слева. После этого сравнение завершится успешно, и значение переменной сохранится в памяти.

Такое поведение оператора `=` это основа «сопоставления с образцом» (pattern matching), которое существует во многих функциональ-

ных языках программирования, хотя его реализация в Erlang обычно считается более гибкой и полной в сравнении с аналогами. Мы рассмотрим сопоставление с образцом более детально, когда будем разбираться с кортежами и списковыми типами, чуть позже в этой же главе, а также в связи с функциями в последующих главах.

Ещё один факт, который можно почерпнуть из команд 1-7 это то, что имена переменных должны начинаться с заглавной буквы. Команда 7 завершилась ошибкой, потому что слово *two* начинается со строчной буквы. Формально имена переменных могут начинатся со знака подчёркивания ('_'), но по соглашению такие имена используются только для переменных, значение которых вас не интересует, но вы всё-таки посчитали, что было бы неплохо описать, что же в них содержится.

Ещё можно завести переменную, имя которой будет состоять просто из знака подчёркивания:

```
10> _ = 14+3.
17
11> _.
* 1: variable '_' is unbound
```

В отличие от любой другой переменной, у этой никогда не будет собственного значения. На данный момент это для нас абсолютно бесполезно, но запомним, что такая переменная существует. Она нам ещё пригодится.

Замечание: если вы экспериментируете в оболочке и присвоили переменной неверное значение, то можно «удалить» эту переменную при помощи функции `f(Variable).`. Чтобы очистить все переменные, используйте `f().`.
Эти функции работают только в оболочке и созданы специально, чтобы помогать вам во время тестирования. В настоящих программах мы не сможем удалять значения таким способом. Смысл этого поведения становится понятен, если представить Erlang в производственном окружении: вполне возможно, что оболочка будет непрерывно запущена на протяжении нескольких лет... Готов поспорить, что переменная X будет использована за это время больше одного раза.

3.3 Атомы

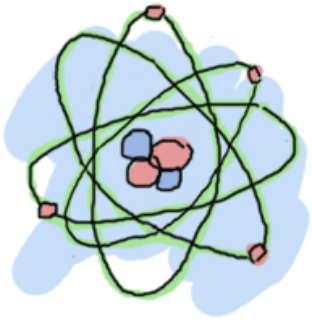
Атомы являются причиной, по которой имена переменных должны начинаться с заглавной буквы. Атомы это литералы – константы с собственным именем для некоторого значения. Атомы это только то, что

вы видите и больше ничего в них нет. Атом *cat* просто означает «cat» и ничего больше. С ним нельзя поиграть, его нельзя изменить, его нельзя разбить вдребезги; это *cat*. Смиритесь с этим.

Кроме записи атомов в виде слов, которые начинаются со строчной буквы, существует ещё несколько способов записи:

```
1> atom.
atom
2> atoms_rule.
atoms_rule
3> atoms_rule@erlang.
atoms_rule@erlang
4> 'Atoms can be cheated!'.
'Atoms can be cheated!'
5> atom = 'atom'.
atom
```

Если атом не начинается со строчной буквы или содержит символы отличные от букв и цифр, знака подчёркивания (_) или @, то он должен обрамляться одиночными кавычками ('). Выражение под номером 5 также показывает, что атом, заключённый в одиночные кавычки, это совершенно то же самое, что и атом без них.



Я сравнил атомы с константами, у которых значением является их собственное имя. Возможно, вы раньше работали с кодом, в котором использовались константы. Пусть, к примеру, у меня есть значения, которые соответствуют цвету глаз: `BLUE -> 1, BROWN -> 2, GREEN -> 3, OTHER -> 4`. Необходимо сопоставить имя константы с некоторым значением. Атомы позволяют без этого обойтись. Цвет глаз может просто быть 'blue', 'brown', 'green' и 'other'. Эти цвета можно использовать в коде где угодно: значения никогда не будут пересекаться, и, вдобавок, такая константа всегда инициализирована. Если вам и в самом деле нужны именно константы со связанными значениями, то всё-таки существует способ их заполучить. Мы рассмотрим его в главе 4.

Таким образом, атом хорош для представления данных, которые с ним связаны. Сложновато без данных найти ему достойное применение. Довольно об атомах. Их время настанет, когда мы сможем компоновать их с другими типами данных.

Не забывайтесь:
Атомы прекрасно подходят для отсылки сообщений и представления констант. Но во многих случаях использование атомов скрывает подвох. Обращение к атому происходит через «таблицу атомов», которая занимает память (4 байт/атом в 32-битной системе, 8 байт/атом в 64-битной системе). Таблица атомов не обрабатывается сборщиком мусора, поэтому атомы будут накапливаться до тех пор, пока система не остановится из-за нехватки памяти, либо потому что было определено максимальное количество атомов – 1048577.
Поэтому атомы нельзя генерировать динамически ни в коем случае. Если вам нужна надёжная система, но данные, введённые пользователем, роняют её из-за конвертации ввода в атомы, то у вас серьёзные проблемы. Атомы необходимо рассматривать только как инструменты разработчика, потому что на самом деле они как раз ими и являются.

Замечание: некоторые атомы являются зарезервированными словами и их нужно использовать так, как было задумано разработчиками языка: для обозначения имён функций, операторов, выражений и т.д. Вот эти атомы: `after` `and` `andalso` `band` `begin` `bnot` `bor` `bsl` `bsr` `bxor` `case` `catch` `cond` `div` `end` `fun` `if` `let` `not` `of` `or` `orelse` `query` `receive` `rem` `try` `when` `xor`

3.4 Булева алгебра и операторы сравнения

Если бы человек не мог отличать большое от малого, истину от лжи, то ему бы пришлось нелегко. В Erlang, как и в любом другом языке, есть возможность применения булевых операций и сравнения элементов. Булева алгебра очень проста:


```
1> true and false.
false
2> false or true.
true
3> true xor false.
true
4> not false.
true
5> not (true and true).
false
```

Замечание: Операторы `and` и `or` всегда вычисляют аргументы, находящиеся по обе стороны от оператора. Если вам нужны операторы, которые вычисляют правую сторону только при необходимости, используйте `andalso` и `orelse`.

Проверка на равенство и неравенство тоже выполняется очень просто, но при этом используют несколько другие символы, чем те, которые можно увидеть во многих других языках:

```
6> 5 =:= 5.
true
7> 1 =:= 0.
false
8> 1 =/= 0.
true
9> 5 =:= 5.0.
false
10> 5 == 5.0.
true
11> 5 /= 5.0.
false
```

Во-первых, если в привычном языке для проверки на равенство и неравенство используется `==` и `!=`, то в Erlang используется `=:=` и `=/=`. Три последних выражения (строки с 9 по 11) также знакомят нас с ловушкой: Erlang не делает различий при выполнении арифметических действий между целыми числами и числами с плавающей запятой, но он будет различать эти числа при сравнении. Впрочем, беспокоиться не о чем, потому что в этом случае на помощь придут операторы `==` и `/=`. Помните об этом различии на случай, если понадобится сравнить числа, не сравнивая их типы.

Остальные операторы сравнения: `<` (меньше чем), `>` (больше чем), `>=` (больше либо равно) и `=<` (меньше либо равно). Мне кажется, что последний оператор записывается задом-наперёд, и из-за него в моём коде часто появляются синтаксические ошибки. Будьте внимательны с этим `=<`.

```
12> 1 < 2.
true
13> 1 < 1.
false
14> 1 >= 1.
true
```

```
15> 1 =< 1.
true
```

Как узнать что произойдёт, если выполнить `5 + llama` или `5 == true`? Нет способа лучше, чем исполнить эти выражения и испугаться сообщений об ошибках!

```
12> 5 + llama.
** exception error: bad argument in an arithmetic
   expression
     in operator  +/2
          called as 5 + llama
```

Ну, что сказать? Erlang не нравится, что вы неправильно используете его базовые типы! Здесь эмулятор возвращает вполне понятное сообщение об ошибке. Он сообщает, что ему не по нраву один из аргументов, которые окружают оператор `+`!

Впрочем, Erlang не всегда сходит с ума из-за неверных типов:

```
13> 5 =:= true.
false
```

Почему для некоторых операторов различие в типах не существено, а для других имеет значение? Хотя Erlang не разрешает суммировать что попало с чем угодно, он позволяет *сравнивать*. Так вышло потому, что создатели Erlang поставили прагматизм выше теории и решили, что было бы неплохо иметь возможность писать, к примеру, общие алгоритмы сортировки, которые могли бы упорядочивать любые элементы. В большинстве случаев это упрощает жизнь.

Когда работаете с булевой алгеброй и сравнениями, стоит помнить ещё об одной вещи:

```
14> 0 == false.
false
15> 1 < false.
true
```

Если вы раньше имели дело преимущественно с процедурными или объектно-ориентированными языками, то, вероятно, сейчас рвёте волосы на голове. Строка 14 должна возвращать *true*, а строка 15 *false*! Везде false означает 0, а true – всё остальное! Но не в Erlang. Потому что я вам солгал. Да, я вам наврал. Стыд мне и позор.

В Erlang нет такого понятия как булевы значения *true* и *false*. Элементы true и false на самом деле являются атомами, но они хорошо ин-

тегрированы в язык, и проблем с ними не будет, пока вы считаете, что false и true – это просто false и true, и ничего больше.

Замечание: при сравнении элементы выстраиваются в следующем порядке:
number < atom < reference < fun < port < pid < tuple < list < bit string
Все эти понятия вам пока неизвестны, но вы будете с ними знакомиться по мере продвижения по тексту книги. Просто помните, что именно благодаря им вы можете сравнивать что угодно с чем угодно! Процитирую Joe Armstrong, одного из создателей Erlang: «Важен не сам порядок, а то, что этот порядок чётко определён.»

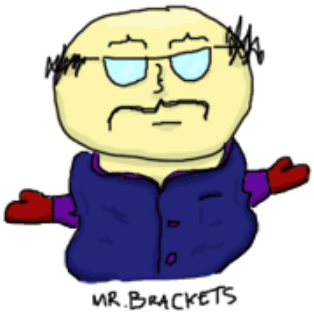
3.5 Кортежи

Кортеж – это способ организации данных. С его помощью можно сгруппировать элементы, когда вам известно их количество. Кортежи в Erlang записываются следующим образом: {Element1, Element2,..., ElementN} . Как пример можно привести координаты (x, y), которые задают положение точки на плоскости. Мы можем представить координаты как кортеж двух элементов:

```
1> X = 10, Y = 4.
4
2> Point = {X,Y}.
{10,4}
```

В этом случае точка будет всегда представлена двумя значениями. Вместо того, чтобы повсюду таскать за собой переменные X и Y, можно определить всего одну. Но что же делать, если у меня есть переменная-точка, а мне необходима лишь её X координата? Эту информацию можно легко получить. Помните, когда мы присваивали значения переменным, Erlang не возражал, если присваиваемые значения были равны содержимому переменных. Давайте это используем! Возможно, вам понадобится очистить командой f() переменные, которые мы определили ранее.

```
3> Point = {4,5}.
{4,5}
4> {X,Y} = Point.
{4,5}
5> X.
4
6> {X,_} = Point.
{4,5}
```



Теперь для получения первого значения в кортеже, мы можем использовать X! Как так получилось? Сначала у X и Y значений не было, и они считались свободными переменными. Мы поместили их в кортеж *{X, Y}* по левую сторону оператора `=`. Оператор `=` сравнивает между собой *{X, Y}* и *{4, 5}*. У Erlang хватает сообразительности, чтобы распаковать значения в кортеже и распределить их по свободным переменным слева от оператора присваивания. После этого сравнение приходит к виду $\{4, 5\} = \{4, 5\}$, и, очевидно, завершается успехом! Это одна из многих форм операции сопоставления с образцом (pattern matching).

Заметьте, что в шестом выражении я использовал анонимную переменную _. Она должна использоваться именно в таких случаях: когда необходимо отбросить значение, которое нам не понадобится. Переменная _ всегда определена как свободная и служит элементом подстановки (wildcard) для сопоставления с образцом. Сопоставление с образцом для распаковки кортежей будет работать только когда количество элементов (длина кортежа) одинаково с обеих сторон.

```
7> {_,_} = {4,5}.
{4,5}
8> {_,_} = {4,5,6}.
** exception error: no match of right hand side value {
   4,5,6}
```

Кортежи полезны и при работе с одиночными значениями. Простейший пример, температура:

```
9> Temperature = 23.213.
23.213
```

В такой денёк неплохо было бы пойти на пляж...Погодите-ка, это температура по шкале Кельвина, Цельсия или Фаренгейта?

```
10> PreciseTemperature = {celsius, 23.213}.
{celsius,23.213}
11> {kelvin, T} = PreciseTemperature.
** exception error: no match of right hand side value {
   celsius,23.213}
```

Операция вызывает ошибку, но это как раз то, что нам необходимо! Мы снова наблюдаем в работе сопоставление с образцом. Оператор `=`

сравнивает *{kelvin, T}* и *{celsius, 23.213}*. Несмотря на то, что переменная T свободна, Erlang при сравнении не посчитает атом *celsius* идентичным атому *kelvin*. Будет брошено исключение, которое остановит выполнение кода. Та часть нашей программы, которая ожидает температуру по шкале Кельвина, не сможет обработать значение, которое будет представлено в градусах Цельсия. Это трюк облегчает отладку кода и даёт программисту представление о передаваемых данных. Кортеж, который содержит атом, а следом за ним одиночный элемент – называется «меченым кортежем». Все элементы кортежа могут принадлежать к разным типам данных, и элементами кортежа могут быть и другие кортежи:

```
12> {point, {X,Y}}.
{point,{4,5}}
```

Что же делать, когда необходимо работать с несколькими точками?

3.6 Списки!

Списки – это хлеб насущный для многих функциональных языков. Их используют для решения множества задач, и они, несомненно, являются наиболее используемой структурой данных в Erlang. Списки могут содержать всё что угодно! Числа, атомы, кортежи, другие списки – всё что пожелаете в одной структуре. Основной способ записи для списков: `[Element1, Element2,...,ElementN]`, и в них можно смешивать несколько разных типов данных:

```
1> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atom].
[1,2,3,{numbers,[4,5,6]},5.34,atom]
```

Всё просто, не так ли?

```
2> [97, 98, 99].
"abc"
```

Ой-ой! Вот одна из тех вещей в Erlang, которая многим не нравится: строки! Строки – тоже списки, поэтому записываются они совершенно так же! Почему же люди их не любят? А вот почему:

```
3> [97,98,99,4,5,6].
[97,98,99,4,5,6]
4> [233].
"é"
```

Erlang напечатает список чисел в числовом представлении, только если хотя бы одно из них невозможно представить в виде буквы! Настоящих строк в Erlang просто нет! Вы ещё не раз встретитесь с этой особенностью, и из-за неё временами будете ненавидеть язык. Но не отчаивайтесь – есть и другие способы записи строк. С ними мы познакомимся чуть позже в этой главе.

Не забывайтесь:
Возможно, вы слышали, что Erlang плохо подходит для работы со строками, потому что, в отличие от большинства других языков, в нём нет встроенных строк. Так вышло потому, что Erlang был создан и использовался в телекоммуникационных компаниях. Они никогда (или редко) не использовали строки, и поэтому им просто не приходило в голову официально добавить их в язык. Тем не менее, недостатки Erlang в области строковых преобразований со временем устраняются. Виртуальная машина имеет встроенную поддержку Unicode-строк, и в целом операции со строками постоянно ускоряются.
Строки также можно хранить в виде двоичной структуры данных, что увеличивает эффективность хранения и скорость обработки. Но всё же в стандартной библиотеке не хватает некоторых функций. Не вызывает сомнений тот факт, что в Erlang можно работать со строками, но для решения задач, в которых нужно делать много строковых преобразований, лучше подходят другие языки, например Perl или Python.

Для склейки списков используют оператор `++`. Противоположное действие выполняет оператор `--`, который удаляет элементы из списка:

```
5> [1,2,3] ++ [4,5].
[1,2,3,4,5]
6> [1,2,3,4,5] -- [1,2,3].
[4,5]
7> [2,4,2] -- [2,4].
[2]
8> [2,4,2] -- [2,4,2].
[]
```

Оба оператора, `++` и `--` – ассоциативны справа. Это означает, что элементы нескольких последовательных операций `++` и `--` будут обрабатываться справа налево, как в следующем примере:

```
9> [1,2,3] -- [1,2] -- [3].
[3]
10> [1,2,3] -- [1,2] -- [2].
```

```
[2,3]
```

Продолжим. Первый элемент списка называется Головой (головным элементом), остальные элементы списка называются Хвостом (хвостовыми элементами). Для их получения будем использовать две встроенные функции:

```
11> hd([1,2,3,4]).
1
12> tl([1,2,3,4]).
[2,3,4]
```

Замечание: встроенными обычно называют функции (ВФ), которые невозможно реализовать на чистом Erlang. Поэтому они реализованы на C или любом другом языке, на котором реализован Erlang (в 80-х это был Prolog). Есть также и ВФ, которые можно реализовать на Erlang, но их всё равно пишут на C для ускорения часто используемых операций. Как пример можно привести функцию `length(List)`, которая (как вы, наверное, догадались) возвращает длину списка, который был передан ей в качестве аргумента.

Операция доступа к головному элементу или добавления к голове списка выполняется быстро и эффективно. Практически во всех случаях, в которых вы сталкиваетесь со списками, необходимо выполнять операции с головой списка. Из-за того, что эта операция используется так часто, существует более простой метод отделения головы от хвоста – при помощи сопоставления с образцом: `[Head|Tail]`. Вот как можно добавить в голову списка новый элемент:

```
13> List = [2,3,4].
[2,3,4]
14> NewList = [1|List].
[1,2,3,4]
```

При обработке списка неплохо было бы иметь возможность быстро сохранить хвост, чтобы чуть позже его обработать. Если вы помните как работают кортежи, и как мы использовали сопоставление с образцом для распаковки координат точки ($\{X,\ Y\}$), то знайте, что мы можем очень похожим методом извлечь первый элемент (головной).

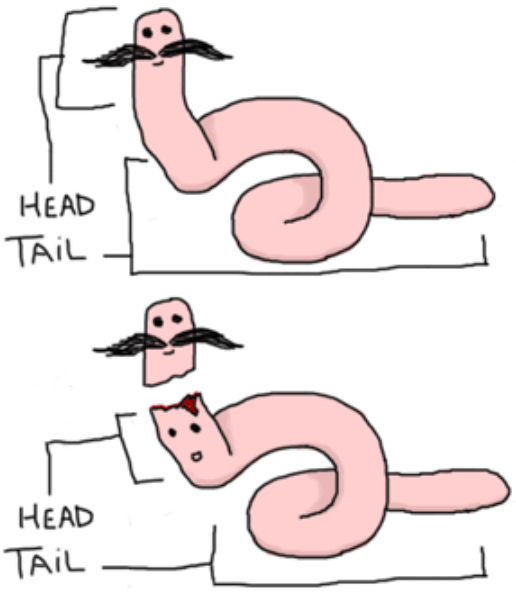
```
15> [Head|Tail] = NewList.
[1,2,3,4]
16> Head.
1
```

```
17> Tail.
[2,3,4]
18> [NewHead|NewTail] = Tail.
[2,3,4]
19> NewHead.
2
```

Оператор `|` называется cons-оператором (конструктором). Любой список можно построить при помощи cons-оператора и значений списка:

```
20> [1 | []].
[1]
21> [2 | [1 | []]].
[2,1]
22> [3 | [2 | [1 | []] ] ].
[3,2,1]
```

Таким образом, любой список можно построить по следующей формуле: `[Term1| [Term2 | [...| [TermN]]]]...`. То есть, списки можно определить рекурсивно как заголовок, за которым следует хвост, который, в свою очередь является заголовком, за которым следуют другие заголовки. В этом смысле можно представить список в виде земляного червя: можно разрубить его пополам, и вот у вас уже два червя.



Способ построения списков в Erlang иногда немного озадачивает людей, которые не имели дело с подобными конструкторами. Чтобы

немного привыкнуть к этой записи, попробуйте прочитать следующие примеры (подсказка: все они равнозначны друг другу):

```
[a, b, c, d]
[a, b, c, d | []]
[a, b | [c, d]]
[a, b | [c | [d]]]
[a | [b | [c | [d]]]]
[a | [b | [c | [d | []]]]]
```

Если вам понятна эта концепция, то будут понятны и списочные выражения.

Замечание: при использовании формы `[1 | 2]` мы получаем так называемый «неправильный список». Неправильные списки будут работать в сопоставлении с образцом вида `[Head|Tail]`, но стандартные функции Erlang (даже `length()`) с ними работать не будут. В этих операциях можно использовать только правильные списки. Последним элементом правильных списков всегда является пустой список. Когда мы объявляем список таким способом: `[2]`, то автоматически формируется правильный список. Поэтому запись `[1|[2]]` сгенерирует правильный список! Хотя неправильные списки синтаксически верны, их редко используют где-то ещё, кроме пользовательских структур данных.

3.7 Списочные выражения

Списочные выражения дают возможность формировать или менять списки. Программы, в которых используются списочные выражения, намного короче и проще для понимания. Этот способ основан на записи математических множеств; если вы когда-либо занимались теорией множеств, или сталкивались с математической записью, то наверняка знаете как она работает. Мы задаём множество через определение свойств, которым должны удовлетворять его элементы. Сначала списочные выражения могут показаться немного сложными, но усилия, потраченные на их понимание, стоят того. С их помощью код становится чище и короче, поэтому вводите примеры и пытайтесь их понять!

Приведём пример математической записи множества: $\{x \in \mathbb{R} : x = x^2\}$. Эта запись говорит о том, что вы хотите получить действительные числа, которые равны собственному квадрату. Результатом будет множество $\{0, 1\}$. В качестве примера более простой записи можно привести

$\{x : x > 0\}$. В результате мы должны получить все числа больше 0.

Списочные выражения в Erlang можно представить как построение множеств из других множеств. Пусть нам дано множество $\{2n : n \text{ in } L\}$, где L – список $[1, 2, 3, 4]$. В Erlang это можно записать как:

```
1> [2*N || N <- [1,2,3,4]].
[2,4,6,8]
```

Если сравнить математическую запись с записью в Erlang, то можно заметить, что они не так уж сильно различаются: фигурные скобки ({}) становятся квадратными ([]), двоеточие (:) становится двумя вертикальными чертами (||), а слово «in» переходит в символ стрелки ($<-$). Мы просто заменяем символы на другие, но логика остаётся прежней. В примере, приведённом выше, каждое значение $[1, 2, 3, 4]$ последовательно помещается в N (проводится операция сопоставления с образцом для каждого значения в списке). Стрелка выполняет ту же функцию, что и оператор $=$, с тем лишь отличием, что она не бросает исключения.

Также в списочные выражения можно добавлять ограничивающие условия, используя операции, которые возвращают булевы значения. Если нам понадобились чётные числа от единицы до десяти, мы можем записать что-то вроде:

```
2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].
[2,4,6,8,10]
```

Где конструкция $X \text{ rem } 2 =:= 0$ проверяет число на чётность. Практическая польза этого подхода становится ясна, когда нам нужно применить функцию к каждому элементу списка или наложить на элементы какие-либо ограничения и т.д. К примеру, мы владеем рестораном. Входит посетитель, видит наше меню и спрашивает, может ли он получить цены всех блюд, которые стоят от $3 до $10, включая налоги (скажем, 7%), и налог должен быть применён после сравнения.

```
3> RestaurantMenu = [{steak, 5.99}, {beer, 3.99}, {
   poutine, 3.50}, {kitten, 20.99}, {water, 0.00}].
[{steak,5.99},
    {beer,3.99},
    {poutine,3.5},
    {kitten,20.99},
{water,0.0}]
4> [{Item, Price*1.07} || {Item, Price} <-
   RestaurantMenu, Price >= 3, Price =< 10].
```

```
[{steak,6.409300000000001},{beer,4.2693},{poutine,3.745
    }]
```

Конечно, неплохо было бы округлить числа для улучшения читаемости, но вы поняли в чём смысл. Таким образом, списочные выражения в Erlang готовят по следующему рецепту: `NewList = [Expression || Pattern <- List, Condition1, Condition2, ... ConditionN]`. Элемент `Pattern <- List` называется Генератором. И их может быть несколько!

```
5> [X+Y || X <- [1,2], Y <- [2,3]].
[3,4,4,5]
```

Это выражение выполняет следующие операции: `1 + 2`, `1 + 3`, `2 + 2`, `2 + 3`. Так что в обобщённом виде вы получите: `NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN, Condition1, Condition2, ... ConditionM]`. Обратите внимание, что выражения-генераторы можно комбинировать с сопоставлением по образцу, и использовать в качестве фильтра:

```
6> Weather = [{toronto, rain}, {montreal, storms}, {
    london, fog},
6>            {paris, sun}, {boston, fog}, {vancouver,
    snow}].
[{toronto,rain},
     {montreal,storms},
     {london,fog},
     {paris,sun},
     {boston,fog},
{vancouver,snow}]
7> FoggyPlaces = [X || {X, fog} <- Weather].
[london,boston]
```

Если элемент списка «Weather» не совпадает с образцом {X, fog}, то в списочном выражении он просто игнорируется, тогда как при использовании с оператором `=` было бы выброшено исключение.

Остался ещё один базовый тип данных, который мы должны рассмотреть. В Erlang встроена функциональность, которая легко и просто позволяет преобразовывать двоичные данные.

3.8 Битовый синтаксис!

Во многих языках есть поддержка манипуляций с числами, атомами, кортежами, списками, записями и/или структурами и т.д. Но большая часть этих языков содержит очень грубые методы манипулирования двоичными данными. Erlang старается предоставить полезные абстракции для работы с двоичными данными при помощи усовершенствованного сопоставления с образцом. Это превращает работу с необработанными двоичными данными в приятное и простое (я не вру) занятие. Метод был разработан для решения телекоммуникационных задач. Битовые манипуляции предоставляют уникальный синтаксис и идиомы, которые на первый взгляд могут показаться странными, но обретают смысл, если вы понимаете, как устроены биты и байты. **В противном случае вам лучше пропустить остаток этой главы.**

В соответствии с битовым синтаксисом, двоичные данные обрамляются символами << и >>, и их разделяют на читаемые сегменты при помощи запятых. Сегмент – это битовая последовательность в двоичном представлении (не обязательно с выравниванием по границе байта, но такое выравнивание происходит по умолчанию). Представим, что мы хотим хранить пиксель оранжевого цвета в 24-битном представлении. Если вам приходилось когда-либо сталкиваться с цветовым представлением в Photoshop или в CSS, то запись в формате #RRGGBB должна быть вам знакома. Оранжевый оттенок можно записать как *#F09A29*. Erlang преобразует эту запись в следующий вид:

```
1> Color = 16#F09A29.
15768105
2> Pixel = <<Color:24>>.
<<240,154,41>>
```

Это означает что-то вроде: «Расположи двоичное представление числа *#F09A29* в 24-х битах (Красный в 8 битах, Зелёный в 8 и Синий тоже в 8 битах) переменной Pixel». Значение можно будет позже взять и в неизменном виде записать в файл. На первый взгляд ничего особенного в этом нет, но если вы откроете записанный файл в текстовом редакторе, то увидите лишь несколько нечитаемых символов. Если этот файл считать при помощи Erlang, то его содержимое в двоичном представлении будет снова преобразовано в удобный формат *<<240,151,41>>*!

Но самое интересное, что для распаковки двоичных данных можно использовать сопоставление с образцом:

```
3> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
```

```
<<213,45,132,64,76,32,76,0,0,234,32,15>>
4> <<Pix1,Pix2,Pix3,Pix4>> = Pixels.
** exception error: no match of right hand side value
    <<213,45,132,64,76,32,76,
0,0,234,32,15>>
5> <<Pix1:24, Pix2:24, Pix3:24, Pix4:24>> = Pixels.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

В 3-й команде мы определили значение, которое соответствует двоичному представлению 4-х пикселов в RGB пространстве. В 4-м выражении мы попытались распаковать 4 значения из двоичной величины. После чего было выброшено исключение, так как в исходной величине не 4 сегмента, а 12! Поэтому мы говорим Erlang, что каждая переменная слева будет содержать 24 бита. Именно это и означает запись `Var:24`. Далее мы можем взять первый пиксел и распаковать его в отдельные цветовые компоненты:

```
6> <<R:8, G:8, B:8>> = <<Pix1:24>>.
<<213,45,132>>
7> R.
213
```

«Да, вышло неплохо. А что если мне нужен лишь первый компонент? Мне всё равно придётся постоянно распаковывать все значения?» Ха! Прочь сомнения! Erlang приходит на помощь, предоставляя комбинацию синтаксического сахара и сопоставления с образцом:

```
8> <<R:8, Rest/binary>> = Pixels.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
9> R.
213
```

Неплохо, да? А всё потому что Erlang воспринимает несколько способов описания двоичного сегмента. Можно использовать все перечисленные ниже:

Value

Value:Size

Value/TypeSpecifierList

Value:Size/TypeSpecifierList

где Size указывает количество бит, а TypeSpecifierList обозначает одно или несколько понятий из следующего списка:

Тип
Возможные значения: integer | float | binary | bytes | bitstring | bits | utf8 | utf16 | utf32
Так определяется вид используемых двоичных данных. Заметьте, что 'bytes' это псевдоним для 'binary', а 'bits'– для 'bitstring'. Если тип не задан, то Erlang по умолчанию использует 'integer'.

Знак
Возможные значения: signed | unsigned
Имеет значение только при сопоставлении, когда тип – integer. По умолчанию используется 'unsigned'.

Порядок байтов
Возможные значения: big | little | native
Порядок байтов имеет значение, когда задан тип integer, utf16, utf32 или float. От порядка зависит то, как система считывает двоичные данные. Например, заголовок изображения в формате BMP содержит размер файла в виде 4-байтного целого числа. Пусть размер файла равен 72 байта, тогда little-endian система представит его в виде «72,0,0,0», а big-endian система в виде «0,0,0,72». Первое будет прочитано как '72', а второе как '1207959552', поэтому старайтесь чтобы порядок байтов был верным. Также существует опция 'native', которая устанавливает порядок байт, используемый процессором. По умолчанию используется порядок 'big'.

Единичный элемент
Записывается как unit:Integer
Это размер одного сегмента в битах. Допустимый диапазон 1..256. По умолчанию равен 1 для integer, float, битовых строк и 8 для двоичных данных. Для типов utf8, utf16 и utf32 единицы определять не нужно. Произведение Размера на Единичный элемент равно количеству бит, которые занимает сегмент, и должно быть кратным 8. Задание размера элемента часто используется для выравнивания по границе байта.

Список TypeSpecifierList строится из атрибутов, разделённых символом '-'.

Несколько примеров помогут разобраться в определениях:

```
10> <<X1/unsigned>> =   <<-44>>.
<<"Ô">>
11> X1.
```

```
212
12> <<X2/signed>> =   <<-44>>.
<<"Ô">>
13> X2.
-44
14> <<X2/integer-signed-little>> =   <<-44>>.
<<"Ô">>
15> X2.
-44
16> <<N:8/unit:1>> = <<72>>.
<<"H">>
17> N.
72
18> <<N/integer>> = <<72>>.
<<"H">>
19> <<Y:4/little-unit:8>> = <<72,0,0,0>>.
<<72,0,0,0>>
20> Y.
72
```

Очевидно, что существует несколько способов читать, хранить и интерпретировать двоичные данные. Немного запутанно, но всё же намного проще, чем обычные инструменты, которые предоставляет большинство языков.

В Erlang также существуют стандартные битовые операции (побитовый сдвиг влево и вправо, битовое 'и', 'или', 'исключающее или' и 'не'). Они реализованы функциями `bsl` (Bit Shift Left (битовое смещение влево)), `bsr` (Bit Shift Right (битовое смещение вправо)), `band`, `bor`, `bxor`, и `bnot`.

```
2#00100 = 2#00010 bsl 1.
2#00001 = 2#00010 bsr 1.
2#10101 = 2#10001 bor 2#00101.
```

С такой записью и с таким битовым синтаксисом задача разбора и сопоставления с образцом двоичных данных сильно облегчается. К примеру, таким кодом можно разобрать сегменты TCP пакета:

```
<<SourcePort:16, DestinationPort:16,
AckNumber:32,
DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
CheckSum: 16, UrgentPointer:16,
Payload/binary>> = SomeBinary.
```

Ту же самую логику можно применять к любым двоичным данным: кодированное видео, изображения, реализация стороннего протокола и т.д.

Не забывайтесь:
Erlang, по сравнению с C и C++, язык медленный. Чтобы заниматься чем-нибудь вроде конвертации видео или изображений на Erlang, нужно быть терпеливым человеком. Применение битового синтаксиса, конечно, делает эти задачи чрезвычайно интересными, о чём я намекнул чуть выше. Но Erlang просто не очень подходит для тяжёлых вычислительных задач.
Имейте в виду, что Erlang весьма быстр там, где не нужно всё время молотить числа: обработка событий, передача сообщений (здесь весьма кстати приходятся чрезвычайно лёгкие атомы) и т.д. Erlang может реагировать на события в сроки, измеряемые миллисекундами, и поэтому прекрасно годится для применения в задачах мягкого реального времени (soft real time).

Для битовой записи также существует совершенно другое приложение: битовые строки. Они не были частью языка, задуманной при проектировании. Их «навесили» позже, как и строки реализованные в виде списков. Но они намного эффективнее строковых списков в отношении занимаемого пространства. Происходит это потому, что обычные списки реализованы как связные списки (1 «узел» на каждую букву), а битовые строки больше похожи на массивы языка C. Для битовых строк используется синтаксис <<“this is a bit string!”>>. Недостатком битовых строк в сравнении со списками является потеря простоты в операциях сопоставления с образцом и в строковых манипуляциях. По этой причине люди используют битовые строки для хранения текста, который не будет интенсивно меняться, либо в случаях, когда необходимо эффективно использовать память.

Замечание: хотя битовые строки довольно легковесны, их лучше не применять для пометки кортежей. Мысль об использовании строкового литерала при записи выражения {<<“temperature”>>,50} выглядит весьма соблазнительно, но в таких случаях всегда используйте атомы. Ранее в этой главе было сказано, что атомы занимают всего 4 или 8 байт, независимо от их длины. Когда вы их используете, нет практически никаких издержек при копировании данных из функции в функцию, или при их пересылке другому Erlang узлу на удалённом сервере.
Верно и обратное: не заменяйте строки атомами ради их лёгкости. Строками можно манипулировать (разбивка, регулярные выражения и т.д.) в то время как атомы можно только сравнивать, и больше с ними ничего не сделаешь.

3.9 Битовые выражения

Битовые выражения являются для битового синтаксиса тем же, чем являются списочные выражения для списков: способом сделать код коротким и ясным. В мире Erlang они появились относительно недавно. Они присутствовали в предыдущих версиях Erlang, но модуль, который их реализовывал, требовал для работы специальный флаг компиляции. Начиная с ревизии R13B (о которой здесь идёт речь), они вошли в стандарт и могут быть использованы где угодно, включая оболочку:

```
1> [ X || <<X>> <= <<1,2,3,4,5>>, X rem 2 == 0].
[2,4]
```

Единственное отличие в синтаксисе от обычных списочных выражений – это замена символа <- на <=, и использование двоичных данных (<<>>) вместо списков ([]). Ранее в этой главе мы видели пример, в котором нам было дано двоичное значение, представляющее собой множество пикселов. К нему мы применили сопоставление с образцом, чтобы вычленить RGB значения для каждого пиксела. В принципе, неплохой метод, но для больших структур такой код будет сложно читать и поддерживать. То же самое можно проделать при помощи однострочного битового выражения, которое выглядит значительно чище:

```
2> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
3> RGB = [ {R,G,B} || <<R:8,G:8,B:8>> <= Pixels ].
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

Замена `<-` на `<=` позволяет использовать двоичный поток как генератор. Битовое выражение, по сути, преобразовывает двоичные данные в кортежи целых значений. Существует также другой синтаксис битовых выражений, который позволяет совершить обратное преобразование:

```
4> << <<R:8, G:8, B:8>> || {R,G,B} <- RGB >>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

Будьте осторожны. Для элементов результирующей двоичной величины необходимо чётко определить размер, в случае если генератор возвратил двоичные данные:

```
5> << <<Bin>> || Bin <- [<<3,7,5,4,7>>] >>.
** exception error: bad argument
6> << <<Bin/binary>> || Bin <- [<<3,7,5,4,7>>] >>.
<<3,7,5,4,7>>
```

Можно также использовать битовые выражения с двоичным генератором, если соблюдать вышеуказанное правило о предопределённом размере:

```
7> << <<(X+1)/integer>> || <<X>> <= <<3,7,5,4,7>> >>.
<<4,8,6,5,8>>
```

Замечание: на момент написания этого текста, битовые выражения использовались достаточно редко и были плохо документированы. Поэтому было принято решение не углубляться дальше основ. Чтобы получить более полное представление о битовом синтаксисе в целом, читайте документ, описывающий его спецификацию.

Глава 4

Модули

4.1 Что такое модули

Работа с интерактивной оболочкой часто считается жизненно важной частью работы с динамическими языками программирования. В ней удобно тестировать различный код и программы. Чтобы использовать большую часть основных типов данных в Erlang, даже не нужно открывать текстовый редактор или сохранять файлы. Можете отставить клавиатуру в сторону, сказать что на сегодня довольно, и пойти гулять. Но если вы на этом остановитесь, то будете ужасным программистом на Erlang. Чтобы использовать код, его нужно где-то хранить!

Для этого и существуют модули. Модуль – это несколько функций, сгруппированных в единый файл под одним именем. Все функции в Erlang должны определяться в модулях. Вы уже использовали модули, возможно даже не догадываясь об этом. Встроенные функции `hd` и `tl`, которые упоминались в предыдущей главе, на самом деле входят в модуль `erlang`, так же как и все арифметические, логические и булевы операторы. ВФ из модуля `erlang` отличаются от других функций тем, что при использовании Erlang они импортируются автоматически. Вызов любой другой функции, определённой в модуле, должен выглядеть так: `Module:Function(Arguments)`.

Смотрите:

```
1> erlang:element(2, {a,b,c}).
b
```

```
2> element(2, {a,b,c}).
b
3> lists:seq(1,4).
[1,2,3,4]
4> seq(1,4).
** exception error: undefined shell command seq/2
```

В этом примере функция `seq` из модуля list не была автоматически импортирована, тогда как `element` была. Ошибку "undefined shell command" генерирует оболочка, которая ищет и не находит команду оболочки (например, такую как `f()`). Некоторые функции из модуля `erlang` не импортируются автоматически, но их используют не так часто.

Согласно логике, вы должны помещать функции, которые касаются похожих вещей, в один модуль. Общие операции над списками хранятся в модуле `lists`, а функции ввода-вывода (которые позволяют выводить данные в консоль или файл), сгруппированы в модуле `io`. Единственный модуль, который не подчиняется этой схеме, это вышеупомянутый модуль `erlang`, который содержит математические функции, функции преобразования, мультипроцессинга, изменения настроек виртуальной машины и т.д. У этих функций нет ничего общего, кроме того что все они считаются встроенными. Лучше не создавать модули, похожие на `erlang`, и сконцентрироваться на ясном логическом разделении функциональности.

4.2 Объявление модуля

Можно объявлять при написании модуля два вида сущностей: функции и атрибуты. Атрибуты это метаданные, которые описывают сам модуль: его имя, функции, которые должны быть видимы снаружи, автора кода и прочее. Эти метаданные весьма полезны, так как подсказывают компилятору, как он должен производить обработку, а также позволяют людям извлекать из скомпилированного кода полезную информацию, не обращаясь к исходному коду.

В Erlang существует большое количество разнообразных атрибутов модулей. Вы и сами можете объявить любые атрибуты на собственный вкус. Но также существуют некоторые предопределённые атрибуты, которые будут появляться в вашем коде чаще других. Все атрибуты модулей записываются в форме `-Name(Attribute).`. Чтобы ваш

модуль можно было скомпилировать, необходим лишь один атрибут:

-module(Name).

Этот атрибут всегда указывается первым оператором в файле и обозначает имя текущего модуля, где *Name* это 3.3 атом. Это имя используется при вызове функций из другого модуля. Вызовы записывают в виде `M:F(A)`, где *M* это имя модуля, *F* это имя функции и *A* её аргументы.

Настало время немного попрограммировать! Наш первый модуль будет простым и бесполезным. Откройте текстовый редактор, введите указанную ниже строку, и сохраните под именем `useless.erl`:

```
-module(useless).
```

Всего лишь одна эта строка уже является рабочим модулем. Конечно, без функций в нём нет никакого смысла. Сначала давайте решим, какие функции будут экспортироваться из нашего «бесполезного» модуля. Для этого нам понадобится ещё один атрибут:

-export([Function1/Arity, Function2/Arity,...,FunctionN/Arity]).

Он используется для определения функций модуля, которые можно вызывать извне. Атрибут содержит список функций с соответствующей им арностью. Арность функции это целое число, которые соответствует количеству аргументов, которые принимает функция. Это важная информация, поскольку разные функции, определённые в модуле, могут иметь одинаковое имя, только если их арность различается. Поэтому функции `add(X, Y)` и `add(X, Y, Z)` будут считаться различными и записываться в виде `add/2` и `add/3` соответственно.

> **Замечание:** экспортируемые функции представляют собой интерфейс модуля. Важно чтобы интерфейс сообщал о модуле только то, что необходимо для его использования и ничего более. Это позволяет менять скрытые детали вашей реализации, не нарушая работу кода, который может полагаться на ваш модуль.

Сначала наш модуль экспортирует полезную функцию под названием «add», которая принимает два аргумента. Атрибут `-export` можно добавить после объявления модуля:

```
-export([add/2]).
```

Теперь напишем функцию:

```
add(A,B) ->
    A + B.
```

Синтаксис функции соответствует виду `Name(Args) -> Body.`, где *Name* должен быть атомом, а *Body* это одно, либо несколько выражений

Erlang, разделённых запятыми. Функция должна заканчиваться точкой. Обратите внимание, что Erlang не использует ключевое слово «return». От «return» никакой пользы! Вместо этого автоматически будет возвращён результат выполнения последнего выражения в функции, и вам ничего для этого не нужно будет делать.

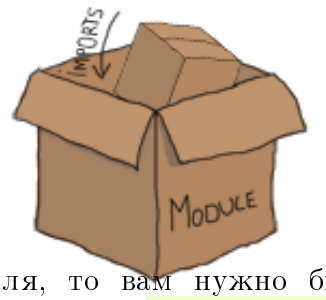
Добавьте следующую функцию (конечно же, какое руководство без «Hello world»! Хоть даже и в четвёртой главе!), и не забудьте добавить её в атрибут `-export`.

```
%% Shows greetings.
%% io:format/1 is the standard function used to output text.
hello() ->
io:format("Hello, world!~n").
```

Из этой функции нам станет понятно, что каждый комментарий должен начинаться с символа `%` и состоять из одной строки (то, что в примере используется `%%`, не более чем вопрос стиля). Также функция `hello/0` демонстрирует как в вашем модуле можно вызывать функции из внешних модулей. В данном случае это функция `io:format/1`, которая является стандартной функцией для вывода текста (что объясняется в комментарии).

Добавим ещё одну функцию, которая будет использовать и `add/2`, и `hello/0`:

```
greet_and_add_two(X) ->
    hello(),
    add(X,2).
```



Не забудьте добавить `greet_and_add_two/1` в список экспортируемых функций. Для вызовов `hello/0` и `add/2` указывать имя модуля не нужно, так как они были объявлены в текущем модуле.

Если бы вы захотели вызвать функцию `io:format/1` так же как `add/2` или любую другую функцию, определённую внутри модуля, то вам нужно было бы добавить в начале файла следующий атрибут: `-import(io, [format/1]).`. После этого можно сделать вызов `format("Hello, World!~n").` напрямую. Общий вид атрибута `-import` подчиняется следующей формуле:

```
-import(Module, [Function1/Arity,..., FunctionN/Arity]).
```

Импорт функции это просто быстрый способ получить к ней доступ. Программистам на Erlang не рекомендуется использовать атрибут `-import`, так как считается, что это уменьшает читаемость кода. К примеру, помимо функции `io:format/2` существует также и функция `io_lib:format/2`. Чтобы понять, какую из них использовал программист, придётся перейти в начало файла и посмотреть, из какого модуля функция была импортирована. Поэтому использование имени модуля в качестве префикса считается хорошим стилем. Обычно импортируют лишь функции, определённые в модуле lists, так как они используются намного чаще других.

Теперь ваш модуль `useless` должен принять следующий вид:

```
-module(useless).
-export([add/2, hello/0, greet_and_add_two/1]).

add(A,B) ->
A + B.

%% Shows greetings.
%% io:format/1 is the standard function used to output text.
hello() ->
io:format("Hello, world!~n").

greet_and_add_two(X) ->
hello(),
add(X,2).
```

Мы закончили работать с нашим модулем «useless». Можете сохранить файл под именем `useless.erl`. Имя файла должно состоять из имени модуля, определённого в атрибуте `-module`, и заканчиваться расширением '.erl', которое стандартно используется для файлов с исходным кодом Erlang.

Перед тем как скомпилировать модуль и, наконец-то, опробовать его в деле, мы увидим как определять и использовать макросы. В Erlang макросы очень похожи на выражения «#define» в языке C, и, главным образом, используются для определения коротких функций и констант. Они представляют собой простые текстовые выражения, которые будут заменены перед компиляцией кода. Макросы полезны для того, чтобы не раскидывать по тексту ваших модулей «магические» значения. Макрос определяется как атрибут модуля в виде `-define(MACRO, some_value).` и его можно использовать внутри модуля как `?MACRO`. Макрос в виде «функции» можно записать как `-define(sub(X, Y), X - Y).` и использовать в виде `?sub(23, 47)`. Такой макрос позже будет заменён компилято-

ром на выражение `23 - 47`. Кто-то использует более сложные макросы, но общий синтаксис не меняется.

4.3 Компилируем код

Чтобы код Erlang мог использоваться виртуальной машиной, его компилируют в байт-код. Компилятор можно вызывать несколькими способами: из командной строки как `$ erlc flags file.erl`, из оболочки или в модуле как `compile:file(FileName)`, в оболочке как `c()` и т.д.

Пора скомпилировать наш бесполезный модуль и опробовать его. Откройте оболочку Erlang и введите:

```
1> cd("/path/to/where/you/saved/the-module/").
"Path Name to the directory you are in"
ok
```

Оболочка будет искать по умолчанию файлы в той же директории, из которой она стартовала, а также в стандартной библиотеке. Функция `cd/1` определена только в оболочке Erlang. Она позволяет изменить текущую директорию. Пользователи Windows должны использовать в качестве разделителя директорий прямой слеш (косую черту `/`). Когда мы поменяли текущую директорию на ту, в которой содержится наш модуль, вводим следующую команду:

```
2> c(useless).
{ok,useless}
```

Если сообщение, которое вы получили, отличается от приведённого выше, то убедитесь что файл назван правильно, что вы находитесь в правильной директории и в вашем модуле нет ошибок. Когда компиляция пройдёт успешно, вы увидите, что в директории помимо `useless.erl` появился ещё один файл – `useless.beam`. Это скомпилированный модуль. Попробуем воспользоваться нашими функциями:

```
3> useless:add(7,2).
9
4> useless:hello().
Hello, world!
ok
5> useless:greet_and_add_two(-3).
Hello, world!
-1
6> useless:not_a_real_function().
** exception error: undefined function useless:
    not_a_real_function/0
```

Функции работают как и было задумано: `add/2` складывает числа, `hello/0` выводит «Hello, world!», а `greet_and_add_two/1` делает и то и другое! Вы, вероятно, задали себе вопрос: а почему функция `hello/0` после вывода текста возвращает атом «ok»? Потому что функции и выражения в Erlang **всегда** должны что-то возвращать, даже когда в других языках они это делать не обязаны. Поэтому функция `io:format/1` возвращает «ok», чтобы обозначить, что выполнение прошло нормально и ошибки отсутствуют.

В выражении 6 отображена ошибка, которая была сгенерирована из-за отсутствия функции. Если вы забыли проэкспортировать функцию, то получите сообщение именно такого типа.

Замечание: расширение '.beam', если вам интересно, означает *Bogdan/Björn's Erlang Abstract Machine* (так называется виртуальная машина). Существуют также и другие виртуальные машины для Erlang, но сейчас они не более чем достояние истории, и их не используют. Среди них JAM (Joe's Abstract Machine, вобравшая черты Prolog WAM, и старая BEAM, которая предпринимала попытки компиляции из Erlang в C, и после – в нативный код. Измерения показали, что выигрыш от применения этого метода был слишком мал, поэтому от него отказались.

Существует много флагов, которые позволяют тоньше контролировать процесс компиляции модуля. Их список можно найти в документации Erlang. Вот наиболее востребованные флаги:

-debug_info
Добавляет в модуль отладочную информацию, которая необходима для работы инструментов Erlang, таких как отладчик, утилиты статического анализа и покрытия кода.

-{outdir,Dir}
Компилятор Erlang будет по умолчанию создавать "beam" файлы в текущей директории. Этот флаг позволяет задать путь к директории, в которой будут сохраняться скомпилированные файлы.

-export_all
Флаг заставляет игнорировать атрибут модуля `-export`. При этом будут проэкспортированы все функции, которые в нём определены. Главным образом этот флаг полезен при тестировании и разработке нового кода, и его не следует использовать на рабочих системах.

{d,Macro} или {d,Macro,Value}
Определяет макрос, который можно будет использовать в модуле. *Macro* должен быть атомом. Чаще всего этот флаг используется при юнит-тестировании, чтобы гарантировать, что тестовые функции будут создаваться и экспортироваться только когда в них есть необходимость. По умолчанию элемент *Value* имеет значение «true», если не указан в списке явно.

Чтобы скомпилировать наш модуль `useless` с использованием флагов, нужно выполнить одну из следующих директив:

```
7> compile:file(useless, [debug_info, export_all]).
{ok,useless}
8> c(useless, [debug_info, export_all]).
{ok,useless}
```

Также можно схитрить и определить флаги компиляции при помощи атрибутов прямо в самом модуле. Чтобы получить такой же результат как в выражениях 7 и 8, необходимо добавить в модуль следующую строку:

```
-compile([debug_info, export_all]).
```

Далее необходимо лишь скомпилировать модуль, и вы получите тот же результат, что и с флагами переданными вручную. А теперь, когда мы можем записывать функции, компилировать их и исполнять, настало время узнать, что же мы сможем со всем этим сделать!

Замечание: модуль также можно скомпилировать в нативный код. Компиляция в нативный код доступна не для каждой платформы и ОС. Для платформ, которые допускают такую компиляцию, можно добиться ускорения программ (по неточным данным приблизительно на 20%). Для компиляции в нативный код необходимо использовать модуль `hipe`. Компиляция осуществляется при помощи команды: `hipe:c(Module,OptionsList).` Также в оболочке можно использовать команду `c(Module,[{hipe,o3}]).` Обратите внимание, что .beam файл, полученный в результате такой компиляции, нельзя переносить между платформами, тогда как обычные файлы можно.

4.4 Подробнее о модулях

Прежде чем переходить к написанию функций и кода, польза которых сомнительна, необходимо упомянуть ещё несколько фактов, которые в будущем могут пригодиться.

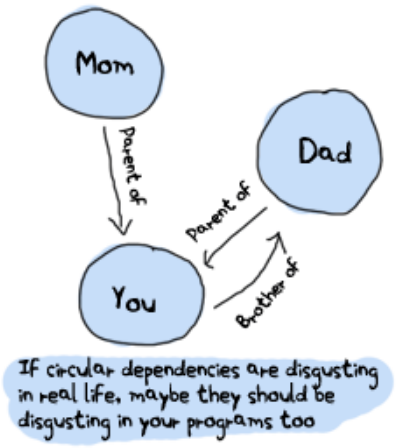
Первый из них касается метаданных в модулях. Я упомянул в начале главы, что атрибуты модуля –это метаданные, которые описывают сам модуль. Как мы можем получить доступ к этим метаданным, если у нас нет доступа к исходному коду модуля? В этом нам поможет компилятор. При компиляции он соберёт атрибуты и сохранит их (вместе с другой информацией) в функции `module_info/0`. Вот так будут выглядеть метаданные модуля `useless`:

```
9> useless:module_info().
[{exports,[{add,2},
           {hello,0},
           {greet_and_add_two,1},
           {module_info,0},
    {module_info,1}]},
    {imports,[]},
    {attributes,[{vsn,[174839656007867314473085021121413256129]}]
        },
    {compile,[{options,[]},
           {version,"4.6.2"},
           {time,{2009,9,9,22,15,50}},
{source,"/home/ferd/learn-you-some-erlang/useless.erl"}]}]
10> useless:module_info(attributes).
[{vsn,[174839656007867314473085021121413256129]}]
```

Также в вышеприведённом тексте есть упоминание дополнительной функции `module_info/1`, которая позволит получить каждый элемент метаданных по отдельности. В метаданных содержатся экспортируемые функции, импортируемые функции (в данном случае ни одной), атрибуты (в них можно хранить метаданные, которые определены вами), информация о компиляции и ключи компиляции. Если бы вы решили добавить в модуль атрибут `-author("An Erlang Champ").`, он оказался бы в том же разделе, где и `vsn`. Когда дело доходит до рабочей системы, для атрибутов модулей маловато применений, но они могут быть полезны для реализации маленьких хитростей: я использую их в тестовом скрипте, чтобы описывать функции для которых юнит-тесты оставляют желать лучшего. Скрипт сканирует атрибуты модуля, находит функции, снабжённые комментариями, и выдаёт о них предупреждения.

Замечание: `vsn` это уникальное значение, которое генерируется автоматически и отличается для каждой версии вашего кода, исключая комментарии. Это значение используется при горячей загрузке кода (обновление приложения во время исполнения, без необходимости его остановки), а также некоторыми инструментами, которые связаны с управлением релизами. Если хотите, можете сами указать значение для `vsn`: просто добавьте в модуль атрибут `-vsn(VersionNumber)`.

Ещё один момент, на который стоит обратить внимание при проектировании модулей: избегайте циклических зависимостей! Модуль A не должен вызывать модуль B, который в свою очередь вызывает модуль A. Такие зависимости приводят к усложнению поддержки кода. Кому хочется проснуться посреди ночи от того, что маньяк-разработчик пытается выдавить вам глаза из-за чудовищного кода, который вы написали.



По той же причине (поддержка кода и забота о вашим зрении), обычно считается хорошим тоном размещение рядом функций близких по назначению. В качестве примера можно привести функции запуска и остановки приложения, или создания и удаления записи в некоторой базе данных.

Ну что, довольно морализаторства. Готовы узнать ещё немного об Erlang?

Глава 5

Синтаксис функций

5.1 Сопоставление с образцом

Теперь, когда у нас есть возможность сохранять и компилировать наш код, мы можем начать писать более сложные функции. Те, что мы написали раньше, были чрезвычайно просты и восторгаться в них было нечем. Перейдём к более интересным вещам. Первая функция, которую мы напишем, будет выдавать различные приветствия в зависимости от переданного ей пола. В большинстве языков нужно было бы написать что-то близкое к этому:

```
function greet(Gender,Name)
    if Gender == male then
        print("Hello, Mr. %s!", Name)
    else if Gender == female then
        print("Hello, Mrs. %s!", Name)
    else
        print("Hello, %s!", Name)
end
```

С использованием сопоставления с образцом, Erlang позволяет избавиться от кучи шаблонного кода. В Erlang подобная функция будет выглядеть так:

```
greet(male, Name) ->
    io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
    io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
    io:format("Hello, ~s!", [Name]).
```

Признаю, что функция вывода в Erlang выглядит намного уродливее, чем в других языках, но смысл не в этом. Главное отличие в том, что при использовании сопоставления с образцом, мы убили сразу двух зайцев: определили, какие части функции будут использованы, и связали значения с переменными. Нам не нужно сначала связывать значения, а потом их сравнивать! Поэтому вместо:

```
function(Args)
    if X then
        Expression
    else if Y then
        Expression
    else
        Expression
```

Мы напишем:

```
function(X) ->
    Expression;
function(Y) ->
    Expression;
function(_) ->
    Expression.
```

и придём к тем же результатам, используя более декларативный стиль. Каждое объявление `function` называется *функциональным выражением*. Функциональные выражения должны разделяться символом точки с запятой (`;`) и вместе они формируют *объявление функции*. Объявление функции считается одной большой конструкцией, поэтому заключительное функциональное выражение завершается точкой. Этот способ использования элементов для определения потока задач (workflow), может показаться немного «странным», но вы к нему привыкнете. Ну или хотя бы надейтесь на то, что это случится, потому что иного пути не существует!

Замечание: форматирование при помощи `io:format` осуществляется посредством токенов, которые определяют замены в строке. Для обозначения токенов используется символ тильды (`~`). Есть встроенные токены, например `~n` . Этот токен будет преобразован в перевод строки. Большинство других токенов обозначают способ форматирования данных. Например, вызов функции `io:format("~s!~n",["Hello"]).` заключает в себе токен `~n` , и токен `~s` , который в качестве аргументов принимает строки и битовые строки. После применения форматирования, строка примет вид `"Hello!n"` . Ещё одним широко используемым токеном является `~p` . Он печатает содержимое переменной Erlang, учитывая форматирование (с добавлением отступов и всего прочего).
Мы ознакомимся с подробностями применения функции `io:format` в последующих главах, когда будем углублённо работать с вводом/выводом. Но сейчас можете попробовать исполнить следующие вызовы функций: `io:format("~s~n",[<<"Hello">>])` , `io:format"~p~n", [<<"Hello">>])` , `io:format("~~~n")` , `io:format("~f~n", [4.0])` , `io:format("~30f~n", [4.0])` . Это лишь малая часть возможных операций. Команды немного похожи на `printf` из другого языка. Если не можете дотерпеть до главы, в которой описывается ввод/вывод, то почитайте документацию онлайн.

В функциях сопоставление с образцом может принимать ещё более сложные и мощные формы. Как вы, может быть, помните, несколько глав назад мы применяли сопоставление с образцом, чтобы получать головную и хвостовую части списков. Давайте попробуем это сделать снова! Создайте новый модуль и назовите его `functions` . В нём мы напишем несколько функций, которые позволят нам исследовать пути использования сопоставления с образцом:

```
-module(functions).
-compile(export_all). %% replace with -export() later, for God's
    sake!
```

Первой нашей функцией станет `head/1` , которая будет вести себя точно так же как `erlang:hd/1` : принимать список в качестве аргумента и возвращать его первый элемент. Мы будем это делать при помощи оператора cons (`|`):

```
head([H|_]) -> H.
```

Если вы введёте в оболочке `functions:head([1,2,3,4]).` (после того как скомпилируете модуль), то вам будет возвращено значение '1'. Чтобы получить второй элемент, следовательно, вам нужно создать функцию:

```
second([_,X|_]) -> X.
```

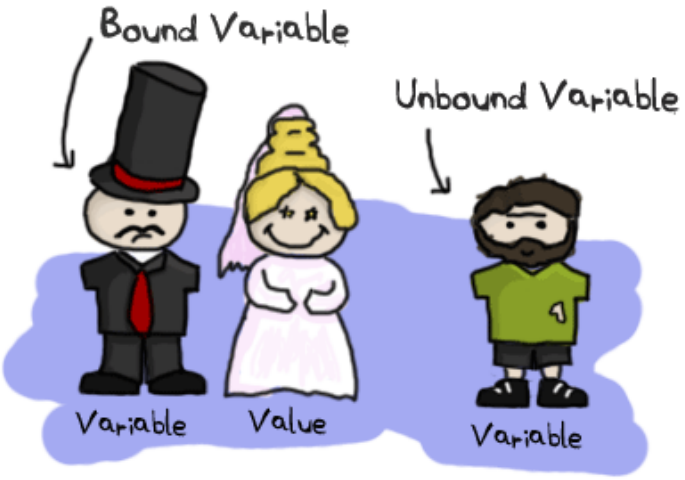
Список будет разобран Erlang при помощи сопоставления с образцом. Поробуйте выполнить эту операцию в оболочке!

```
1> c(functions).
{ok, functions}
2> functions:head([1,2,3,4]).
1
3> functions:second([1,2,3,4]).
2
```

Можно повторять этот процесс для списков сколько угодно, но для тысячи значений это было бы непрактично. Для того, чтобы это исправить, мы будем писать рекурсивные функции, которые разберём чуть позже. А сейчас давайте сосредоточимся на сопоставлении с образцом. Концепция свободной и связанной переменной, которую мы обсуждали в главе 3 также справедлива и для функций. Мы можем сравнивать и выяснять, одинаковы ли два параметра, которые переданы в функцию. Для этого мы создадим функцию `same/2`, которая принимает два аргумента и сообщает, совпадают ли они друг с другом:

```
same(X,X) ->
true;
same(_,_) ->
false.
```

Вот так всё просто. Прежде чем объяснять как работают функции, мы ещё раз повторим концепцию связанной и свободной переменной, на всякий случай:



На этой картинке жених опечален, потому что в Erlang переменные никогда не могут менять значение: конец свободе! А если серьёзно, то свободными называются переменные, у которых ничего нет (как у нашего маленького бомжика справа). Процесс связывания пе-

ременной заключается в простом присоединении значения к свободной переменной. Если вы захотите присвоить значение связанной переменной, Erlang сгенерирует ошибку. Но ошибки не будет, если новое значение совпадает со старым. Представим, что парень слева женится на девушке, у которой есть сестра-близнец. Если рядом появится сестра, жених не отличит её от невесты, и никак не отреагирует. Если же появится посторонняя женщина, жених будет недоволен. Если вам неясна эта концепция, можете вернуться к разделу о 3.2 Неизменных переменных.

Вернёмся к нашему коду. Когда вы вызовете функцию `same(a,a)`, первая переменная X считается свободной и автоматически принимает значение `a`. Далее Erlang переходит ко второму аргументу, видит что переменная X уже связана. После этого он сравнивает её со значением `a`, которое было передано в качестве второго аргумента, и определяет, совпадают ли они друг с другом. Операция сопоставления с образцом завершается успешно и функция возвращает `true`. Если два значения различаются, то сравнение завершится неудачей, и управление перейдёт ко второму функциональному выражению, которое не проверяет аргументы (когда выбирать не из чего, нечего перебирать!), а просто возвращает false. Заметьте, что эта функция может фактически принимать абсолютно любые аргументы! Она работает не только со списками или с одиночными переменными, а с любыми типами данных. Рассмотрим пример посложнее: функцию, которая печатает дату, но лишь тогда, когда она правильно отформатирована:

```
valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
    io:format("The Date tuple (~p) says today is: ~p/~p/~p,~n",[
        Date,Y,M,D]),
    io:format("The time tuple (~p) indicates: ~p:~p:~p.~n", [Time
        ,H,Min,S]);
valid_time(_) ->
    io:format("Stop feeding me wrong data!~n").
```

Есть также возможность использовать оператор `=`, который позволяет сопоставлять не только содержимое кортежа (`{Y,M,D}`), но и сам кортеж в целом (*Date*). Функцию можно протестировать следующим образом:

```
4> c(functions).
{ok, functions}
5> functions:valid_time({{2011,09,06},{09,04,43}}).
The Date tuple ({2011,9,6}) says today is: 2011/9/6,
The time tuple ({9,4,43}) indicates: 9:4:43.
ok
```

```
6> functions:valid_time({{2011,09,06},{09,04}}).
Stop feeding me wrong data!
ok
```

Правда, есть одна проблема! Эта функция будет принимать в качестве входящих данных что угодно, даже текст или атомы. Достаточно лишь, чтобы кортежи имели вид `{{A,B,C}, {D,E,F}}`. Вот мы и пришли к одному из ограничений сопоставления с образцом: с его помощью можно указать либо очень точные значения, например, известное число или атом, либо абстрактные значения, такие как голова|хвост списка, кортеж из N элементов, или что угодно другое (`_` и свободные переменные), и т.д. Чтобы решить эту проблему мы используем охранные выражения, стражи (guards).

5.2 Стража! Стража!

Стражи – это дополнительные выражения, которые можно добавлять в заголовки функций, чтобы сделать сопоставление с образцом более выразительным. Как уже упоминалось выше, в сопоставлении с образцом есть ограничения, так как с его помощью нельзя сопоставлять, к примеру, диапазон значений, или определённые типы данных. Мы не можем выразить концепцию счёта: слишком ли низок этот двенадцатилетний баскетболист, чтобы играть с профессионалами? Слишком ли велика эта дистанция, чтобы пройти её на руках? Ты слишком стар, или слишком молод, чтобы водить машину? На такие вопросы не ответишь, применяя лишь простое сопоставление с образцом. Конечно, можно представить вопрос про вождение в таком виде:

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
```

Но это уж слишком громоздко. Если хотите, можете так делать, но будьте готовы работать над своим кодом в гордом одиночестве. Если всё же хотите со временем обзавестись друзьями, создайте новый модуль `guards`, в котором мы реализуем «правильное» решение для вопроса о вождении:

```
old_enough(X) when X >= 16 -> true;
```

```
old_enough(_) -> false.
```

Вот и всё! Как видите, такая запись намного короче и понятнее. Основное правило для охранных выражений: чтобы выражение сработало, оно должно возвращать `true`. Если страж возвращает `false`, или бросает исключение, то оно не срабатывает. Предположим, что мы хотим запретить садиться за руль людям старше 104 лет. Водить можно только в возрасте от 16 до 104 лет. Как же нам выполнить это условие? Давайте просто добавим второе охранное выражение:

```
right_age(X) when X >= 16, X =< 104 ->
    true;
right_age(_) ->
    false.
```

Запятая (`,`) по выполняемой роли похожа на оператор `andalso`, а точка с запятой (`;`) ведёт себя приблизительно как `orelse` (эти операторы описаны в разделе 3.4 Булева алгебра и операторы сравнения). Чтобы всё выражение успешно выполнилось, необходимо чтобы было удовлетворено условие в обоих охранных выражениях. Также можно поменять условия в функции на противоположные:

```
wrong_age(X) when X < 16; X > 104 ->
    true;
wrong_age(_) ->
    false.
```

И с этим выражением мы всё равно получим правильные результаты. Если хотите, можете протестировать (вы всегда должны всё проверять!). В охранных выражениях точка с запятой (`;`) ведёт себя как оператор `orelse`: если первый страж не исполняется, то исполнение переходит ко второму, потом к следующему, до тех пор пока хотя бы один страж возвратит истину или все стражи возвратят ложь.

Помимо функций сравнения и булевых операторов, можно использовать и другие функции, включая математические операции (`A*B/C >= 0`) и функции определения типа, такие как `is_integer/1`, `is_atom/1`, и т.д. (мы вернёмся к ним в следующей главе). Одним из недостатков охранных выражений является то, что они не принимают функции определённые пользователем из-за возможных побочных эффектов. Erlang не чистый функциональный язык программирования, (коим является, к примеру,

Haskell) потому что во многом полагается на побочные эффекты: можно выполнять операции ввода-вывода, пересылать между акторами сообщения, выбрасывать исключения когда угодно и где угодно. Не существует простого способа определить, что функция, которая используется в охранном выражении, печатает текст. А может быть не печатает. А может быть она перехватывает важные сообщения об ошибках при тестировании в нескольких функциональных выражениях. Поэтому Erlang просто вам не доверяет (и, скорее всего, правильно делает!)

После всего сказанного, у вас должно было появиться понимание базового синтаксиса охранных выражений, достаточное для того чтобы не теряться при встрече с ними.

Замечание: я сравнивал символы `,` и `;` в охранных выражениях с операторами `andalso` и `orelse`. По правде говоря, они не совсем эквивалентны. Первые будут захватывать возникающие исключения, а вторые не будут. Это означает, что если в первой части охранного выражения `X >= N; N >= 0` будет сгенерирована ошибка, то вторая часть всё же будет исполнена, и выражение может сработать. Если ошибка была выброшена в первой части выражения `X >= N orelse N >= 0`, то вторая часть будет пропущена, и всё охранное выражение не сработает.
Однако, (всегда есть какое-нибудь «однако») только операторы `andalso` и `orelse` могут помещаться в охранное выражение. Это означает, что `(A orelse B) andalso C` это валидное выражение, а `(A; B), C` – нет. Нужно учитывать их различное назначение и использовать в сочетании друг с другом.

5.3 Что ещё за «If»?!

`If`-ы ведут себя подобно охранным выражениям и имеют тот же синтаксис, но используются за пределами заголовка функции. Они даже называются *Охранными шаблонами*. `if`-ы в Erlang отличаются от `if`-ов в большинстве других языков. По сравнению с ними, это странные создания, которых принимали бы охотнее, называйся они иначе. Вступая в страну Erlang, оставьте всё что вы знаете об `if` на пороге. Займите своё место – мы отправляемся в путешествие.

Чтобы понять как похожи выражения `if` на охранные выражения, взгляните на следующие примеры:

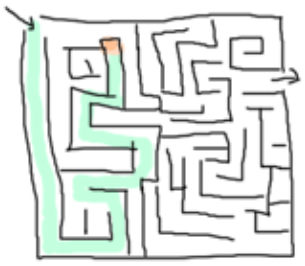
```
-module(what_the_if).
-export([heh_fine/0]).

heh_fine() ->
```

```
    if 1 =:= 1 ->
        works
    end,
    if 1 =:= 2; 1 =:= 1 ->
        works
    end,
    if 1 =:= 2, 1 =:= 1 ->
        fails
    end.
```

Сохраним этот код в файл what_the_if.erl, и попробуем его исполнить:

```
1> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever
   match
./what_the_if.erl:12: Warning: the guard for this
   clause evaluates to 'false'
{ok,what_the_if}
2> what_the_if:heh_fine().
** exception error: no true branch found when
   evaluating an if expression
     in function  what_the_if:heh_fine/0
```

Ой! Компилятор нас предупреждает, что условие if в строке 12 (1 =:= 2, 1 =:= 1) никогда не будет использовано, так как его единственное охранное выражение всегда возвращает false. Помните, в Erlang всё должно что-то возвращать, и if- выражения не являются исключением из этого правила. Таким образом, когда Erlang не может найти случай, в котором



страж успешно выполнится, будет сгенерирована ошибка: возвращать в этом случае нечего. Поэтому мы должны добавить ветвь, которая будет успешно исполняться не смотря ни на что. В большинстве языков это бы называлось 'else'. В Erlang мы используем слово 'true' (это объясняет, почему VM сгенерировала сообщение «no true branch found»)

```
oh_god(N) ->
```

```
    if N =:= 2 -> might_succeed;
        true -> always_does  %% this is Erlang's if's 'else!'
    end.
```

Если мы теперь протестируем эту новую функцию (старая будет продолжать плеваться предупреждениями. Можно их игнорировать или оставить как напоминание о том, как делать нельзя):

```
3> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause
    evaluates to 'false'
{ok,what_the_if}
4> what_the_if:oh_god(2).
might_succeed
5> what_the_if:oh_god(3).
always_does
```

Вот ещё одна функция, которая показывает как использовать несколько стражей в `if`-выражении. Эта же функция также демонстрирует, что любое выражение должно что-нибудь возвращать: к переменной *Talk* привязывается результат выражения `if` и затем конкатенируется в строку, которая входит в состав кортежа. Читая код, легко увидеть как отсутствие ветки `true` могло бы всё испортить, если учитывать, что в Erlang не существует такого понятия как null-значение (как nil в lisp, NULL в C, None в Python и т.д.):

```
%% note, this one would be better as a pattern match in function
    heads!
%% I'm doing it this way for the sake of the example.
help_me(Animal) ->
    Talk = if Animal == cat  -> "meow";
              Animal == beef -> "mooo";
              Animal == dog  -> "bark";
              Animal == tree -> "bark";
              true -> "fgdadfgna"
           end,
    {Animal, "says␣" ++ Talk ++ "!"}.
```

А теперь попробуем исполнить:

```
6> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause
    evaluates to 'false'
{ok,what_the_if}
7> what_the_if:help_me(dog).
{dog,"says␣bark!"}
8> what_the_if:help_me("it␣hurts!").
```

```
{"it hurts!","says fgdadfgna!"}
```

Как и множество программистов на Erlang, вы, должно быть, недоумеваете – почему 'true' взяло верх над 'else' в качестве атома для управления потоком исполнения? В конце концов, 'else' более привычен. Richard O'Keefe дал в почтовой рассылке Erlang следующий ответ на этот вопрос. Я привожу его здесь без купюр, потому что сам не смог бы сформулировать лучше:

> Может быть, 'else' и привычнее, но это не означает, что 'else' это хорошо. Я понимаю, что при помощи выражения '; true ->' очень легко получить в Erlang 'else', но результаты двух десятков лет наблюдений за психологией программирования показывают, что ни к чему хорошему это не приведёт. Я начал заменять:

```
if X > Y -> a()
     ; true  -> b()
if X > Y -> a()
     ; X < Y -> b()
     ; true  -> c()
end
```

> на

```
if X > Y  -> a()
     ; X =< Y -> b()
end

if X > Y -> a()
     ; X < Y -> b()
     ; X == Y -> C()
end
```

> это немного раздражает, когда я **пишу** код, но чрезвычайно помогает, когда я его **читаю**.

Лучше всего «избегать» и 'else' и 'true': `if`-ы обычно легче читать, когда явно покрываются все логические исходы, без использования выражения, которое «ловит всё подряд».

Как упоминалось ранее, в охранных выражениях можно использовать ограниченный набор функций (мы ещё с ними встретимся в 6 Типы (или их отсутствие). Настало время явить подлинную мощь условных операторов Erlang. Я представляю вам выражение `case`!

Замечание: весь ужас, отражённый в названиях функций файла what_the_if.erl, относится к языковой конструкции if, если рассматривать её с точки зрения if любого другого языка. В контексте Erlang эта конструкция оказывается абсолютно логичной, просто её имя сбивает с толку.

5.4 В случае... если

Если считать, что выражения if похожи на стражей, то case ... of похоже на заголовок функции в целом. Для каждого аргумента можно применять сложные выражения сопоставления с образцом и, вдобавок, использовать стражи!

Так как вы уже неплохо знакомы с синтаксисом, нам не понадобится много примеров. На этот раз мы напишем функцию дополнения (append) для множеств (sets) (набор уникальных значений), который мы представим как неупорядоченный список. В смысле эффективности это, пожалуй, наихудшая возможная реализация наборов, но сейчас мы заботимся не об эффективности, а о синтаксисе:

```
insert(X,[]) ->
    [X];
insert(X,Set) ->
    case lists:member(X,Set) of
        true  -> Set;
        false -> [X|Set]
    end.
```

Если мы передаём в функцию пустое множество (список) и терм X, она возвращает нам список, который содержит лишь X. Иначе, функция lists:member/2 проверяет, не является ли элемент частью списка, и возвращает true, если является и false, если нет. В случае, когда в нашем множестве уже присутствует элемент X, нам не нужно изменять список. Если элемента в списке нет, то мы добавляем X первым элементом списка.

В нашем примере, сопоставление с образцом было очень простым. Оно может усложняться (можете сравнить ваш код с моим):

```
beach(Temperature) ->
    case Temperature of
        {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
        {kelvin, N} when N >= 293, N =< 318 ->
            'scientifically_favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
```

```
            'favorable in the US';
        _ ->
            'avoid beach'
    end.
```

Здесь представлен ответ на вопрос «не сходить ли на пляж?» для трёх температурных систем: Цельсия, Кельвина и Фаренгейта. Сопоставление с образцом комбинируется со стражами и, в конце концов, возвращает ответ, который удовлетворяет всем вариантам использования. Как было указано ранее, выражения `case...of` это практически то же самое, что и группа заголовков функций со стражами. Мы даже могли бы записать наш код в следующем виде:

```
beachf({celsius, N}) when N >= 20, N =< 45 ->
    'favorable';
...
beachf(_) ->
    'avoid beach'.
```

Это вызывает вопрос: когда нужно использовать `if`, а когда `case ... of` или функции для записи условных выражений?

5.5 Что же использовать?

На вопрос «Что же использовать?», – ответить довольно сложно. Между вызовами функций и конструкцией `case ... of` разница очень невелика. На самом деле, на низком уровне они реализованы одинаково, поэтому с точки зрения производительности они ничем не отличаются.

Различия появляются, когда обрабатывается более чем один аргумент: `function(A,B) -> ... end.` может использовать стражи, чтобы сопоставлять с образцом A и B, тогда как case выражение было бы необходимо сформулировать приблизительно следующим образом:

```
case {A,B} of
    Pattern Guards -> ...
end.
```

Такая форма встречается редко, и наверняка застанет читателя врасплох. В похожих ситуациях правильнее было бы использовать вызов функции. С другой стороны, функция `insert/2`, которую мы записали ранее, определённо выглядит яснее в своём текущем виде, без прямого вызова функции, которая проходит по веткам `true` или `false`.

Следующий вопрос: зачем тогда вообще использовать `if`, если `case` и функции достаточно гибки, чтобы реализовать `if` посредством стражей? За `if` стоит простая логика: эта конструкция была добавлена в язык, чтобы иметь возможность использовать стражи, не описывая сопоставления с образцом, когда в них нет необходимости.

Конечно же, всё это относится больше к индивидуальным предпочтениям. Определённого ответа на этот вопрос не существует. Эта тема до сих пор время от времени обсуждается сообществом Erlang. Если ваш код прост для понимания, никто вас бить не будет, какой бы путь вы не избрали. Как однажды сказал Ward Cunningham: «Код можно считать ясным, когда вы смотрите на подпрограмму, и она выглядит приблизительно так, как вы и ожидали.»

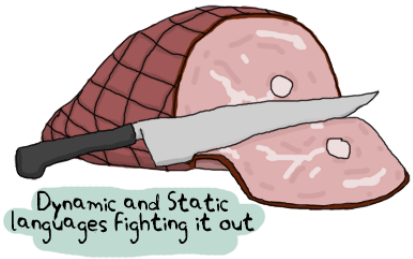
Глава 6

Типы (или их отсутствие)

6.1 Сильнейшая типизация

Как вы, вероятно, заметили во время ввода примеров из 3, а потом модулей и функций из 4 и 5, нам не нужно было вводить тип переменной или тип функции. При сопоставлении с образцом нашему коду не нужно было знать, с чем производится сопоставление. Кортеж `{X,Y}` с одинаковым успехом можно сопоставить как с `{atom, 123}`, так и с `{“A string”, «“binary stuff!”»}`, `{2.0, [“strings”, “and”, atoms]}` или вообще с чем угодно.

При неудачном сопоставлении просто генерировалась ошибка, но она генерировалась только во время исполнения кода. Причина этого – *динамическая типизация* в языке Erlang. Каждая ошибка ловится во время исполнения. Если исполнение кода потенциально может закончиться аварией, то компилятор об этом не всегда будет истошно вопить, как в примере `“llama + 5”` из главы 3.



Одной из классических точек трения между сторонниками статической и динамической типизации является безопасность програмного обеспечения. Очень часто насаждается мысль, что хорошая статическая

система типизации, за соблюдением которой с усердием следят компиляторы, будет ловить большинство ошибок ещё до того как код начнёт исполняться. Поэтому языки со статической типизацией считаются более надёжными, чем их динамические собратья. Хотя для многих динамических языков это действительно так, но к Erlang это относится не в полной мере, и его послужной список это подтверждает. Лучшим примером служит степень готовности к обслуживанию (availability) *девять девяток* (99.9999999%), которая обеспечивается коммутаторами Ericsson AXD 301 ATM. Количество строк Erlang-кода для этих устройств – свыше 1 миллиона. Обратите внимание – этот факт не свидетельствует о том, что ни один из компонентов системы, написанной на Erlang, не давал сбой. Это означает, что коммутатор как единая система был готов к работе 99.9999999% времени, включая запланированные перерывы. Отчасти это заслуга проектировщиков, которые в разработке Erlang руководствовались идеей, что сбой в одной из компонент не должен влиять на всю систему. При этом учитываются ошибки, которые допускает программист, сбои оборудования или сети. Язык содержит средства, позволяющие распределять программы на различные узлы, обрабатывать непредвиденные ошибки и *никогда* не останавливаться.

Иначе говоря, в то время как большинство языков и систем типизации стараются избавить программы от ошибок, Erlang использует стратегию, согласно которой считается, что ошибки будут возникать в любом случае, и старается эти случаи предотвратить. Система динамической типизации в Erlang – не преграда для надёжности и безопасности программ. Всё сказанное мной напоминает речи проповедника, но в последующих главах вы увидите как это происходит.

Замечание: динамическая типизация в ретроспективе была избрана по простой причине. Люди, которые занимались реализацией Erlang, в большинстве своём имели опыт программирования на языках с динамической типизацией, и поэтому динамическая типизация в Erlang была для них самым естественным выбором.

Кроме того, Erlang – язык с сильной типизацией. Языки со слабой типизацией производят неявные преобразования типов между термами. Если бы Erlang был языком со слабой типизацией, то мы, вероятно, смогли бы исполнить операцию $6 = 5 + "1"$. В действительности будет выброшено исключение, которое сообщает о неверных аргументах:

```
1> 6 + "1".
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as 6 + "1"
```

Конечно же, есть случаи, когда вам хотелось бы преобразовать один вид данных в другой. Например, привести обычный строковый тип к битовым строкам на время хранения, или целое значение привести к числу с плавающей запятой. Для таких ситуаций Стандартная библиотека Erlang предлагает множество функций.

6.2 Преобразование типов

Как и многие языки, Erlang меняет тип терма посредством приведения его к другому типу. Для этого применяются встроенные функции, поскольку многие преобразования невозможно реализовать на чистом Erlang. Каждая такая функция имеет форму <тип>_to_<тип> и находится в модуле `erlang`. Вот некоторые из этих функций:

```
1> erlang:list_to_integer("54").
54
2> erlang:integer_to_list(54).
"54"
3> erlang:list_to_integer("54.32").
** exception error: bad argument
     in function  list_to_integer/1
        called as list_to_integer("54.32")
4> erlang:list_to_float("54.32").
54.32
5> erlang:atom_to_list(true).
"true"
6> erlang:list_to_bitstring("hi there").
<<"hi there">>
7> erlang:bitstring_to_list(<<"hi there">>).
"hi there"
```

И так далее. Здесь мы сталкиваемся с неприятной особенностью языка. Для именования функций используется схема <тип>_to_<тип>, поэтому при добавлении нового типа в язык приходится добавлять множество встроенных функций! Вот их полный список:

`atom_to_binary/2, atom_to_list/1, binary_to_atom/2`
`binary_to_existing_atom/2, binary_to_list/1,`
`bitstring_to_list/1, binary_to_term/1, float_to_list/1,`
`fun_to_list/1, integer_to_list/1, integer_to_list/2,`
`iolist_to_binary/1, iolist_to_atom/1, list_to_atom/1,`
`list_to_binary/1, list_to_bitstring/1,`
`list_to_existing_atom/1, list_to_float/1, list_to_integer/2,`

list_to_pid/1, list_to_tuple/1, pid_to_list/1, port_to_list/1, ref_to_list/1, term_to_binary/1, term_to_binary/2 и tuple_to_list/1.

Многовато функций. Большинство из них, если не все, мы увидим в этой книге. Впрочем, все они нам вряд ли понадобятся.

6.3 Охранять тип данных

Базовые типы Erlang просто заметить: у кортежей есть фигурные скобки, у списков – квадратные, строки заключены в двойные кавычки и т.д. Поэтому определённый тип данных можно использовать в сопоставлении с образцом: функция `head/1`, которая принимает в качестве аргумента список, может принимать списки потому, что для других типов не сработало бы сопоставление (`[H|_]`).

Тем не менее, у нас уже возникали проблемы с числовыми значениями, для которых мы не могли задавать диапазоны. Для решения этой проблемы мы использовали стражи в функциях, которые были связаны с температурой, водительским возрастом и т.д. Перед нами встаёт ещё одно препятствие. Как нам записать страж, который применил бы сопоставление с образцом к данным лишь одного определённого типа, такого как числа, атомы или битовые строки?

Для решения этой задачи предназначены функции, которые возвращают истину, если переданный им аргумент имеет верный тип, и ложь в противоположном случае. Они составляют группу функций, которые допускаются в охранных выражениях и называются «встроенными функциями проверки типов»:

```
is_atom/1          is_binary/1
is_bitstring/1     is_boolean/1       is_builtin/3
is_float/1         is_function/1      is_function/2
is_integer/1       is_list/1          is_number/1
is_pid/1           is_port/1          is_record/2
is_record/3        is_reference/1     is_tuple/1
```

Их можно использовать, как и любое другое охранное выражение, в любом месте, где допускается охранное выражение. Вы, вероятно, зададёте себе вопрос: почему функция просто не возвращает тип терма, который ей передали (что-то вроде `type_of(X) -> Type`). Ответ прост. Erlang концентрируется на программировании для корректных ситуаций: вы составляете программу только для событий, которые точно должно произойти, для тех событий, которые вы ожидаете. Всё прочее должно как можно раньше вызвать ошибку. Может быть это покажется абсурдом, но объяснения, которые вы получите в главе 9, я надеюсь, прояснят ситуацию. А пока что просто поверьте мне на слово.

Замечание: встроенные функции проверки типов более чем на половину состоят из инструкций, которые можно использовать в охранных выражениях. Остальные функции также являются встроенными, но не относятся к функциям проверки типов. Среди них:
`abs(Number)`, `bit_size(Bitstring)`, `byte_size(Bitstring)`,
`element(N, Tuple)`, `float(Term)`, `hd(List)`, `length(List)`,
`node()`, `node(Pid|Ref|Port)`, `round(Number)`, `self()`,
`size(Tuple|Bitstring)`, `tl(List)`, `trunc(Number)`, `tuple_size(Tuple)`.
Функции `node/1` и `self/0` принадлежат к распределённым средствам Erlang и разделу процессов/акторов. Вскоре мы будем их использовать, но до той поры мы ещё должны изучить много других вещей.

Может показаться, что структуры данных в Erlang относительно ограничены, но списков и кортежей обычно достаточно для создания других более сложных структур. Например, узел двоичного дерева можно представить как `{node, Value, Left, Right}`, где *Left* и *Right* это или узлы, одинаковые по структуре, или пустые кортежи. Я мог бы представить информацию о себе в таком виде:

```
{person, {name, <<"Fred T-H">>},
    {qualities, ["handsome", "smart", "honest", "objective"]},
    {faults, ["liar"]},
    {skills, ["programming", "bass guitar", "underwater
       breakdancing"]}}.
```

Этот пример показывает, как можно получить сложные структуры данных. Для этого необходимо вложить списки в кортежи, заполнить их данными, и создать функции, которые будут работать с полученной структурой.

Дополнение: в релизе R13B04 появилась встроенная функция `binary_to_term/2`, которая позволяет десериализовать данные так же, как это делает `binary_to_term/1`, с тем лишь отличием, что вторым аргументом можно передать список опций. Если передать опцию `[safe]`, то двоичные данные, содержащие неизвестные атомы или 8.1 анонимные функции, которые могут привести к исчерпанию памяти, декодированы не будут.

6.4 Для «подсевших на систему» типизации

Этот раздел предназначен для программистов, которые по той или иной причине не представляют своё существование без статических систем типизации. Он содержит слегка усложнённую теорию, которая не всем будет ясна. Я кратко опишу инструменты, которые используются для статического анализа типов в Erlang, определения специализированных типов, и то, как всё это позволяет получить более безопасный код. Эти средства будут описаны в книге намного позже, потому как они совсем не обязательны для разработки надёжных программ на Erlang. Так как мы откладываем их рассмотрение на потом, я дам лишь основные сведения об их установке, запуске и т.д. Повторюсь: этот раздел предназначен для тех, кто действительно не может жить без развитых систем типизации.

На протяжении нескольких лет предпринимались попытки построения системы типов поверх Erlang. Одна из таких попыток случилась в 1997 году под руководством Simon Marlow, одного из ведущих разработчиков Glasgow Haskell Compiler и Philip Wadler, который работал над проектированием Haskell и внёс свою лепту в теорию, которая лежит в основе монад (здесь вы можете прочитать работу, посвящённую упомянутой системе типов). Joe Armstrong позже так прокомментировал этот документ:

Однажды мне позвонил Фил и заявил, что а) Erlang необходима система типов, б) он написал непольшой прототип такой системы и в) у него есть возможность взять творческий отпуск, в течение которого он собирался написать систему типов для Erlang, и спрашивал: «будет ли нам это интересно?». Я ответил: «Да.»
Phil Wadler и Simon Marlow работали над системой типов больше года, и результаты были опубликованы в [20]. Результаты оказались немного разочаровывающими. Начнём с того, что на соответствие типов можно было проверять не весь язык, а лишь его подмножество. Большим упущением было отсутствие типизированных процессов и отсутствие проверки типов для сообщений, пересылаемых между процессами.

Процессы и сообщения относятся к основным средствам Erlang. Может быть именно поэтому система так никогда и не была внедрена. Были и другие неудачные попытки типизировать Erlang. В результате усилий проекта HiPE (попытка увеличить производительность Erlang) появился Dialyzer, инструмент статического анализа, который используется до сих пор. Он имеет свой собственный механизм вывода типов (type inference).

Система типов, которая в нём используется, основана на успешных типизациях (success typings) – концепции, которая отличается от системы типов Hindley-Milner или мягкой типизации (soft-typing). Концепция успешной типизации проста: метод определение типов не пытается найти точный тип каждого выражения, но он гарантирует, что выведенные им типы точны, и что ошибки типов, которые он находит, действительно являются ошибками.

В качестве примера лучше всего привести реализацию функции `and`, которая обычно принимает два булевых значения и возвращает 'true', если оба параметра истинны, иначе возвращается 'false'. В системе типов Haskell это можно записать как `and :: bool -> bool -> bool`. Если бы нужно было реализовать функцию `and` на Erlang, то это можно было бы сделать следующим образом:

```
and(false, _) -> false;
and(_, false) -> false;
and(true,true) -> true.
```

После применения метода успешной типизации, был бы выведен тип `and(_,_) -> bool()`, где _ означает «любое значение». Причина появления такого типа проста: при вызыве функции с аргументами `false` и `42`, будет возвращён результат 'false'. Использование шаблона подстановки `_` в сопоставлении с образцом привело к тому, что для работы функции достаточно передать хотя бы один аргумент, который равен

'false'. У системы типов ML случился бы припадок (а у её пользователей сердечный приступ), если бы вы вызвали функцию таким образом. Но не у Erlang. В этих строках появится больше смысла, если вы решите прочитать документ про реализацию успешной типизации, в котором объясняется логика такого поведения. Я призываю каждого наркозависимого от типов прочитать эту статью. Она представляет собой интересное и полезное описание реализации метода.

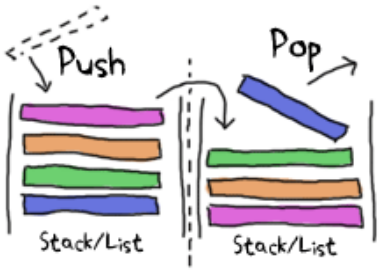
Подробности процесса определения типов и аннотирования функций описаны в Предложении об Улучшении Erlang №8 (EEP8). Если вас заинтересовало использование успешной типизации в Erlang, взгляните на приложение TypEr и Dialyzer. Оба входят в стандартный дистрибутив. Чтобы ими воспользоваться, введите `$ typer –help` и `$ dialyzer –help` (для Windows команды `typer.exe –help` и `dialyzer –help`, если они доступны из текущей директории).

TypEr используется для аннотации типов функций. Запуск TypEr для этой маленькой реализации FIFO очереди генерирует следующее описание типов:

```
%% File: fifo.erl
%% --------------
-spec new() -> {'fifo',[],[]}.
-spec push({'fifo',_,_},_) -> {'fifo',
    nonempty_maybe_improper_list(),_}.
-spec pop({'fifo',_,maybe_improper_list()}) -> {_,{'fifo',_,_}}.
-spec empty({'fifo',_,_}) -> bool().
```

Что, в целом, верно. Improper списки лучше не использовать, так как с ними не работает функция `lists:reverse/1`, а кто-нибудь всё-таки может попытаться протолкнуть такой список, минуя интерфейс модуля. В этом случае функции `push/2` и `pop/2` всё-таки отработают несколько вызовов, а затем будет выброшено исключение. Поэтому нам необходимо добавить стражи, либо вручную оптимизировать определения типов. Предположим, что мы добавили в модуль сигнатуру
`-spec push({fifo,list(),list()},_) -> {fifo,nonempty_list(),list()}.`
и функцию, которая передаёт improper список для `push/2`. При сканировании Dialyzer-ом (который проверяет типы на соответствие), будет выведено сообщение об ошибке «The call fifo:push(fifo,[1|2],[],3) breaks the contract '<Type definition here>>́>>.

Dialyzer будет сообщать об ошибках лишь тогда, когда код будет нарушать работу другого кода, и эти сообщения, скорее всего, будут верными (он также будет сообщать и о других проблемах, например о ветках кода, до которых никогда не дойдёт исполнение или об общем



рассогласовании). При помощи Dialyzer-а можно также анализировать полиморфные типы данных. Функцию `hd()` можно описать выражением `-spec([A]) -> A.` и корректно проанализировать, впрочем программисты на Erlang редко используют этот синтаксис описания типов.

Не забывайтесь:
Dialyzer и TypEr не обрабатывают классы типов с конструкторами, типы первого порядка и рекурсивные типы. Типы в Erlang – это лишь аннотация, которая никак не влияет на компиляцию и не ограничивает её, кроме ситуаций, когда вы сами накладываете эти ограничения. Программа проверки типов никогда не сообщит вам, что в приложении, которое работает прямо сейчас (или исполняется уже на протяжении двух лет), есть ошибка типов, которая никак себя не проявляет во время исполнения (корректное исполнение не говорит о том, что в коде нет ошибок...)
Неплохо было бы иметь возможность создания рекурсивных типов, но они вряд ли когда-нибудь появятся для TypEr и Dialyzer в текущем состоянии (объяснение можно найти в статье, указанной выше). На текущий момент можно лишь вручную симулировать рекурсивные типы добавлением нескольких уровней вложенности.
Конечно же, это нельзя назвать всеобъемлющей системой типов, сравнимой со строгостью и мощью систем типизации в Scala, Haskell или OCaml. Предупреждения и сообщения об ошибках обычно слегка запутаны и не всегда ясны пользователю. Но если вы просто не можете существовать в мире динамического языка или жаждете дополнительной надёжности – это решение предлагает очень неплохой компромисс. Относитесь к нему как к инструменту в вашем арсенале, не более.

Дополнение:
Начиная с версии R13B04, в Dialyzer была добавлена экспериментальная возможность работы с рекурсивными типами. Этот факт делает предыдущий раздел отчасти неверным. Стыд мне и срам.
Также заметьте, что документация, описывающая типы, стала официальной (хотя она и может в будущем измениться) и более полной, чем в EEP8.

Глава 7

Рекурсия

7.1 Привет, рекурсия!

Некоторые читатели, знакомые с императивными и объектно-ориентированными языками программирования, должно быть, недоумевают, почему до сих пор не были показаны циклы. Ответить на это можно вопросом: «Что такое цикл?» По правде говоря, функциональные языки программирования обычно не предлагают средства построения циклов, такие как `for` и `while`. Вместо них программисты-функциональщики полагаются на незамысловатую концепцию, именуемую *рекурсией*.

Вы, должно быть, помните как во вводной главе объяснялись неизменяемые переменные. Если не помните, то уделите им 3.2 ещё немного внимания! Рекурсию тоже можно объяснить при помощи математических концепций и функций. Хорошим примером функции, которую можно выразить в рекурсивном виде, может послужить примитивная функция вычисления факториала. Факториал числа n это произведение последовательности $1 \times 2 \times 3 \times \ldots \times n$ или $n \times n-1 \times n-2 \times \ldots \times 1$. Например, факториал числа 3 равен $3! = 3 \times 2 \times 1 = 6$. Факториал 4 равен $4! = 4 \times 3 \times 2 \times 1 = 24$. Эту функцию можно выразить при помощи математической записи в следующем виде:

$$n! = \begin{cases} 1 & \text{if } x = 0 \\ n((n-1)!) & \text{if } x < 0 \end{cases}$$

Это означает, что если n равно 0, то в качестве результата возвра-

щается 1. Для любого значения больше 0 возвращается n умноженное на факториал `n - 1`, значение которого тоже раскрывается, до тех пор, пока не достигнет 1.

$$4! = 4 \times 3!$$
$$4! = 4 \times 3 \times 2!$$
$$4! = 4 \times 3 \times 2 \times 1!$$
$$4! = 4 \times 3 \times 2 \times 1 \times 1$$

Как же записать такую математическую функцию на Erlang? Очень просто. Взгляните на части записи: `n!`, 1 и `n((n-1)!)`, а затем на `if`-ы. Мы можем различить имя функции (`n!`), стражей (им соответствуют `if`-ы) и тело функции (1 и `n((n-1)!)`). Переименуем `n!` в `fac(N)`, чтобы немного ограничить наш синтаксис, и получим следующее:

```
-module(recursive).
-export([fac/1]).

fac(N) when N == 0 -> 1;
fac(N) when N > 0  -> N*fac(N-1).
```

Вот и готова наша функция для вычисления факториала! Она очень похожа на её математическое определение. Применяя сопоставление с образцом, объявление функции можно немного сократить:

```
fac(0) -> 1;
fac(N) when N > 0 -> N*fac(N-1).
```

Для математического определения, которое рекурсивно по своей природе, трансляция на Erlang происходит просто и быстро. Мы записали цикл! Определение рекурсивной функции можно сократить до «функция, которая вызывает саму себя». Также нам необходимо определить условие остановки вычислений (это называется частным случаем), так как без такого условия мы будем находиться в цикле бесконечно. В нашем примере условие остановки – это состояние, когда n равно 0. В момент, когда это условие истинно, мы перестаём вызывать функцию, и её исполнение прекращается.

7.2 Длина

Попробуем перейти к более практичным вещам. Напишем функцию, которая считает количество элементов, содержащихся в списке.

Сразу же ясно, что нам понадобится:

- частный случай;
- функция, которая вызывает сама себя;
- список, к которому мы применим нашу функцию.

Мне кажется, что для большинства рекурсивных функций проще всего сначала записать частный случай. Что представляет собой самый простой входящий параметр, для которого мы можем найти длину? Это, конечно же, пустой список, длина которого равна 0. Поэтому запомним, что `[] = 0`. Следующий по простоте список имеет длину 1: `[_] = 1`. Похоже, у нас есть всё необходимое, чтобы записать определение нашей функции:

```
len([]) -> 0;
len([_]) -> 1.
```

Прекрасно! Мы можем подсчитать длину списка, если он содержит 0 либо 1 элемент! Очень полезная возможность. Правда, пользы в ней не так много, потому что ей не хватает рекурсивности. Это приводит нас к самому сложному моменту: нам необходимо расширить функцию таким образом, чтобы она вызывала сама себя для списков с длиной больше 0 и 1. Ранее мы 3.6 упоминали, что списки определяются рекурсивно как `[1 | [2| ...[n| []]]]`. Это означает, что мы можем использовать образец `[H|T]` для сопоставления со списками одного и более элементов, так как список с одним элементом можно определить как `[X|[]]`, а список с двумя элементами как `[X|[Y|[]]]`. Обратите внимание, что второй элемент сам является списком. Поэтому нам нужно посчитать лишь первый элемент, а потом функция может вызвать себя для второго элемента. Если учесть, что каждый элемент в списке прибавляет к длине 1, то функцию можно переписать в следующем виде:

```
len([]) -> 0;
len([_|T]) -> 1 + len(T).
```

Вот у нас и появилась рекурсивная функция, которая определяет длину списка. Давайте посмотрим как она будет вести себя при испол-

нении. Попробуем применить её, например, к списку `[1,2,3,4]`:

$$\begin{aligned} len([1,2,3,4]) &= len([1|[2,3,4]) \\ &= 1 + len([2|[3,4,]]) \\ &= 1 + 1 + len([3|[4]]) \\ &= 1 + 1 + 1 + len([4|[]]) \\ &= 1 + 1 + 1 + 1 + len([]) \\ &= 1 + 1 + 1 + 1 + 0 \\ &= 1 + 1 + 1 + 1 \\ &= 1 + 1 + 2 \\ &= 1 + 3 \\ &= 4 \end{aligned}$$

Мы получили верный результат. Поздравляю вас с первой полезной рекурсивной функцией на Erlang!

7.3 Длина хвостовой рекурсии

Возможно, вы заметили, что для списка из четырёх термов, мы разложили вызов нашей функций на цепь из пяти операций суммирования. Хотя этот принцип хорошо работает для коротких списков, он может создать проблемы для случая, когда ваш список содержит несколько миллионов значений. Для такого простого вычисления совсем не обязательно хранить в памяти миллионы чисел. Это слишком расточительно, к тому же, есть способ получше. Познакомьтесь с *хвостовой рекурсией*.

Хвостовая рекурсия – это способ трансформации вышеописанного линейного процесса (который растёт прямо пропорционально количеству элементов) в итеративный (в котором никакого роста нет вообще). Чтобы вызов функции стал хвостовым, он должен быть «одиночным». Здесь необходимо пояснить: наши предыдущие вызовы росли из-за того, что результат первого вызова зависел от вычисления второго. Чтобы найти ответ для `1 + len(Rest)`, необходимо вычислить результат `len(Rest)`. А для вычисления `len(Rest)`, в свою очередь, понадобится найти результат ещё одного вызова функции. Операции суммирования будут накапливаться до тех пор, пока не будет найден результат последней, и

только после этого можно будет вычислить конечный результат. Хвостовая рекурсия позволяет избавиться от этого накопления посредством вычисления операций по мере их возникновения.

Чтобы этого добиться, нам необходимо завести временную переменную и передавать её при каждом вызове функции в качестве дополнительного параметра. Я проиллюстрирую эту концепцию при помощи функции вычисления факториала. На этот раз определим её с использованием хвостовой рекурсии. Вышеупомянутую временную переменную иногда называют *аккумулятором*. Для ограничения роста вызовов нашей функции, мы сохраняем в аккумуляторе результаты вычислений, по мере того как они происходят:

```
tail_fac(N) -> tail_fac(N,1).

tail_fac(0,Acc) -> Acc;
tail_fac(N,Acc) when N > 0 -> tail_fac(N-1,N*Acc).
```

Я определил две функции `tail_fac/1` и `tail_fac/2`. Сделал я это по причине того, что Erlang не позволяет указывать для параметров значения по умолчанию (функции с разной арностью, которые имеют одинаковые имена – это разные функции), поэтому мы создаём аналогичный эффект вручную. В данном случае `tail_fac/1` реализует абстракцию поверх функции `tail_fac/2`, которая использует хвостовую рекурсию. Никого не интересуют детали реализации скрытого аккумулятора, который использует `tail_fac/2`, поэтому мы будем экспортировать из модуля лишь `tail_fac/1`. Исполнение функции можно записать в следующем виде:

$$
\begin{aligned}
tail\_fac(4) &= tail\_fac(4,1) \\
tail\_fac(4,1) &= tail\_fac(4-1,4*1) \\
tail\_fac(3,4) &= tail\_fac(3-1,3*4) \\
tail\_fac(2,12) &= tail\_fac(2-1,2*12) \\
tail\_fac(1,24) &= tail\_fac(1-1,1*24) \\
tail\_fac(0,24) &= 24
\end{aligned}
$$

Видите разницу? Теперь нам не нужно держать в памяти одновременно более двух термов. Количество места, выделяемого под хранение данных, не меняется. Для вычисления факториала числа 4 понадобится столько же места, сколько для вычисления факториала 1000000 (если не учитывать, что 4! занимает в числовом представлении намного меньше места, чем 1000000!).

Теперь, когда вы видели функцию вычисления факториала с использованием хвостовой рекурсии, вам, наверняка, ясно как этот шаблон можно применить к нашей функции `len/1`. Необходимо лишь сделать так, чтобы к результату рекурсивного вызова больше не применялись никакие операции. Для тех, кто любит визуальные примеры, представьте, что вы помещаете операцию `+1` внутрь вызова функции путём добавления к параметру:

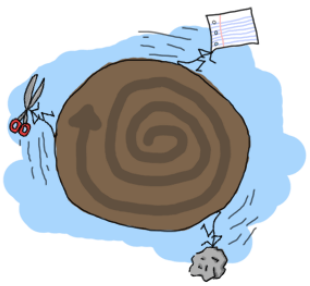
```
len([]) -> 0;
len([_|T]) -> 1 + len(T).
```

принимает вид:

```
tail_len(L) -> tail_len(L,0).

tail_len([], Acc) -> Acc;
tail_len([_|T], Acc) -> tail_len(T,Acc+1).
```

Теперь ваша функция вычисления длины списка использует хвостовую рекурсию.

7.4 Снова рекурсивные функции



Чтобы немного привыкнуть к рекурсивным функциям, мы напишем ещё несколько. В конце концов, рекурсия – это единственный способ организации итераций, который существует в Erlang (кроме списковых выражений), поэтому эту концепцию очень важно понимать. Это знание будет также полезно для любого другого функционального языка программирования, с которым вы можете столкнуться позже, поэтому запоминайте!

Первая функция, которую мы напишем – `duplicate/2`. Она принимает первым параметром целое число, а вторым – произвольный терм. Затем она создаёт список, который содержит столько копий терма, сколько указано в первом параметре. Как и ранее, для начала неплохо подумать о частном случае. Самое простое, что может сделать функция `duplicate/2`, это повторить что-либо 0 раз. Для этого она должна просто возвратить пустой список, не учитывая передаваемый терм. В любом другом случае мы должны пытаться добраться до частного, путём рекурсивного вызова функции. Вдобавок, мы запретим целому параметру

принимать отрицательные значения, потому что нельзя повторить что-либо `-n` раз:

```
duplicate(0,_) ->
    [];
duplicate(N,Term) when N > 0 ->
    [Term|duplicate(N-1,Term)].
```

Как только мы определили основную рекурсивную функцию, её становится проще записать в виде, который использует хвостовую рекурсию. Делается это путём перемещения операции создания списка во временную переменную:

```
tail_duplicate(N,Term) ->
    tail_duplicate(N,Term,[]).

tail_duplicate(0,_,List) ->
    List;
tail_duplicate(N,Term,List) when N > 0 ->
    tail_duplicate(N-1, Term, [Term|List]).
```

Получилось! Теперь я хотел бы немного изменить предмет обсуждения, чтобы обозначить связь между хвостовой рекурсией и циклом while. Наша функция `tail_duplicate/2` содержит все части, которые присущи циклу while. Если бы мы представили себе цикл while в выдуманном языке с синтаксисом похожим на Erlang, то наша функция могла бы выглядеть следующим образом:

```
function(N, Term) ->
    while N > 0 ->
        List = [Term|List],
        N = N-1
    end,
    List.
```

Обратите внимание, что все элементы цикла присутствуют как в воображаемом языке, так и в Erlang. Отличается лишь их расположение. Это показывает, что правильно написанная функция, использующая хвостовую рекурсию, подобна итеративному процессу, такому как цикл while.

Есть также ещё одно интересное свойство, которое мы «откроем» путём сравнении рекурсивной функции и функции с хвостовой рекурсией. Мы напишем функцию `reverse/1`, которая будет разворачивать список термов задом-наперёд. Базовым случаем в этой функции является пустой список, в котором разворачивать нечего. Для пустого списка мы можем просто вернуть пустой список. В любой другой ситуации функция должна сходиться к частному случаю, вызывая саму себя, как это

было в `duplicate/2`. Наша функция будет перебирать элементы списка при помощи сопоставления с образцом `[H|T]`, а потом добавлять H в конец списка:

```
reverse([]) -> [];
reverse([H|T]) -> reverse(T)++[H].
```

Для длинных списков это может оказаться настоящим кошмаром: будут накапливаться вызовы функции добавления, и ситуация усугубится ещё и тем, что для каждой операции добавления в конец списка нам придётся проходить список полностью от начала до конца! Визуально это можно представить так:

$$\begin{aligned} reverse([1,2,3,4]) &= [4] + +[3] + +[2] + +[1] \\ &= [4,3] + +[2] + +[1] \\ &= [4,3,2] + +[1] \\ &= [4,3,2,1] \end{aligned}$$

К нам на помощь приходит хвостовая рекурсия. На каждой итерации мы будем добавлять к аккумулятору новый головной элемент. Это автоматически развернёт список в противоположном направлении. Посмотрим на реализацию:

```
tail_reverse(L) -> tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

Если мы распишем шаги исполнения этой функции, то получим:

$$\begin{aligned} tail\_reverse([1,2,3,4]) &= tail\_reverse([2,3,4],[1]) \\ &= tail\_reverse([3,4,],[2,1]) \\ &= tail\_reverse([4],[3,2,1]) \\ &= tail\_reverse([],[4,3,2,1]) \\ &= [4,3,2,1] \end{aligned}$$

Можно заметить, что теперь мы посещаем линейное количество элементов. Наш стек не будет расти, и эффективность выполнения операций добавления элементов в этом случае намного выше!

Реализуем ещё одну функцию – `sublist/2`. Она принимает список L и целое число N, а возвращает N начальных элементов списка. Например, вызов `sublist([1,2,3,4,5,6],3)` возвратит [1,2,3]. И снова частный

случай – попытка получить 0 элементов из списка. Впрочем, не теряйте бдительность: в случае `sublist/2` есть отличия. Есть и второй частный случай, в котором передаётся пустой список! Если мы не сделаем проверку списка на пустоту, то при вызове `recursive:sublist([1],2).` будет выброшена ошибка, хотя в качестве результата мы ожидали получить `[1]`. Как только эти проблемы будут нами преодолены, рекурсивной части функции остаётся лишь пройти по списку, сохраняя элементы, пока она не упрётся в один из частных случаев:

```
sublist(_,0) -> [];
sublist([],_) -> [];
sublist([H|T],N) when N > 0 -> [H|sublist(T,N-1)].
```

Функцию можно привести к форме, использующей хвостовую рекурсию, тем же путём, что и прежде:

```
tail_sublist(L, N) -> tail_sublist(L, N, []).

tail_sublist(_, 0, SubList) -> SubList;
tail_sublist([], _, SubList) -> SubList;
tail_sublist([H|T], N, SubList) when N > 0 ->
tail_sublist(T, N-1, [H|SubList]).
```

Но в этой функции есть изъян. Роковой изъян! Мы используем в качестве аккумулятора список, так же как мы поступали, когда разворачивали список задом-наперёд. Если вы скомпилируете функцию, то вызов `sublist([1,2,3,4,5,6],3)` возвратит не [1,2,3], а [3,2,1]. Поэтому нам нужно взять окончательный результат и развернуть его самостоятельно. Просто поменяйте вызов `tail_sublist/2`, а рекурсивную логику оставьте прежней:

```
tail_sublist(L, N) -> reverse(tail_sublist(L, N, [])).
```

Окончательный результат будет упорядочен правильно. Может показаться, что разворот списка после вызова хвостовой рекурсии – напрасная потеря времени, и это отчасти правда (таким образом мы всё равно экономим память). Вы сможете заметить, что из-за необходимости разворота, на коротких списках ваш код с использованием обычной рекурсии работает быстрее, чем код с хвостовой рекурсией. Но по мере увеличения объёма данных, на фоне остальных операций разворачивание списка будет значить всё меньше и меньше.

Замечание: вместо собственноручно написанной функции `reverse/1` вы должны использовать `lists:reverse/1`. Функция так часто использовалась для вызовов с хвостовой рекурсией, что разработчики Erlang решили превратить её во встроенную. Списки будут разворачиваться чрезвычайно быстро (благодаря тому, что функция написана на языке C), что сделает замедление, которое вносит разворот, менее заметным. Остальной код в этой главе будет использовать нашу собственную функцию разворота элементов, но больше вы её никогда использовать не должны.

Чтобы немного развить тему, мы напишем zip-функцию. Такая функция принимает в качестве параметров два списка одинаковой длины и попарно комбинирует их элементы в виде списка кортежей. Наша функция `zip/2` будет вести себя следующим образом:

```
1> recursive:zip([a,b,c],[1,2,3]).
[{a,1},{b,2},{c,3}]
```

Так как мы хотим, чтобы оба параметра имели одинаковую длину, нашим частным случаем будет комбинирование двух пустых списков:

```
zip([],[]) -> [];
zip([X|Xs],[Y|Ys]) -> [{X,Y}|zip(Xs,Ys)].
```

Однако, если вы хотите, чтобы функция относилась к параметрам менее строго, лучше позволить ей заканчивать обработку, когда опустеет один из списков. Для такого сценария у вас появится два частных случая:

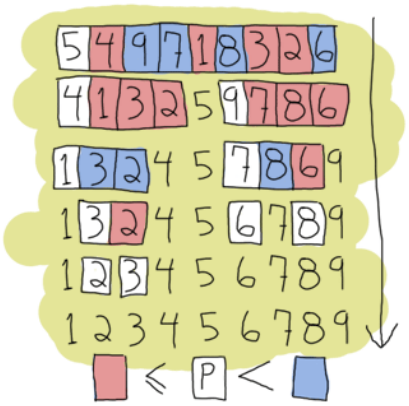
```
lenient_zip([],_) -> [];
lenient_zip(_,[]) -> [];
lenient_zip([X|Xs],[Y|Ys]) -> [{X,Y}|lenient_zip(Xs,Ys)].
```

Обратите внимание, что вне зависимости от того, какие мы определяем частные случаи, рекурсивная часть функции остаётся неизменной. Я бы посоветовал вам попробовать написать собственные версии `zip/2` и `lenient_zip/2` с использованием хвостовой рекурсии, чтобы убедиться, что вы полностью понимаете как создаются такие функции. Хвостовая рекурсия будет одной из центральных концепций, лежащих в основе большого приложения, главный цикл которого будет организован именно таким образом.

Если хотите проверить то, что у вас получилось, взгляните на мою реализацию в recursive.erl, а именно на функции `tail_zip/2` и `tail_lenient_zip/3`.

Замечание: хвостовая рекурсия не раздувает используемую память, так как виртуальная машина видит, что рекурсивный вызов происходит из хвостовой позиции (последнее выражение, которое обрабатывает функция), и устраняет текущий стековый кадр. Эта техника называется оптимизацией хвостового вызова и является специальным случаем более общей оптимизации под названием *оптимизация последнего вызова*.
Оптимизация последнего вызова происходит, когда последнее выражение в теле функции является ещё одним вызовом функции. В этом случае, как и в случае оптимизации хвостовой рекурсии, виртуальная машина Erlang не сохраняет стековый кадр. Поэтому хвостовая рекурсия возможна между несколькими функциями. Например, цепь функций `a() -> b(). b() -> c(). c() -> a().` создаст в результате бесконечный цикл, который не приведёт к исчерпанию памяти, так как оптимизация последнего вызова предотвращает переполнение стека. Этот принцип в сочетании с использованием аккумуляторов делает хвостовую рекурсию полезной техникой.

7.5 Быстро, сортируй!



Я могу (и буду) считать, что рекурсия и хвостовая рекурсия вам ясна. Но просто чтобы в этом убедиться, я приведу более сложный пример – реализацию алгоритма быстрой сортировки. Да, традиционный «эй, смотри, я могу писать сжатый функциональный код» канонический пример. Наивная реализация алгоритма быстой сортировки выбирает первый элемент списка, *опорный элемент*, и перемещает все элементы меньшие, либо равные опорному элементу в новый список, а все элементы больше опорного элемента – в другой список. Затем мы повторяем эту процедуру для полученных списков. Этот процесс продложается до тех пор, пока для сортировки не останется ничего, кроме пустых списков, которые и станут нашим частным случаем. Эта реализация считается простой, потому что более эффективные версии быстрой сортировки будут пытаться выбрать оптимальные опорные элементы с целью ускорения. Впрочем, в нашем примере это не столь важно.

Нам понадобятся две функции: первая будет разбивать список на

части, содержащие меньшие и большие элементы, а вторая будет применять функцию разбиения к новым спискам и соединять их в единое целое. Для начала мы напишем функцию соединения:

```
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
{Smaller, Larger} = partition(Pivot,Rest,[],[]),
quicksort(Smaller) ++ [Pivot] ++ quicksort(Larger).
```

Здесь мы видим: частный случай; список, который разбит при помощи ещё одной функции на части с большими и меньшими элементами; опорный элемент, к которому слева и справа присоединены отсортированные списки. Этот код отвечает за соединение списков. Перейдём к функции разбиения:

```
partition(_,[], Smaller, Larger) -> {Smaller, Larger};
partition(Pivot, [H|T], Smaller, Larger) ->
    if H =< Pivot -> partition(Pivot, T, [H|Smaller], Larger);
       H >  Pivot -> partition(Pivot, T, Smaller, [H|Larger])
    end.
```

Теперь функцию быстрой сортировки можно запустить. Если вы искали в Интернет примеры программ на Erlang, то, скорее всего, наталкивались на другую реализацию быстрой сортировки, ту, которая выглядит проще и легче читается, но использует списковые выражения. Их легко можно применить в месте, где создаются новые списки, в функции `partition/4`:

```
lc_quicksort([]) -> [];
lc_quicksort([Pivot|Rest]) ->
    lc_quicksort([Smaller || Smaller <- Rest, Smaller =< Pivot])
    ++ [Pivot] ++
    lc_quicksort([Larger || Larger <- Rest, Larger > Pivot]).
```

Главное отличие этой версии в том, что её код легче читать, но за лёгкость чтения приходится платить тем, что для разбиения списка, необходимо перебрать все его элементы. В этом проявляется борьба ясности кода против скорости его исполнения. Но проигрываете в этой борьбе лишь вы, потому что для нужд сортировки уже создана функция `lists:sort/1`. Используйте лучше её.

Не забывайтесь:
Выразительность кода хороша для обучения, но не всегда полезна для производительности. Множество руководств по функциональному программированию ни слова об этом не говорят! Во-первых, обеим реализациям, приведённым здесь, приходится неоднократно обрабатывать элементы равные опорному элементу. Для увеличения эффективности можно было бы возвращать три списка: элементы меньше, больше и равные опорному элементу.
Ещё одна проблема связана с тем, что нам необходимо неоднократно проходить по разбитым спискам при их слиянии с опорным элементом. Можно немного уменьшить накладные расходы, если производить объединение во время разбиения списков на три части. Тем, кто заинтересовался реализацией, я предлагаю взглянуть на последнюю функцию (`bestest_qsort/1`) из файла recursive/erl.
Приятно отметить, что все рассмотренные реализации быстрой сортировки будут работать со списками, содержащими любые типы, даже кортежи или что-то подобное. Попробуйте, они работают!

7.6 Больше чем списки

Читая эту главу, вы, возможно, начинаете думать, что рекурсия в Erlang, главным образом, связана со списками. Хотя списки и могут служить хорошим примером структуры, которую можно определить через рекурсию, но на ней, конечно, всё не заканчивается. Мы рассмотрим, для разнообразия, как можно создавать двоичные деревья и считывать из них данные.

Для начала неплохо было бы определить, что такое дерево. В нашем случае оно снизу доверху состоит из узлов. Узлы – это кортежи, которые содержат ключ, данные связанные с ключом, и два других узла. Эти два узла мы разделяем на узел с ключом меньшим и большим, чем ключ узла, который их содержит. И вот тут появляется рекурсия! Дерево – это узел, который содержит узлы, каждый из которых содержит узлы, которые, в

свою очередь, содержат узлы. Бесконечно это не может продолжаться (у нас нет бесконечного пространства для хранения данных), поэтому мы скажем, что узлы, которые содержатся в наших узлах, также могут быть пустыми.

Кортежи – подходящая структура для представления узлов. Для нашей реализации мы определим кортежи вида `{node, {Key, Value, Smaller, Larger}}` (меченый кортеж!), где *Smaller* и *Larger* могут быть другим подобным узлом, или пустым узлом (`{node, nil}` . Более сложные концепции нам не понадобятся.

Начнём создавать модуль для нашей очень простой реализации дерева. Первая функция `empty/0` – возвращает пустой узел. Пустой узел – это начальная точка нового дерева, которая также называется *корневым узлом*:

```
-module(tree).
-export([empty/0, insert/3, lookup/2]).

empty() -> {node, 'nil'}.
```

Мы скрываем реализацию, используя эту функцию, и, затем, инкапсулируем её в одинаковое представление узлов, чтобы пользователю не пришлось задумываться о том, как устроен наш код. Вся эта информация будет существовать лишь в рамках модуля. Если вам когда-либо придёт в голову изменить представление узла – вы сможете сделать изменения, не нарушая работу внешнего кода.

Для того, чтобы наполнить дерево содержимым, нам необходимо, для начала, понять, как по нему рекурсивно перемещаться. Давайте поступим так же, как мы поступали в любом другом примере с рекурсией – попытаемся найти частный случай. Так как пустое дерево состоит из пустого узла, то, рассуждая логически, нашим частным случаем будет пустой узел. Мы сможем добавить новую пару ключ/значение, когда наткнёмся на пустой узел. В остальных случаях наш код будет ходить по дереву в поисках пустого узла, который можно наполнить данными.

Мы будем искать пустой узел, начиная с корневого узла. Для этого мы должны использовать знание о том, что *Меньший* и *Больший* узлы позволяют нам определять направление перемещения, сравнивая новый ключ, который необходимо добавить, с ключом текущего узла. Если новый ключ меньше ключа текущего узла, то мы продолжаем искать пустой узел внутри *Меньшего* узла, и если больше, то внутри *Большего*. Нельзя забывать ещё об одном случае: что будет, если новый ключ равен ключу текущего узла? У нас есть две возможности: сгенерировать ошибку, либо заменить значение узла новым. Здесь мы используем за-

мену. Все эти рассуждения, заключённые в код, выглядят следующим образом:

```
insert(Key, Val, {node, 'nil'}) ->
    {node, {Key, Val, {node, 'nil'}, {node, 'nil'}}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when
   NewKey < Key ->
    {node, {Key, Val, insert(NewKey, NewVal, Smaller), Larger}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when
   NewKey > Key ->
    {node, {Key, Val, Smaller, insert(NewKey, NewVal, Larger)}};
insert(Key, Val, {node, {Key, _, Smaller, Larger}}) ->
    {node, {Key, Val, Smaller, Larger}}.
```

Обратите внимание, что функция возвращает совершенно новое дерево. Это характерная черта функциональных языков программирования, в которых присваивание происходит лишь один раз. Хотя это и может показаться неэффективным, но большая часть структур, которые лежат в основе обеих версий дерева, остаются теми же и используются совместно, а поэтому копируются виртуальной машиной лишь по необходимости.

Для завершения этой демонстрационной реализации дерева, нам необходимо создать функцию `lookup/2`, которая позволит найти значение узла по ключу. Мы будем использовать принцип, чрезвычайно похожий на тот, который мы использовали при добавлении новых данных в дерево: мы проходим по узлам дерева, шаг за шагом проверяя, что искомый ключ больше, меньше, либо равен ключу текущего узла. У нас есть два частных случая: первый – когда узел пуст (ключа в дереве нет), и второй – когда ключ найден. Мы не хотим, чтобы наша программа сбоила каждый раз, когда мы ищем ключ, который в дереве отсутствует, а поэтому в такой ситуации мы будем возвращать атом 'undefined'. В случае удачи, мы возвращаем `{ok, Value}`. Если бы мы просто возвращали *Value*, а искомый узел содержал атом 'undefined', то мы не могли бы понять, было найдено верное значение или нет. Обёртывание успешно найденных результатов позволяет легко отличать их от неудачной попытки поиска. Вот реализация функции:

```
lookup(_, {node, 'nil'}) ->
    undefined;
lookup(Key, {node, {Key, Val, _, _}}) ->
    {ok, Val};
lookup(Key, {node, {NodeKey, _, Smaller, _}}) when Key < NodeKey
   ->
    lookup(Key, Smaller);
lookup(Key, {node, {_, _, _, Larger}}) ->
    lookup(Key, Larger).
```

Всё, мы закончили. Давайте потестируем нашу структуру – напишем небольшую адресную книгу для электронной почты. Скомпилируйте файл и запустите оболочку:

```
1> T1 = tree:insert("Jim Woodland", "jim.woodland@gmail.com",
    tree:empty()).
{node,{"Jim Woodland","jim.woodland@gmail.com",
     {node,nil},
     {node,nil}}}
2> T2 = tree:insert("Mark Anderson", "i.am.a@hotmail.com", T1).
{node,{"Jim Woodland","jim.woodland@gmail.com",
     {node,nil},
     {node,{"Mark Anderson","i.am.a@hotmail.com",
          {node,nil},
          {node,nil}}}}}
3> Addresses = tree:insert("Anita Bath", "abath@someuni.edu",
    tree:insert("Kevin Robert", "myfairy@yahoo.com", tree:insert(
    "Wilson Longbrow", "longwil@gmail.com", T2))).
{node,{"Jim Woodland","jim.woodland@gmail.com",
     {node,{"Anita Bath","abath@someuni.edu",
          {node,nil},
          {node,nil}}},
     {node,{"Mark Anderson","i.am.a@hotmail.com",
          {node,{"Kevin Robert","myfairy@yahoo.com",
               {node,nil},
               {node,nil}}},
          {node,{"Wilson Longbrow","longwil@gmail.com",
               {node,nil},
               {node,nil}}}}}}}
```

Теперь можно поискать адреса с помощью нашей книги:

```
4> tree:lookup("Anita Bath", Addresses).
{ok, "abath@someuni.edu"}
5> tree:lookup("Jacques Requin", Addresses).
undefined
```

На этом мы завершаем рассмотрение примера функциональной адресной книги, построенной при помощи рекурсивной структуры данных отличной от списка!

Замечание: наша реализация весьма примитивна: мы не поддерживаем часто используемые операции, такие как удаление узлов или перебалансировка дерева для ускорения последующих операций поиска. Если вам интересно было бы реализовать и/или изучить эти операции – обратитесь к реализации модуля `gb_trees` (`otp_src_R<version>B<revision>/lib/stdlib/src/gb_trees.erl`). Кроме того, при работе с деревьями используйте именно этот модуль и не пытайтесь изобрести велосипед.

7.7 Рекурсивное мышление

Если вы поняли всё сказанное в этой главе, то рекурсивное мышление, вероятно, начинает становиться частью вашей интуиции. Главное отличие рекурсивных определений от их императивных аналогов (обычно это циклы while или for) в том, что вместо пошагового выполнения («сделай это, потом то, затем вот это, после этого заверши исполнение») мы используем декларативный подход («если ты получишь такой входящий параметр, сделай это, в противном случае – вот это»). Такое свойство становится более очевидным при использовании сопоставления с образцом в заголовке функции.

Если вы до сих пор не поняли как работает рекурсия, может быть вам нужно прочитать вот 7 это.

Отбросив шутки в сторону, нужно признать, что иногда рекурсия в сочетании с сопоставлением с образцом – это оптимальный способ записи ясных и понятных алгоритмов. После того как каждая часть задачи поделена на функции, которые не поддаются дальнейшему упрощению, алгоритм превращается в компоновку правильных ответов, получаемых из коротких подпрограмм (что-то подбное мы делали с быстрой сортировкой). С обычными циклами можно тоже использовать такой тип мысленной абстракции, но на практике это получается лучше именно с рекурсией. Ваш опыт может свидетельствовать об ином.

А теперь, леди и джентльмены – дискуссия автора с самим собой

– Хорошо, кажется я понял рекурсию. Я понимаю, что она связана с декларативностью, что корни её уходят в математику, как и у неизменяемых переменных. Я понимаю, что в некоторых случаях её проще использовать. Что ещё?

– Она имеет регулярную структуру. Найди частный случай, запиши его. Для получения результата нужно, чтобы любой другой случай сходился к одному из частных. Так проще записывать функции.

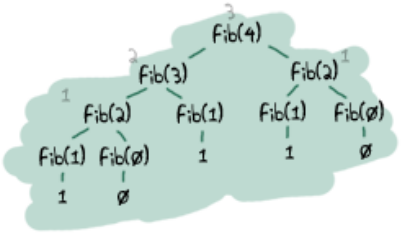
– Ага, понял, ты это уже говорил несколько раз. Я то же самое могу сделать циклами.

– Да, можешь. Не буду отрицать!

– Хорошо. Мне не совсем ясно, зачем ты писал функции, которые не используют хвостовую рекурсию, раз уж они намного хуже тех, что её используют.

– А, ну это просто чтобы было легче понять. Мне показалось, что переход от обычный рекурсии, которая выглядит красивее и проще для понимания, к хвостовой рекурсии, которая теоретически более эффективна, неплохо показал все плюсы и минусы этих подходов.

– Хорошо, значит ни для чего кроме обучения они не годятся, я понял.



– Это не совсем так. Разница в производительности между функцией с хвостовой рекурсией и обычной рекурсией будет не сильно заметна. Хвостовая рекурсия хорошо подходит для функций, которые итерируют бесконечно, например для главных циклов. Есть также функции, которые порождают очень большой стековый кадр, медленно исполняются и, если их не записывать в виде хвостовой рекурсии, могут рано приводить к аварийному завершению. Неплохим примером может служить задача вычисления чисел Фибоначчи, сложность которой растёт экспоненциально, если не решать её в виде итеративного процесса или хвостовой рекурсии. Также необходимо профилировать написанный код (позже я покажу как это делать, я обещаю), определять и исправлять места, которые замедляют исполнение.

– Но циклы – это всегда итеративный процесс, проще использовать их и не знать проблем.

– Да, но... но... мой прекрасный Erlang...

– Ну, зашибись. Весь сыр-бор из-за того, что в Erlang нет «while» и «for». Спасибо большое, я пошёл дальше программировать свой тостер на C!

– Погоди-ка! У функциональных языков программирования есть и другие возможности! Кроме шаблона для поиска частных случаев, который облегчил нам жизнь при написании рекурсивных функций, существует ещё много других шаблонов, которые придумали умные люди. Благодаря этому необходимость самостоятельно писать рекурсивные функции значительно уменьшается. Если ты продолжишь занятия, я покажу как строить такие абстракции. Но для этого нам нужно взять что-

то помощнее. Позволь мне рассказать о функциях высшего порядка...

Глава 8

Функции высшего порядка

8.1 Подбавим функциональщины

Важной частью всех функциональных языков является возможность передачи функции как параметра для другой функции. Это, в свою очередь, связывает параметр-функцию с переменной, которую можно использовать внутри функции как любую другую. Функция, которая может таким способом принимать другие функции, называется функцией высшего порядка. Функции высшего порядка – это мощный способ абстракции и один из лучших инструментов, которым предлагает овладеть Erlang.

Опять же, эта концепция берёт начало в математике, а именно в лямбда-исчислении. Не буду углубляться в детали лямбда-исчисления, потому что эта теория довольно сложна для понимания, и немного выходит за рамки контекста, который мы рассматриваем. Тем не менее, я бы кратко охарактеризовал её как систему, в которой всё представлено в виде функций, даже числа. Так как любая сущность – это функция, то в качестве параметров мы должны передавать функциям другие функции и оперировать ими, опять же, при помощи функций!

Ну ладно, наверняка всё сказанное звучит немного странно, поэтому начнём с примера:

```
-module(hhfuns).
-compile(export_all).

one() -> 1.
```

```
two() -> 2.

add(X,Y) -> X() + Y().
```

А теперь откройте оболочку Erlang, скомпилируйте модуль и посмотрите как он работает:

```
1> c(hhfuns).
{ok, hhfuns}
2> hhfuns:add(one,two).
** exception error: bad function one
     in function  hhfuns:add/2
3> hhfuns:add(1,2).
** exception error: bad function 1
     in function  hhfuns:add/2
4> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
```

Не совсем понятно? Когда разберётесь в принципе работы, сразу станет яснее (всегда так, не правда ли?). В команде под номером 2 атомы `one` и `two` передаются в функцию `add/2`, которая затем использует оба атома в качестве имён для функций (`X() + Y()`). Если имена функций записаны без списка параметров, то эти имена интерпретируются как атомы, а атомы не могут быть функциями, поэтому вызов заканчивается неудачей. По той же причине не удаётся исполнить выражение 3. Значения 1 и 2 тоже невозможно использовать как функции, а нам нужны именно функции!

Поэтому мы должны использовать новый способ записи, который позволит передавать функции, размещённые за пределами модуля. Именно такую задачу выполняет `fun Module:Function/Arity`. Эта строка указывает VM, что та должна взять определённую функцию и привязать её к переменной.

Так что же мы приобретаем, используя функции таким образом? Для понимания рассмотрим маленький пример. Мы добавим в модуль `hhfuns` пару функций, которые будут рекурсивно проходить по списку и прибавлять или вычитать единицу из каждого элемента:

```
increment([]) -> [];
increment([H|T]) -> [H+1|increment(T)].

decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].
```

Видите как эти функции похожи друг на друга? Они практически делают одно и то же: проходят по списку, применяют к каждому элементу функцию (`+` или `-`) и затем снова вызывают себя. В этом коде

практически ничего не меняется, кроме применяемой функции и рекурсивного вызова. Для такого рекурсивного вызова над списком, основа всегда остаётся неизменной. Мы обобщим все похожие части в единую функцию (`map/2`), которая будет принимать в качестве аргумента ещё одну функцию:

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].

incr(X) -> X + 1.
decr(X) -> X - 1.
```

Её можно протестировать в оболочке:

```
1> c(hhfuns).
{ok, hhfuns}
2> L = [1,2,3,4,5].
[1,2,3,4,5]
3> hhfuns:increment(L).
[2,3,4,5,6]
4> hhfuns:decrement(L).
[0,1,2,3,4]
5> hhfuns:map(fun hhfuns:incr/1, L).
[2,3,4,5,6]
6> hhfuns:map(fun hhfuns:decr/1, L).
[0,1,2,3,4]
```

Вычисление даёт тот же самый результат, и, вдобавок мы получаем изящную абстракцию! Каждый раз, когда вы хотите применить функцию к каждому элементу в списке, вам нужно лишь вызвать `map/2` и передать ей в качестве параметра собственную функцию. Впрочем, было бы утомительно помещать каждую функцию, которую мы хотим передать `map/2` в качестве параметра, в модуль, затем её экспортировать, компилировать и т.д. В сущности, это просто непрактично. Нам нужны функции, которые можно было бы определять на ходу...

8.2 Анонимные функции

Анонимные функции, или *funs*, справляются с этой проблемой, позволяя вам декларировать особенный вид функций без необходимости их именования. Такие функции могут делать практически всё, что умеют обычные функции, кроме рекурсивных вызовов (как бы они их делали? Они же анонимные!) Вот их синтаксис:

```
fun(Args1) ->
    Expression1, Exp2, ..., ExpN;
```

```
    (Args2) ->
    Expression1, Exp2, ..., ExpN;
    (Args3) ->
    Expression1, Exp2, ..., ExpN
end
```

А использовать их можно следующим способом:

```
7> Fn = fun() -> a end.
#Fun<erl_eval.20.67289768>
8> Fn().
a
9> hhfuns:map(fun(X) -> X + 1 end, L).
[2,3,4,5,6]
10> hhfuns:map(fun(X) -> X - 1 end, L).
[0,1,2,3,4]
```

Теперь-то вам, должно быть, становится ясна одна из причин, по которой людям так нравится функциональное программирование: в коде можно создавать абстракции на очень низком уровне. Поэтому основные концепции, такие как циклы, можно игнорировать, и сконцентрироваться на том, *что* необходимо сделать, вместо того *как* это должно быть сделано.

Анонимные функции сами по себе – довольно хорошая абстракция, но в них заключены дополнительные скрытые силы:

```
11> PrepareAlarm = fun(Room) ->
11>                    io:format("Alarm set in ~s.~n",[Room]),
11>                    fun() -> io:format("Alarm tripped in ~s!
    Call Batman!~n",[Room]) end
11>                end.
#Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.6.13229925>
13> AlarmReady().
Alarm tripped in bathroom! Call Batman!
ok
```

Бэтмэн, оставайся на связи! Что тут происходит? Во-первых, мы декларируем анонимную функцию, которую присваиваем переменной *PrepareAlarm*. Эта функция ещё не запускалась: её исполнят во время вызова `PrepareAlarm("bathroom")`.

В этот момент будет обработан вызов функции `io:format/2`, и на экране отобразится текст “Alarm set”. Второе выражение (ещё одна анонимная функция), возвращается вызывающему коду и присваивается переменной *AlarmReady*. Обратите внимание, что в этой функции значение переменной *Room* берётся из «родительской» функции (*PrepareAlarm*). Этот

эффект имеет отношение к концепции *замыканий* (closures).

Чтобы понять замыкания, необходимо понять что такое область действия (контекст). Область действия функции можно представить как место, в котором хранятся все переменные и их значения. В функции `base(A) -> B = A + 1.`, *A* и *B* определены в контексте функции `base/1`. Это означает, что в любом месте функции `base/1` можно обратиться к переменной *A* или *B*, и ожидать, что с ними связано значение. А когда я говорю "в любом месте" – я, сынок, не шучу; к анонимным функциям это тоже относится:

```
base(A) ->
    B = A + 1,
    F = fun() -> A * B end,
    F().
```

Переменные B и A всё ещё связаны с областью действия функции `base/1`, поэтому функция F может беспрепятственно к ним обращаться. Так происходит потому, что F наследует область действия функции `base/1`. Как и в правилах наследования, которые действуют в реальной жизни, у родителей обычно нет доступа к тому, чем владеют их дети:

```
base(A) ->
    B = A + 1,
    F = fun() -> C = A * B end,
    F(),
    C.
```

В этой версии нашей функции, переменная *B* всё ещё равна `A + 1`, а *F* по прежнему исполняется без проблем. Но вот переменная *C* существует лишь в контексте анонимной функции *F*. Когда `base/1` пытается в последней строке обратиться к значению *C*, она натыкается на свободную переменную. По правде говоря, если бы вы попытались эту функцию скомпилировать, компилятор пришёл бы в бешенство. Наследование работает лишь в одну сторону.

Важно отметить, что наследуемый контекст повсюду сопровождает анонимную функцию, даже когда её передают другой функции:

```
a() ->
    Secret = "pony",
    fun() -> Secret end.
```

```
b(F) ->
    "a/0's password is "++F().
```

А если мы её скомпилируем:

```
14> c(hhfuns).
{ok, hhfuns}
15> hhfuns:b(hhfuns:a()).
"a/0's password is pony"
```

Кто разболтал пароль функции `a/0`? Сама функция `a/0` и разболтала. Так как анонимная функция в момент объявления находится в контексте `a/0`, то она всё ещё будет иметь доступ к этому контексту во время её исполнения в функции `b/1`, как я объяснял выше. Это свойство очень полезно, так как позволяет переносить параметры и их содержимое за пределы оригинального контекста, туда где в полном контексте больше нет необходимости (в точности так, как мы поступили с Бэтмэном в предыдущем примере).

Анонимные функции чаще всего используют для переноса состояния, когда ваша функция принимает несколько аргументов, но один из этих аргументов – константа:

```
16> math:pow(5,2).
25.0
17> Base = 2.
2
18> PowerOfTwo = fun(X) -> math:pow(Base,X) end.
#Fun<erl_eval.6.13229925>
17> hhfuns:map(PowerOfTwo, [1,2,3,4]).
[2.0,4.0,8.0,16.0]
```

Обернув вызов функции `math:pow/2` анонимной функцией, в контексте которой переменная *Base* получает своё значение, мы используем числа из списка как степени для *Base* при каждом вызове функции *PowerOfTwo*.

Если внутри анонимной функции попытаться переопределить контекст, можно столкнуться с проблемой:

```
base() ->
    A = 1,
    (fun() -> A = 2 end)().
```

Этот код объявит анонимную функцию, а затем исполнит её. Так как анонимная функция наследует контекст функции `base/0`, то попытка использовать оператор `=` приведёт к сравнению числа 2 с перемен-

ной A (которая связана со значением 1). Эта операция гарантированно закончится неудачей. Тем не менее, переменную всё же можно переопределить, если сделать это во вложенном заголовке функции:

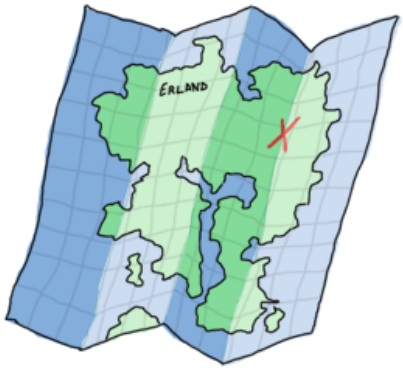
```
base() ->
    A = 1,
    (fun(A) -> A = 2 end)(2).
```

Этот код сработает. При компиляции вы получите предупреждение о *затенении* (shadowing) (“*Warning: variable 'A' shadowed in 'fun'*”). Затенение – это термин, который используют для описания ситуации, когда новая переменная имеет имя, совпадающее с именем переменной из родительского контекста. Это предупреждение призвано предотвратить вероятные ошибки (и чаще всего ему это удаётся), поэтому в подобных обстоятельствах подумайте, а не переименовать ли вам одну из переменных.

Как я и обещал в конце предыдущей главы, давайте немного отвлечёмся от теории анонимных функций, и исследуем некоторые общие абстракции, которые позволяют нам избавиться от необходимости писать рекурсивные функции.

8.3 Отображения (maps), фильтры, свёртки (folds) и прочее

В начале этой главы я кратко продемонстрировал, как преобразовать две подобные функции в функцию `map/2`. Также я утверждал, что такую функцию можно использовать для любого списка, в котором мы хотим производить какое-либо действие с каждым элементом. Функция выглядела следующим образом:




```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].
```

Однако, существует много других абстракций, подобных этой, которые можно построить из часто встречающихся рекурсивных функций. Давайте для начала взглянем на парочку функций:

```
%% only keep even numbers
```

```
even(L) -> lists:reverse(even(L,[])).

even([], Acc) -> Acc;
even([H|T], Acc) when H rem 2 == 0 ->
    even(T, [H|Acc]);
even([_|T], Acc) ->
    even(T, Acc).

%% only keep men older than 60
old_men(L) -> lists:reverse(old_men(L,[])).

old_men([], Acc) -> Acc;
old_men([Person = {male, Age}|People], Acc) when Age > 60 ->
    old_men(People, [Person|Acc]);
old_men([_|People], Acc) ->
    old_men(People, Acc).
```

Первая принимает список чисел и возвращает только чётные. Вторая проходит по списку с данными вида {Пол, Возраст} и возвращает лишь записи о мужчинах старше 60 лет. В этих функциях найти общее немного сложнее, но некоторое подобие всё же есть. Обе функции работают над списками, возвращают элементы, которые прошли некий тест (его ещё называют *предикатом*), а остальные элементы отбрасывают. Из этого обобщения мы можем извлечь всю необходимую нам информацию и преобразовать её в функцию:

```
filter(Pred, L) -> lists:reverse(filter(Pred, L,[])).

filter(_, [], Acc) -> Acc;
filter(Pred, [H|T], Acc) ->
    case Pred(H) of
        true  -> filter(Pred, T, [H|Acc]);
        false -> filter(Pred, T, Acc)
    end.
```

Чтобы воспользоваться функцией-фильтром, нам необходимо лишь получить извне тестирующую функцию. Скомпилируйте модуль `hhfuns` и попробуйте им воспользоваться:

```
1> c(hhfuns).
{ok, hhfuns}
2> Numbers = lists:seq(1,10).
[1,2,3,4,5,6,7,8,9,10]
3> hhfuns:filter(fun(X) -> X rem 2 == 0 end, Numbers).
[2,4,6,8,10]
4> People = [{male,45},{female,67},{male,66},{female,12},{unkown
    ,174},{male,74}].
```

```
[{male,45},{female,67},{male,66},{female,12},{unkown,174},{male
    ,74}]
5> hhfuns:filter(fun({Gender,Age}) -> Gender == male andalso Age
    > 60 end, People).
[{male,66},{male,74}]
```

Эти примеры демонстрируют, что программисту, который использует функцию `filter/2`, нужно лишь задать предикат и указать список, который будет фильтроваться. О самом процессе перемещения по списку и об отбрасывании ненужных элементов думать больше не нужно. При обобщении функционального кода происходит одна важная вещь: мы пытаемся избавиться от того, что остаётся постоянным, и позволяем программисту предоставить лишь то, что подвергается изменению.

В предыдущей главе мы применяли к спискам другой вид рекурсивных манипуляций, в котором мы последовательно рассматривали каждый элемент списка и приводили их к одному ответу. Эта операция называется *свёрткой* (fold) и может применяться для следующих функций:

```
%% find the maximum of a list
max([H|T]) -> max2(T, H).

max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).

%% find the minimum of a list
min([H|T]) -> min2(T,H).

min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).

%% sum of all the elements of a list
sum(L) -> sum(L,0).

sum([], Sum) -> Sum;
sum([H|T], Sum) -> sum(T, H+Sum).
```

Чтобы понять как себя ведёт свёртка, нам необходимо найти общее в этих действиях, а после найти различия. Как было указано выше, мы всегда приводим список к единичному значению. Следовательно, наша свёртка должна повторять действие, сохраняя один элемент. Нам не нужно строить новый список. Стражи нам не понадобятся, так как они присутствуют не всегда, а поэтому должны находиться в пользовательской функции. В этом отношении наша функция свёртки будет, скорее всего, походить на функцию sum.

Ещё одной неприметной частью всех трёх функций, о которой мы пока не упоминали, является то, что у каждой функции должно быть начальное значение, от которого будет начинаться отсчёт. В случае `sum/2` мы используем 0, так как проводим операцию сложения, которая нейтральна относительно 0, и при вычислении `X = X + 0` мы никак не повлияем на результат. Если бы мы использовали операцию умножения `X = X * 1`, то в качестве стартового значения избрали бы 1. У функций `min/1` и `max/1` стартового значения по умолчанию быть не может. Если список полностью состоит из отрицательных чисел, а мы начнём с 0, то получим неверный результат. Поэтому нам необходимо использовать в качестве начальной точки первый элемент списка. К сожалению, мы не можем заранее применить такие рассуждения к любой ситуации, а поэтому оставляем решение этой проблемы программисту. Принимая во внимание все перечисленные элементы, мы можем построить следующую абстракцию:

```
fold(_, Start, []) -> Start;
fold(F, Start, [H|T]) -> fold(F, F(H,Start), T).
```

А затем исполнить:

```
6> c(hhfuns).
{ok, hhfuns}
7> [H|T] = [1,7,3,5,9,0,2,3].
[1,7,3,5,9,0,2,3]
8> hhfuns:fold(fun(A,B) when A > B -> A; (_,B) -> B end, H, T).
9
9> hhfuns:fold(fun(A,B) when A < B -> A; (_,B) -> B end, H, T).
0
10> hhfuns:fold(fun(A,B) -> A + B end, 0, lists:seq(1,6)).
21
```

Практически любую функцию, которая сводит список значений к одному элементу, можно выразить в виде свёртки.

Занятно, что вы можете представить аккумулятор как единичный элемент (или единичную переменную), а аккумулятор может быть списком. Так мы можем использовать свёртку для построения списков. Это значит, что свёртка универсальна в том смысле, что с её помощью можно реализовать практически любую другую рекурсивную функцию для списков, даже отображение и фильтр:

```
reverse(L) ->
```

```
    fold(fun(X,Acc) -> [X|Acc] end, [], L).

map2(F,L) ->
    reverse(fold(fun(X,Acc) -> [F(X)|Acc] end, [], L)).

filter2(Pred, L) ->
    F = fun(X,Acc) ->
        case Pred(X) of
            true  -> [X|Acc];
            false -> Acc
        end
    end,
    reverse(fold(F, [], L)).
```

И эти функции будут работать так же как и те, что мы написали ранее. Ну как, мощные абстракции или нет?

Отображение, фильтры и свёртки – это лишь часть функций для списков, которые предоставляются стандартной библиотекой Erlang (см. `lists:map/2`, `lists:filter/2`, `lists:foldl/3` и `lists:foldr/3`. Стоит отметить также функции `all/2` и `any/2`, которые принимают предикат и проверяют, что для каждого элемента предикат возвращает true, или что хотя бы для одного из них предикат возвращает true, соответственно. Ещё есть функция `dropwhile/2`, которая игнорирует элементы списка до тех пор, пока не находит один, удовлетворяющий предикату, и противоположная функция `takewhile/2`, которая будет возвращать элементы до тех пор, пока не встретит такой, для которого предикат не возвращает true. Две предыдущие функции объединяет `partition/2`. Эта функция принимает один список, а возвращает два. Первый содержит термы, которые удовлетворяют предикату, а второй – остальные элементы. Кроме того, для обработки списков часто используются такие функции как `flatten/1`, `flatlength/1`, `flatmap/2`, `merge/1`, `nth/2`, `nthtail/2`, `split/2` и другие.

Также в этом модуле можно найти и другие функции, такие как zip-функция (которую мы рассматривали в предыдущей главе), unzip--функция, комбинации отображений и свёрток и т.д. Рекомендую вам прочитать описание списков. В этом документе вы найдёте примеры использования списковых функций. Применяя те концепции, которые были заключены умными людьми в этих абстракциях, у вас редко будет появляться необходимость в самостоятельном создании рекурсивных функций.

Глава 9

Ошибки и исключения

9.1 Придержи коней!

Для этой главы невозможно подобрать подходящее место в книге. К этому моменту вы изучили уже достаточно, чтобы начать натыкаться на ошибки, но ещё недостаточно, чтобы знать, как их контролировать. По правде говоря, в этой главе мы не сможем рассмотреть все механизмы управления ошибками. Отчасти это невозможно потому, что в Erlang существуют две главные парадигмы: функциональная и параллельная (concurrent). О функциональной я рассказываю с самого начала книги: ссылочная прозрачность (чистота, referntial transparency), рекурсия, функции высшего порядка и т.д. Но именно параллельная часть прославила Erlang: акторы, тысячи и тысячи параллельных (concurrent) процессов, деревья контроля и т.д.

Я считаю, что прежде чем переходить к параллельной части, совершенно необходимо изучить функциональную. Поэтому я затрону лишь функциональное подмножество языка. Чтобы управлять ошибками, нам нужно сначала их понять.

Замечание: хоть Erlang и позволяет использовать сразу несколько способов управления ошибками в функциональном коде, но чаще всего вы будете слышать, что делать ничего не нужно, а нужно позволить процессу упасть. Я уже намекал на это во 1 введении. Механизмы, которые позволяют вам программировать в таком стиле, находятся в параллельной части языка.

9.2 Компиляция ошибок

Ошибки подразделяются на множество видов: ошибки времени компиляции, логические ошибки, ошибки времени исполнения и сгенерированные ошибки. В этом разделе я сконцентрирую внимание на ошибках времени компиляции, а о других расскажу чуть позже.

Ошибки времени компиляции чаще всего являются синтаксическими ошибками: нужно проверить наименование функций, языковые токены (скобки, квадратные скобки, точки, запятые), арность ваших функций и т.д. Вот список некоторых часто встречающихся проблем времени компиляции и потенциальные способы их разрешения:

module.beam: Module name 'madule' does not match file name 'module'
Имя модуля, которое вы указали в атрибуте `-module` не совпадает с именем файла.

./module.erl:2: Warning: function some_function/0 is unused
Вы не проэкспортировали функцию, либо в месте её использования указано неверное имя или арность. Возможно также, что вы записали функцию, которая больше нигде не вызывается. Проверьте ваш код!

./module.erl:2: function some_function/1 undefined
Функция не существует. Вы ввели неверное имя или указали неверную арность либо в атрибуте `-export`, либо во время определения функции. Это сообщение также можно увидеть, когда данная фунция не может быть скомпилирована. Обычно такое случается из-за какой-либо синтаксической ошибки, например, если вы забыли поставить точку в конце функции.

./module.erl:5: syntax error before: 'SomeCharacterOrWord'
Эта ошибка может быть вызвана множеством причин, а именно: незакрытыми скобками, неверным завершением кортежа или выражения (когда вы, к примеру, закрыли заключительную ветку `case` при помощи запятой). Среди других причин можно выделить использование зарезервированного атома в вашем коде, либо юникодного символа, который был искажён при перекодировке (я видел и такое!)

./module.erl:5: syntax error before:
Да уж, смысл сообщения не вполне очевиден. Эта ошибка выдаётся, когда неверно завершена одна из строк. Она является частным случаем предыдущей ошибки. Просто будьте внимательны.

./module.erl:5: Warning: this expression will fail with a 'badarith' exception
Всё в Erlang вертится вокруг динамической типизации, но не забывайте, что типизация в Erlang ещё и сильная. В этом примере у компилятора хватило сообразительности, чтобы определить, что в одном из арифметических выражений кроется ошибка (например, `llama + 5`). Впрочем, более сложные ошибки типов пройдут незамеченными.

./module.erl:5: Warning: variable 'Var' is unused
Вы объявили переменную и нигде её не использовали. Это может оказаться ошибкой, так что перепроверьте ваш код. Если вы это сделали намеренно, то наверняка лучше поменять имя переменной на `_` или поместить перед именем переменной знак подчёркивания (что-то вроде *_Var*). Так следует поступать, когда вы считаете, что наличие имени улучшит читабельность кода.

./module.erl:5: Warning: a term is constructed, but never used
Вы создали в одной из ваших функций кортеж, список, или анонимную функцию, которую впоследствии не связали с переменной и не использовали в качестве возвращаемого значения. Это предупреждение сообщает, что вы создаёте что-либо впустую или совершили какую-либо ошибку.

./module.erl:5: head mismatch
Возможно, у вашей функции несколько заголовков с различной арностью. Не забывайте, что изменяя арность можно создавать разные функции с одинаковыми именами. В объявлении одной функции нельзя чередовать заголовки с разной арностью. Эту ошибку также можно встретить, когда между заголовками одной функции вставляют определение другой.

./module.erl:5: Warning: this clause cannot match because a previous clause at line 4 always matches
Модуль содержит функцию, в которой после универсального (catch-all) условия определено конкретное (specific). Поэтому компилятор предупреждает, что до конкретного условия исполнение не дойдёт.

./module.erl:9: variable 'A' unsafe in 'case' (line 5)
Переменная, определённая в одной из веток выражения `case...of`, используется вне этого выражения. Такое поведение считается небезопасным. Если вам понадобились такие переменные, их лучше объявлять при помощи конструкции `MyVar = case ...of...`

Этот перечень покрывает большинство ошибок компиляции, с которыми вы можете столкнуться на текущий момент. Их не так уж и

много, поэтому зачастую сложнее всего найти ошибку, которая вызвала каскад ошибок, найденных в других функциях. Ошибки, которые выдаёт компилятор, лучше всего исправлять в том порядке, в котором они указаны в списке, чтобы не начать исправлять ошибки, которые могут ими и не являться. Иногда можно наткнуться и на другие сообщения, которые в список не попали. Если вы с таким столкнулись, то напишите мне письмо, и я как можно быстрее добавлю это сообщение в список вместе с комментариями.

9.3 Нет, это у ТЕБЯ ложная логика!

Логические ошибки искать и отлаживать сложнее всего. Чаще всего эти ошибки делает сам программист: ветвления и условия (например if-ы и case-ы), в которых не учитываются все случаи, употребление умножения вместо деления и т.д. Такие ошибки не приводят к аварийному завершению программы, но из-за них программа может просто выдать неверные данные или работать незапланированным образом.

С такими ошибками вам, скорее всего, придётся справляться самостоятельно, но в Erlang есть много средств, которые придут вам на помощь. В их числе тестовые фреймворки, TypEr и Dialyzer (которые описывались в 6.4 главе о типах), ?? отладчик, ?? модуль трассировки и т.д. Лучшая защита от таких ошибок – это тестирование. В карьере любого программиста, к сожалению, таких ошибок хватит на пару дюжин книг, поэтому я не буду тратить на них много времени. Легче сконцентрироваться на тех ошибках, которые приводят к аварийному завершению, так как момент их появления ясен, и они не всплывут на поверхность через 50 уровней вложенности. Это соображение, как раз, и служит источником мантры: «пусть процесс падает», о которой я уже несколько раз упоминал.

9.4 Ошибки времени исполнения

Ошибки времени исполнения (run-time errors) весьма разрушительно влияют на ваш код. Они приводят к аварийной ситуации. Хотя в Erlang и существуют методы их контроля, но никогда не бывает лишним

умение эти ошибки различать. Поэтому я составил небольшой список таких ошибок с объяснением и примерами кода, который может их вызвать.

function_clause

```
1> lists:sort([3,2,1]).
[1,2,3]
2> lists:sort(fffffff).
** exception error: no function clause matching lists:sort(
    fffffff)
```

Все охранные выражения завершились неудачей, либо для функции не сработал ни один из шаблонов для сопоставления с образцом.

case_clause

```
3> case "Unexpected Value" of
3>     expected_value -> ok;
3>     other_expected_value -> 'also ok'
3> end.
** exception error: no case clause matching "Unexpected Value"
```

Похоже, что кто-то забыл указать шаблон в выражении `case`, передал данные не того типа, или забыл задать условие для выбора по умолчанию (catch-all clause)!

if_clause

```
4> if 2 > 4 -> ok;
4>    0 > 1 -> ok
4> end.
** exception error: no true branch found when evaluating an if
    expression
```

Это сообщение очень похоже на ошибки `case_clause`. Не получается найти ветку, которая принимает значение `true`. Скорее всего, необходимо убедиться, что обрабатываются все возможные случаи, либо добавить вариант `true` для выбора по умолчанию.

badmatch

```
5> [X,Y] = {4,5}.
** exception error: no match of right hand side value {4,5}
```

Такие ошибки возникают в случае, когда не удаётся провести операцию сопоставления с образцом. Скорее всего, вы пытаетесь провести невозможное сопоставление, пытаетесь связать переменную со значением во второй раз, или по обе стороны оператора `=` находятся неравные

значения (что, в общем-то и приводит к тому, что операция связывания завершается неудачей!). Заметьте, что иногда эта ошибка возникает, потому что по мнению программиста переменная вида *_MyVar* означает то же самое, что и `_`. Переменные, имя которых начинается со знака подчёркивания – это обычные переменные. Единственное их отличие в том, что компилятор не генерирует предупреждение, если эти переменные не используются после объявления. Их можно связать со значением лишь один раз.

badarg

```
6> erlang:binary_to_list("heh, already a list").
** exception error: bad argument
     in function  binary_to_list/1
        called as binary_to_list("heh, already a list")
```

Эта ошибка похожа на `function_clause` тем, что она сообщает о вызове функции с некорректными аргументами. Главное отличие в том, что ошибка генерируется программистом, который проверяет аргументы в теле функции, а не в охранных выражениях (стражах). Чуть позже в этой главе я покажу как генерировать такие ошибки.

undef

```
7> lists:random([1,2,3]).
** exception error: undefined function lists:random/1
```

Ошибка генерируется, когда вы пытаетесь вызвать несуществующую функцию. Убедитесь, что функция экспортируется из модуля с правильной арностью (если вы вызываете её вне модуля), и перепроверьте правильность написания имени функции и модуля. Ещё одной причиной получения этого сообщения может служить то, что модуль находится вне пути поиска Erlang. По умолчанию поиск происходит в текущей директории. Добавлять пути можно при помощи функции `code:add_patha/1` или `code:add_pathz/1`. Если ничего из перечисленного не помогает, то убедитесь, что модуль был скомпилирован!

badarith

```
8> 5 + llama.
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as 5 + llama
```

Это сообщение появляется, когда вы пытаетесь выполнить несуществующее арифметическое действие. Например, делите на ноль или

пытаетесь выполнять арифметические операции между атомами и числами.

badfun

```
9> hhfuns:add(one,two).
** exception error: bad function one
in function  hhfuns:add/2
```

Чаще всего причиной возникновения этой ошибки становится попытка использовать переменные в качестве функций, но при этом переменные функций не содержат. В приведённом выше примере я использую функцию `hhfuns` из предыдущей главы 8.1 и при этом передаю в качестве параметров-функций пару атомов. Такой код не сработает, и будет выброшена ошибка `badfun`.

badarity

```
10> F = fun(_) -> ok end.
#Fun<erl_eval.6.13229925>
11> F(a,b).
** exception error: interpreted function with arity 1 called with
     two arguments
```

Ошибка `badarity` – это частный случай ошибки `badfun`. Она происходит, когда вы используете функции высшего порядка, которым передаёте больше (или меньше) аргументов, чем они требуют.

system_limit

Есть много причин, по которым может быть сгенерирована ошибка `system_limit`: слишком много процессов (мы до них ещё доберёмся), слишком длинные атомы, у функции слишком много аргументов, количество атомов слишком велико, слишком много подсоединённых узлов и т.д. Полный список можно прочитать в Erlang Efficiency Guide в разделе о системных ограничениях (system limits). Примите к сведению: некоторые из этих ошибок настолько серьёзны, что могут привести к аварии всей виртуальной машины.

9.5 Вызываем исключения

Чтобы следить за исполнением кода и защититься от логических ошибок, иногда полезно провоцировать аварийную остановку во время исполнения кода, чтобы как можно раньше выявить возможные проблемы. В Erlang существуют три вида исключений: *ошибки (errors)*, *броски*

(throws) и *завершения (exits)*. Все они используются в разных случаях (ну или почти в разных):

9.5.1 Ошибки

Вызов `erlang:error(Reason)` завершит исполнение в текущем процесе и, после того как исключение будет поймано, предоставит трассировку стека и список аргументов для последних вызовов функций. Это тот самый вид исключений, который вызывает ошибки времени исполнения, о которых я упоминал выше.

При помощи ошибок (errors) можно останавливать исполнение, в случае когда вызываемый код не сможет сам справиться с создавшейся ситуацией. Если вам была возвращена ошибка `if_clause`, что вы сделаете? Измените код, перекомпилируете его – на этом список доступных вам действий заканчивается (ну, ещё можно отобразить красивое сообщение об ошибке). Примером того, где не следует использовать ошибки, может служить наш модуль для работы с деревьями из главы о рекурсии 7. Поиск по дереву, реализованный в модуле, не всегда в состоянии найти в дереве заданный ключ. В такой ситуации имеет смысл ожидать, что с неизвестным результатом разберётся сам пользователь. Он сможет использовать значение по умолчанию, удалить дерево, вставить новое значение после проверки и т.д. В этой ситуации вместо генерирования ошибки лучше было бы возвратить кортеж `{ok, Value}`, или атом `undefined`.

Но область применения ошибок не ограничивается лишь этими примерами. Можно также определить собственный вид ошибок:

```
1> erlang:error(badarith).
** exception error: bad argument in an arithmetic expression
2> erlang:error(custom_error).
** exception error: custom_error
```

В этом примере оболочка Erlang не распознала `custom_error` и не вывела сообщение вида “bad argument in ...”, но тем не менее эту ошибку можно использовать и обрабатывать так же как обычно (скоро мы увидим как это делать).

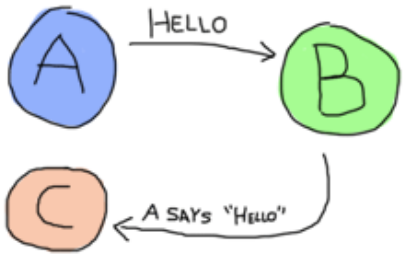
9.5.2 Завершения

Существует два вида завершений: «внутренние» (internal) и «внешние» (external). Внутренние завершения выполняются посредством вызова функции `exit/1` и приводят к остановке текущего процесса. Внешние завершения выполняются функцией `exit/2` и имеют отношение к множеству процессов, принадлежащих к параллельной (concurrent) части Erlang; поэтому здесь мы сосредоточимся на внутренних завершениях, а к внешним обратимся чуть позже.

Внутренние завершения очень похожи на ошибки (errors). На самом деле, раньше они никак не различались и существовала лишь одна функция `exit/1`. Использовали их в приблизительно одинаковых случаях. Так как же определить, что использовать в конкретной ситуации? Выбор неочевиден. Чтобы понять, когда нужно использовать ту или иную конструкцию, ничего не остаётся кроме как начать издалека поглядывать на концепции акторов и процессов.

Во введении я сравнивал процессы с людьми, которые общаются посредством почты. К этой аналогии, в общем-то, добавить нечего, поэтому перейду к диаграммам и кружкам.

Процессы на этой картинке могут посылать друг другу сообщения. Процесс также может ожидать прихода каких-либо сообщений. Можно выбирать, какие сообщения нужно ждать, какие отклонять, а какие игнорировать, через какой период времени прекращать ожидание и т.д.



Эти основные концепции позволяют создателям Erlang использовать особенный вид сообщений, при помощи которых между процессами передаются исключения. Они работают как своего рода «последний

вздох» процесса; их посылают прямо перед тем, как процесс умирает, и его код перестаёт выполняться. Другие процессы, которые ожидали этот вид сообщений, смогут узнать об этом событии и распорядиться этим знанием как угодно. Сделать запись в журнале, перезапустить умерший процесс и т.д.

Ну а теперь, когда мы имеем представление об этой концепции, нам будет проще понять разницу между `erlang:error/1` и `exit/1`. Обе функции можно использовать очень похожим способом, но настоящая разница заключается в нашем намерении. После получения такой ошибки можно решить, была ли это «просто» ошибка или ошибка из-за которой стоит убить текущий процесс. Это соображение подкрепляется тем, что функция `erlang:error/1` возвращает трассировку стека, а `exit/1` этого не делает. Если бы у текущей функции был длинный стек вызовов или много аргументов, то отсылка сообщений о её завершении каждому процессу, который их ожидает, означала бы, что эти данные пришлось бы копировать. В некоторых случаях это может стать непрактичным.

9.5.3 Броски

Броски – это класс исключений, используемых в случаях, которые должен обрабатывать программист. По сравнению с завершениями или ошибками, броски не имеют коннотации «роняй процесс!». Они больше относятся к управлению логикой программы. Так как вы используете броски в ожидании, что о них позаботится программист, то не лишним будет задокументировать этот факт в модуле, который их содержит.

Синтаксис бросков следующий:

```
1> throw(permission_denied).
** exception throw: permission_denied
```

`permission_denied` можно заменить чем угодно (даже сообщением `'всё в порядке'`, но этим вы вряд ли кому-либо поможете, а вот друзей с таким кодом вы точно не приобретёте).

Броски также можно использовать для нелокального возврата из глубокой рекурсии. Можно привести модуль `ssl` в качестве примера. Он использует функцию `throw/1` для того, чтобы вернуть кортежи `{error, Reason}` вызывающей функции. А она, в свою очередь, просто возвращает кортеж

пользователю. Благодаря этому механизму разработчик может писать код, который рассматривает только благополучные варианты развития событий, а все исключения обрабатывает лишь в одной вышестоящей функции.

Ещё одним примером может послужить модуль для работы с массивами, в котором есть функция поиска. В случае, если нужный элемент не был найден, эта функция возвращает значение по умолчанию, предоставленное пользователем. Когда элемент не удаётся найти, в исключении бросается значение `default`, которое обрабатывает вышестоящая функция и заменяет на пользовательское значение по умолчанию. Это избавляет создателя модуля от необходимости передавать значение по умолчанию в каждую функцию алгоритма поиска и позволяет сконцентрироваться лишь на благополучных исходах.

Чтобы облегчить отладку кода, старайтесь сосредоточить все случаи использования бросков для нелокальных возвратов в одном модуле. К тому же, это позволит изменять внутреннее устройство модуля без необходимости менять его интерфейс.

9.6 Расправляемся с исключениями

Я уже много раз упоминал о том, что бросками, ошибками и завершениями можно управлять. Для этого используют выражения `try ... catch`.

При помощи `try ... catch` можно исполнить какое-либо выражение, и обработать как случай успешного выполнения кода, так и возможные ошибки. Для таких выражений используется следующий синтаксис:

```
try Expression of
    SuccessfulPattern1 [Guards] ->
        Expression1;
    SuccessfulPattern2 [Guards] ->
    Expression2
catch
    TypeOfError:ExceptionPattern1 ->
        Expression3;
    TypeOfError:ExceptionPattern2 ->
        Expression4
end.
```

О *Выражении*, которое находится между `try` и `of` говорят, что оно *защищено*. Это означает, что будет поймано любое исключение, вызванное этим выражением. Шаблоны и выражения, которые используются в `try ... of` и `catch`

ведут себя так же как и в `case ... of`. Также в `catch` можно заменить *TypeOfError* на `error`, `throw` либо `exit`, для каждого соответствующего типа исключений, которые мы рассмотрели в этой главе. Если не указывать тип исключения, то по умолчанию используется тип `throw`. Попробуем применить наши знания на практике.

Для начала создадим модуль под названием `exceptions`. Ничего сложного выдумывать не будем:

```
-module(exceptions).
-compile(export_all).

throws(F) ->
    try F() of
        _ -> ok
    catch
        Throw -> {throw, caught, Throw}
    end.
```

Давайте скомпилируем модуль и опробуем его на разных видах исключений:

```
1> c(exceptions).
{ok,exceptions}
2> exceptions:throws(fun() -> throw(thrown) end).
{throw,caught,thrown}
3> exceptions:throws(fun() -> erlang:error(pang) end).
** exception error: pang
```

Как видите, `try ... catch` принимает лишь броски (throws). Как я и говорил ранее, без явного указания типа исключения подразумевается, что используются броски. А теперь запишем функции с catch-условием для каждого типа:

```
errors(F) ->
    try F() of
        _ -> ok
    catch
        error:Error -> {error, caught, Error}
    end.

exits(F) ->
    try F() of
        _ -> ok
    catch
        exit:Exit -> {exit, caught, Exit}
    end.
```

И исполним их:

```
4> c(exceptions).
{ok,exceptions}
5> exceptions:errors(fun() -> erlang:error("Die!") end).
{error,caught,"Die!"}
6> exceptions:exits(fun() -> exit(goodbye) end).
{exit,caught,goodbye}
```

Следующий пример показывает как можно скомбинировать все виды исключений в одном `try ... catch`. Сначала объявим функцию, которая будет генерировать все нужные нам исключения:

```
sword(1) -> throw(slice);
sword(2) -> erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

black_knight(Attack) when is_function(Attack, 0) ->
    try Attack() of
        _ -> "None shall pass."
    catch
        throw:slice -> "It is but a scratch.";
        error:cut_arm -> "I've had worse.";
        exit:cut_leg -> "Come on you pansy!";
        _:_ -> "Just a flesh wound."
    end.
```

В этом примере используется встроенная функция `is_function/2`, при помощи которой проверяется, что переменная *Attack* – это функция арности 0. А теперь добавим ещё одну:

```
talk() -> "blah blah".
```

А теперь кое-что совершенно иное (MP):

```
7> c(exceptions).
{ok,exceptions}
8> exceptions:talk().
"blah blah"
9> exceptions:black_knight(fun exceptions:talk/0).
"None shall pass."
10> exceptions:black_knight(fun() -> exceptions:sword(1) end).
"It is but a scratch."
11> exceptions:black_knight(fun() -> exceptions:sword(2) end).
"I've had worse."
12> exceptions:black_knight(fun() -> exceptions:sword(3) end).
"Come on you pansy!"
13> exceptions:black_knight(fun() -> exceptions:sword(4) end).
"Just a flesh wound."
```

```
14> exceptions:black_knight(fun() -> exceptions:sword(5) end).
"Just a flesh wound."
```

Выражение из 9-й строки демонстрирует нормальное поведение чёрного рыцаря для случая, когда функция выполняется успешно. В каждой последующей строчке показано сопоставление с образцом для исключений согласно их класса (бросок, ошибка, завершение), и причина их возникновения (`slice`, `cut_arm`, `cut_leg`).

В строках 13 и 14 показано выражение, которое используется для ловли исключений, когда все предыдущие выражения не сработали. Шаблон `_:_` используется в случае, когда вы хотите словить любое исключение какого угодно типа. На практике следует использовать этот шаблон с осторожностью: старайтесь защитить ваш код от того, с чем можете справиться, но не более того. Чтобы справиться со всем остальным, в Erlang есть другие средства.

После `try ... catch` можно добавить дополнительное выражение, которое будет исполнено в любом случае. Оно эквивалентно блоку 'finally', который используется во многих других языках:

```
try Expr of
    Pattern -> Expr1
catch
    Type:Exception -> Expr2
after % this always gets executed
    Expr3
end
```

Выражения внутри `after` будут гарантированно исполнены независимо от того, возникли ошибки или нет. Стоит отметить, что конструкция `after` не возвращает результат. Поэтому её можно использовать лишь для исполнения кода с побочными эффектами. Каноническим примером использования этой конструкции может служит ситуация, когда вы хотите убедиться, что файл, из которого вы читаете данные, будет закрыт независимо от того, возникали исключения или нет.

Теперь мы знаем как управлять в Erlang тремя классами исключений при помощи блоков catch. Но я кое о чём умолчал! Между `try` и of можно поместить не одно выражение, а несколько!

```
whoa() ->
```

```
    try
        talk(),
        _Knight = "None shall Pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    of
        tequila -> "hey this worked!"
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

При вызове `exceptions:whoa()` мы, очевидно, получим `{caught, throw, up}` из-за `throw(up)`. Короче говоря, да, между `try` и `of` можно помещать несколько выражений...

В `exceptions:whoa/0` я попытался показать (а вы, вероятно этого не заметили), что когда мы используем много выражений, нам не всегда важен возвращаемый результат. Поэтому часть `of` становится чуть-чуть бесполезной. Ну и прекрасно, ведь её можно просто убрать:

```
im_impressed() ->
    try
        talk(),
        _Knight = "None shall Pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

Так-то лучше!

Замечание: важно понимать, что защищённая от исключений часть кода не может использовать хвостовую рекурсию. Виртуальная машина должна постоянно держать ссылку на этот код, на случай если возникнет исключение.
Так как в конструкции `try ... catch` без части `of` нет ничего кроме защищённой части, то вызов рекурсивной функции из этой области кода может быть небезопасным для программы, которая должна находиться в запущенном состоянии долгое время (такие программы, как раз – ниша Erlang). Через определённое количество итераций на вашей машине закончится память, или ваша программа начнёт замедлятся без явных на то причин. Если вы поместите рекурсивные вызовы между `of` и `catch`, то они будут происходить вне защищённой части, и для них будет выполняться оптимизация последнего вызова.
Некоторые люди используют по умолчанию `try ... of ... catch` вместо `try ... catch`, чтобы избежать неожиданных ошибок такого типа. Такая осторожность излишня лишь в коде, который явно не содержит рекурсивные вызовы, и результаты выполнения которого не будут больше нигде использованы. Ну, вы и сами наверняка можете решить, как поступать в такой ситуации!

9.7 Но это ещё не всё!

Казалось бы, этих средств и так достаточно, чтобы быть наравне с другими языками программирования, но в Erlang есть ещё одна структура для управления ошибками. Эта структура определяется ключевым словом `catch` и, по сути, вдобавок к верным результатам может возвращать все типы исключений. Она может показаться несколько странной, так как отображает исключения немного иначе:

```
1> catch throw(whoa).
whoa
2> catch exit(die).
{'EXIT',die}
3> catch 1/0.
{'EXIT',{badarith,[{erlang,'/',[1,0]},
            {erl_eval,do_apply,5},
            {erl_eval,expr,5},
            {shell,exprs,6},
            {shell,eval_exprs,6},
{shell,eval_loop,3}]}}
4> catch 2+2.
4
```

В этом примере мы видим, что отображение бросков осталось прежним, а завершения и ошибки представлены в виде кортежей `{'EXIT', Reason}`. Это вызвано тем, что ошибки были вмонтированы в язык после завершений (для них оставили схожее представление для обратной совместимости).

Вот как следует читать эту трассировку стека:

```
5> catch doesnt:exist(a,4).
{'EXIT',{undef,[{doesnt,exist,[a,4]},
                {erl_eval,do_apply,5},
                {erl_eval,expr,5},
                {shell,exprs,6},
                {shell,eval_exprs,6},
{shell,eval_loop,3}]}}
```

- Тип этой ошибки – `undef`. Это означает, что вызванная функция не определена (см. список в начале главы).
- Сразу за типом ошибки следует трассировка стека.
- Кортеж в самом начале трассировки обозначает последнюю функцию, которая должна была быть вызвана (`{Module, Function, Arguments}`). Это та самая неопределённая функция.
- Кортежи, следующие за первым – это функции, которые вызывались перед тем как произошла ошибка. Здесь они принимают форму кортежа `{Module, Function, Arity}`.
- В общем-то, вот и всё.

Также можно вручную получить трассировку стека, вызывая функцию `erlang:get_stacktrace/0` внутри аварийного процесса.

Часто можно увидеть `catch`, записанную следующим образом (мы всё ещё рассматриваем exceptions.erl):

```
catcher(X,Y) ->
    case catch X/Y of
        {'EXIT', {badarith,_}} -> "uh oh";
        N -> N
    end.
```

И, как мы и ожидали:

```
6> c(exceptions).
{ok,exceptions}
7> exceptions:catcher(3,3).
```

```
1.0
8> exceptions:catcher(6,3).
2.0
9> exceptions:catcher(6,0).
"uh oh"
```

На первый взгляд этот способ перехвата исключений выглядит компактным и простым, но у `catch` есть несколько особенностей. Первая – это порядок выполнения операций:

```
10> X = catch 4+2.
* 1: syntax error before: 'catch'
10> X = (catch 4+2).
6
```

То, что в примере необходимы скобки, не вполне согласуется с интуицией, если учесть, что большинство выражений в скобки заключать не нужно. Ещё одна проблема заключается в том, что невозможно отличить представление исключения от настоящего исключения:

```
11> catch erlang:boat().
{'EXIT',{undef,[{erlang,boat,[]},
                {erl_eval,do_apply,5},
                {erl_eval,expr,5},
                {shell,exprs,6},
                {shell,eval_exprs,6},
{shell,eval_loop,3}]}}
12> catch exit({undef, [{erlang,boat,[]}, {erl_eval,do_apply,5},
    {erl_eval,expr,5}, {shell,exprs,6}, {shell,eval_exprs,6}, {
    shell,eval_loop,3}]}).
{'EXIT',{undef,[{erlang,boat,[]},
                {erl_eval,do_apply,5},
                {erl_eval,expr,5},
                {shell,exprs,6},
                {shell,eval_exprs,6},
{shell,eval_loop,3}]}}
```

И ошибку невозможно отличить от, собственно, самого завершения. Чтобы сгенерировать исключение, приведённое выше, можно было бы также использовать функцию `throw/1`. Надо сказать, что использование `throw/1` в `catch` может также представлять собой проблему в другом контексте:

```
one_or_two(1) -> return;
one_or_two(2) -> throw(return).
```

А вот и страшная проблема:

```
13> c(exceptions).
```

```
{ok,exceptions}
14> catch exceptions:one_or_two(1).
return
15> catch exceptions:one_or_two(2).
return
```

Так как мы находимся за `catch`, то никогда не знаем наверняка, бросила ли функция исключение или вернула результат! На практике это случается не очень часто, но эта особенность повлекла за собой появление конструкции `try ...catch` в релизе R10B.

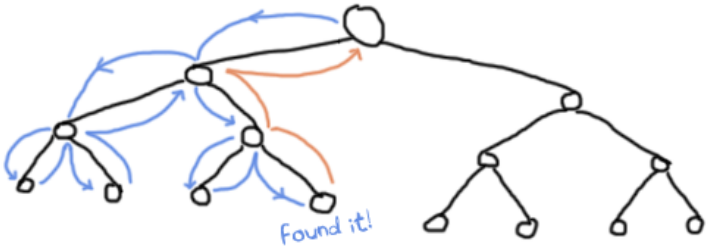
9.8 Попробуй try в дереве

Чтобы применить на практике наши знания об исключениях, мы проделаем небольшое упражнение, для которого нам придётся раскопать модуль `tree`. Мы добавим в него функцию, которая позволит нам передвигаться по дереву, и выяснять, присутствует ли в нём заданное значение. Дерево упорядочено по ключам, а в этом случае с ключами мы дела не имеем. Следовательно, чтобы найти значение, нам придётся пройти по всему дереву.

Проход будет выполняться приблизительно так же как и в функции `tree:lookup/2`, но в этот раз мы всегда будем выполнять поиск как в левой, так и в правой ветке. Чтобы написать функцию, необходимо лишь помнить, что узел дерева – это либо `{node, {Key, Value, NodeLeft, NodeRight}}`, либо `{node, 'nil'}` если узел пуст. Вооружившись этими знаниями, мы можем написать базовую реализацию, не используя исключения:

```
%% looks for a given value 'Val' in the tree.
has_value(_, {node, 'nil'}) ->
    false;
has_value(Val, {node, {_, Val, _, _}}) ->
    true;
has_value(Val, {node, {_, _, Left, Right}}) ->
    case has_value(Val, Left) of
        true -> true;
        false -> has_value(Val, Right)
    end.
```

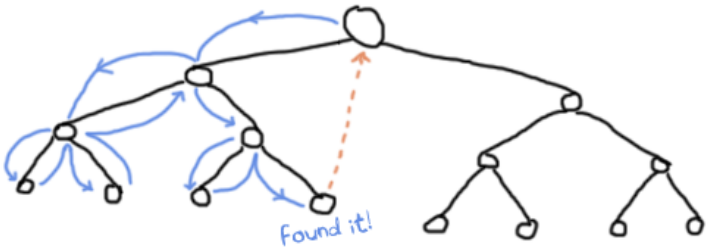
Проблема этой реализации в том, что для каждого узла, в котором мы делаем ветвление, нам необходимо проверять результат предыдущей ветки:



Этот факт немного раздражает. Но при помощи бросков мы можем прийти к коду, который требует меньше операций сравнения:

```
has_value(Val, Tree) ->
    try has_value1(Val, Tree) of
        false -> false
    catch
        true -> true
    end.

has_value1(_, {node, 'nil'}) ->
    false;
has_value1(Val, {node, {_, Val, _, _}}) ->
    throw(true);
has_value1(Val, {node, {_, _, Left, Right}}) ->
    has_value1(Val, Left),
    has_value1(Val, Right).
```

Код, приведённый выше, исполняется приблизительно так же как и его предыдущая версия, за исключением того, что теперь возвращаемое значение мы нигде проверять не будем. Нам оно совершенно не интересно. В этой версии об обнаружении значения в дереве сигнализирует только бросок. Когда значение найдено, обработка дерева прекращается, и исполнение переходит в верхний `catch`. В противном случае исполнение продолжается, пока не будет возвращён последний *false*, и вот что тогда видит пользователь:



Конечно же, такая реализация занимает больше строк кода, чем предыдущая. Применяя нелокальные возвраты с броском, можно вы-

играть в скорости и ясности кода, но этот выигрыш будет зависеть от действий, которые выполняет ваша программа. Этот пример – простое сравнение и ничего особенного собой не представляет, а вот показанный в нём приём имеет смысл применять при работе с более сложными операциями и структурами данных.

Ну а теперь мы, наверное, готовы решать настоящие проблемы при помощи последовательного кода на Erlang.

Глава 10

Решаем задачи в функциональном стиле

Похоже, что мы уже выпили достаточно эрлангового сока, чтобы создать что-нибудь полезное. В этой главе не будет нового материала. Я просто покажу как применять элементы увиденного ранее. Задачи были взяты из книги Miran-a Learn You a Haskell. Для всех примеров я использовал аналогичный способ решения, чтобы любопытный читатель смог сравнивать код на Erlang и Haskell как ему заблагорассудится. Проведя такое сравнение, вы, наверняка, придёте к заключению, что для двух языков с очень разными синтаксисами, решения очень похожи друг на друга. Это сходство обусловлено тем, что изученные концепции функционального программирования можно сравнительно легко переносить на другие функциональные языки.

10.1 Калькулятор в обратной польской записи

Большинство людей обучены записи арифметических выражений, согласно которой оператор помещают между чисел (`(2 + 2) / 5`). Так вводятся математические выражения в большинстве калькуляторов, и, скорее всего, такую запись вас обучили использовать при счёте на школьных уроках. У этой записи есть недостаток: вам необходимо знать порядок вычисления операторов. Умножение и деление считается более важным (имеет более высокий приоритет), чем сложение и вычитание.

Существует ещё один способ записи, который называется *префиксной записью* или *польской записью*, в которой оператор записывается перед операндами. В этой записи выражение `(2 + 2) / 5` примет вид

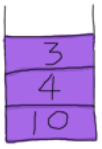
`(/ (+ 2 2) 5)`. Если мы решим, что `+` и `/` должны всегда принимать два аргумента, то `(/ (+ 2 2) 5)` можно просто записать как `/ + 2 2 5`.

Однако, мы сосредоточимся на *обратной польской записи* (или просто ОПЗ), которая противоположна префиксной записи: в ней оператор следует за операндами. Тот же самый пример, приведённый выше, в ОПЗ будет записан как `2 2 + 5 /`. Можно привести и другие примеры: `9 * 5 + 7` и `10 * 2 * (3 + 5) / 2` транслируются соответственно в `9 5 * 7` и `10 2 * 3 4 + * 2 /`. Эта запись очень широко использовалась в ранних моделях калькуляторов, так как экономно использует память. А некоторые люди до сих пор таскают с собой калькуляторы с ОПЗ. Именно такой калькулятор мы и напишем.

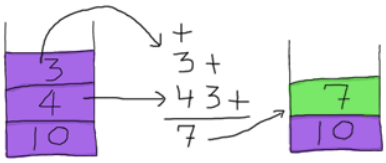
Для начала, неплохо было бы понимать, как читаются выражения ОПЗ. Первый способ: по одному находим операторы и, учитывая арность, группируем их с операндами.

$$
\begin{aligned}
&10\ 4\ 3\ +\ 2\ *\ - \\
&10\ (4\ 3\ +)\ 2\ *\ - \\
&10\ ((4\ 3\ +)\ 2\ *) - \\
&(10\ ((4\ 3\ +)\ 2\ *) -) \\
&(10\ (7\ 2\ *) -) \\
&(10\ 14\ -) \\
&-\ 4
\end{aligned}
$$

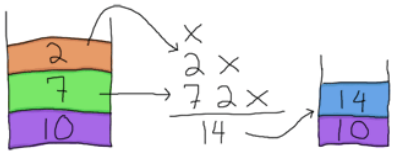
Однако, в компьютере или калькуляторе намного проще создать *стек* всех операндов в порядке их расположения. Возьмём математическое выражение `10 4 3 + 2 * -`. Первый операнд 10. Мы добавляем его в стек. Затем идёт 4. Его мы тоже кладём на вершину стека. Третьим идёт число 3. Мы помещаем в стек и его. Теперь стек принял следующий вид:



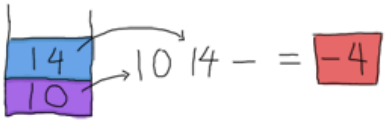
Следующий символ это `+`. Он представляет собой функцию арности 2. Чтобы ею воспользоваться, нам необходимо передать ей два операнда, которые мы извлечём из стека:



Полученную цифру 7 мы загоняем в верхушку стека (фу, мы же не хотим, чтобы эти грязные числа шатались повсюду!) Теперь стек содержит [7, 10], и от первоначального выражения осталось лишь `2 * -`. Мы берём 2 и помещаем его в вершину стека. Затем мы видим операцию `*`, которой для работы необходимо передать два операнда. И снова мы извлекаем их из стека:



И помещаем 14 в вершину стека. Остаётся лишь `-`, которому необходимо передать два операнда. Невероятная удача! В стеке как раз осталось два операнда. Пустим их в дело!



Ну вот мы и получили результат. Этот стек-ориентированный подход сравнительно надёжен. Тот факт, что для начала вычислений требуется разбирать мало данных, объясняет почему этот способ годится даже для старых калькуляторов. ОПЗ стоит использовать и по другим причинам, но их обсуждение выходит за рамки этого руководства.. Заинтересовавшимся лучше обратиться к статье на Wikipedia.

Как только мы справились со сложными моментами, записать решение на Erlang становится достаточно просто. Оказывается, самое сложное – определить шаги для получения конечного результата. А это мы только что проделали. Неплохо. Откройте файл *calc.erl*.

Для начала озаботимся представлением математических выражений. Чтобы упростить задачу мы, наверное, представим их в виде строки: `"10 4 3 + 2 * -"`. В этой строке между термами есть пробелы, которые не

оговорены в нашем решении, но для работы простого лексического анализатора они необходимы. После обработки входящей строки анализатором, мы ожидаем получить список термов вида `["10","4","3","+","2","*","-"]`. Оказывается, функция `string:tokens/2` делает именно то, что нам нужно:

```
1> string:tokens("10 4 3 + 2 * -", " ").
["10","4","3","+","2","*","-"]
```

Такое представление выражения нам подойдёт. Следующим шагом мы должны определить стек. Как мы это сделаем? Возможно, вы заметили, что поведение списков в Erlang очень похоже на стек. Оператор (`|`) в шаблоне `[Head|Tail]` ведёт себя так же как и операция добавления *Head* в вершину стека (которой в данном случае является *Tail*). Список в роли стека нам вполне подойдёт.

Чтобы прочитать выражение, нам необходимо просто повторить те же действия, которые мы выполняли при решении задачи вручную. Последовательно считываем каждое значение в выражении. Если мы прочитали число – кладём его на стек. Если функцию – извлекаем все необходимые ей значения из стека, а результат вычисления возвращаем обратно в стек. Если обобщить задачу, то нам нужно один раз пройти в цикле по всему выражению и по ходу движения накапливать результаты. С этим прекрасно справится свёртка (fold)!

Теперь мы должны выяснить, как будет выглядеть функция, которую `lists:foldl/3` будет применять к каждому оператору и операнду в нашем выражении. Функция будет запускаться в свёртке, следовательно она должна принимать два аргумента: первый будет содержать элемент выражения, который будет обрабатывать функция, а вторым будет передаваться стек.

Начнём редактировать наш код в файле `calc.erl`. Напишем функцию, в которой будет происходить итерация, а также удаление пробелов из выражения:

```
-module(calc).
-export([rpn/1]).

rpn(L) when is_list(L) ->
    [Res] = lists:foldl(fun rpn/2, [], string:tokens(L, " ")),
    Res.
```

Следующим шагом реализуем функцию `rpn/2`. Обратите внимание: каждый оператор и операнд из выражения в конце концов попадает на вершину стека, а поэтому и конечный результат вычислений также

окажется в стеке. Мы извлекаем последнее значение из стека и возвращаем его пользователю. Именно поэтому в сопоставлении с образцом мы используем шаблон `[Res]` и возвращаем *Res*.

Хорошо, а теперь более сложный момент. Наша функция `rpn/2` должна обрабатывать стек для всех переданных ей значений. Заголовок функции, скорее всего, будет выглядеть как `rpn(Op,Stack)`, а возвращаемое значение примет вид `[NewVal|Stack]`. Для обычных чисел будет выполняться операция:

```
rpn(X, Stack) -> [read(X)|Stack].
```

Функция `read/1` конвертирует строку в целое число или число с плавающей точкой. К сожалению, в Erlang нет встроенной функции, которая делает и то и другое, поэтому мы создадим её сами:

```
read(N) ->
    case string:to_float(N) of
        {error,no_float} -> list_to_integer(N);
        {F,_} -> F
    end.
```

Где `string:to_float/1` производит конвертацию строк вида ”13.37” в их числовой эквивалент. Если значение числа с плавающей запятой определить не удаётся, функция возвращает `{error,no_float}`. После чего мы должны вызвать функцию `list_to_integer/1`.

А теперь снова вернёмся к `rpn/2`. Все найденные числа мы отправляем в стек. Но так как наше сопоставление с образцом захватывает любые значения (см. **??**), то кроме чисел в стек также будут попадать и операторы. Чтобы этого избежать, мы добавим для всех операторов конкретные шаблоны, которые будут предварять общий. Начнём со сложения:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn(X, Stack) -> [read(X)|Stack].
```

Очевидно, что каждый раз когда нам попадается строка `”+”`, мы извлекаем из стека два числа *(N1,N2)*, складываем их и возвращаем результат в стек. Именно такие действия мы выполняли, когда решали задачу вручную. Запустив программу, мы можем убедиться в её работоспособности:

```
1> c(calc).
{ok,calc}
2> calc:rpn("3 5 +").
```

```
8
3> calc:rpn("7 3 + 5 +").
15
```

Остальное – тривиально. Нужно лишь добавить такие же строки для всех оставшихся операторов:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn("-", [N1,N2|S]) -> [N2-N1|S];
rpn("*", [N1,N2|S]) -> [N2*N1|S];
rpn("/", [N1,N2|S]) -> [N2/N1|S];
rpn("^", [N1,N2|S]) -> [math:pow(N2,N1)|S];
rpn("ln", [N|S])    -> [math:log(N)|S];
rpn("log10", [N|S]) -> [math:log10(N)|S];
rpn(X, Stack) -> [read(X)|Stack].
```

Обратите внимание, что функции, которые принимают лишь один аргумент (например функция логарифмирования), должны извлекать из стека один элемент. Пусть читатель в качестве упражнения добавит функции 'sum' или 'prod', которые, соответственно, возвращают сумму и произведение всех считанных элементов. Если у вас возникнут затруднения, обратитесь к моей реализации этих функций в `calc.erl`.

Мы напишем несколько простых юнит-тестов, чтобы убедиться, что всё работает как положено. Оператор `=` в Erlang можно использовать как функцию *утверждения (assertion)*. Если после проверки утверждения обнаружены значения, которые ему не соответствуют – должен происходить сбой. Это как раз то, что нам нужно. Конечно, в Erlang есть и более совершенные тестовые фреймворки, например Common Test и EUnit. Мы поговорим о них позже, а сейчас нам хватит и `=`.

```
rpn_test() ->
    5 = rpn("2 3 +"),
    87 = rpn("90 3 -"),
    -4 = rpn("10 4 3 + 2 * -"),
    -2.0 = rpn("10 4 3 + 2 * - 2 /"),
    ok = try
        rpn("90 34 12 33 55 66 + * - +")
    catch
        error:{badmatch,[_|_]} -> ok
    end,
    4037 = rpn("90 34 12 33 55 66 + * - + -"),
    8.0 = rpn("2 3 ^"),
    true = math:sqrt(2) == rpn("2 0.5 ^"),
    true = math:log(2.7) == rpn("2.7 ln"),
    true = math:log10(2.7) == rpn("2.7 log10"),
    50 = rpn("10 10 10 20 sum"),
    10.0 = rpn("10 10 10 20 sum 5 /"),
```

```
    1000.0 = rpn("10 10 20 0.5 prod"),
    ok.
```

Тестовая функция пытается исполнить все операции. Тест считается пройденным, если не было возбуждено исключение. Четыре первых теста проверяют корректную работу арифметических функций. Пятый тест задаёт поведение, которое я пока ещё не обсуждал. Выражение `try ...catch` ожидает, что будет брошена ошибка badmatch, так как выражение невозможно вычислить:

```
90 34 12 33 55 66 + * - +
90 (34 (12 (33 (55 66 +) *) -) +)
```

Под конец выполнения функции `rpn/1` значения -3947 и 90 остаются в стеке, так как не хватает оператора, который бы произвёл действие над числом 90. Эта проблема имеет два решения: её можно проигнорировать и просто взять значение, которое находится на вершине стека (это будет последний вычисленный результат), или сгенерировать ошибку, которая сообщит о неверных арифметических действиях. Так как политика Erlang для таких случаев требует, чтобы мы позволили процессу упасть, то мы в этой ситуации сделаем выбор в пользу именно этого требования. Сбой происходит в шаблоне `[Res]` функции `rpn/1`. Шаблон проверяет, что в стеке остался лишь один элемент, и этот элемент – конечный результат вычислений.

Несколько тестов вида `true = FunctionCall1 == FunctionCall2` были добавлены, так как по левую сторону от `=` нельзя использовать вызов функции. Но они всё равно работают аналогично утверждениям, потому что мы сравниваем результат сравнения с *true*.

Также я добавил тесты для операторов sum и prod, чтобы вы смогли поупражняться в их реализации. Если все тесты прошли успешно, то вы должны увидеть следующее:

```
1> c(calc).
{ok,calc}
2> calc:rpn_test().
ok
3> calc:rpn("1 2 ^ 2 2 ^ 3 2 ^ 4 2 ^ sum 2 -").
28.0
```

Где число 28 действительно равно результату вычисления $sum(1^2 + 2^2 + 3^2 + 4^2) - 2$.

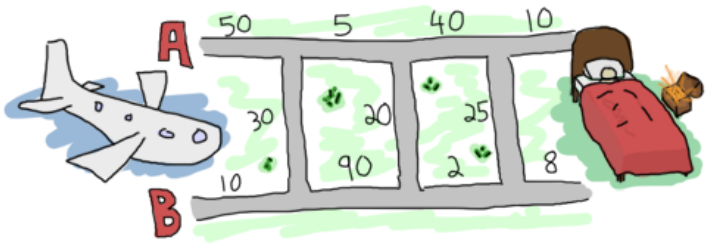
Наш калькулятор можно улучшить, добавив возбуждение исключений `badarith` в случае аварийного завершения из-за неизвестных операторов или из-за оставленных на стеке необработанных значений. Такое

исключение обозначает проблему чётче, чем ошибка `badmatch`, которую мы генерируем сейчас. Этим мы значительно облегчим отладку для пользователя модуля calc.

10.2 Из Хитроу в Лондон

Следующая задача также взята из Learn You a Haskell. Вы летите в самолёте, который через несколько часов должен приземлиться в аэропорте Хитроу. После приземления необходимо как можно быстрее добраться до Лондона. Ваш богатый дядя при смерти, и вы хотите первым предъявить права на его недвижимость.

Из Хитроу в Лондон проложены две дороги, которые сообщаются при помощи нескольких переулков. Некоторые части дорог и переулков, из-за скоростных ограничений и заторов, позволяют ехать медленнее, чем другие. Перед посадкой вы решаете найти оптимальный путь, ведущий к дому, и, тем самым, максимизировать свои шансы. Вот карта, которую вы нашли при помощи своего лэптопа:



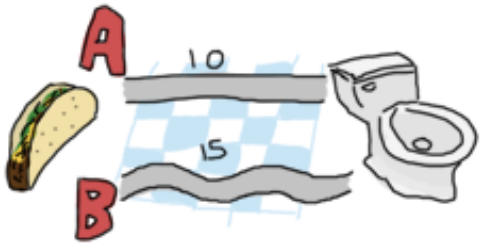
Так как после прочтения нескольких онлайн-книг вы стали ярым фанатом Erlang-а, то, конечно же, хотите решить эту задачу на вашем любимом языке. Чтобы облегчить работу с картой, вы сохраняете исходные данные в файле road.txt следующим образом:

```
50
10
30
5
90
20
40
2
25
10
8
0
```

Информация о дороге организована по шаблону: A1, B1, X1, A2, B2, X2, ..., An, Bn, Xn, где x – это один из переулков, соединяющих между собой части A и B. В качестве последнего сегмента x мы используем 0, так как в этот момент мы гарантированно будем находиться в пункте назначения. Данные можно организовать в кортежи по 3 элемента (тройки) вида {A,B,X}.

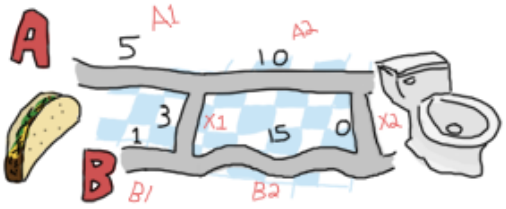
И тут вы понимаете, что если вы не знаете как решить эту задачу вручную, то нечего даже пытаться решить её на Erlang. Для анализа задачи будем использовать то, чему нас научила рекурсия.

Первым шагом мы пытаемся найти частный случай. Для нашей задачи это ситуация, когда осталось проанализировать лишь один кортеж. То есть, сделать выбор между *A*, *B* (и переулком *x*, который в данном случае бесполезен, так как мы находимся в пункте назначения):



Поэтому остаётся лишь определить, какой путь короче – путь A или путь B. Если вы хорошо изучили рекурсию, то знаете, что мы обязаны сходиться к частному случаю, то есть, на каждом шаге мы будем пытаться свести задачу к выбору между A и B.

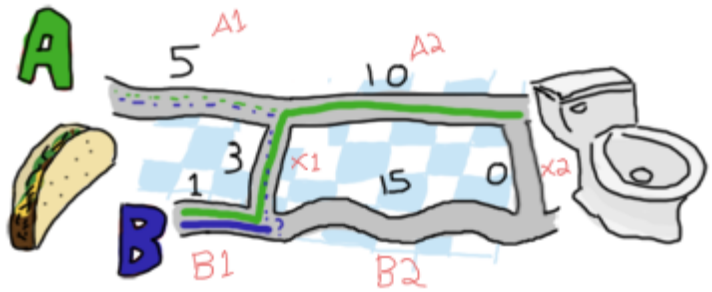
Расширим нашу карту и начнём заново:



О, это уже интереснее! Как мы можем перейти от тройки {5,1,3} к чёткому выбору между A и B? Определим, сколько существует вариантов для A. Чтобы добраться до пересечения *A1* и *A2* (я буду называть это место *точкой A1*), я могу поехать напрямую по дороге *A1* (5), либо проехать по *B1* (1), а затем по переулку *X1* (3). В этом случае первый вариант (5) длиннее второго (4). Кратчайший путь для варианта A это [B, X]. А какие есть варианты для B? Можно проехать по *A1* (5), затем по переулку *X1* (3), или сразу же избрать путь *B1* (1).

Хорошо! У нас есть путь длины 4 `[B, X]` до первого пересечения A, и путь длины 1 `[B]` до пересечения *B1* и *B2*. Теперь мы должны сделать выбор между второй точкой A (пересечение *A2* и конечной точки *X2*) и второй точкой B (пересечение *B2* и *X2*). Предлагаю сделать то же самое, что и раньше. Так как тексты здесь пишу я, то моё решение вам оспорить не удастся. Ну что ж, продолжим!

Все возможные пути для этого случая можно найти таким же способом, как и в предыдущей ситуации. До точки A мы можем добраться либо по пути *A2* из `[B, X]`, который даёт нам длину 14 (`14 = 4 + 10`, либо по *B2*, а затем *X2* из точки `[B]`, что даёт нам длину 16 (`16 = 1 + 15 + 0`). Очевидно, что путь `[B, X, A]` лучше пути `[B, B, X]`.



Также мы можем добраться до следующей точки B по пути *A2* из `[B, X]` и переулку *X2*, что даст нам длину 14 (`14 = 4 + 10 + 0`). Либо по дороге *B2* из *[B]*, которая даст нам путь длины 16 (`16 = 1 + 15`). Мы выберем первый вариант: `[B, X, A, X]`.

В конце у нас остаются два пути A и B. Длина обоих равна 14. Любой из этих маршрутов можно считать верным. Окончательный выбор всегда будет происходить между двумя маршрутами равной длины, если последний сегмент X имеет длину 0. Наше рекурсивное решение даёт уверенность, что на выходе мы всегда получаем кратчайший путь. Совсем неплохо, согласитесь.

Постепенно мы снабдили себя основными деталями, необходимыми для постройки рекурсивной функции. Можете, конечно, её реализовать, если хотите, но я пообещал, что самостоятельно рекурсивные функции мы будем писать очень редко. Здесь мы задействуем свёртку (fold).

Замечание: хоть я и показывал, что свёртки используются для списков и создаются с их помощью, но свёртки представляют собой более универсальную концепцию перебора элементов структуры данных при помощи аккумулятора. Поэтому свёртки можно реализовывать над деревьями, словарями (dictionaries), массивами, таблицами баз данных и т.д.
Во время экспериментов зачастую полезно использовать отображения (maps) и свёртки (folds). С их помощью можно впоследствии легко изменить структуру данных, с которой работает логика вашей программы.

Итак, на чём мы остановились? Ах, да! Мы подготовили файл, который будем использовать как вход для нашей программы. Для файловых манипуляций лучшим инструментом является файловый модуль. Он содержит множество функций, которые встречаются в большинстве языков программирования и используются для работы с файлами (установка прав доступа, перемещение файлов, переименование и удаление и т.д.)

Модуль также содержит стандартные функции для чтения и/или записи из/в файлы, такие как `file:open/2` и `file:close/1`, которые делают именно то, о чём сообщают их имена (открывают и закрывают файлы!) `file:read/2` извлекает из файла содержимое (в двоичном виде или в виде строки), `file:read_line/1` считывает единичную строку, `file:position/3` перемещает указатель в открытом файле в указанную позицию и т.д.

В модуле есть и несколько упрощённых функций, например `file:read_file/1` (открывает файл и читает его содержимое в двоичном виде), `file:consult/1` (открывает и разбирает файл на термы Erlang) или `file:pread/2` (меняет текущую позицию в файле, а потом считывает данные) и `pwrite/2` (меняет текущую позицию и записывает данные).

С таким богатым выбором мы легко найдём функцию, которая позволит считать наш файл road.txt. Так как нам известно, что описание дороги относительно невелико, то мы воспользуемся вызовом `file:read_file("road.txt").`:

```
1> {ok, Binary} = file:read_file("road.txt").
{ok,<<"50\r\n10\r\n30\r\n5\r\n90\r\n20\r\n40\r\n2\r\n25\r\n10\r\
    n8\r\n0\r\n">>}
2> S = string:tokens(binary_to_list(Binary), "\r\n\t ").
["50","10","30","5","90","20","40","2","25","10","8","0"]
```

Обратите внимание, что в данном случае я добавил пробел (`" "`) и символ табуляции (`"\t"`) в список валидных токенов, чтобы иметь возможность считывать файлы вида "50 10 30 5 90 20 40 2 25 10 8 0". После считывания этого списка, нам нужно преобразовать строки в целые чис-

ла. Применим способ, подобный тому, который мы использовали в нашем ОПЗ калькуляторе:

```
3> [list_to_integer(X) || X <- S].
[50,10,30,5,90,20,40,2,25,10,8,0]
```

Давайте создадим новый модуль, назовём его road.erl и запишем наши рассуждения в виде кода:

```
-module(road).
-compile(export_all).

main() ->
    File = "road.txt",
    {ok, Bin} = file:read_file(File),
    parse_map(Bin).

parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    [list_to_integer(X) || X <- string:tokens(Str,"\r\n\t ")].
```

Функция `main/0` отвечает за чтение содержимого файла и его передачу в функцию `parse_map/1`. Так как для чтения мы используем функцию `file:read_file/1`, то полученный результат будет представлен в виде двоичных данных. Поэтому я сделал так, чтобы функция `parse_map/1` проводила сопоставление и для списков, и для двоичных данных. В случае двоичных данных мы просто повторно вызываем функцию и передаём ей строку, преобразованную в список (наша функция для разбиения строк работает только со списками).

Следующим шагом разбора данных, описывающих карту, будет перегруппировка чисел в кортежи `{A,B,X}`, описанные ранее. К сожалению, простого универсального способа выбрать из списка по 3 элемента за раз не существует, а поэтому нам придётся использовать сопоставление с образцом в рекурсивной функции:

```
group_vals([], Acc) ->
    lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
    group_vals(Rest, [{A,B,X} | Acc]).
```

Эта функция оптимизируется в хвостовую рекурсию и ничего особенно сложного в ней не происходит. Её нужно будет вызвать из слегка модифицированной функции `parse_map/1`:

```
parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
```

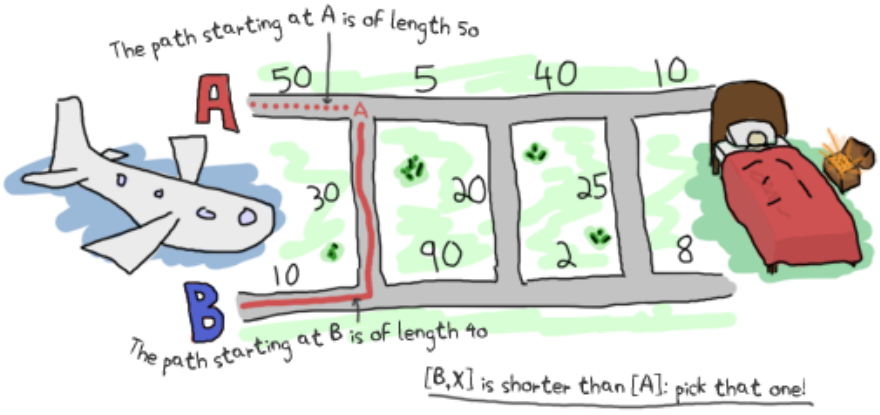
```
parse_map(Str) when is_list(Str) ->
    Values = [list_to_integer(X) || X <- string:tokens(Str,"\r\n\
        t ")],
    group_vals(Values, []).
```

Если мы попробуем скомпилировать всё вместе, то получим осмысленное описание дороги:

```
1> c(road).
{ok,road}
2> road:main().
[{50,10,30},{5,90,20},{40,2,25},{10,8,0}]
```

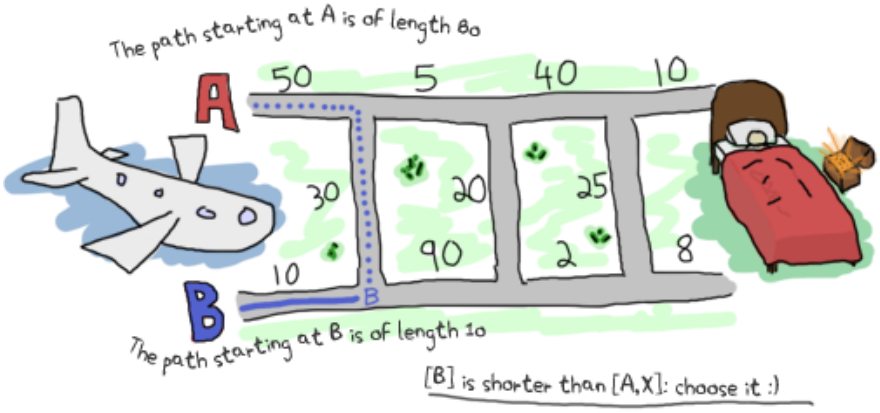
Ага, похоже всё верно. У нас появился ещё один элемент функции, которая потом попадёт в свёртку. Теперь для осуществления задуманного нам нужен хороший аккумулятор.

Для выбора аккумулятора я использую следующий метод: представляю, что я внутри запущенного алгоритма. Для этой конкретной задачи я представлю, что пытаюсь определить кратчайший путь для второй тройки (`{5,90,20}`). Чтобы определить лучший маршрут, мне нужно владеть результатом для предыдущей тройки. К счастью, способ решения этой задачи нам уже известен, так как для него нам не понадобится аккумулятор, и всю необходимую логику мы уже записали. Итак, для A:



И из этих двух путей выбираем кратчайший.

Для B задача имеет похожее решение:



Теперь нам известно, что лучший путь в A на текущий момент – это `[B,X]`. Нам также известно, что его длина равна 40. Оптимальный путь для B – просто `[B]` и его длина 10. Эту информацию мы можем использовать, чтобы найти следующий лучший маршрут для A и B, применяя похожие рассуждения, но при этом учитывая предыдущие решения. Нам также пригодится пройденный путь, чтобы мы его могли показывать пользователю. Необходимо найти два пути (один для A, второй для B) и две аккумулированные длины, поэтому аккумулятор можно выразить в виде кортежа `{{DistanceA, PathA}, {DistanceB, PathB}}`. Таким образом, каждая итерация свёртки имеет полный доступ к состоянию, и мы накапливаем результат, чтобы в конце показать его пользователю.

Теперь у нашей функции есть все необходимые параметры: тройки вида `{A,B,X}` и аккумулятор `{{DistanceA,PathA}, {DistanceB,PathB}}`.

Аккумулятор можно получить при помощи следующего кода:

```
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.
```

Переменной *OptA1* присваивается первый вариант для A (перебираются элементы *A*), переменной *OptA2* – второй вариант (проходим по *B*, затем по *X*). Для точки из B производим похожие манипуляции с переменными *OptB1* и *OptB2*. В результате возвращаем аккумулятор с полученным маршрутом.

Заметьте, что для хранения маршрутов я решил использовать представление `[{x,X}]` вместо `[x]`, чтобы пользователь имел возможность

узнать длину каждого сегмента. Стоит также обратить внимание на то, что я накапливаю пути в обратном порядке (`{x,X}` перед `{b,B}`). Это происходит из-за того, что свёртка использует хвостовую рекурсию: весь список разворачивается задом-наперёд, поэтому необходимо размещать последний пройденный элемент впереди остальных.

И, наконец, я использую функцию `erlang:min/2`, чтобы найти кратчайший путь. Применение этой функции сравнения к кортежам может показаться странным, но вспомните: каждый терм Erlang можно сравнивать с любым другим! Благодаря тому, что первым элементом кортежа является длина, мы можем их сортировать.

Осталось вставить эту функцию в свёртку:

```
optimal_path(Map) ->
    {A,B} = lists:foldl(fun shortest_step/2, {{0,[]}, {0,[]}},
        Map),
    {_Dist,Path} = if hd(element(2,A)) =/= {x,0} -> A;
            hd(element(2,B)) =/= {x,0} -> B
        end,
    lists:reverse(Path).
```

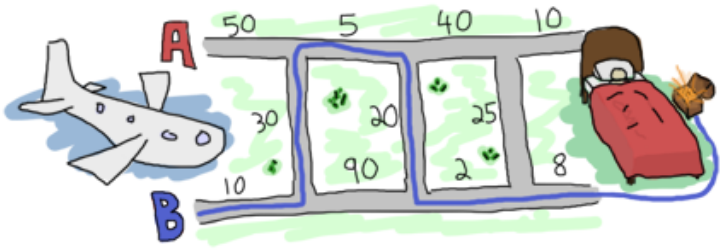
В результате выполнения свёртки, оба пути имеют одинаковую длину, но один из путей проходит через последний сегмент `{x,0}`. В конструкции `if` мы смотрим на последний элемент обоих путей и возвращаем путь, который не проходит через `{x,0}`. Можно также просто выбрать путь с наименьшим числом шагов (сравнить результаты `length/1`). Как только кратчайший путь был найден, мы разворачиваем его задом-наперёд (так как он был построен при помощи хвостовой рекурсии, его **нужно** развернуть). Теперь маршрут можно показать всему миру или держать в секрете, чтобы заполучить богатое дядюшкино наследство. Для успешной компиляции кода нужно добавить в функцию main вызов `optimal_path/1`.

```
main() ->
    File = "road.txt",
    {ok, Bin} = file:read_file(File),
    optimal_path(parse_map(Bin)).
```

О, глядите-ка! Мы получили верный ответ! Отличная работа!

```
1> c(road).
{ok,road}
2> road:main().
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]
```

А знаете что ещё бы нам пригодилось? Возможность запускать программу вне оболочки Erlang. Опять слегка изменим нашу функцию main:




```
main([FileName]) ->
    {ok, Bin} = file:read_file(FileName),
    Map = parse_map(Bin),
    io:format("~p~n",[optimal_path(Map)]),
    erlang:halt().
```

Теперь арность функции main равна 1. Это позволит нам получить параметры командной строки. Я добавил вызов функции `erlang:halt/0`, после которого виртуальная машина Erlang завершает работу, и завернул вызов `optimal_path/1` в `io:format/2`, потому что вне оболочки Erlang текст можно отобразить только таким способом.

С этими поправками файл road.erl примет следующий вид (не учитывая комментарии):

```
-module(road).
-compile(export_all).

main([FileName]) ->
    {ok, Bin} = file:read_file(FileName),
    Map = parse_map(Bin),
    io:format("~p~n",[optimal_path(Map)]),
    erlang:halt(0).

%% Transform a string into a readable map of triples
parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    Values = [list_to_integer(X) || X <- string:tokens(Str,"\r\n\
        t ")],
    group_vals(Values, []).

group_vals([], Acc) ->
    lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
    group_vals(Rest, [{A,B,X} | Acc]).

%% Picks the best of all paths, woo!
optimal_path(Map) ->
```

```
    {A,B} = lists:foldl(fun shortest_step/2, {{0,[]}, {0,[]}},
        Map),
    {_Dist,Path} = if hd(element(2,A)) =/= {x,0} -> A;
                      hd(element(2,B)) =/= {x,0} -> B
                   end,
    lists:reverse(Path).

%% actual problem solving
%% change triples of the form {A,B,X}
%% where A,B,X are distances and a,b,x are possible paths
%% to the form {DistanceSum, PathList}.
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.
```

Исполним код:

```
$ erlc road.erl
$ erl -noshell -run road main road.txt
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]
```

Да, всё верно! Для запуска программы больше ничего делать не нужно. Эти две строки можно завернуть в единый bash/bat файл, или применить для этого escript.

На примере этих двух упражнений мы увидели, что решать задачу становится проще, если сначала разбить её на мелкие части, по отдельности решить каждую подзадачу, а затем соединить всё воедино. Мы также увидели, что нет смысла начинать программирование до полного прояснения задачи. Ну и несколько тестов, конечно же, никогда не будут лишними. С их помощью вы убедитесь, что всё работает как положено и сможете менять код без изменения конечного результата.

Глава 11

Краткий обзор общеизвестных структур данных

11.1 Обзор недолгий, обещаю!

Функциональное подмножество Erlang вы, скорее всего, теперь понимаете достаточно неплохо, и текст многих программ смогли бы читать без проблем. Но, тем не менее, готов поспорить, что вам не вполне ясно как построить настоящую полезную программу, хоть последняя глава и рассказывала о том, как решать задачи в функциональном стиле. Говорю я об этом потому, что испытывал точно такие же чувства приблизительно в этот же момент моего обучения. Впрочем, если у вас такой проблемы нет, то поздравляю!

В любом случае, я исхожу из того, что мы рассмотрели много всего: большинство базовых типов данных, оболочку, то как писать модули и функции (с рекурсией), различные способы компиляции, управление логикой программы, перехват исключений, абстракция некоторых общих операций и т.д. Также мы познакомились с хранением данных в кортежах, списках, и рассмотрели незаконченную реализацию двоичного дерева поиска. Но мы совсем не рассматривали другие структуры данных, которые предоставляет программисту стандартная библиотека Erlang.

11.2 Записи

Прежде всего, записи – это хак. Они – что-то вроде запоздалой мысли, пришедшей в голову разработчикам языка, и поэтому имеют свои недостатки. О недостатках я расскажу позже. Записи очень пригождаются,

если нам нужна небольшая структура данных, которая предоставляет прямой доступ к именованным атрибутам. Поэтому записи в Erlang очень похожи на структуры в C (если вы знакомы с языком C).

Они объявляются так же как и атрибуты модуля:

```
-module(records).
-compile(export_all).

 -record(robot, {name,
                 type=industrial,
                 hobbies,
                 details=[]}).
```

Эта запись представляет информацию о роботе и имеет 4 поля: имя, тип робота, его хобби и подробности. Для полей, хранящих тип и подробности, кроме того, задано значение по умолчанию: соответственно `industrial` и `[]`. Вот так в модуле records можно объявить запись:

```
first_robot() ->
    #robot{name="Mechatron",
           type=handmade,
           details=["Moved by a small man inside"]}.
```

Запускаем код:

```
1> c(records).
{ok,records}
2> records:first_robot().
{robot,"Mechatron",handmade,undefined,
    ["Moved by a small man inside"]}
```

Опа! Вот и хак! Записи в Erlang – всего лишь синтаксический сахар поверх кортежей. К счастью, ситуацию можно улучшить. В оболочке Erlang есть команда `rr(Module)`, которая позволяет загружать определения записей из *Module*:

```
3> rr(records).
[robot]
4> records:first_robot().
#robot{name = "Mechatron",type = handmade,
    hobbies = undefined,
    details = ["Moved by a small man inside"]}
```

Совсем другое дело! Так с записями намного легче работать. Вы можете заметить, что в `first_robot/0` мы не определяли поле `hobbies`, и в его определении не было значения по умолчанию. Erlang неявно присваивает неинициализированным полям значение undefined.

Следующая функция поможет нам увидеть, как ведут себя значения по умолчанию, установленные в определении записи `robot`:

```
car_factory(CorpName) ->
    #robot{name=CorpName, hobbies="building cars"}.
```

И запускаем:

```
5> c(records).
{ok,records}
6> records:car_factory("Jokeswagen").
#robot{name = "Jokeswagen",type = industrial,
    hobbies = "building cars",details = []}
```

Теперь у нас есть промышленный робот, который любит заниматься постройкой машин.

Замечание: функция `rr()` может принимать не только имя модуля, но и шаблоны (вида `rr("*")`) и список загружаемых записей вторым аргументом.
В оболочке есть ещё несколько функций, с помощью которых можно манипулировать записями: `rd(Name, Definition)` позволяет определять запись способом похожим на `-record(Name, Definition)`, который мы использовали в нашем модуле. Команду `rf()` можно использовать для «выгрузки» всех записей, а `rf(Name)` и `rf([Names])` – чтобы избавиться только от указанных определений.
Можно использовать команду `rl()` для вывода определений всех записей в виде пригодном для непосредственного копирования в модуль, или её формы `rl(Name)` и `rl([Names])` для вывода лишь указанных записей.
И, наконец, команда `rp(Term)` конвертирует кортеж в запись (при условии, что запись пределена).

Но на одних записях далеко не уедешь. Нужно уметь извлекать из них значения. Делается это двумя способами. Первый – специальный «синтаксис с точкой». Представим, что определение записи о роботе уже загружено:

```
5> Crusher = #robot{name="Crusher", hobbies=["Crushing people","
    petting cats"]}.
#robot{name = "Crusher",type = industrial,
    hobbies = ["Crushing people","petting cats"],
    details = []}
6> Crusher#robot.hobbies.
["Crushing people","petting cats"]
```

Синтаксис красотой не блещет. Он выглядит так потому, что записи по природе своей – кортежи. Не более чем уловка компилятора. Поэтому для записи приходится хранить ключевые слова, которые определяют, к какой переменной она относится. Именно поэтому `#robot` – часть `Crusher#robot.hobbies`. Всё это, конечно, огорчает, но ничего не поделаешь. Вложенные записи выглядят ещё уродливее:

```
7> NestedBot = #robot{details=#robot{name="erNest"}}.
#robot{name = undefined,type = industrial,
       hobbies = undefined,
       details = #robot{name = "erNest",type = industrial,
                        hobbies = undefined,details = []}}
8> (NestedBot#robot.details)#robot.name.
"erNest"
```

И да, скобки обязательны.

Дополнение:
Начиная с ревизии R14A появилась возможность записывать вложенные записи без скобок. Вышеприведённый пример *NestedBot* можно также записать как `NestedRobot#robot.details#robot.name`. Работать такая запись будет аналогично.

Рассмотрим следующий пример, чтобы глубже обозначить зависимость записей от кортежей:

```
9> #robot.type.
3
```

Код выводит номер элемента кортежа, который реализует данное поле.

У записей есть одна оправдывающая их особенность – записи можно использовать в заголовках функции для сопоставления с образцом, а также в охранных выражениях. Определите в начале файла новую запись, и добавьте несколько функций:

```
-record(user, {id, name, group, age}).

%% use pattern matching to filter
admin_panel(#user{name=Name, group=admin}) ->
    Name ++ " is allowed!";
admin_panel(#user{name=Name}) ->
    Name ++ " is not allowed".

%% can extend user without problem
adult_section(U = #user{}) when U#user.age >= 18 ->
    %% Show stuff that can't be written in such a text
    allowed;
adult_section(_) ->
```

```
    %% redirect to sesame street site
    forbidden.
```

В функции `admin_panel/1` показан синтаксис, который позволяет связывать переменную с любым полем записи (можно связывать переменные сразу с несколькими полями). Следует отметить, что для связывания целой записи с переменной в функции `adult_section/1`, необходимо выполнить код `SomeVar = #some_record{}`. И, как обычно, скомпилировать:

```
10> c(records).
{ok,records}
11> rr(records).
[robot,user]
12> records:admin_panel(#user{id=1, name="ferd", group=admin, age
    =96}).
"ferd is allowed!"
13> records:admin_panel(#user{id=2, name="you", group=users, age
    =66}).
"you is not allowed"
14> records:adult_section(#user{id=21, name="Bill", group=users,
    age=72}).
allowed
15> records:adult_section(#user{id=22, name="Noah", group=users,
    age=13}).
forbidden
```

На этом примере можно увидеть, что нет необходимости делать сопоставление по всем элементам кортежа, или вообще иметь во время написания функции представление о количестве этих элементов. Можно провести сопоставление только по возрасту или группе, а об остальных полях даже и не вспоминать. Если бы нам пришлось использовать обычный кортеж, то определение функции выглядело бы приблизительно так: `function({record, _, _, ICareAboutThis, _, _}) -> ...`. Как только кто-либо решит добавить в кортеж новый элемент, кто-то другой (скорее всего весьма недовольный этой ситуацией) будет обязан исправить все функции, в которых используется этот кортеж.

Эта функция показывает как обновлять запись (иначе пользы от них было бы мало):

```
repairman(Rob) ->
    Details = Rob#robot.details,
    NewRob = Rob#robot{details=["Repaired by repairman"|Details]}
        ,
    {repaired, NewRob}.
```

А затем:

```
16> c(records).
{ok,records}
17> records:repairman(#robot{name="Ulbert", hobbies=["trying␣to␣
    have␣feelings"]}).
{repaired,#robot{name = "Ulbert",type = industrial,
     hobbies = ["trying␣to␣have␣feelings"],
     details = ["Repaired␣by␣repairman"]}}
```

Как видите, моего робота починили. Здесь для обновления записей используется особенный синтаксис. Может показаться, что мы обновляем ту же самую запись (`Rob#robot{Field=NewValue}`), но на самом деле это просто уловки компилятора, которые маскируют скрытый вызов функции `erlang:setelement/3`.

И последнее о записях. Записи весьма полезны, а дублирование кода не очень. Поэтому программисты на Erlang часто совмещают использование одних и тех же записей в нескольких модулях при помощи *заголовочных файлов*. Файл заголовков в Erlang очень похож на их аналог в языке C. Это просто фрагмент кода, который добавляется в модуль, как будто он там был всегда. Создайте файл records.hrl и добавьте в него следующий код:

```
%% this is a .hrl (header) file.
-record(included, {some_field,
                   some_default = "yeah!",
                   unimaginative_name}).
```

Чтобы включить этот заголовок в records.erl, нужно просто добавить в модуль эту строчку:

```
-include("records.hrl").
```

И в следующей функции применить запись:

```
included() -> #included{some_field="Some␣value"}.
```

А теперь, как обычно, проверим:

```
18> c(records).
{ok,records}
19> rr(records).
[included,robot,user]
20> records:included().
#included{some_field = "Some␣value",some_default = "yeah!",
          unimaginative_name = undefined}
```

Ура! Довольно о записях. Они, конечно, неприятные, но полезные. У них некрасивый синтаксис, они реализованы при помощи хака, но они

играют сравнительно важную роль в удобстве поддержки вашего кода.

Замечание: в проектах с открытым кодом вы сможете часто встретить описанный здесь метод. В проект добавляется `.hrl` файл, содержащий записи, которые используются во всех модулях приложения. Хоть я и считаю, что обязан задокументировать этот способ использования записей, но я настоятельно рекомендую применять только локальные определения, которые действуют в пределах одного модуля. Если вам нужно, чтобы какой-либо другой модуль имел доступ к внутренему устройству записи – напишите несколько функций, которые предоставляют доступ к полям, а подробности реализации сделайте как можно более закрытыми. Это правило позволяет предотвратить конфликт имён, помогает избежать проблем с обновлением кода, просто улучшает общую читабельность и упрощает поддержку вашего кода.

11.3 Хранилища пар ключ-значение

Несколько глав назад вы построили дерево и собирались его использовать в качестве хранилища пар ключ-значение для адресной книги. Книга получилась хреновая: удалять значения из неё мы не могли, да и найти ей достойное применение тоже не получилось. С её помощью нам удалось хорошо продемонстрировать принцип работы рекурсии, но не более. Пришло время познакомить вас с несколькими полезными структурами, предназначение которых – хранение данных, связанных с определённым ключом. Я не буду рассказывать подробно о каждой функции, или делать полный обзор модулей. Я просто дам вам ссылку на страницы документации. Считайте, что моей обязанностью является «улучшение осведомлённости о хранилищах пар ключ-значение в Erlang» или что-то вроде того. Ну а что, звание звучит неплохо. Осталось только нашивку сделать.

Для небольших объёмов данных можно использовать два вида структур. Первая называется *proplist* (список свойств). Proplist – это любой список кортежей вида `[{Key, Value}]`. На этом правила, определяющие её устройство, заканчиваются. Немного странная структура. Список может также содержать булевы значения, целые числа и вообще всё что вам угодно. Но сейчас нас интересует именно список, наполненный кортежами с ключом и значением. Для работы с proplist-ами можно использовать модуль proplists. Он содержит функции `proplists:delete/2`, `proplists:get_value/2`, `proplists:get_all_values/2`, `proplists:lookup/2` и

`proplists:lookup_all/2`.

Вы заметите, что в этом списке нет функций для добавления и удаления элементов. Это даёт представление о том, насколько нечётко определены proplist-ы как структура данных. Чтобы получить возможность добавлять и удалять элементы, придётся вручную добавлять их при помощи конструктора (`[NewElement|OldList]`) и использовать, например, функцию `lists:keyreplace/4`. Для работы с одной небольшой структурой данных приходится использовать два модуля, и это решение не назовёшь кристально чистым. Но благодаря нечёткому определению proplist-ов, их часто используют для представления списка настроек или общего описания какого-либо предмета. Proplist-ы нельзя назвать полной структурой данных. Их скорее можно отнести к общему шаблону, который используется для представления некоторого объекта или предмета при помощи списков и кортежей. А модуль proplists – что-то вроде инструментария поверх этого шаблона.

Если вам понадобилось хранилище пар ключ-значение для небольших объёмов данных с более полной функциональностью, то модуль orddict – это то что нужно. Orddict-ы (упорядоченные словари) – это proplist-ы со склонностью к формализму. Каждый ключ может присутствовать в структуре только один раз, список упорядочивается для ускорения поиска и т.д. Обычные CRUD-операции представлены функциями `orddict:store/3`, `orddict:find/2` (на случай, если вам точно не известно, есть ли такой ключ в словаре), `orddict:fetch/2` (когда вы знаете, что такой ключ в словаре присутствует, или что он там **должен** быть) и `orddict:erase/2`.

Orddict-ы представляют собой неплохой компромисс между сложностью и эффективностью, при условии, что количество элементов приблизительно равно 75 (ознакомьтесь с моими замерами). Если количество элементов превышает это число, то вам нужно переключиться на какое-либо другое хранилище пар ключ-значение.

В принципе, существует два модуля/структуры для работы с большими объёмами данных: dicts и gb_trees. У словарей такой же интерфейс как и у orddict-ов: `dict:store/3`, `dict:find/2`, `dict:fetch/2`, `dict:erase/2` и любая другая функция, например `dict:map/2` и `dict:fold/2` (их очень удобно использовать при работе с целой структурой). Так что если вам

нужно расширить orddict – словари очень хорошо справятся с этой задачей..

Зато у сбалансированных деревьев общего назначения есть несколько функций, которые предоставляют прямой доступ к управлению структурой. По сути, у gb_trees есть два режима: в первом режиме вам известно о структуре всё (я называю этот режим «умным (smart) режимом»), и второй, в котором вы не можете делать какие-либо предположения о структуре (я называю этот режим «наивным (naive) режимом»). В наивном режиме доступны функции `gb_trees:enter/2`, `gb_trees:lookup/2` и `gb_trees:delete_any/2`. В умном режиме им соответствуют функции `gb_trees:insert/3`, `gb_trees:get/2`, `gb_trees:update/3` и `gb_trees:delete/2`. Ещё можно использовать `gb_trees:map/2`, а такая возможность никогда не бывает лишней.

Недостатком «наивных» функций, в сравнении с «умными» является то, что после вставки нового элемента (или удаления нескольких), может появиться необходимость перебалансировки дерева. Это займёт определённое время и потребует некоторого количества памяти (даже если просто нужно будет убедиться в том, что перебалансировка не нужна). Все «умные» функции предполагают, что искомый ключ присутствует в дереве. Это позволяет пропустить все проверки и уменьшает время исполнения операций.

Когда же всё-таки необходимо использовать gb_trees вместо dict? Решение не вполне очевидно. Результаты исполнения моего измерительного модуля, свидетельствуют о том, что gb_trees и dicts показывают во многих случаях приблизительно одинаковый результат. Но измерения также говорят о том, что dicts показывает самую высокую скорость при считывании значений, а gb_trees отрабатывает немного быстрее в других операциях. Вы сможете делать выбор, отталкиваясь от того, какие операции для вашей задачи важнее других.

Да, ещё запомните, что у dicts есть функция свёртки (fold), а у gb_trees – нет. Вместо неё у gb_trees есть функция-*итератор*, которая возвращает часть дерева, к которой можно применить вызов `gb_trees:next(Iterator)` и получить следующее по порядку значение. Так что к структуре gb_trees не получится применять универсальный fold, а придётся писать собственные рекурсивные функции. Зато gb_trees предоставляет быстрый доступ к наименьшему и наибольшему элементу в структуре при помощи функций `gb_trees:smallest/1` и `gb_trees:largest/1`.

Можно сказать, что при выборе хранилища пар ключ-значение вы должны руководствоваться нуждами вашего приложения. На ваше ре-

шение могут оказать влияние различные факторы, каждый из которых по-своему важен: сколько данных вам необходимо хранить, какие операции с этими данными нужно выполнять и прочее. Чтобы убедиться в том, что вы не ошиблись – проводите измерения, профилируйте код, сравнивайте скорость исполнения.

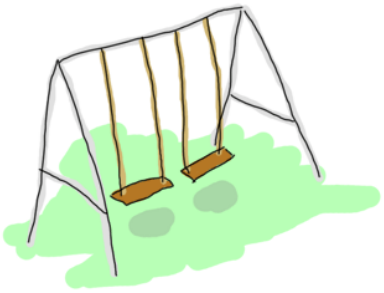
Замечание: существуют специальные хранилища пар ключ-значение для работы с ресурсами различного размера. Это ETS таблицы, DETS таблицы и база данных mnesia. Но так как их использование тесно связано с концепцией множественных процессов и распределённостью, мы затронем их позже. Я оставляю эту информацию как ориентир, призванный разжечь ваше любопытство и направить заинтересовавшихся в нужном направлении.

11.4 Массивы

А как быть с кодом, для которого нужны структуры данных, принимающие только числовые данные? В нём можно использовать массивы. К их элементам можно обращаться при помощи числовой идексации, а также применять ко всей структуре операцию свёртки. При этом есть возможность игнорировать позиции с неопределённым значением.

Не забывайтесь:
В противоположность своим императивным собратьям, массивы Erlang не могут выполнять операции вставки и поиска за константное время. На практике массивы используют редко, так как они обычно медленнее массивов в языках, подерживающих деструктивное присваивание, и стиль программирования на Erlang не очень хорошо стыкуется с массивами и матрицами.
В задачах, где необходимо использовать матричные манипуляции и массивы, программисты склоняются к использованию концепции, которая называется Ports. С её помощью исполнение тяжёлых операций перекладывается на другие языки. К той же концепции можно отнести C-Nodes, Linked in drivers и NIF (экспериментальная поддержка в R13B03+).
Ещё одна странность массивов заключается в том, что их индексация начинается с 0 (в противоположность кортежам или спискам). Такая же индексация используется и в модуле для работы с регулярными выражениями. Будьте внимательны.

11.5 Набор наборов



Если вам когда-либо приходилось изучать теорию множеств, то у вас есть представление о свойствах наборов. Если не приходилось, то вы, возможно, не захотите читать дальше. Но я, всё-таки скажу, что наборы – это группы уникальных элементов, которые можно сравнивать. Над ними также можно совершать различные действия: найти элементы, которые одновременно принадлежат двум группам, или не принадлежат ни одной группе, или принадлежат лишь одной из групп и т.д. Можно также производить более сложные операции, которые позволяют определить связи между группами, совершать над ними действия, и ещё много всего. Не буду погружаться в теорию (опять же, она выходит за рамки этой книги), а просто опишу наборы как они есть.

В Erlang существует 4 основных модуля, предназначенных для работы с наборами. Сперва такое обилие может показаться странным, но оно порождено согласием разработчиков, что «лучшего» способа реализации наборов не существует. Вот эти четыре модуля: ordsets, sets, gb_sets и sofs (наборы наборов):

ordsets
Ordset-ы реализованы в виде упорядоченного списка. Главным образом, их хорошо использовать для небольших объёмов данных, и они – самый медленный вид набора. Но из всех наборов они имеют самый простой и легкочитаемый способ представления. Для них реализованы следующие стандартные функции: `ordsets:new/0`, `ordsets:is_element/2`, `ordsets:add_element/2`, `ordsets:del_element/2`, `ordsets:union/1`, `ordsets:intersection/1` и другие.

sets
Модуль sets использует в своей основе структуру, очень похожую на ту, что используется в `dict`. Sets реализует тот же самый интерфейс, что и ordsets, но по сравнению с ним намного лучше масштабируется. Как и словари, эту структуру очень удобно использовать для действий, требующих большого количества операций чтения, например для проверки вхождения какого-либо элемента в набор.

gb_sets
Этот модуль построен поверх сбалансированных деревьев, похожих на структуру, используемую в модуле gb_trees. gb_sets относится к sets так же как и gb_tree к dict. Они быстрее в операциях отличных от чтения, и дают больше контроля над структурой. Модуль gb_sets не только реализует тот же самый интерфейс, что sets и ordsets, но и дополняет его множеством функций. Как и в gb_trees у вас есть умные (smart) и наивные (naive) функции, итераторы, быстрый доступ к наименьшему и наибольшему значению и т.д.

sofs
Наборы наборов (osfs) реализованы при помощи упорядоченных списков, скомпонованных с метаданными в кортеж. Этот модуль следует использовать, если вы хотите полностью управлять связями наборов, их объединениями, обеспечивать для наборов соблюдение типов и т.д. Если вам требуется именно математическая концепция наборов, а не «просто» группы уникальных элементов, то вам необходим именно этот модуль.

Не забывайтесь:
Такое многообразие можно, конечно, рассматривать как преимущество, но детали реализации могут свести эти преимущества на нет. К примеру, gb_sets, ordsets и sofs – все используют оператор `==` для сравнения значений. Числа 2 и 2.0 при сравнении будут рассматриваться как одно и то же число.

А вот, модуль sets использует оператор `=:=`. Так что просто переключаться по желанию между реализациями получится не всегда. В некоторых случаях вам будет подходить лишь одно поведение, и тогда все преимущества, которые даёт множество реализаций, просто исчезнут.

Такой широкий выбор немного сбивает с толку. Разработчик Björn Gustavsson из команды Erlang/OTP, который также разрабатывает Wings3D, рекомендует в большинстве случаев использовать gb_sets; ordset использовать, когда для последующей обработки в собственном коде необходимо простое представление данных, а sets оставить на случай, когда для сравнения нужно использовать оператор `=:=` (источник).

Как бы то ни было, обычно лучше провести замеры производительности вашего кода, и определить структуру, которая наилучшим образом подходит для вашей задачи.

11.6 Ориентированные графы

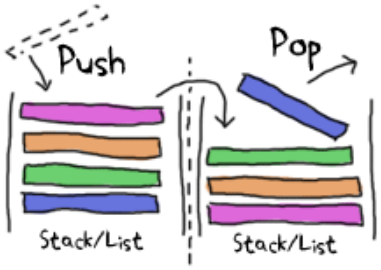
Хочу упомянуть ещё одну структуру данных (я не хочу сказать, что помимо упомянутых в этой главе структур ничего больше нет, как раз наоборот): ориентированные графы. Эта структура, опять же, будет полезна читателям, знакомым с математической основой орграфов.

В Erlang взаимодействие с ориентированными графами реализуется двумя модулями: digraph и digraph_utils. Модуль digraph позволяет создавать и модифицировать орграфы: совершать манипуляции с рёбрами и вершинами, находить пути и циклы и т.д. А digraph_utils позволяет передвигаться по графу (прямой порядок, обратный порядок), проверять на наличие циклов, деревьев, находить соседние вершины и так далее.

Так как орграфы тесно связаны с теорией множеств – модуль «sofs» содержит несколько функций, которые позволяют конвертировать семейства в орграфы и орграфы в семейства.

11.7 Очереди

Модуль queue реализует двунаправленную FIFO (First In, First Out) очередь: На картинке представлена примерная реализация: два списка (в



этом контексте стеки), которые позволяют быстро добавлять элементы в голову и хвост очереди.

Модуль queue содержит несколько функций, которые мысленно можно отнести к 3-м интерфейсам (или API) различной сложности. Они называются «Оригинальный API», «Расширенный API», и «Okasaki API»:

Оригинальный API
Оригинальный API содержит функции, реализующие базовые свойства очереди. В их число входят: `new/0` создаёт пустую очередь, `in/2` вставляет новый элемент, `out/1` удаляет элемент. Здесь также есть функции для конвертации очереди в список, функция для изменения порядка элементов на противоположный, для определения вхождения какого-либо значения в очередь и т.д.

Расширенный API
В расширенном API, главным образом, введены средства, придающие структуре гибкость, и позволяющие осуществлять интроспекцию. С их помощью можно, например, извлекать головной элемент без его удаления (см. `get/1` или `peek/1`), просто удалять элементы, не заботясь об их значении (`drop/1`), и т.д. Эти функции нельзя отнести к основам, составляющим концепцию очереди, но всё равно они весьма полезны.

Okasaki API
Okasaki API немного странный. Он взят из книги Chris Okasaki Purely Functional Data Structures. Этот API предоставляет набор операций, похожий на предыдущие два, но некоторые имена функций в нём записываются задом-наперёд, и в целом этот API весьма своеобразен. Если вы не вполне уверены, что вам нужен именно этот API, то лучше с ним не связывайтесь.

Очереди обычно используют, когда необходимо убедиться, что первый по порядку элемент точно будет обработан первым. Те примеры, которые я показывал ранее, в качестве аккумуляторов в основном использовали списки, которые мы разворачивали в конце вычислений. Если вы не можете развернуть сразу все элементы, или в аккумулятор приходится часто добавлять новые, то вам скорее всего нужен модуль queue (конечно же, вы сперва должны провести измерения и всё проверить! Первым делом всё всегда нужно измерять и проверять!)

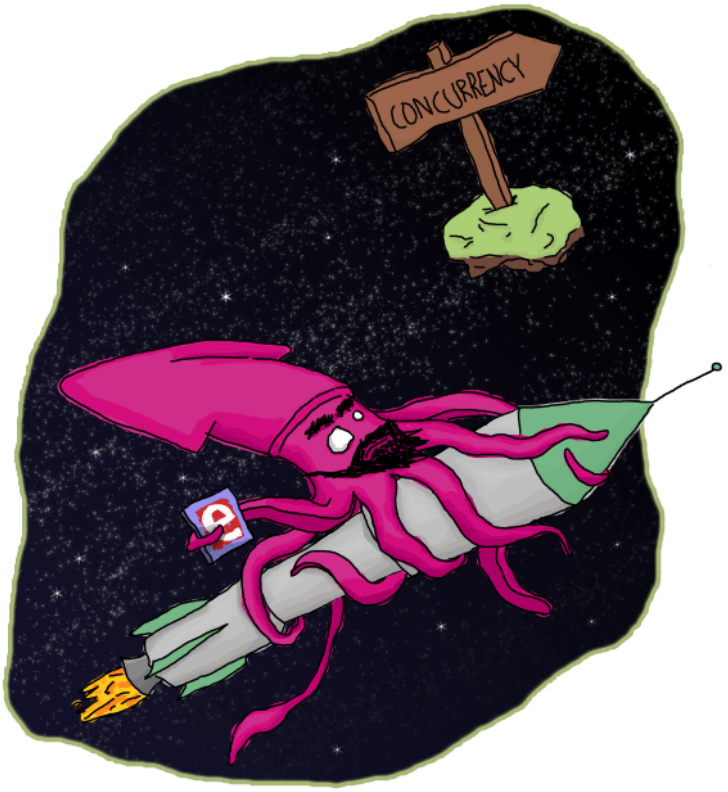
11.8 Окончание краткого обзора

Вот и закончен наш краткий экскурс в структуры данных Erlang. Спасибо, что во время путешествия не высовывали руки за пределы транспортного средства. Есть, конечно же, и другие структуры данных, которые помогают в решении различных задач. Я осветил лишь те, которые вам пригодятся больше всех, и с которыми, учитывая специфику

задач решаемых с помощью Erlang, вы точно столкнётесь. Для поиска дополнительных сведений рекомендую обратиться к стандартной и расширенной библиотеке.

Вас наверняка порадует, что на этом мы завершаем путешествие в последовательный (sequential) (функциональный) Erlang. Я знаком со многими людьми, которые начали изучать Erlang из-за параллелизма, процессов и всего прочего. И это неудивительно, ведь Erlang блестяще справляется с задачами именно в этих областях. К вашим услугам: деревья контроля, развитая система управления ошибками, распределённость и многое другое. Знаю, что из-за собственной нетерпеливости я уже касался этих тем, а некоторые нетерпеливые читатели уже наверняка о них успели прочитать.

Тем не менее, я посчитал, что перед переходом к параллельному Erlang нужно привыкнуть к его функциональной стороне. Так нам будет легче продвигаться по материалу и концентрироваться лишь на новых концепциях. Начинаем!



Глава 12

Путеводитель по параллелизму для путешествующих автостопом

Далеко-далеко в закоулках нефешенебельного начала 21-го века, которого даже нет на карте, находится маленькое подмножество человеческих знаний.

В этом подмножестве заключена совершенно невзрачная маленькая дисциплина, чья Фон-Неймановская архитектура столь примитивна, что согласно ей ОПЗ-калькуляторы считаются чем-то выдающимся.

У этой дисциплины есть, а точнее была – проблема: большинство людей, изучающих её, почти всегда оставались недовольны, создавая параллельное ПО. Предлагалось множество решений этой проблемы, но чаще всего они были связаны с перемещением маленьких логических блоков, которые назывались локами, мутексами и всякими другими именами, что несколько странно, поскольку этим самым блокам параллелизм был совершенно не нужен.

Так проблема и оставалась нерешённой. Многие люди теряли человечесий облик, на кого-то просто было жалко смотреть, и даже ОПЗ-калькуляторы им не помогали.

Кое-кто был убеждён, что людям не стоило добавлять параллелизм в языки программирования. И что программы вообще не должны были выходить из их исходного потока.

Замечание: неплохое развлечение – писать пародии на «Путеводитель по галактике для путешествующих автостопом». Если вам ещё не попадалась эта книга, то обязательно её прочтите. Она стоит того!

12.1 Без паники

Привет. Сегодня (или в любой из дней, когда вы читаете эти строки, может быть даже завтра) я хочу рассказать вам о параллельном Erlang. Скорее всего вы уже читали о параллелизме, или когда-либо сталкивались с ним. Может быть, вы интересуетесь истоками появления программирования для множества ядер, или читаете эту книгу, наслушавшись разговоров, которыми в наше время окружён параллелизм.

Впрочем, хочу вас предупредить, что в этой главе основной упор сделан на теорию. Если у вас болит голова, или вы питаете отвращение к истории языков программирования, или вам просто хотелось попрограммировать, то перейдите лучше к **??** концу главы, или сразу к следующей (там как раз освещается более практическая сторона вопроса).

Я уже объяснил во введении к книге, что параллелизм в Erlang основан на передаче сообщений и модели акторов. Мой пример рассказывал о людях, общение которых происходит исключительно при помощи писем. Чуть позже я ещё вернусь к этой модели, а сейчас нам необходимо первым делом обозначить разницу между *конкурентностью* и *параллелизмом*.

Оба слова во многих ситуациях имеют одно и то же значение. Но в контексте Erlang они часто относятся к двум разным концепциям. По мнению большинства эрлангистов понятие конкурентности описывает несколько акторов, которые исполняются независимо друг от друга, но их исполнение не обязательно происходит в один тот же момент. Параллелизм же означает, что несколько акторов исполняются одновременно. Взгляды различных областей computer science на правильность этих определений могут не совпадать, но в этом руководстве я буду пользоваться именно такими определениями. Не удивляйтесь, если в других источниках вы увидите, как кто-то употребляет те же самые термины для обозначения других понятий.

Конкурентность была в Erlang с самого начала, даже в восьмедисятые, когда всё запускалось на одноядерном процессоре. Каждому процессу Erlang для исполнения отводился свой собственный временной отрезок, совсем как в эру десктопных приложений, которая предшествовала появлению многоядерных систем.

Уже в то время можно было, в принципе, реализовать параллелизм.

Для этого нам бы потребовался второй компьютер, который исполнял бы код и обменивался информацией с первым. Но даже такая система смогла бы исполнять параллельно всего лишь два актора. Современные многоядерные системы позволяют реализовать параллелизм в рамках одного компьютера (некоторые промышленные чипы могут содержать десятки ядер), и Erlang использует эту возможность в полной мере.

Не забывайтесь:
Важно понимать разницу между конкурентностью и параллелизмом. Многие программисты верят, что Erlang был готов к использованию на многоядерных компьютерах задолго до того, как это произошло в действительности. Erlang стал использовать истинную симметричную мультипроцессорность (symmetric multiprocessing) лишь в середине двухтысячных, а большая часть реализации была завершена в релизе R13B в 2009 году. До этого часто приходилось отключать SMP, чтобы избежать потерь производительности. На многоядерном компьютере без SMP можно получить параллелизм, если запустить одновременно несколько экземпляров виртуальной машины.

Интересно отметить, что для внесения истинного параллелизма в язык, не потребовалось производить какие-либо концептуальные изменения на языковом уровне. Благодаря тому, что конкурентность в Erlang строится вокруг изолированных процессов, все изменения были сделаны внутри VM, подальше от глаз обычного программиста.

12.2 Принципы конкурентности

Когда-то разработка языка Erlang проходила в быстром темпе, и от инженеров, которые занимались разработкой на Erlang для телефонных коммутаторов, поступал плотный поток обратной связи. Их отчёты подтвердили, что конкурентность, основанная на процессах, и асинхронная передача сообщений позволяли хорошо моделировать задачи, возникающие перед разработчиками. Кроме того, ещё до появления Erlang, в мире телефонии уже сформировалась некая культура, тяготеющая к параллелизму. Она была унаследована от языка PLEX, который ранее был создан в Ericsson, и использовался в коммутаторах AXE. Erlang унаследовал эту тенденцию, и попытался усовершенство-

вать существующие инструменты.

Для того, чтобы Erlang считался хорошим инструментом, он должен был удовлетворять нескольким требованиям. Главным условием была возможность масштабирования и поддержки тысяч и тысяч пользователей на множестве коммутаторов. Также эти коммутаторы должны были обеспечивать высокую надёжность работы – вплоть до того, что исполнение кода никогда не должно было останавливаться.

12.2.1 Масштабируемость

Для начала я расскажу о масштабировании. Для достижения масштабируемости была необходима система, обладающая определёнными свойствами. В такой системе пользователи были бы представлены при помощи процессов, которые реагируют на определённые события (например, приём звонка, завершение разговора и т.д.). Идеальная система должна поддерживать процессы, выполняющие малые объёмы вычислений, и быстрое переключение между процессами при поступлении событий. Высокоэффективная обработка процессов предполагала возможность их очень быстрого старта, очень быстрого уничтожения, и очень быстрой коммутации. Обязательным условием для такого поведения была легковесность процессов. Это было также необходимо, чтобы избавиться от наличия пула процессов (фиксированное множество процессов, между которыми распределяется работа). Намного легче создавать программы, которые используют сразу столько процессов, сколько нужно.

Ещё один важный аспект масшабируемости – это возможность преодоления ограниченности ресурсов оборудования. Для решения этой задачи выделяют два направления: можно улучшать характеристики оборудования, а можно увеличивать его количество. Первое решение будет работать до определённого момента, после которого за улучшение придётся очень дорого платить (необходимо, например, покупать суперкомпьютер). Второе решение, как правило, обходится дешевле. Для выполнения тех же задач нужно просто добавлять больше компьютеров. Вот где вашему языку может пригодиться распределённость.

Вернёмся к обсуждению лёгких процессов. Высокая надёжность очень важна для нужд телефонии, поэтому разработчики решили, что правильнее всего будет запретить процессам иметь общую память. Некоторые аварийные ситуации с участием разделяемой памяти могут привести к противоречивому состоянию (особенно если данные разделяются между несколькими узлами) и осложнениям. Вместо этого процессы должны общаться при помощи сообщений, которые содержат полные копии данных. Этим мы рискуем получить более медленное, но зато более

надёжное решение.

12.2.2 Устойчивость к сбоям

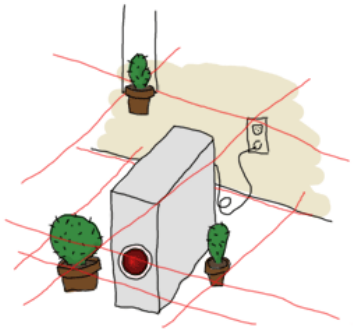
Мы приходим ко второму требованию, которому должен соответствовать Erlang: надёжность. Первые разработчики на Erlang всегда помнили о том, что сбои происходят повсеместно. Можно сколько угодно пытаться предотвратить ошибки, но в большинстве случаев от некоторых из них не получится избавиться полностью. И даже если не будет ошибок – от сбоев оборудования никуда не денешься. Поэтому вместо того, чтобы пытаться полностью предотвратить ошибки, лучше найти хороший способ их обрабатывать.

Оказывается, что подход к проектированию при помощи множественных процессов с передачей сообщений, оказался верным, так как в него можно относительно легко встроить обработку ошибок. Возьмём, к примеру, легковесные процессы (созданные для быстрых перезапусков и выключений). Исследования показали, что для масштабных программных комплексов главным источником простоя являются ошибки, которые нерегулярно себя проявляют и спонтанно исчезают (источник). Существует правило, говорящее что ошибки, которые искажают данные, должны как можно быстрее приводить к остановке неисправной части системы, чтобы предотвратить проникновение ошибок и плохих данных в остальные узлы. Есть также ещё одна концепция, согласно которой существует множество различных способов остановки системы. Двумя такими способами являются корректная остановка и сбой (завершение, вызванное непредвиденной ошибкой).

Очевидно, что наихудшим исходом будет сбой. Для надёжного устранения проблемы сбоев можно сделать так, чтобы все аварийные ситуации проходили так же, как и корректные остановки. Для этого необходимо использовать ряд методов, к которым можно отнести принцип неразделяемости ресурсов (shared-nothing) и единичное присваивание (single assignment) (которое позволяет изолировать память процесса), уход от блокировок (после аварии блокировка может остаться закрытой, и будет перекрывать другим процессам доступ к данным, или просто приводить данные в нестабильное состояние), а также другие техники, которые я не буду подробно описывать, но они также использовались при проектировании Erlang. Таким образом, идеальным решением аварийной ситуации в Erlang считается быстрое уничтожение процессов, которое позволяет избежать порчи данных и случайных, нерегулярных ошибок. Ключевым элементом этой схемы являются лёгкие процессы. В языке также присутствуют механизмы обработки ошибок, которые позволяют процессам

следить за другими процессами (подробнее о них в главе 14 Ошибки и процессы), определять момент смерти процесса и планировать действия, связанные с этим событием.

Предположим, что быстрый перезапуск процессов позволил нам справиться с аварийными ситуациями. Следующая проблема – сбои оборудования. Как же сделать так, чтобы программа работала даже тогда, когда кто-то пинает ногами компьютер, на котором она запущена? Сложнейший защитный механизм, состоящий из лазерных детекторов и стратегически расставленных кактусов, конечно же, проработает какое--то время, но надолго его не хватит. Напрашивается мысль, что можно просто запустить программу сразу на нескольких компьютерах. Ведь мы это и так уже делаем при масштабировании. Вот вам и ещё одно преимущество независимых процессов, единственным средством коммуникации которых является передача сообщений. Они будут работать одинаково и на локальном и на удалённом компьютере. Такая отказоустойчивость посредством распределённости будет работать практически без участия программиста.



Распределённость напрямую влияет на общение процессов между собой. Мы не можем быть уверены, что если узел (удалённый компьютер) существовал в момент вызова функции, то он будет существовать и во время передачи вызова, и что вызов вообще будет правильно выполнен. Это одно из самых серьёзных препятствий для внедрения распределённости. Если кто-то споткнётся о кабель питания или выключит компьютер, то ваше приложение зависнет. А может быть оно аварийно завершится. Кто знает?

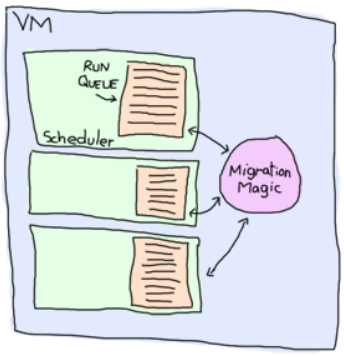
Выбор асинхронной передачи сообщений, оказывается, тоже был верным шагом. Согласно модели «процессы с асинхронной передачей сообщений», сообщения передаются от одного процесса к другому и хранятся в почтовом ящике принимающего процесса до тех пор, пока они не будут извлечены и прочитаны. При отсылке сообщения мы даже не проверяем, что получающий процесс существует, так как пользы от этой проверки нет никакой. Как сказано в предыдущем абзаце, невозможно узнать заранее, что за время, прошедшее между отсылкой сообщения и его получением, не случится аварийное завершение процесса. А если сообщение всё же получено, то невозможно узнать, что процесс-получатель хоть как-то отреагирует на сообщение, или вообще доживёт до этого мо-

мента. Асинхронные сообщения позволяют безопасно вызывать удалённые функции, так как не делают никаких прогнозов о возможном развитии событий. Прогнозы должен делать программист. Если вам нужно подтвердить факт доставки, то в ответ необходимо послать оригинальному процессу подтверждающее сообщение. Для сообщения, а также для любой программы или библиотеки, построенной на этом принципе, будут выполняться те же самые принципы безопасности.

12.2.3 Реализация

В общем, было принято решение, что легковесные процессы с асинхронной передачей сообщений полностью подходят Erlang. Как теперь всё это заставить работать? Ну, во-первых, нельзя доверять операционной системе управление процессами. Операционные системы имеют слишком много методов регуляции процессов, причём эти методы очень сильно различаются по своей производительности. Большинство методов, если не все, слишком медленны или слишком тяжелы для нужд стандартных областей применения Erlang. Перенося эти механизмы регуляции внутрь VM, разработчики Erlang сохраняют контроль над оптимизацией и надёжностью. Современный процесс Erlang занимает около 300 слов памяти, и может создаваться за микросекунды. В большинстве сегодняшних операционных систем такое не увидишь.

Чтобы справляться с множеством потенциальных процессов, которые может создать ваша программа, VM создаёт по одному потоку на каждое ядро. Эти потоки будут выполнять функцию *планировщика* (scheduler). Каждый планировщик обслуживает *очередь исполнения* (run queue), которая представляет собой список процессов Erlang, между которыми распределяются отрезки времени. Когда в очереди исполнения какого-либо из планировщиков появляется слишком много задач, часть из них переносят в другую очередь. Так что каждая виртуальная машина Erlang самостоятельно осуществляет балансировку нагрузки, и программиста это не должно беспокоить. Выполняются также и другие оптимизации, например ограничение частоты отсылки сообщений перегруженным процессам, что позволяет регулировать и перераспределять нагрузку.



Всё самое сложное будет делать вместо вас вирутальная машина. Именно благодаря этой автоматизации в Erlang можно легко перейти к

параллельному исполнению задач. Переход к параллельному исполнению означает, что скорость работы вашей программы возрастёт в два раза, если вы добавите второе вычислительное ядро, в четыре раза, если добавить ещё 4 и так далее, правильно? Не всегда. Такой рост скорости, зависящий от количества ядер или процессоров, называется *линейным масштабированием* (linear scaling) (см. график ниже). К сожалению, в реальности не бывает бесплатных обедов (бывают на похоронах, но за них всё равно кто-то платит).

12.3 Почти линейное масштабирование, но не совсем

Линейное масштабирование трудно осуществлять не из-за самого языка, а из-за природы решаемых задач. О задачах, которые очень хорошо масштабируются, иногда говорят, что они *очевидно параллельны* (embarassingly parallel). Если вы поищете в Интернет очевидно параллельные задачи, то скорее всего наткнётесь на примеры алгоритмов трассировки лучей (ray-tracing) (метод создания 3D изображений), поиска грубой силой в криптографии, предсказания погоды и т.д.

Время от времени на IRC каналах, форумах или почтовых рассылках появляются люди с вопросом, можно ли применять Erlang для решения таких задач, и можно ли при этом использовать GPU. Ответ на этот вопрос почти всегда отрицательный, и причина для этого сравнительно проста: все эти задачи обычно решаются при помощи численных алгоритмов, перемалывающих большие объёмы данных. Erlang справляется с такими задачами не очень хорошо.

Задачи, которые являются очевидно параллельными для Erlang, относятся к более высокому логическому уровню. Обычно они связаны с такими понятиями как чат-серверы, телефонные коммутаторы, веб-серверы, очереди сообщений, поисковые роботы, ну или любые другие задачи, в которых действия могут быть представлены как независимые логические элементы (кто-нибудь вспомнил об акторах?). Такой тип задач можно эффективно решать с масштабированием, приближающимся к линейному.

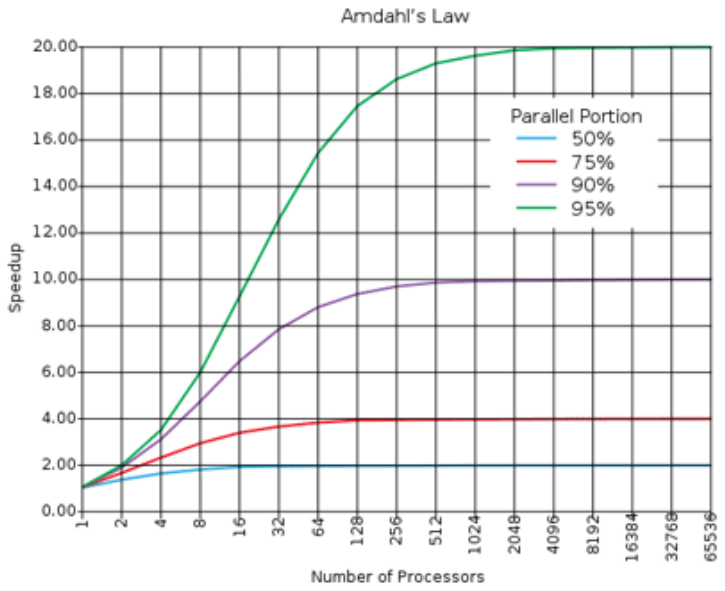
Но для некоторых задач такие характеристики масштабирования недостижимы. Если в решении присутствует одна централизованная последовательность операций, то о линейности масштабирования можно забыть. **Ваша параллельная программа не может работать быстрее, чем её самая медленная последовательная часть.** Пример

такого феномена можно наблюдать при любом походе в супермаркет. Сотни людей могут одновременно выбирать товары в зале, при этом почти не мешая друг другу. Но если количество покупателей превышает число кассиров, то при оплате покупок сразу образуются очереди.

Можно было бы добавлять кассиров до тех пор, пока на каждого покупателя не будет приходиться по одному кассиру, но тогда вам придётся для каждого покупателя добавлять и собственную дверь, потому что они не смогут все одновременно входить и выходить из магазина.

Иначе говоря, хоть покупатели и могли выбирать необходимые товары параллельно, и это отнимало бы столько же времени, сколько нужно для выбора товаров в одиночестве, но в конце концов им всё равно пришлось бы ожидать своей очереди на пути к кассе.

В обобщённом виде этот принцип называется законом Амдала. Он демонстрирует ускорение, которое можно ожидать от системы после добавления параллелизма, учитывая соотношение объёма параллельного и последовательного кода:



Согласно закону Амдала, код параллельный на 50% никогда не сможет стать быстрее более чем в два раза, а код параллельный на 95% теоретически может работать в 20 раз быстрее, при наличии достаточного количества процессоров. Интересно видеть на этом графике, что удаление последних непараллельных частей программы теоретически даст огромный прирост скорости, по сравнению с удалением последовательного кода из программы, которая и до удаления была не очень-то параллельной.

Не забывайтесь:
Параллелизмом *не получится* решить любую проблему. В некоторых случаях добавление параллелизации замедлит ваше приложение. Это может произойти, если ваша программа на 100% состоит из последовательного кода, но при этом пытается использовать несколько процессов.

Одним из лучших примеров такого поведения является *кольцевой тест* (ring benchmark). В этом тесте несколько тысяч процессов последовательно по кругу передают друг другу данные. Это очень похоже на игру в телефон, если такая аналогия вам покажется знакомой. В этом тесте в каждую единицу времени лишь один процесс выполняет полезную работу, но виртуальная машина Erlang в это время всё равно не прекращает распределять нагрузку между ядрами, и отдаёт каждому процессу положенную ему часть времени.

Такая ситуация оказывает негативный эффект на множество оптимизаций, применяемых в оборудовании, и заставляет виртуальную машину тратить время на бесполезные вычисления. Зачастую это приводит к тому, что приложение, полностью состоящее из последовательного кода, исполняется на нескольких ядрах значительно медленнее, чем на одном. В таком случае не помешает отключить симметричную мультипроцессорность (`$ erl -smp disable`).

12.4 Прощайте, и спасибо за рыбку!

Эта глава не была бы полна без рассказа о трёх базовых компонентах, необходимых для осуществления конкурентности в Erlang: создание новых процессов, отсылка сообщений и получение сообщений. Есть ещё и другие механизмы, необходимые для создания по-настоящему надёжных приложений, но на данном этапе будет довольно перечисленных.

В своём рассказе я много раз обходил стороной объяснение сущности процесса. Он является ни чем иным, как функцией. Вот и всё объяснение. Запускается функция, и, как только она завершает исполнение, процесс исчезает. Формально у процесса есть скрытое состояние (к нему относится, например, ящик для почтовых сообщений), но на данный момент нам будет достаточно обсуждения функций.

Новые процессы в Erlang создаются функцией `spawn/1`, которая принимает в качестве параметра ещё одну функцию и запускает её.

```
1> F = fun() -> 2 + 2 end.
#Fun<erl_eval.20.67289768>
```

```
2> spawn(F).
<0.44.0>
```

`spawn/1` возвращает результат (`<0.44.0>`), который называется *идентификатором процесса* (process identifier), и чаще всего именуется сообществом как *PID*, *Pid* или *pid*. Идентификатор процесса является произвольным значением, которое представляет любой процесс, существующий (или существовавший) в какой-либо момент жизни виртуальной машины. Его используют как адрес, необходимый для общения с процессом.

Вы, должно быть, заметили, что мы не видим результат вычисления функции *F*. Мы получаем лишь pid. Так происходит потому, что процессы ничего не возвращают.

Как же нам в таком случае увидеть результат выполнения F? Есть два способа. Проще всего просто отобразить полученный результат:

```
3> spawn(fun() -> io:format("~p~n",[2 + 2]) end).
4
<0.46.0>
```

Для настоящих программ это не очень удобно, но так мы сможем получить представление об организации процессов в Erlang. К счастью, для экспериментов, нам будет достаточно функции `io:format/2`. Мы быстро создадим 10 процессов и приостановим их выполнение на некоторый период времени при помощи функции `timer:sleep/1`. Она принимает целое значение *N*, ожидает пока пройдут *N* миллисекунд и продолжает выполнение кода. После задержки на экран выводится переданное процессу значение.

```
4> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end.
#Fun<erl_eval.6.13229925>
5> [spawn(fun() -> G(X) end) || X <- lists:seq(1,10)].
[<0.273.0>,<0.274.0>,<0.275.0>,<0.276.0>,<0.277.0>,
<0.278.0>,<0.279.0>,<0.280.0>,<0.281.0>,<0.282.0>]
2
1
4
3
5
8
7
6
10
9
```

Откуда же взялся такой порядок чисел? Добро пожаловать в параллелизм. Порядок процессов невозможно гарантировать, так как все процессы стартуют одновременно. Виртуальная машина Erlang использует множество ухищрений для определения момента запуска того или иного процесса, чтобы каждому процессу гарантированно хватило времени для старта. Множество сервисов в Erlang реализовано в виде процессов, включая оболочку, в которой вы вводите команды. Системе приходится балансировать между вашими процессами и процессами, которые необходимы ей самой. Это может явиться причиной странного порядка запускаемых процессов.

Замечание: если включить или отключить симметричное мультипроцессирование, то результаты не будут сильно отличаться. Можете сами в этом убедиться, запустив виртуальную машину Erlang командой `$ erl -smp disable`.

Чтобы узнать, используется ли SMP в запущенной виртуальной машине, запустите новую VM без каких-либо параметров и обратите внимание на первую выведенную строку. Если вы видите там такой текст: [smp:2:2] [rq:2], то в вашей виртуальной машине включена поддержка SMP, и 2 очереди исполнения (rq, или планировщики) работают на двух ядрах. Если вы видите только [rq:1], то поддержка SMP отключена.

Знайте – [smp:2:2] означает, что вам доступны два ядра с двумя планировщиками. [rq:2] означает, что активны две очереди исполнения. В более ранних версиях Erlang можно было иметь множество планировщиков, но у всех у них была единственная общая очередь исполнения. Начиная с версии R13B, по умолчанию для каждого планировщика выделяется своя очередь исполнения. Это способствует улучшению параллелизма.

Я хочу доказать вам, что оболочка реализована как обычный процесс, и для этого я буду использовать встроенную функцию `self/0`, которая возвращает pid текущего процесса:

```
6> self().
<0.41.0>
7> exit(self()).
** exception exit: <0.41.0>
8> self().
<0.285.0>
```

pid поменялся из-за перезапуска процесса. Подробности работы этого механизма мы рассмотрим позже. А сейчас нам необходимо разобраться в более простых вещах. Первым делом рассмотрим, как пересылать сооб-

щения, ведь никому не хочется постоянно выводить результаты работы процессов на экран, а затем вручную вводить их в других процессах (по крайней мере, я это делать совершенно точно не хочу).

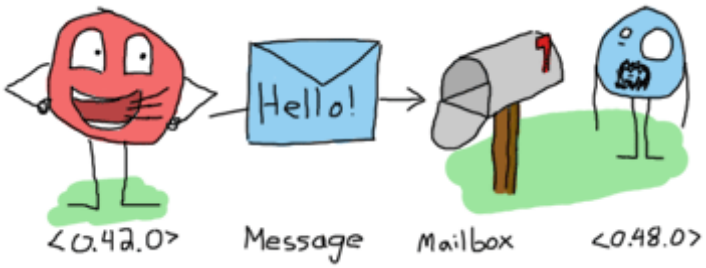
Для пересылки сообщений нам понадобится следующий элемент: оператор `!`, известный также как *bang*-символ. Слева он принимает в качестве параметра pid процесса, а справа должен находиться любой терм Erlang. Затем указанный терм пересылается процессу, который обозначен указанным pid, и тот получает к терму доступ:

```
9> self() ! hello.
hello
```

Сообщение уже попало в почтовый ящик процесса, но ещё не было прочитано. Второе `hello` это значение, возвращённое операцией отсылки. Таким способом можно отправлять одно и то же сообщение сразу нескольким процессам:

```
10> self() ! self() ! double.
double
```

Эта запись эквивалентна выражению `self() ! (self() ! double)`. Нужно сделать ремарку насчёт почтового ящика процесса: сообщения хранятся в ящике в том порядке, в котором они были получены. Для прочтения каждое сообщение извлекается из ящика. Опять же, этот процесс немного напоминает пример из введения, рассказывающий о людях, которые пишут друг другу письма.



Вы можете использовать в оболочке команду `flush()` для просмотра содержимого текущего почтового ящика:

```
11> flush().
Shell got hello
Shell got double
Shell got double
ok
```

Эта функция – просто средство для вывода полученных сообщений. То есть, поместить результат работы процесса в переменную мы всё равно не сможем, но теперь мы хотя бы знаем как отослать сообщение процессу, а потом проверить, что оно получено.

От пересылки сообщений, которые никто не прочитает, толку столько же, сколько от написания эмо-поэзии, то есть не очень много. Вот почему нам нужно выражение `receive`. Не будем слишком долго рассиживаться в оболочке – напишем короткую программу о дельфинах, используя новую конструкцию:

```
-module(dolphins).
-compile(export_all).

dolphin1() ->
    receive
        do_a_flip ->
            io:format("How␣about␣no?~n");
        fish ->
            io:format("So␣long␣and␣thanks␣for␣all␣the␣fish!~n");
        _ ->
            io:format("Heh,␣we're␣smarter␣than␣you␣humans.~n")
    end.
```

Как видите, `receive` похожа синтаксически на `case...of`. И они даже работают совершенно одинаково, за исключением того, что receive связывает не выражения между `case` и `of`, а переменные, извлечённые из сообщений. В receive можно использовать стражи:

```
receive
    Pattern1 when Guard1 -> Expr1;
    Pattern2 when Guard2 -> Expr2;
    Pattern3 -> Expr3
end
```

Теперь этот модуль можно скомпилировать, запустить и начать переговоры с дельфинами:

```
11> c(dolphins).
{ok,dolphins}
12> Dolphin = spawn(dolphins, dolphin1, []).
<0.40.0>
13> Dolphin ! "oh,␣hello␣dolphin!".
Heh, we're␣smarter␣than␣you␣humans.
"oh,␣hello␣dolphin!"
14>␣Dolphin␣!␣fish.
fish
15>
```

Здесь мы знакомимся с новым способом создания процессов при помощи `spawn/3`. Эта модификация spawn принимает не одну функцию, а модуль, функцию и её аргументы. После запуска функции происходит следующее:

1. Исполнение доходит до выражения `receive`. Так как почтовый ящик процесса пуст, наш дельфин ждёт появления сообщения;

2. Получено сообщение «oh, hello dolphin!». Функция пытается провести сопоставление с образцом `do_a_flip`. Сопоставление не удаётся, и происходит попытка сопоставления с `fish`, которая тоже заканчивается неудачей. Наконец, сообщение доходит до универсального шаблона (`_`) и сопоставление завершается удачно.

3. Процесс выводит сообщение «Heh, we're smarter than you humans.»

Необходимо заметить, что если первое посланное нами выражение сработало, то на второе процесс `<0.40.0>` никак не отреагировал. Это произошло из-за того, что после вывода «Heh, we're smarter than you humans.» наша функция завершилась, а вместе с ней завершился и процесс. Придётся нам перезапустить дельфина:

```
8> f(Dolphin).
ok
9> Dolphin = spawn(dolphins, dolphin1, []).
<0.53.0>
10> Dolphin ! fish.
So long and thanks for all the fish!
fish
```

И на этот раз сообщение о рыбе срабатывает. Но ведь было бы полезнее, если бы мы вместо вывода посредством функции `io:format/2` могли получить от дельфина ответ? Конечно, да! (зачем я вообще спрашиваю?) Чуть раньше в этой главе я упомянул, что мы можем узнать, получил ли процесс наше сообщение, только если он пошлёт нам ответное. Нашему дельфиньему процессу нужно знать кому отвечать. Всё как на почте. Если мы хотим, чтобы адресат нам ответил, нужно сообщить ему наш адрес. В рамках Erlang это осуществляется путём упаковки pid процесса в кортеж. В результате получается сообщение, которое выглядит примерно так: `{Pid, Message}`. Давайте создадим новую функцию-дельфина, которая будет принимать такие сообщения:

```
dolphin2() ->
    receive
        {From, do_a_flip} ->
```

```
            From ! "How about no?";
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n")
    end.
```

Как видите, мы больше не принимаем сообщения `do_a_flip` и `fish`. Теперь мы ожидаем переменную *From*. Именно там будет находиться идентификатор процесса.

```
11> c(dolphins).
{ok,dolphins}
12> Dolphin2 = spawn(dolphins, dolphin2, []).
<0.65.0>
13> Dolphin2 ! {self(), do_a_flip}.
{<0.32.0>,do_a_flip}
14> flush().
Shell got "How about no?"
ok
```

Вроде бы работает неплохо. Мы можем получать ответы на посылаемые нами сообщения (к каждому сообщению мы должны прикладывать свой адрес), но нам всё так же приходится стартовать новый процесс для каждого вызова. Эту проблему мы можем решить при помощи рекурсии. Нам нужна функция, которая будет бесконечно вызывать саму себя и постоянно ожидать прихода сообщений. Эту задачу будет реализовывать `dolphin3/0`:

```
dolphin3() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?",
            dolphin3();
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n"),
            dolphin3()
    end.
```

В этом примере и универсальное условие, и `do_a_flip` входят в цикл при помощи `dolphin3/0`. Заметьте, что функция оптимизируется в хвостовую рекурсию, и стек она раздувать не будет. Процесс-дельфин будет находиться в бесконечном цикле, пока ему будут приходить только эти сообщения. Но как только мы пошлём ему сообщение `fish` – процесс сразу же остановится:

```
15> Dolphin3 = spawn(dolphins, dolphin3, []).
<0.75.0>
16> Dolphin3 ! Dolphin3 ! {self(), do_a_flip}.
{<0.32.0>,do_a_flip}
17> flush().
Shell got "How about no?"
Shell got "How about no?"
ok
18> Dolphin3 ! {self(), unknown_message}.
Heh, we're smarter than you humans.
{<0.32.0>,unknown_message}
19> Dolphin3 ! Dolphin3 ! {self(), fish}.
{<0.32.0>,fish}
20> flush().
Shell got "So long and thanks for all the fish!"
ok
```

Вот, пожалуй и всё, что касается dolphins.erl. Как видите, эта программа соответствует ожидаемому нами поведению. Она однократно отвечает на каждое сообщение и продолжает работать, пока не получит сообщение `fish`. Дельфину надоели наши нелепые выходки, и он покидает нас навсегда.

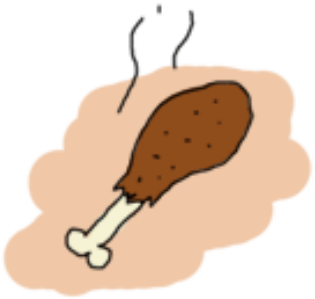
Ну вот, теперь вы знакомы с основой всех конкурентных механизмов Erlang. Мы рассмотрели процессы и примитивную передачу сообщений, но чтобы писать действительно полезные и надёжные программы,

нам понадобится ещё много чего изучить. Некоторые вопросы мы затронем в следующей главе, и продолжим их изучение в последующих.

Глава 13

Снова о многопроцессорности

13.1 Предъявите ваше состояние



Примеры, показанные в предыдущей главе, годились для использования в качестве демонстрационного материала, но с таким ограниченным инструментарием далеко не уйдёшь. Нет, примеры неплохие, но от процессов и акторов пользы мало, когда они представлены только функциями и сообщениями. Для устранения этого недостатка нам необходимо уметь сохранять в процессе состояние.

Давайте, для начала, создадим функцию в новом модуле kitchen.erl, которая позволит процессу выполнять функции холодильника. Процессу разрешено совершать две операции: хранить еду в холодильнике и вынимать её оттуда. Вынимать можно только ту еду, которая была заранее помещена в холодильник. Пусть основой нашего процесса будет следующая функция:

```
-module(kitchen).
-compile(export_all).

fridge1() ->
    receive
        {From, {store, _Food}} ->
        From ! {self(), ok},
        fridge1();
    {From, {take, _Food}} ->
        %% uh....
        From ! {self(), not_found},
```

```
        fridge1();
    terminate ->
        ok
    end.
```

Что-то здесь не так. Когда мы делаем запрос на хранение еды, процесс возвращает результат *ok*, но фактически еда нигде не сохраняется. Будет вызвана функция *fridge1()*, и она начнёт исполняться с чистого листа, без сохранённого состояния. Также очевидно, что когда мы просим процесс извлечь еду из холодильника, её просто неоткуда взять, и остаётся просто вернуть в качестве результата *not_found*. Ясно, что для хранения и извлечения провизии нам необходимо добавить в функцию состояние.

Благодаря рекурсии, состояние процесса может целиком содержаться в параметрах, которые передаются в функцию. Для нашего процесса-холодильника можно хранить весь провиант в виде списка, и когда кто-нибудь захочет поесть, мы сможем поискать в нём необходимый продукт:

```
fridge2(FoodList) ->
    receive
        {From, {store, Food}} ->
            From ! {self(), ok},
            fridge2([Food|FoodList]);
        {From, {take, Food}} ->
            case lists:member(Food, FoodList) of
                true ->
                    From ! {self(), {ok, Food}},
                    fridge2(lists:delete(Food, FoodList));
                false ->
                    From ! {self(), not_found},
                    fridge2(FoodList)
            end;
        terminate ->
            ok
    end.
```

Сразу можно заметить, что `fridge2/1` принимает один аргумент -- *FoodList*. Когда мы пошлём сообщение вида `{From, {store, Food}}` -- функция добавит значение *Food* в *FoodList* перед следующей итерацией. На следующей итерации рекурсивного вызова можно будет извлечь из списка тот же самый элемент, который мы туда поместили ранее. Я даже реализовал эту возможность. Функция использует `lists:member/2` для проверки наличия `Food` в `FoodList`. В зависимости от результата, полученный элемент либо пересылается вызывающему процессу (и удаляется из *FoodList*), либо получателю отсылается атом *not_found*:

```
1> c(kitchen).
{ok,kitchen}
2> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.51.0>
3> Pid ! {self(), {store, milk}}.
{<0.33.0>,{store,milk}}
4> flush().
Shell got {<0.51.0>,ok}
ok
```

Функция хранения продуктов в холодильнике вроде бы работает. Теперь попробуем поместить туда различные продукты, а затем их извлечь.

```
5> Pid ! {self(), {store, bacon}}.
{<0.33.0>,{store,bacon}}
6> Pid ! {self(), {take, bacon}}.
{<0.33.0>,{take,bacon}}
7> Pid ! {self(), {take, turkey}}.
{<0.33.0>,{take,turkey}}
8> flush().
Shell got {<0.51.0>,ok}
Shell got {<0.51.0>,{ok,bacon}}
Shell got {<0.51.0>,not_found}
ok
```

В соответствии с нашими ожиданиями, мы можем достать из холодильника бекон, потому что мы его туда поместили первым по счёту (вместе с молоком и пищевой содой), но когда мы просим процесс-холодильник достать немного мяса индейки, у него ничего не выходит. Поэтому мы получаем последнее сообщение `{<0.51.0>,not_found}`.

13.2 Мы любим послания, но держим их в секрете

В предыдущем примере немного раздражает то, что программист, который собирается воспользоваться холодильником, должен иметь представление о протоколе, который был изобретён специально для этого процесса. Что ведёт к усложнению без видимой на то необходимости. Этот недостаток можно устранить, поместив обработку сообщений в функции, которые эти сообщения получают и отправляют:

```
store(Pid, Food) ->
    Pid ! {self(), {store, Food}},
    receive
```

```
        {Pid, Msg} -> Msg
    end.

take(Pid, Food) ->
    Pid ! {self(), {take, Food}},
    receive
        {Pid, Msg} -> Msg
    end.
```

В таком виде взаимодействие с процессом выглядит гораздо опрятнее:

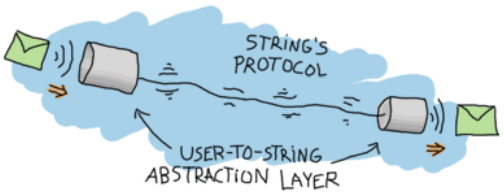
```
9> c(kitchen).
{ok,kitchen}
10> f().
ok
11> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.73.0>
12> kitchen:store(Pid, water).
ok
13> kitchen:take(Pid, water).
{ok,water}
14> kitchen:take(Pid, juice).
not_found
```

Нам больше не нужно переживать о том, по какому принципу работают сообщения, нужно ли нам посылать `self()` или какой-то конкретный атом из числа `take` или `store`. Мы должны знать лишь pid и то, какую функцию нужно вызвать. Это позволяет спрятать подальше от глаз всю грязную работу и облегчает создание процесса-холодильника.

Осталось только спрятать саму необходимость порождения процесса. Мы позаботились о сокрытии сообщений, но обязанности по созданию процесса мы возложили на программиста. Я добавлю следующую функцию `start/1`:

```
start(FoodList) ->
    spawn(?MODULE, fridge2, [FoodList]).
```

`?MODULE` – это макрос, который возвращает имя текущего модуля. Казалось бы, запись такой функции не несёт никаких преимуществ, но на самом деле некоторые преимущества есть. Самым главным из них является согласованность с вызовами функций



`take/2` и `store/2`. Все операции процесса-холодильника теперь обрабатываются модулем kitchen. Если вам необходимо добавить запись о времени старта процесса-холодильника, или запустить второй процесс (пусть это будет морозильник), то это можно легко сделать внутри нашей функции `start/1`. Но если порождение процесса будет делать пользователь при помощи `spawn/3`, то в каждом месте, где запускается холодильник, мы будем обязаны добавить вызовы новых функций. Здесь очень просто совершить ошибку, а ошибки это плохо.

Поглядим на функцию в деле:

```
15> f().
ok
16> c(kitchen).
{ok,kitchen}
17> Pid = kitchen:start([rhubarb, dog, hotdog]).
<0.84.0>
18> kitchen:take(Pid, dog).
{ok,dog}
19> kitchen:take(Pid, dog).
not_found
```

Ура! Собака выбралась из холодильника, и наша абстракция готова к использованию!

13.3 Тайм-аут

Давайте попробуем сделать что-нибудь с использованием команды `pid(A,B,C)`, которая позволяет нам преобразовать 3 целых числа A, B и C в pid. Попробуем нарочно передать функции `kitchen:take/2` несуществующий pid:

```
20> kitchen:take(pid(0,250,0), dog).
```

Ой, оболочка зависла. Зависание связано с реализацией функции `take/2`. Попробуем понять, почему так получилось, и для начала рассмотрим события, которые происходят в случае нормального выполнения:

1. От вас (от оболочки) пересылается сообщение процессу-холодильнику, с указанием сохранить еду;

2. Ваш процесс переключается в режим приёма и ожидает появления нового сообщения;

3. Холодильник сохраняет элемент и посылает вашему процессу сообщение 'ok';

4. Ваш процесс получает это подтверждение и продолжает заниматься своими делами.

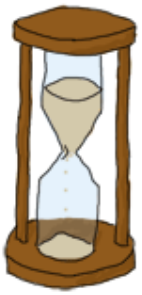
А вот что происходит при зависании оболочки:

1. От вас (от оболочки) к неизвестному процессу уходит сообщение с указанием сохранить еду;

2. Ваш процесс переключается в режим приёма и ожидает новое сообщение;

3. Неизвестный процесс не существует вовсе, либо не ожидает ваше сообщение, и после его получения ничего с ним не делает;

4. Процесс вашей оболочки застревает в режиме приёма.



Досадно, особенно если учесть, что эту ошибку нельзя обработать. Ничего плохого не происходит, программа просто находится в режиме ожидания. Как правило, любой код, имеющий дело с асинхронными операциями (а именно так организована передача сообщений в Erlang), должен иметь возможность прекращать ожидание по прошествии некоторого времени, если информация так и не была получена. Похожую операцию проделывает веб-браузер, когда загрузка страницы или изображения продолжается слишком долго. Вы и сами проделываете то же самое, если при совершении телефонного звонка абонент долго не берёт трубку, или когда кто-то не приходит на встречу вовремя. Само собой, в Erlang на этот случай имеется соответствующий механизм, который является частью конструкции `receive`.

```
receive
    Match -> Expression1
after Delay ->
    Expression2
end.
```

За выполнение того, о чём мы только что говорили, отвечает часть кода, которая находится между `receive` и `after`. Код в `after` будет выполнен, если за время равное *Delay* (целое значение, заданное в миллисекундах) не будет получено сообщение, которое соответствует шаблону *Match*. В этом случае выполняется *Expression2*.

Напишем пару новых функций интерфейса `store2/2` и `take2/2`, которые ведут себя абсолютно так же как `store/2` и `take/2`, но будут прекращать ожидание после 3-х секунд:

```
store2(Pid, Food) ->
    Pid ! {self(), {store, Food}},
    receive
        {Pid, Msg} -> Msg
    after 3000 ->
        timeout
    end.

take2(Pid, Food) ->
    Pid ! {self(), {take, Food}},
    receive
        {Pid, Msg} -> Msg
    after 3000 ->
        timeout
    end.
```

Теперь вы можете вывести оболочку из зависшего состояния нажатием Ctrl-G, и испытать новые интерфейсные функции:

```
User switch command
 --> k
 --> s
 --> c
Eshell V5.7.5  (abort with ^G)
1> c(kitchen).
{ok,kitchen}
2> kitchen:take2(pid(0,250,0), dog).
timeout
```

Вот теперь всё работает.

Замечание: я говорил, что `after` принимает значения только в миллисекундах, но ему также можно передавать атом `infinity`. Во многих случаях от этой возможности мало проку (тогда выражение `after` можно было бы полностью удалить), но иногда её используют, если программист имеет возможность передать время ожидания в функцию, в которой предполагается получение результата. Ожидание в такой ситуации действительно может продолжаться вечно, если того захочет программист.

Помимо случаев, когда нужно прекращать слишком долгое ожидание, такие таймеры могут пригодиться и в других ситуациях. Простым примером может послужить то, как работает функция `time:sleep/1`, которую мы использовали ранее. Вот как она реализована (поместим её в

новый модуль multiproc.erl):

```
sleep(T) ->
    receive
    after T -> ok
    end.
```

В `receive` не будет найдено совпадение ни с одним из сообщений, так как для сопоставления с образцом не задан шаблон. По истечении периода T просто будет выполнена `after` часть конструкции.

А вот ещё один особый случай, в котором время таймаута равно 0:

```
flush() ->
    receive
        _ -> flush()
    after 0 ->
        ok
    end.
```

В такой ситуации виртуальная машина Erlang будет пытаться найти сообщение, совпадающее с одним из указанных шаблонов. В вышеприведённом случае может подойти что угодно. Функция `flush/0` будет рекурсивно вызывать себя до тех пор, пока в почтовом ящике не закончатся сообщения. Когда ящик будет опустошён, выполнится код `after 0 -> ok`, и функция завершит выполнение.

13.4 Выборочный приём сообщений

Рассмотренная концепция «сброса» (flushing) позволяет реализовать *выборочный приём сообщений*, который даёт возможность назначать приоритет полученным сообщениям при помощи вложенных вызовов:

```
important() ->
    receive
        {Priority, Message} when Priority > 10 ->
            [Message | important()]
    after 0 ->
        normal()
    end.

normal() ->
    receive
        {_, Message} ->
            [Message | normal()]
    after 0 ->
        []
```

```
    end.
```

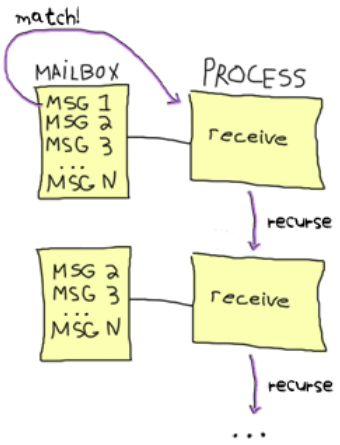
Эта функция построит список всех сообщений. Первыми в нём будут стоять элементы с приоритетом больше 10:

```
1> c(multiproc).
{ok,multiproc}
2> self() ! {15, high}, self() ! {7, low}, self() ! {1, low},
   self() ! {17, high}.
{17,high}
3> multiproc:important().
[high,high,low,low]
```

Я использовал в коде условие `after 0`, а значит процесс получит все сообщения до последнего. Но он будет получать сообщения с приоритетом выше 10, совершенно не принимая в расчёт остальные послания. А они будут накапливаться в `normal/0`.

Если такой способ обработки вас заинтересовал, то имейте в виду. что иногда он может оказаться небезопасным из-за особенностей работы выборочного приёма сообщений в Erlang.

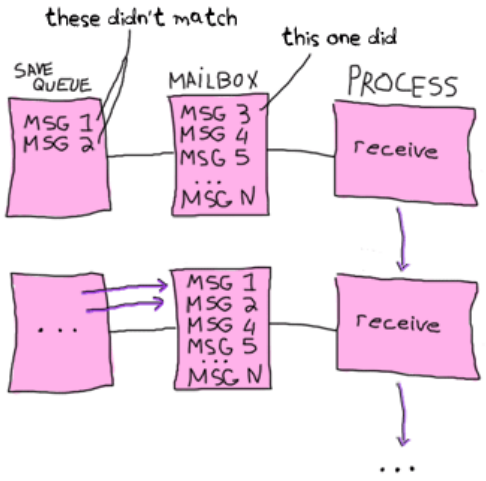
Когда процессу отсылаются сообщения, они хранятся в почтовом ящике до тех пор, пока процесс не прочитает его и не сопоставит с шаблоном. Как было сказано в 12 предыдущей главе, порядок хранения сообщений определяется порядком их получения. Поэтому когда вы попытаетесь провести над сообщением операцию сопоставления, она будет проведена для сообщения, полученного раньше всех остальных.



Самое старое сообщение сопоставляется с каждым шаблоном в `receive`, пока совпадение не будет найдено. После этого сообщение изымается из почтового ящика, и код процесса исполняется в обычном порядке до сле-

дующего `receive`. При обработке очередного `receive`, виртуальная машина будет искать в почтовом ящике самое старое сообщение (следующее за тем, которое мы изъяли) и так далее.

Когда данное сообщение не совпадает ни с одним шаблоном, его помещают в *очередь хранения (save queue)* и переходят к следующему. Если второе сообщение совпадает с одним из шаблонов, то первое снова помещается в вершину почтовой очереди, и будет обработано повторно чуть позже.



Благодаря этому механизму, вы сможете сосредоточиться только на нужных вам сообщениях. Способ игнорирования некоторых сообщений с целью последующей их обработки, описанный выше, как раз и является сущностью *выборочного приёма сообщений (selective receives (selective receives))*. Не смотря на полезность этого метода, у него есть один недостаток: если процессу приходит много сообщений, которые никогда не понадобятся, то поиск нужных сообщений будет отнимать всё больше и больше времени (и размер процесса также будет увеличиваться).

Взгляните на рисунок, изображённый выше. Представьте, что нам необходимо 367-е сообщение, но предыдущие 366 – просто мусор, и наш код их проигнорировал. Для получения 367-го сообщения, процессу необходимо провести сопоставление для 366-ти предшествующих сообщений. Как только сопоставление проведено, и все ненужные сообщения были помещены в очередь, извлекается 367-е сообщение, а все предыдущие 366 возвращаются обратно в почтовый ящик. Следующее полезное сообщение может погрузиться ещё глубже и его придётся искать ещё дольше.

Эта проблема часто является причиной снижения производительности в Erlang. Если ваше приложение исполняется медленно, и вы знае-

те, что оно пересылает много сообщений, то проблема может быть именно в этом.

Когда выборочный приём сообщений является причиной серьёзного замедления вашего кода, первым делом спросите себя, почему вы получаете сообщения, которые вам не нужны? Посылаются ли эти сообщения нужным процессам? Используете ли вы правильные шаблоны? Может быть, сообщения отформатированы неверно? Может, там, где необходимо использовать множество процессов, вы используете один? Ответы на эти вопросы могут оказаться решением вашей проблемы.

Пытаясь уменьшить риск появления ненужных сообщений, загрязняющих почтовый ящик процесса, программисты на Erlang зачастую принимают меры, которые предотвращают появление таких событий. Стандартный приём для такого случая выглядит следующим образом:

```
receive
    Pattern1 -> Expression1;
    Pattern2 -> Expression2;
    Pattern3 -> Expression3;
    ...
    PatternN -> ExpressionN;
    Unexpected ->
        io:format("unexpected␣message␣~p~n", [Unexpected])
end.
```

Этот код заботится о том, чтобы любое сообщение совпало хотя бы с одним шаблоном. Переменная *Unexpected* будет захватывать любое неожиданное сообщение, изымать его из почтового ящика и отображать предупреждение об этом событии. Скорее всего, вы захотите сохранить сообщение при помощи какого-либо регистрирующего средства, чтобы впоследствии иметь возможность получить о нём информацию. Было бы досадно навсегда потерять сообщения, отосланные по ошибке, а потом недоумевать, почему какой-то из процессов не получил то, что должен был получить.

Если вам необходимо следить за приоритетом сообщений, и использование универсального шаблона не представляется возможным, то вы можете реализовать min-кучу или использовать модуль `gb_trees` для сохранения каждого полученного сообщения (проследите, чтобы приоритет стоял в ключе первым – так он будет задействован при сортировке). Затем вы сможете просто извлекать наименьший или наибольший элемент в структуре, в соответствии с вашими пожеланиями.

В большинстве случаев этот приём позволит извлекать сообщения с установленным приоритетом более эффективно, чем при помощи избирательного приёма сообщений. Но если большинство получаемых сооб-

щений имеют наибольший возможный приоритет, то этот метод может оказать на код замедляющее действие. Как обычно, перед оптимизацией необходимо провести измерения и профилирование кода.

Замечание: начиная с версии R14A, в компиляторе Erlang появилась новая оптимизация. Она упрощает избирательный приём сообщений для некоторых специальных случаев двустороннего взаимодействия процессов. Пример её использования можно найти в функции `optimized/1` модуля multiproc.erl.

Чтобы эта оптимизация сработала, в функции необходимо создать ссылку (`make_ref()`) и отослать её в сообщении. Затем, в той же самой функции осуществляется избирательный приём сообщений. Так как ни одно сообщение не может совпасть с шаблоном, если оно не содержит сгенерированную ссылку, то компилятор автоматически заставляет виртуальную машину пропускать сообщения, полученные до момента создания этой ссылки.

Заметьте, что вы не должны пытаться привести ваш код в соответствие с оптимизациями. Разработчики Erlang ищут часто используемый шаблонный код, и ускоряют его исполнение. Если ваш код достаточно идиоматичен, то оптимизации сами к вам придут. Не наоборот.

Теперь у нас есть представление о рассмотренных в этой главе концепциях, и на следующем этапе мы увидим как осуществляется обработка ошибок для нескольких процессов.

Глава 14

Ошибки и процессы

14.1 Связи

Связи – это особые взаимоотношения, которые могут быть установлены между двумя процессами. Когда один из процессов, который участвует в таких отношениях, умирает от неожиданного броска, ошибки или завершения (см. 9 Ошибки и исключения), то связанный с ним процесс тоже завершается.

Эта концепция может оказаться полезной, когда необходимо как можно быстрее завершить процесс, чтобы предотвратить появление ошибок. Если процесс, в котором появилась ошибка, завершился аварией, а процессы, которые на него полагаются, продолжили работать, то все эти зависимые процессы должны что-то предпринять. Обычно приемлемым вариантом развития событий можно считать остановку и перезапуск всей группы процессов. Именно это и позволяют нам сделать связи.

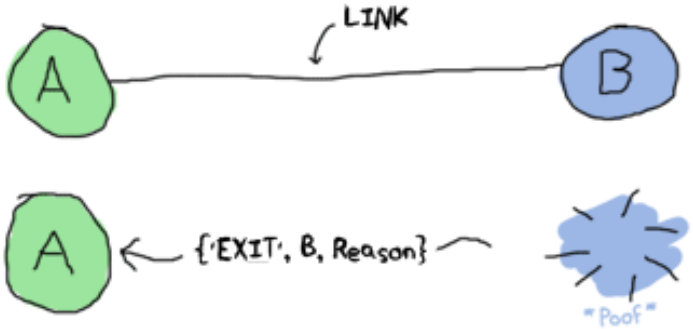
Для создания связи между парой процессов, в Erlang существует базовая функция link/1, принимающая в качестве аргумента Pid. После запуска эта функция создаст связь между текущим процессом и процессом, который отождествляется с указанным *Pid*. Для разрушения связи используют функцию unlink/1. При аварийном завершении одного из связанных процессов, отсылается сообщение особого вида, уведомляющее о произошедшем событии. Если процесс умирает по естественным причинам (читай: завершает исполнение своих функций), то такое сообщение не отсылается. Поговорим для начала об этой новой функции как об элементе модуля linkmon.erl:

```
myproc() ->
    timer:sleep(5000),
    exit(reason).
```

Если вы попробуете исполнить следущие вызовы функций (и сделаете между каждой командой spawn пятисекундную паузу), то увидите, что оболочка завершится с ошибкой 'reason', только если между двумя процессами была установлена связь.

```
1> c(linkmon).
{ok,linkmon}
2> spawn(fun linkmon:myproc/0).
<0.52.0>
3> link(spawn(fun linkmon:myproc/0)).
true
** exception error: reason
```

Изобразим это на картинке:



Для захвата сообщения {'EXIT', B, Reason} не получится использовать стандартную структуру try ... catch. Для этого существуют другие средства, которые мы рассмотрим позже.

Важно отметить, что связи можно также использовать для организации больших групп процессов, которые должны прекращать исполнения как единая группа, все вместе:

```
chain(0) ->
    receive
        _ -> ok
    after 2000 ->
        exit("chain dies here")
    end;
chain(N) ->
    Pid = spawn(fun() -> chain(N-1) end),
    link(Pid),
    receive
        _ -> ok
    end.
```

Эта функция принимает целое число *N*, запускает *N* связанных между собой процессов. Для передачи *N-1* аргумента следующему процессу в «цепи» я оборачиваю вызов в анонимную функцию, чтобы она перестала принимать аргументы. Подобный эффект можно получить при помощи вызова `spawn(?MODULE, chain, [N-1])`.

В этом примере я создам множество связанных процессов, каждый из которых будет умирать сразу после завершения процесса, идущего следом:

```
4> c(linkmon).
{ok,linkmon}
5> link(spawn(linkmon, chain, [3])).
true
** exception error: "chain dies here"
```

Как видите, оболочка получает сообщение о смерти от одного из процессов. Вот подробное описание завершения запущенных процессов и уничтожения связей:

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] == *dead*
*dead, error message shown*
[shell] <-- restarted
```

После завершения процесса, который исполняет функцию `linkmon:chain(0)`, ошибка передаётся по цепи связей, и в результате из-за неё умирает процесс оболочки. Авария могла произойти в любом из связанных процессов. Связи работают в обе стороны, поэтому для завершения всей группы процессов достаточно умереть лишь одному из них.

Замечание: если вам необходимо убить из оболочки какой-либо процесс, то это можно сделать при помощи функции exit/2. Её можно вызвать следующим образом: `exit(Pid, Reason)`. Можете попробовать ею воспользоваться, если хотите.

Замечание: связи не накапливаются. Если вы вызвали `link/1` 15 раз для одной и той же пары процессов, то между ними будет существовать лишь одна связь, и для её разрушения будет достаточно однократного вызова `unlink/1`.

Нужно сказать, что вызовы `link(spawn(Function))` или `link(spawn(M,F,A))` совершаются в несколько шагов. Иногда процесс может умереть до того как была установлена связь, и это спровоцирует неожиданное поведение исполняемого кода. На такой случай в языке существует функция

spawn_link/1-3. Она принимает те же самые параметры, что и `spawn/1-3`, создаёт процесс и связывает его, так же как это делает `link/1`, но вся цепь действий осуществляется одной атомарной операцией (все действия объединяются в одно, и результатом их исполнения может стать лишь совокупный успех или неудача, никаких других результатов быть не может). Такой способ создания процессов и их связей считается более надёжным. К тому же, на нём можно сэкономить пару скобок.

14.2 Это ловушка!

Вернёмся к связям и умирающим процессам. Распространение ошибок от процесса к процессу осуществляется методом, похожим на передачу сообщений, но при этом используется их особый тип – сигналы. Сигналы о завершении – это «тайные» сообщения, которые автоматически действуют на процессы, убивая их во время исполнения.

Я уже неоднократно упоминал, что надёжность приложения зависит от его умения быстро убивать и перезапускать процессы. В настоящее время мы можем использовать связи в роли орудия убийства. Осталось разобраться с перезапуском.

Чтобы перезапустить процесс, сначала нужно как-то узнать, что он умер. Мы можем сделать это, добавив поверх связей ещё один слой (вкусную глазурь на торте), содержащий концепцию, которая называется *системные процессы* (system processes). Системные процессы, по сути, это обычные процессы, но они могут конвертировать сигналы о завершении в обычные сообщения. Делают они это при помощи вызова `process_flag(trap_exit, true)` в работающем процессе. Ничто не сможет сказать об этом механизме больше, чем пример. Поэтому сейчас мы его и рассмотрим. Я немного видоизменю пример с chain, поместив в начале системный процесс:

```
1> process_flag(trap_exit, true).
true
2> spawn_link(fun() -> linkmon:chain(3) end).
<0.49.0>
3> receive X -> X end.
{'EXIT',<0.49.0>,"chain dies here"}
```

Ага! Это уже интереснее. Вернёмся к нашей иллюстрации. Теперь картина событий выглядит приблизительно так:

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] <-- {'EXIT,Pid,"chain dies here"} -- *dead*
[shell] <-- still alive!
```

Вот он, механизм позволяющий быстро перезапускать процессы. Используя системные процессы при написании программ, можно легко создать процесс, единственная функция которого – следить за тем, чтобы все процессы оставались живы, и рестартовать их в случае аварии. Мы поговорим об этом подробнее в следующей главе, когда будем применять этот приём всерьёз.

А сейчас я хочу вернуться к функциям для работы с исключениями, которые мы рассмотрели в главе 9 об ошибках и исключениях, и увидим как они ведут себя в процессах, которые улавливают факт завершения других процессов. Сначала зададим основу эксперимента, не используя при этом системный процесс. Я последовательно приведу результаты, которые возвращают непойманные броски, ошибки и завершения (exits) в соседних процессах:

- **Источник исключения:** `spawn_link(fun() -> ok end)`
  Результат, который не был перехвачен: - nothing -
  Пойманный результат: {'EXIT', <0.61.0>, normal}
  Процесс завершился в обычном порядке, без проблем. Заметьте, что этот результат выглядит почти так же как и результат `catch_exit(normal)`, с тем лишь отличием, что для идентификации проблемного процесса в кортеж добавлен PID.

- **Источник исключения:** `spawn_link(fun() -> exit(reason) end)`
  Результат, который не был перехвачен: ** exception exit: reason
  Пойманный результат: {'EXIT', <0.55.0>, reason}
  Процесс завершился по причине, указанной пользователем.
  Если завершение (exit) не было поймано, то в этой ситуации процесс аварийно прекращает работу.
  В противном случае вы получите сообщение, указанное выше.

- **Источник исключения:** `spawn_link(fun() -> exit(normal) end)`
  Результат, который не был перехвачен: - nothing -

Пойманный результат: {'EXIT', <0.58.0>, normal}
Этот вызов эмулирует нормальное завершение процесса. Иногда в ходе обычного исполнения программы нужно убить процесс, не создавая никаких исключений. В этом случае применяют именно такой метод.

- **Источник исключения:** `spawn_link(fun() -> 1/0 end)`
Результат, который не был перехвачен: Error in process <0.44.0> with exit value: {badarith, [{erlang, '/', [1,0]}]}
Пойманный результат: {'EXIT', <0.52.0>, {badarith, [{erlang, '/', [1,0]}]}}
Ошибка (`{badarith, Reason}`) не перехватывается блоком `try ... catch` и передаётся выше, достигая 'EXIT'. С этого момента вызов ведёт себя так же как и `exit(reason)`, с тем отличием, что трассировка стека будет содержать более полную информацию о произошедшем.

- **Источник исключения:** `spawn_link(fun() -> erlang:error(reason) end)`
Результат, который не был перехвачен: Error in process <0.47.0> with exit value: {reason, [{erlang, apply, 2}]}
Пойманный результат: {'EXIT', <0.74.0>, {reason, [{erlang, apply, 2}]}}
Этот вызов почти такой же как `1/0`. Ничего особенного в этом нет, так как `erlang:error/1` создан для использования именно в такой ситуации.

- **Источник исключения:** `spawn_link(fun() -> throw(rocks) end)`
Результат, который не был перехвачен: Error in process <0.51.0> with exit value: {{nocatch, rocks}, [{erlang, apply, 2}]}
Пойманный результат: {'EXIT', <0.79.0>, {{nocatch, rocks}, [{erlang, apply, 2}]}}
Бросок (throw) не перехватывается блоком `try...catch` и переходит в ошибку (error), которая переходит в EXIT. Если процесс не ловит exit, его исполнение заканчивается аварией. В противном случае всё проходит без каких-либо проблем.

Вот, пожалуй, и всё, что я хотел рассказать об обычных исключениях. Если дела идут как обычно – всё проходит нормально. Случается что-то исключительное – умирает процесс, рассылаются различные сигналы.

А ещё есть `exit/2`. Этот вызов – что-то вроде пистолета, который могут использовать процессы в Erlang. Он позволяет процессу убить другой процесс, сохраняя безопасную дистанцию. Вот некоторые вызовы, которые можно делать с его помощью:

- **Источник исключения:** `exit(self(), normal)`
  Результат, который не был перехвачен: ** exception exit: normal
  Пойманный результат: {'EXIT', <0.31.0>, normal}
  Когда вызов `exit(self(), normal)` не улавливает завершения (exits), он действует так же как и `exit(normal)`. В противном случае вы получаете сообщение того же формата, который можно получить при прослушивании связей (links) от умирающих внешних (foreign) процессов.

- **Источник исключения:** `exit(spawn_link(fun() -> timer:sleep(50000) end), normal)`
  Результат, который не был перехвачен: - nothing -
  Пойманный результат: - nothing -
  Фактически, это вызов функции `exit(Pid, normal)`. Ничего полезного эта команда не делает, так как процесс нельзя убить удалённо с причиной `normal` в качестве аргумента.

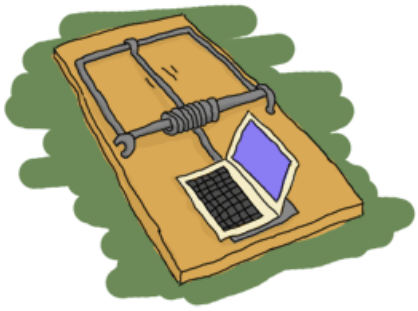
- **Источник исключения:** `exit(spawn_link(fun() -> timer:sleep(50000) end), reason)`
  Результат, который не был перехвачен: ** exception exit: reason
  Пойманный результат: {'EXIT', <0.52.0>, reason}
  Внешний (foreign) процесс завершает себя по причине *reason*. Выглядит это так, как будто внешний процесс вызвал сам для себя `exit(reason)`.

- **Источник исключения:** `exit(spawn_link(fun() -> timer:sleep(50000) end), kill)`
  Результат, который не был перехвачен: ** exception exit: killed
  Пойманный результат: {'EXIT', <0.58.0>, killed}
  По пути от умирающего процесса к его создателю, сообщение неожиданно меняется. Теперь создатель процесса получает `killed` вместо kill. Происходит это потому, что `kill` – особый сигнал завершения. Подробнее об этом чуть позже.

- **Источник исключения:** `exit(self(), kill)`
  Результат, который не был перехвачен: ** exception exit: killed
  Пойманный результат: ** exception exit: killed
  Ого, глядите-ка. Похоже это исключение словить невозможно. Нука проверим.

- **Источник исключения:** `spawn_link(fun() -> exit(kill) end)`
  Результат, который не был перехвачен: ** exception exit: killed
  Пойманный результат: {'EXIT', <0.67.0>, kill}
  Здесь происходит что-то непонятное. Когда какой-либо процесс убивает себя вызовом `exit(kill)`, и мы не ловим завершения (exits)– наш собственный процесс умирает с причиной `killed`. Но если мы перехватываем завершения, этого не происходит.

Хотя большинство причин завершения поддаются перехвату, всё же есть ситуации, в которых вам понадобилось бы жестоко расправиться с неким процессом. Такой процесс, к примеру, может ловить завершения (exits), но в то же время застрял в бесконечном цикле и игнорирует все сообщения. Причина завершения `kill` действует как особый сигнал, который невозможно перехватить. Так мы можем гарантировать, что любой процесс, завершаемый по этой причине, действительно будет мёртв. К `kill` обычно прибегают в крайнем случае, когда больше ничего не срабатывает.



Так как причину завершения `kill` словить невозможно, её необходимо заменять на `killed` до получения другим процессом. Если такую замену не сделать, то каждый процесс, связанный с умирающим, умрёт по той же самой причине `kill`, и, в свою очередь, убьёт своих соседей, и так далее. Смерть будет распространяться каскадами.

Этот факт также объясняет, почему `exit(kill)` выглядит так же как и `killed` при его получении другими связанными процессами (перед получением сигнал видоизменяется, и цепной реакции не происходит), но будучи захвачен локально, он выглядит как `kill`.

Если вас всё это запутало, не переживайте. Множество программистов испытывают те же чувства. Сигналы завершения - странные твари. К счастью, кроме описанных выше ситуаций, существует не так уж много вариантов их использования. После усвоения этих примеров, вам будет ясна большая часть механизмов конкурентного управления ошибками в Erlang.

14.3 Мониторы

Ну, ладно. Может, вам совсем не хочется жестоко убивать процессы. Может, вы не собираетесь разрушить весь мир и погибнуть вместе с ним. Может, вам больше нравится наблюдать. Тогда мониторы придутся вам по вкусу.

Если серьёзно, то мониторы представляют собой особый тип связей, обладающий двумя отличиями:

- они работают в одну сторону;
- их можно накапливать (stack).

Мониторы могут пригодиться, когда какой-либо процесс хочет знать, что происходит с другим процессом, но это знание ни для кого из них не представляет жизненной ценности.

Ещё одна причина, которая приведена в списке -- возможность накопления (stacking) ссылок. На первый взгляд она может показаться бесполезной, но эта характеристика очень ценна при написании библиотек, для функционирования которых необходимо знать, что происходит с другими процессами.

Нужно понимать, что связи в большей степени относятся к средствам организации. При проектировании архитектуры вашего приложения, различным процессам назначаются их функции, между ними строятся зависимости. Некоторые процессы будут присматривать за другими, кто-то из них не сможет существовать без парного процесса и т.д. Обычно эта структура существует как нечто предрешённое, известное наперёд. Именно в такую ситуацию органично вписываются связи, и использовать их в другом окружении совсем не обязательно.

Но что делать, если вы пользуетесь 2-мя или 3-мя разными библиотеками, и всем им необходимо знать, жив процесс или нет? Использование связей может создать проблему при попытке их разрушения. Нужно учитывать, что связи нельзя накапливать, поэтому как только вы разрушите одну связь, больше связей не останется, и все предположения об их существовании, которые делают различные библиотеки, будут нарушены. Хорошего мало. Поэтому вам понадобятся связи, которые можно накапливать, наслаивать поверх друг друга, и мониторы эту проблему

решают. Их можно удалять по отдельности. К тому же, их однонаправленность удобно применять в библиотеках, так как другим процессам о существовании этих библиотек знать совершенно не нужно.

Так как же выглядит монитор? Давайте просто его создадим. Для порождения монитора используют функцию erlang:monitor/2, первым параметром которой является атом *process*, а вторым – pid процесса:

```
1> erlang:monitor(process, spawn(fun() -> timer:sleep(500) end)).
#Ref<0.0.0.77>
2> flush().
Shell got {'DOWN',#Ref<0.0.0.77>,process,<0.63.0>,normal}
ok
```

Каждый раз, когда наблюдаемый процесс прекращает работать, вы получаете сообщение вида `{'DOWN', MonitorReference, process, Pid, Reason}`. Ссылка позволяет прекратить наблюдение за процессом. Помните, что мониторы можно накапливать, так что при желании можно не ограничиваться выключением только одного монитора. Ссылки позволяют следить за каждым по отдельности. Как и для ссылок, для мониторов существует атомарная функция, которая позволяет создать процесс и сразу начать за ним наблюдение – spawn_monitor/1-3:

```
3> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom)
    end end).
{<0.73.0>,#Ref<0.0.0.100>}
4> erlang:demonitor(Ref).
true
5> Pid ! die.
die
6> flush().
ok
```

В этом примере мы прекращаем наблюдать за процессом до его завершения, и поэтому не видим как он умирает. Существует также функция demonitor/2, которая позволяет извлечь чуть больше информации. Вторым параметром ей можно передавать список опций. Их всего лишь две: `info` и `flush`:

```
7> f().
ok
8> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom)
    end end).
{<0.35.0>,#Ref<0.0.0.35>}
9> Pid ! die.
die
10> erlang:demonitor(Ref, [flush, info]).
```

```
false
11> flush().
ok
```

Опция `info` сообщает о том, существовал ли монитор в момент попытки его удаления. Именно поэтому выражение под номером 10 вернуло `false`. При помощи `flush` можно удалить сообщение `DOWN` из почтового ящика, если оно там, конечно, было. После этого функция `flush()` ничего в ящике текущего процесса не найдёт.

14.4 Присваиваем процессам имена

Мы разобрались со связями и мониторами, осталось решить ещё одну проблему. Давайте воспользуемся следующей функцией из модуля linkmon.erl:

```
start_critic() ->
spawn(?MODULE, critic, []).

judge(Pid, Band, Album) ->
    Pid ! {self(), {Band, Album}},
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.

critic() ->
    receive
        {From, {"Rage Against the Turing Machine", "Unit Testify"
            }} ->
            From ! {self(), "They are great!"};
        {From, {"System of a Downtime", "Memoize"}} ->
            From ! {self(), "They're not Johnny Crash but they're
                good."};
        {From, {"Johnny Crash", "The Token Ring of Fire"}} ->
            From ! {self(), "Simply incredible."};
        {From, {_Band, _Album}} ->
            From ! {self(), "They are terrible!"}
    end,
    critic().
```

А теперь представьте, что мы ходим по магазинам, покупаем музыкальные записи. Некоторые альбомы звучат неплохо, но полной уверенности на их счёт у вас нет. Вы решаете позвонить другу – музыкальному критику.

```
1> c(linkmon).
{ok,linkmon}
2> Critic = linkmon:start_critic().
<0.47.0>
3> linkmon:judge(Critic, "Genesis", "The␣Lambda␣Lies␣Down␣on␣
    Broadway").
"They␣are␣terrible!"
```

Из-за солнечной бури (я стараюсь придумать что-то правдоподобное), соединение разрывается:

```
4> exit(Critic, solar_storm).
true
5> linkmon:judge(Critic, "Genesis", "A␣trick␣of␣the␣Tail␣
    Recursion").
timeout
```

Вот досада. Теперь мы не можем получать критические отзывы об интересующих нас альбомах. Чтобы критик оставался на связи, мы напишем простой процесс «супервизор» (supervisor), единственная задача которого – перезапуск критика, если тот прекращает исполнение:

```
start_critic2() ->
    spawn(?MODULE, restarter, []).

restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    receive
        {'EXIT', Pid, normal} -> % not a crash
            ok;
        {'EXIT', Pid, shutdown} -> % manual termination, not a
            crash
            ok;
        {'EXIT', Pid, _} ->
            restarter()
    end.
```

Ответственный за перезапуск будет сам представлен в виде процесса. В свою очередь, он запустит процесс-критик, и если тот когда-либо умрёт по причине аварии, `restarter/0` пройдёт одну итерацию цикла и создаст нового критика. Обратите внимание, что шаблон `{'EXIT', Pid, shutdown}` я добавил как средство для ручного останова критика, на случай если у нас когда-либо появится такая необходимость.

В нашем подходе к решению этой проблемы есть один изъян: у нас нет возможности узнать Pid критика. Поэтому мы не можем просто

ему позвонить, чтобы осведомиться о его мнении. Одним из способов решения этой проблемы в Erlang является именование процессов.

Присвоение имени процессу позволяет вам заменять непредсказуемый pid на атом. Затем этот атом можно использовать при отсылке сообщений совершенно так же как Pid. Для именования процесса используется функция erlang:register/2. Если процесс умирает, он автоматически теряет имя. Для удаления имени вручную можно использовать unregister/1. Получить список зарегистрированных процессов можно при помощи registered/0, а более подробный список возвращает команда оболочки `regs()`. Давайте придадим функции `restarter/0` следующий вид:

```
restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    register(critic, Pid),
    receive
        {'EXIT', Pid, normal} -> % not a crash
            ok;
        {'EXIT', Pid, shutdown} -> % manual termination, not a
            crash
            ok;
        {'EXIT', Pid, _} ->
            restarter()
    end.
```

Как видите, независимо от Pid критика, функция `register/2` всегда будет давать ему имя «critic». Теперь нам нужно убрать из функций абстракции необходимость передачи Pid. Попробуем сделать это так:

```
judge2(Band, Album) ->
    critic ! {self(), {Band, Album}},
    Pid = whereis(critic),
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.
```

В этом примере строка `Pid = whereis(critic)` используется для определения идентификатора процесса-критика, чтобы подставить его в сопоставление с образцом выражения `receive`. Это сопоставление необходимо нам, чтобы удостовериться, что мы будем обрабатывать правильное сообщение (на этот момент в почтовом ящике может быть, скажем, 500 сообщений!) Но этот подход может стать источником затруднений. Вышеприведённый код предполагает, что pid критика не будет изменяться между первыми двумя строками функции. Но вполне можно допустить,

что случится следующее:

1. critic ! Message
2. critic получает сообщение
3. critic отвечает
4. critic умирает
5. в функции whereis происходит сбой
6. critic перезапускается
7. происходит аварийное завершение кода

Нельзя исключать и такой вариант:

1. critic ! Message
2. critic получает сообщение
3. critic отвечает
4. critic умирает
5. critic перезапускается
6. функция whereis получает неверный pid
7. сообщение никогда не пройдёт сопоставление с образцом

Существует вероятность, что если мы не сделаем всё как положено, то ошибка в одном процессе вызовет ошибку в другом. В этом случае значение атома *critic* будет доступно сразу в нескольких процессах. Это называется *разделяемым состоянием*. Проблема в том, что значение *critic* может считываться *и* изменяться разными процессами практически в один и тот же момент. Это приводит данные к противоречивому виду и может вызвать ошибки в программном обеспечении. Такое состояние обычно называют «состоянием гонки» *race condition*. Они очень опасны тем, что зависят от распределения событий во времени. Практически в любом конкурентном и параллельном языке это распределение зависит от непредсказуемых факторов, таких, например, как загрузка процессора, распределение процессов и того, какие данные обрабатываются вашей программой.

Не забывайтесь: Возможно, вы слышали, что обычно в Erlang нет «состояний гонки» (race conditions) или взаимных блокировок (deadlocks), что, в свою очередь, обеспечивает надёжность параллельного кода. Во многих случаях это действительно так, но никогда не полагайтесь на то, что ваш код действительно абсолютно надёжен. Помимо именованных процессов существует ещё много способов заставить параллельный код исполняться не так как было задумано.

Как пример можно привести доступ к файлам, размещённым на диске компьютера (с целью их модификации), обновление одних и тех же записей в базе данных сразу из нескольких процессов и т.д.

К счастью, код, приведённый выше, сравнительно легко можно исправить. Необходимо лишь предположить, что именованный процесс не всегда будет оставаться тем же, и для идентификации сообщений использовать ссылки (которые можно создавать при помощи `make_ref()`.

Нам нужно заменить функцию `critic/0` на `critic2/0`, а `judge/3` на `judge2/2`:

```
judge2(Band, Album) ->
    Ref = make_ref(),
    critic ! {self(), Ref, {Band, Album}},
    receive
        {Ref, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.

critic2() ->
    receive
        {From, Ref, {"Rage Against the Turing Machine", "Unit
            Testify"}} ->
            From ! {Ref, "They are great!"};
        {From, Ref, {"System of a Downtime", "Memoize"}} ->
            From ! {Ref, "They're not Johnny Crash but they're
                good."};
        {From, Ref, {"Johnny Crash", "The Token Ring of Fire"}}
            ->
            From ! {Ref, "Simply incredible."};
        {From, Ref, {_Band, _Album}} ->
            From ! {Ref, "They are terrible!"}
    end,
    critic2().
```

А затем поменять `restarter/0` таким образом, чтобы он вместо `critic/0` запускал `critic2/0`. Все остальные функции можно оставить без изменений.

Для пользователя эта модификация пройдёт незамеченной. Точнее, он заметит, что мы переименовали функции и изменили несколько параметров, но не узнает подробностей реализации, и причину, по которой произошли эти изменения. Он лишь увидит, что его код стал проще, и что при вызове функций больше не нужно постоянно передавать pid:

```
6> c(linkmon).
{ok,linkmon}
7> linkmon:start_critic2().
<0.55.0>
8> linkmon:judge2("The Doors", "Light my Firewall").
"They are terrible!"
9> exit(whereis(critic), kill).
true
10> linkmon:judge2("Rage Against the Turing Machine", "Unit
    Testify").
"They are great!"
```

Вот теперь, даже если мы убьём критика, сразу же появится новый и решит наши проблемы. Вот она, польза именованных процесов. Если бы вы попытались вызвать `linkmon:judge/2` без зарегистрированного процесса, то в функции оператором `!` была бы выброшена ошибка *bad argument*, которая гарантирует, что процессы, зависящие от именованного процесса, не смогут продолжить исполнение без него.

Замечание: я уже упоминал, что атомы можно использовать для ограниченного круга ситуаций (хоть этот круг и достаточно велик). Никогда не создавайте атомы динамически. Именованные процессы должны использоваться только для важных сервисов, уникальных в рамках одной VM, и эти процессы должны быть запущены на протяжении всего времени работы приложения.

Если вам понадобились именованные процессы, но их существование скоротечно, или их уникальность в рамках VM не может быть обеспечена, то скорее всего их нужно представить в виде группы. По сравнению с использованием динамических имён, связывание и одновременный перезапуск группы процессов в случае аварии, может оказаться более разумным выбором.

В следующей главе мы применим свежеприобретённые знания о конкурентном программировании в Erlang для написания настоящих приложений.

Глава 15

Проектируем конкурентное приложение

Всё это, конечно, здорово. Вы ознакомились с базовыми принципами, но, опять же, мы с самого начала книги занимались лишь игрушечными примерами: калькуляторами, деревьями, ездили из Хитроу в Лондон и т.д. Пора сделать что-нибудь более интересное с точки зрения обучения. Мы напишем небольшое приложение на конкурентном Erlang. Приложение будет простым и взаимодействие с ним будет осуществляться посредством строковых команд. Кроме того, оно будет приносить пользу и его функциональность можно будет наращивать.

Меня нельзя назвать организованным человеком. Я теряюсь в домашних заданиях, благоустройстве квартиры, этой книге, работе, совещаниях, встречах и прочем. У меня есть дюжина списков с задачами, которые я забываю сделать, или просто не нахожу для них времени. Надеюсь, вам тоже иногда нужно напоминать о делах (хоть ваш разум и не блуждает так же часто как мой). Мы напишем приложение, которое уведомляет вас о необходимости что-либо сделать и напоминает о встречах.

15.1 Разбираем задачу

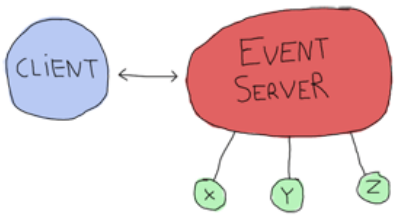
Перво-наперво нужно понять, что мы вообще собираемся сделать. Вы скажете: «Напоминалку». «Ну, конечно», – скажу я. Но это только

начало. Как мы собираемся взаимодействовать с программой? Что она должна для нас делать? Как представить программу при помощи процессов? Как узнать, какие нужно отсылать сообщения?

Как говорится: «Ходить по воде и разрабатывать программное обеспечение по техническому заданию одинаково просто, если и то и другое заморожено.» Так что давайте разработаем технические условия и будем их придерживаться. Наша программа позволит совершать следующие действия:

- Добавлять событие. У событий может быть крайний срок исполнения (момент времени, о котором необходимо предупредить), наименование события и его описание.
- Показывать предупреждение, когда подошло время.
- Отменять событие по его имени.
- Не хранить данные на диске. Для демонстрации архитектурных концепций, которые мы рассмотрим, хранение совершенно излишне. Для настоящего приложения это, конечно никуда не годится, но я покажу, куда можно вставить код, если вам захочется реализовать эту функциональность, и укажу на несколько полезных функций, которые могли бы вам в этом помочь.
- Так как постоянного хранилища у нас нет, нам нужно иметь возможность изменять код во время исполнения.
- Общение с программой будет осуществляться через командную строку, но мы должны предусмотреть возможность последующего расширения средств взаимодействия (добавить, скажем, графический интерфейс, доступ через веб-страницу, через систему обмена мгновенными сообщениями (instant messaging), электронную почту и прочее).

  Для нашей программы я избрал такой способ организации:



Где клиент, сервер событий и x, y, z представлены в виде процессов. Вот что каждый из них может делать:

15.1.1 Сервер событий

- Принимает подписки от клиентов.
- Передаёт уведомления от процессов, генерирующих события, каждому подписчику.
- Принимает сообщения о добавлении событий (и необходимости запуска процессов x, y, z).
- Может принимать сообщения об отмене события, и последующем убийстве процессов-генераторов событий.
- Может быть остановлен клиентом.
- Его код может быть перезагружен из оболочки.

15.1.2 Клиент

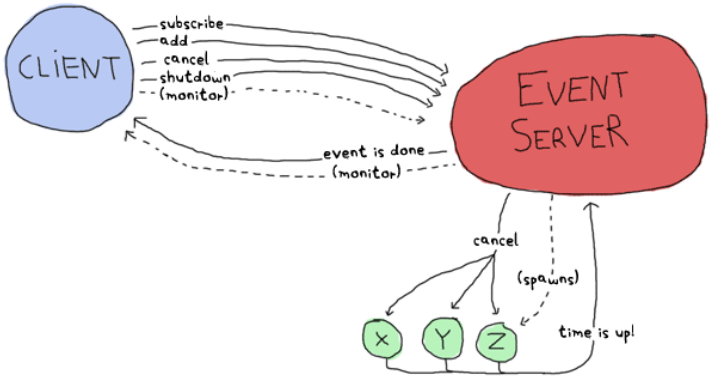
- Подписывается на события у сервера событий и получает уведомления посредством сообщений. Используя этот механизм, можно легко спроектировать группу клиентов, которые создают подписку на сервере событий. Каждый клиент потенциально может служить шлюзом к различным точкам взаимодействия, упомянутым выше (графический интерфейс, веб-страница, программа обмена мгновенными сообщениями, электронная почта и т.д.).
- Запрашивает у сервера создание события с необходимыми параметрами.
- Совершает к серверу запрос на отмену события.
- Отслеживает сервер (на случай если тот прекратит работу).
- При необходимости останавливает сервер событий.

15.1.3 x, y и z

- С их помощью обозначаются уведомления, готовые к запуску (они реализованы в виде таймеров, связанных с сервером событий).
- Отсылают сообщение серверу событий по истечении заданного периода.
- Получают сообщение об отмене и умирают.

Обратите внимание, что все клиенты (IM, почта и т.д., которые в этой книге не реализованы) получают уведомления обо всех событиях, а отмена не входит в список вещей, о которых следует предупреждать клиентов. Эта программа написана для нас с вами, поэтому предполагается, что её будет запускать только один пользователь.

Вот более сложная схема, с указанием всех возможных сообщений:

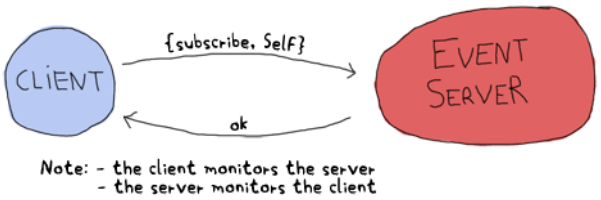


Здесь указан каждый процесс, который мы будем использовать. Стрелки обозначают передаваемые сообщения. С их помощью мы записали высокоуровневый протокол взаимодействия, ну или хотя бы его основу.

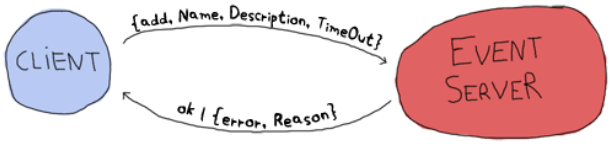
Нужно заметить, что для уведомлений мы используем по процессу на каждое событие. Это слишком расточительно, и такое решение будет плохо масштабироваться в реальной задаче. Но для приложения, единственным пользователем которого будете только вы, это вполне уместно. Можно было бы решить эту проблему иначе, и использовать, к примеру, функцию timer:send_after/2-3, позволив тем самым избежать порождения большого количества процессов.

15.2 Определяем протокол

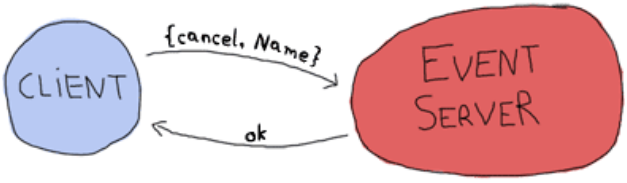
Теперь, когда мы знаем что должен передавать каждый компонент и каковы его функции, неплохо было бы составить список всех передаваемых сообщений, и установить их вид. Начнём с взаимодействия между клиентом и сервером событий:



Я решил использовать два монитора, так как между клиентом и сервером нет явной зависимости. Конечно, клиент без сервера работать не сможет, но сервер без клиента будет существовать без проблем. Здесь можно было бы использовать связь (link), но мы хотим, чтобы функциональность нашей системы могла расширяться за счёт различных клиентов, поэтому мы не можем просто предположить, что после остановки сервера любой клиент тоже захочет аварийно завершиться. Мы также не можем рассчитывать на то, что клиента можно превратить в системный процесс, и он начнёт улавливать завершения (exits) в случае смерти сервера. Перейдём к следующему набору сообщений:

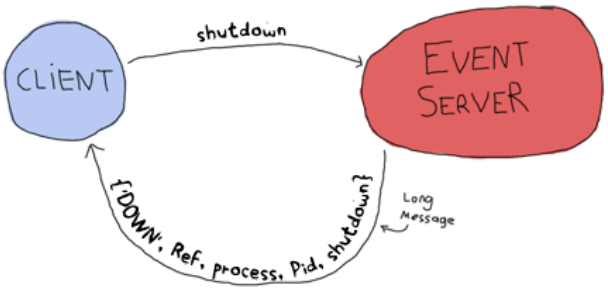


Здесь мы добавляем на сервере событий ещё одно событие. Клиенту высылается подтверждение в виде атома `ok`, за исключением случаев, когда что-то пошло не так (например, TimeOut был передан в неверном формате.) Обратная операция удаления событий может быть совершена следующим образом:



Позже сервер событий может отослать уведомление о том, что событие наступило:

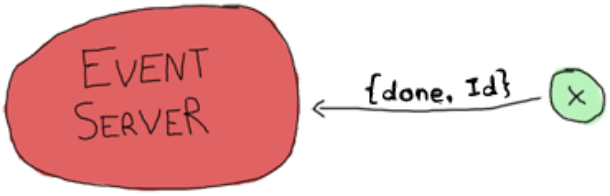
Нам осталось определить пару особых случаев: когда необходимо остановить сервер, и когда сервер аварийно завершается:



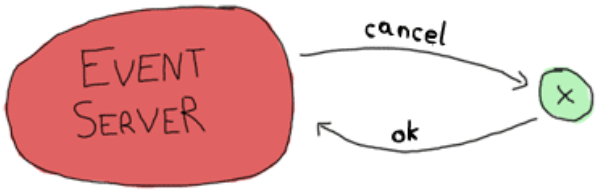
Прямое подтверждение об остановке сервера не отсылается, так как о случившемся нас предупредит монитор. Вот, в общем-то и всё, что будет происходить между клиентом и сервером событий. Перейдём к сообщениям, передаваемым между сервером событий и самими процессами событий.

Перед тем как мы приступим к их описанию, я бы хотел заметить, что неплохо было бы установить связи (links) между сервером событий и событиями. Сделать это нужно по той причине, что если сервер умирает, нам нужно чтобы все события умерли вместе с ним – без сервера в их существовании нет никакого смысла.

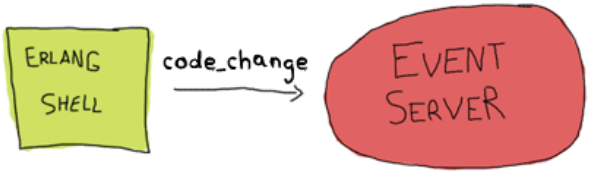
Итак, вернёмся к событиям. Когда сервер событий их создаёт, он присваивает каждому особый идентификатор (имя события). Как только приходит время какого-нибудь события, сервер должен отослать об этом уведомление:



Событие, в свою очередь, должно ожидать от сервера событий сигналы об отмене:



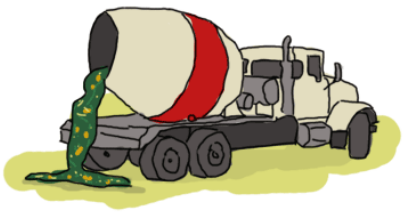
Вот и всё. Последний штрих - нам потребуется сообщение, которое позволит обновлять код сервера:



Отвечать на это сообщение нет необходимости. Когда мы реализуем эту часть нашей программы, вы поймёте, почему мы можем так сделать.

Теперь у нас есть и протокол общения, и приблизительная структура иерархии процессов. Можно наконец-то приступить к воплощению нашего проекта в жизнь.

15.3 Заложим-ка основы



Для начала мы должны создать стандартный набор директорий, принятый в Erlang. Вот как он выглядит:

ebin/
include/
priv/
src/

В директорию `ebin/` попадают скомпилированные файлы. Директория `include/` используется для хранения файлов `.hrl`, предназначенных для включения другими приложениями; файлы с расширением `.hrl`, доступные лишь текущему приложению (private), обычно хранятся в директории `src/`. Директория `priv/` содержит исполняемые файлы,

которые могут взаимодействовать с Erlang. К их числу относятся некоторые драйверы и прочее в этом духе. В нашем проекте мы эту директорию использовать не будем. Ну и последняя директория – `src/`, в ней находятся все файлы `.erl`.

Описанная структура директорий может немного варьироваться в стандартных проектах Erlang. Для хранения некоторых конфигурационных файлов может добавляться директория `conf/`, для документации `doc/` и `lib/` – для сторонних библиотек, которые ваше приложение использует во время исполнения. Зачастую различные Erlang-продукты, представленные на рынке, используют имена отличные от указанных, но четыре имени, упомянутые выше, обычно не меняются, так как они являются частью стандартных приёмов (standard practices) OTP.

15.4 Модуль для работы с событиями

Зайдите в директорию `src/` и откройте модуль event.erl, который реализует события x, y и z, отмеченные на приведённых ранее рисунках. Я начинаю именно с этого модуля, так как у него меньше всего зависимостей. Мы сможем попытаться его запустить, не реализовывая сервер событий или функции клиента.

Перед тем как мы начнём писать код, я должен упомянуть, что разработанный нами протокол неполон. Он помогает представить те данные, которые будут пересылаться между процессами, но не описывает подробности пересылки. Как работает адресация? Что мы при этом используем -- ссылки или имена и т.д. Большинство сообщений будут завёрнуты в кортеж вида `{Pid, Ref, Message}`, где *Pid* – отправитель и *Ref* – уникальный идентификатор сообщения, который помогает определить, от кого был получен ответ. Если бы перед ожиданием ответов мы отослали множество сообщений, то без ссылок нам не удалось бы понять, какому сообщению соответствует каждый из ответов.

Начинаем. Ядром процессов, исполняющих код модуля `event.erl` будет функция `loop/1`, основа которой будет выглядеть приблизительно следующим образом, если вы помните протокол:

```
loop(State) ->
    receive
        {Server, Ref, cancel} ->
            ...
    after Delay ->
        ...
    end.
```

Здесь показан поддерживаемый нами тайм-аут, который оповещает о наступлении события, а также способ отмены события сервером. В цикле вы можете заметить переменную *State*. Эта переменная будет содержать значение тайм-аута (в секундах) и имя события (необходимое для отсылки сообщения `{done, Id}`.) Чтобы отсылать уведомления серверу событий, нам также понадобится его pid.

Вся эта информация вполне годится для размещения в состоянии цикла. Для этого объявим в начале файла запись `state`:

```
-module(event).
-compile(export_all).
-record(state, {server,
                name="",
                to_go=0}).
```

Состояние объявлено, теперь можно немного усовершенствовать цикл:

```
loop(S = #state{server=Server}) ->
    receive
        {Server, Ref, cancel} ->
            Server ! {Ref, ok}
    after S#state.to_go*1000 ->
        Server ! {done, S#state.name}
    end.
```

Умножение на тысячу используется для перевода значения `to_go` из секунд в миллисекунды.

Не забывайтесь:
Далее речь пойдёт о языковом недостатке! Переменная «Server» используется при сопоставлении с образцом в секции receive, и поэтому в заголовке функции я связываю её со значением. Помните, что 11.2 записи – это хак! `S#state.server` тихонько разворачивается в выражение `element(2, S)`, использовать которое в качестве шаблона для сопоставления не получится.
Для выражения `S#state.to_go`, следующего за `after`, этот механизм срабатывает нормально, так как его вычисление можно отложить на потом.

А теперь проверим цикл:

```
6> c(event).
{ok,event}
7> rr(event, state).
[state]
8> spawn(event, loop, [#state{server=self(), name="test", to_go=5
   }]).
```

```
<0.60.0>
9> flush().
ok
10> flush().
Shell got {done,"test"}
ok
11> Pid = spawn(event, loop, [#state{server=self(), name="test",
    to_go=500}]).
<0.64.0>
12> ReplyRef = make_ref().
#Ref<0.0.0.210>
13> Pid ! {self(), ReplyRef, cancel}.
{<0.50.0>,#Ref<0.0.0.210>,cancel}
14> flush().
Shell got {#Ref<0.0.0.210>,ok}
ok
```

Довольно насыщенный пример. Сначала мы импортируем запись (record) из модуля обработки событий командой `rr(Mod)`. Затем запускаем цикл обработки событий, для которого сервером выступает оболочка (`self()`). Заданное событие должно произойти через 5 секунд. Выражение в 9-ой строке было запущено через 3 секунды, в 10-ой – через 6 секунд. Как видите, со второй попытки мы получили сообщение `{done, "test"}`.

Далее я пытаюсь воспользоваться командой отмены (для ввода которой с запасом выделяется 500 секунд). Видно, как я создаю ссылку, посылаю сообщение и получаю ответ, используя ту же самую ссылку. Так мы можем определить, что полученное сообщение `ok` пришло именно от заявленного процесса, а не какого-либо другого существующего в системе.

Сообщение отмены завёрнуто в ссылку, а сообщение `done` – нет. Делаем мы так просто потому, что не ожидаем получить `done` от какого-либо определённого процесса (сгодится любой, мы не будем проводить сопоставление в receive), и отвечать на это сообщение мы тоже не намерены. Я бы хотел провести заранее ещё одну проверку. Что произойдёт, если событие случится в следующем году?

```
15> spawn(event, loop, [#state{server=self(), name="test", to_go
    =365*24*60*60}]).
<0.69.0>
16>
=ERROR REPORT==== DD-MM-YYYY::HH:mm:SS ===
Error in process <0.69.0> with exit value: {timeout_value,[{event
    ,loop,1}]}
```

Ой. Кажется мы наткнулись на ограничение, обусловленное реализацей. Оказывается на значение тайм-аута в Erlang накладывается ограничение

в 50 дней (в миллисекундах). Может быть, эта особенность и не столь важна, но у меня есть целых три причины для демонстрации этой ошибки:

1. Я напоролся на этот нюанс, когда глава была наполовину написана, и я разрабатывал и тестировал для неё сопровождающий модуль.

2. Само собой разумеется, что Erlang не всегда идеально подходит для решения любой задачи. Здесь перед нами предстаёт результат использования таймеров способом, не предусмотренным разработчиками.

3. Но это, в общем-то, не проблема; давайте придумаем как обойти это ограничение.

Я решил устранить этот недостаток при помощи функции, которая разделяла бы значение длинного тайм-аута на несколько частей. Функцию `loop/1` тоже потребуется немного изменить. Проще говоря, мы разделим временной промежуток на равные части по 49 дней (так как ограничение равно приблизительно 50 дням), а затем к этим равным частям добавим остаток. Полная сумма значений в этом списке должна быть равна исходному временному отрезку:

```
%% Because Erlang is limited to about 49 days (49*24*60*60*1000)
    in
%% milliseconds, the following function is used
normalize(N) ->
    Limit = 49*24*60*60,
    [N rem Limit | lists:duplicate(N div Limit, Limit)].
```

Функция `lists:duplicate/2` принимает вторым аргументом данное выражение и повторяет его столько раз, сколько задано значением первого аргумента (`[a,a,a] = lists:duplicate(3, a)`). Если бы мы передали функции `normalize/1` значение `98*24*60*60+4`, то она бы возвратила `[4,4233600,423360]`. Для поддержки нового формата, функция `loop/2` принимает следующий вид:

```
%% Loop uses a list for times in order to go around the ~49 days
    limit
%% on timeouts.
loop(S = #state{server=Server, to_go=[T|Next]}) ->
    receive
        {Server, Ref, cancel} ->
            Server ! {Ref, ok}
    after T*1000 ->
        if Next =:= [] ->
            Server ! {done, S#state.name};
            Next =/= [] ->
```

```
                loop(S#state{to_go=Next})
            end
    end.
```

Можете попробовать её запустить, она будет работать как и прежде, но ко всему прочему сможет обрабатывать тайм-ауты длиной в несколько лет. Работает этот механизм следующим образом: функция извлекает из списка `to_go` первый элемент и переходит в состояние ожидания на период, равный значению этого элемента. Когда ожидание окончено, проверяется, есть ли в списке следующий элемент. Если список пуст, то тайм-аут окончен, и сервер получает об этом уведомление. В противном случае процесс повторяется для всех остальных элементов списка, до полного их исчерпания.

Было бы очень досадно, если каждый раз при запуске процесса--события приходилось бы вручную вызывать что-нибудь вроде `event:normalize(N)`, особенно если учесть, что этот обходной манёвр совсем не должен заботить программистов, использующих наш код. Стандартное решение для этой проблемы – завести функцию `init`, которая будет содержать инициализацию данных, необходимых для правильной работы функции цикла. Раз уж мы об этом заговорили, давайте заодно добавим стандартные функции `start` и `start_link`:

```
start(EventName, Delay) ->
    spawn(?MODULE, init, [self(), EventName, Delay]).

start_link(EventName, Delay) ->
    spawn_link(?MODULE, init, [self(), EventName, Delay]).

%%% Event's innards
init(Server, EventName, Delay) ->
    loop(#state{server=Server,
                name=EventName,
                to_go=normalize(Delay)}).
```

Теперь интерфейс стал намного чище. Но прежде чем приступить к его тестированию, было бы неплохо иметь для единственного сообщения, которое мы можем посылать и отменять, свою собственную интерфейсную функцию:

```
cancel(Pid) ->
    %% Monitor in case the process is already dead
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, cancel},
    receive
        {Ref, ok} ->
            erlang:demonitor(Ref, [flush]),
```

```
            ok;
        {'DOWN', Ref, process, Pid, _Reason} ->
            ok
    end.
```

А вот и новый трюк! Я использую монитор для проверки существования процесса. Если процесс уже умер, я не трачу время на бессмысленное ожидание, а сразу возвращаю `ok`, как и указано в протоколе. Если процесс отвечает ссылкой, я знаю, что он скоро умрёт: я убираю ссылку, так как необходимости в ней больше нет. Обратите внимание, что я также указываю опцию `flush`, которая удалит сообщение `DOWN`, если оно было послано до момента отключения мониторинга.

Протестируем описанные функции:

```
17> c(event).
{ok,event}
18> f().
ok
19> event:start("Event", 0).
<0.103.0>
20> flush().
Shell got {done,"Event"}
ok
21> Pid = event:start("Event", 500).
<0.106.0>
22> event:cancel(Pid).
ok
```

И они работают! Осталась последняя особенность модуля событий, которая нас беспокоит – нам необходимо указывать оставшееся время в секундах. Было бы намного удобнее использовать время в стандартном формате, таком как datetime в Erlang (`{{Year, Month, Day}, {Hour, Minute, Second}}`). Просто добавим следующую функцию, которая будет вычислять разницу между текущим временем на вашем компьютере и введённой задержкой:

```
time_to_go(TimeOut={{_,_,_}, {_,_,_}}) ->
    Now = calendar:local_time(),
    ToGo = calendar:datetime_to_gregorian_seconds(TimeOut) -
        calendar:datetime_to_gregorian_seconds(Now),
    Secs = if ToGo > 0  -> ToGo;
            ToGo =< 0 -> 0
        end,
    normalize(Secs).
```

Да уж, у функций в календарном модуле прикольные имена. Как было отмечено выше, этот код вычисляет число секунд между текущим моментом и моментом, когда должно произойти событие. Если событие

уже прошло, мы возвращаем 0, чтобы как можно быстрее уведомить об этом сервер. Исправьте функцию init, чтобы она вызывала этот код вместо `normalize/1`. Также можно переименовать переменную *Delay* в, скажем, *DateTime*, если хотите, чтобы имя лучше описывало смысл происходящего:

```
init(Server, EventName, DateTime) ->
    loop(#state{server=Server,
        name=EventName,
        to_go=time_to_go(DateTime)}).
```

Всё, с этим закончили. Теперь можно сделать перерыв, создать новое событие, сходить выпить пинту (пол-литра) молока/пива и вернуться точно к моменту, когда придёт сообщение о том, что событие наступило.

15.5 Сервер событий

Теперь давайте разберёмся с сервером событий. Согласно протоколу, его каркас должен выглядеть приблизительно таким образом:

```
-module(evserv).
-compile(export_all).

loop(State) ->
    receive
        {Pid, MsgRef, {subscribe, Client}} ->
            ...
        {Pid, MsgRef, {add, Name, Description, TimeOut}} ->
            ...
        {Pid, MsgRef, {cancel, Name}} ->
            ...
        {done, Name} ->
            ...
        shutdown ->
            ...
        {'DOWN', Ref, process, _Pid, _Reason} ->
            ...
        code_change ->
            ...
        Unknown ->
            io:format("Unknown message: ~p~n",[Unknown]),
            loop(State)
    end.
```

Вы можете заметить, что я завернул вызовы, которые требуют ответа, используя тот же формат `{Pid, Ref, Message}`, который употреблялся ранее. Теперь серверу нужно будет хранить в состоянии две вещи: список

подписавшихся клиентов и список всех запущенных процессов-событий. Возможно, вы обратили внимание: в протоколе говорится, что когда наступает событие, сервер событий должен получать `{done, Name}`, но посылать `{done, Name, Description}`. Идея заключается в том, чтобы генерировать как можно меньше трафика, и позволять процессам-событиям заниматься только самым необходимым. Да, вернёмся к списку клиентов и списку событий:

```
-record(state, {events,     %% list of #event{} records
                clients}). %% list of Pids

-record(event, {name="",
        description="",
        pid,
        timeout={{1970,1,1},{0,0,0}}}).
```

Теперь в заголовке цикла у нас находится объявление записи (record definition):

```
loop(S = #state{}) ->
    receive
    ...
    end.
```

Неплохо было бы задействовать для хранения и событий и клиентов упорядоченные словари (orddicts). Вряд ли количество элементов, помещённых в такой контейнер, будет исчисляться многими сотнями. Вспоминая главу о 11.3 структурах данных, мы можем убедиться, что orddict-ы очень хорошо подходят для наших нужд. Для реализации этой функциональности мы напишем функцию `init`:

```
init() ->
    %% Loading events from a static file could be done here.
    %% You would need to pass an argument to init telling where
       the
    %% resource to find the events is. Then load it from here.
    %% Another option is to just pass the events straight to the
       server
    %% through this function.
    loop(#state{events=orddict:new(),
                clients=orddict:new()}).
```

С каркасом и инициализацией мы закончили. Теперь я запишу реализацию каждого сообщения по отдельности. Первое сообщение касается подписки. Мы хотим хранить список всех подписчиков для того, чтобы иметь возможность уведомить их о наступлении события. Также в описанном выше протоколе упоминается, что мы должны наблюдать (monitor) за

подписчиками. В этом намерении есть рациональное зерно, так как хранить сведения о нерабочих клиентах и посылать без причины ненужные сообщения мы не хотим. Как бы то ни было, код должен выглядеть вот так:

```
{Pid, MsgRef, {subscribe, Client}} ->
    Ref = erlang:monitor(process, Client),
    NewClients = orddict:store(Ref, Client, S#state.clients),
    Pid ! {MsgRef, ok},
    loop(S#state{clients=NewClients});
```

Эта часть функции `loop/1` делает вот что: запускает монитор и сохраняет информацию о клиенте в orddict-е, используя ключ *Ref*. Мы поступаем так по простой причине: в следующий раз нам понадобится извлечь ID клиента, только если мы получим от монитора сообщение `EXIT`, в котором будет содержаться ссылка (которая позволит нам избавиться от записи в orddict-е).

Следующее сообщение, которое могло бы нас заинтересовать, позволяет добавлять события. Выполняя эту операцию, у нас есть возможность вернуть состояние ошибки. Контроль данных мы будем осуществлять только в одном месте – при проверке входящих временных меток (timestamps). Мы могли бы просто проверять входящие данные на соответствие шаблону `{{Year,Month,Day}, {Hour,Minute,seconds}}`, но нам нужно убедиться, что мы не допустим, к примеру, событие, запланированное на 29 февраля невисокосного года, или не примем любую другую несуществующую дату. И тем более мы не хотим принимать даты, которые не могут существовать в принципе (пример такой даты: «5 часов, минус 1 минута и 75 секунд»). С проверкой всех этих условий можно справиться силами одной функции.

Первым кирпичиком, которым мы воспользуемся, будет функция calendar:valid_date/1. Как можно догадаться по имени, эта функция проверяет дату на валидность. К сожалению, странности календарного модуля на чудны́х именах не заканчиваются: в модуле не существует функции, которая может подтвердить, что кортеж `{H,M,S}` содержит валидные значения. Придётся нам самим написать такую функцию, при этом следуя странным правилам именования, присущим календарному модулю:

```
valid_datetime({Date,Time}) ->
    try
```

```
        calendar:valid_date(Date) andalso valid_time(Time)
    catch
        error:function_clause -> %% not in {{Y,M,D},{H,Min,S}}
            format
        false
    end;
valid_datetime(_) ->
    false.

valid_time({H,M,S}) -> valid_time(H,M,S).
valid_time(H,M,S) when H >= 0, H < 24,
                       M >= 0, M < 60,
                       S >= 0, S < 60 -> true;
valid_time(_,_,_) -> false.
```

Теперь можно воспользоваться функцией `valid_datetime/1` в том месте, где мы пытаемся добавить сообщение:

```
{Pid, MsgRef, {add, Name, Description, TimeOut}} ->
    case valid_datetime(TimeOut) of
        true ->
            EventPid = event:start_link(Name, TimeOut),
            NewEvents = orddict:store(Name,
                                      #event{name=Name,
                                           description=Description,
                                           pid=EventPid,
                                      timeout=TimeOut},
                                      S#state.events),
            Pid ! {MsgRef, ok},
            loop(S#state{events=NewEvents});
        false ->
            Pid ! {MsgRef, {error, bad_timeout}},
            loop(S)
    end;
```

Если время сформировано верно, мы порождаем новый процесс--событие, затем сохраняем его данные в состоянии сервера событий и посылаем вызывающему процессу подтверждение. Если тайм-аут сформирован неверно, мы не позволяем ошибке пройти незамеченной, и не переводим сервер в аварийное состояние – вместо этого мы сообщаем об ошибке клиенту. Для обнаружения конфликта имён, или наложения каких-либо других ограничений, можно ввести дополнительные проверки (главное, не забудьте сделать изменения в документации протокола!)

Следующее сообщение, определённое в нашем протоколе, позволяет отменять событие. Отмена никогда не сможет пройти неудачно с точки зрения клиента, а потому и код будет попроще. Нужно просто проверить, существует ли событие в записи (record) состояния процесса. Если оно

там есть, мы используем функцию `event:cancel/1` для остановки события, и отсылки сообщения ok. Если событие не было найдено, мы всё равно говорим пользователю, что всё прошло хорошо – событие не запущено, а это именно то, что пользователь и хотел.

```
{Pid, MsgRef, {cancel, Name}} ->
    Events = case orddict:find(Name, S#state.events) of
                 {ok, E} ->
                     event:cancel(E#event.pid),
                     orddict:erase(Name, S#state.events);
                 error ->
                     S#state.events
             end,
    Pid ! {MsgRef, ok},
    loop(S#state{events=Events});
```

Неплохо, неплохо. Теперь нами покрыты все произвольные действия, которые может инициировать клиент по отношению к серверу событий. Давайте займёмся тем, что происходит непосредственно между сервером и событиями. Нам нужно обрабатывать два сообщения: отмена событий (это мы уже сделали), и превышение лимита времени (timing out) для событий. Это сообщение выглядит просто как `{done, Name}`:

```
{done, Name} ->
    case orddict:find(Name, S#state.events) of
        {ok, E} ->
            send_to_clients({done, E#event.name, E#event.
                description},
                            S#state.clients),
            NewEvents = orddict:erase(Name, S#state.events),
            loop(S#state{events=NewEvents});
        error ->
            %% This may happen if we cancel an event and
            %% it fires at the same time
            loop(S)
    end;
```

А функция `send_to_clients/2` делает именно то, о чём говорит её имя, и определяется следующим образом:

```
send_to_clients(Msg, ClientDict) ->
    orddict:map(fun(_Ref, Pid) -> Pid ! Msg end, ClientDict).
```

С большей частью кода цикла мы закончили. Осталось записать код обработки различных сообщений, меняющих состояние: клиент прекращает работу, остановка, апгрейд кода и т.д. Вот как они реализованы:

```
shutdown ->
```

```
    exit(shutdown);
{'DOWN', Ref, process, _Pid, _Reason} ->
    loop(S#state{clients=orddict:erase(Ref, S#state.clients)});
code_change ->
    ?MODULE:loop(S);
Unknown ->
    io:format("Unknown message: ~p~n",[Unknown]),
    loop(S)
```

Обработка первого сообщения (`shutdown`) довольно очевидна. Вы получаете сообщение о прекращении работы, после чего позволяете процессу умереть. Если бы вы захотели сохранить состояние на диск, это место неплохо подошло бы для осуществления такой задачи. Если бы вам понадобилась более безопасная семантика сохранения/выхода, то её можно обеспечить при раздельной обработке каждого сообщения `add`, `cancel` или `done`. А загрузку событий с диска можно было бы производить в функции `init`, запуская процессы-события по мере их появления.

Действия, производимые для сообщения «`DOWN`», также достаточно просты. Получение этого сообщения означает, что клиент умер, а значит мы просто убираем его из списка клиентов, который хранится в состоянии процесса.

Неизвестные сообщения просто будут отображаться при помощи `io:format/2` для отладочных целей. Нужно учитывать, что в настоящем рабочем приложении, лучше использовать отдельный модуль логирования.

Ну и напоследок рассмотрим сообщение для изменения кода. Мне оно кажется достаточно интересным для того, чтобы посвятить ему целый раздел.

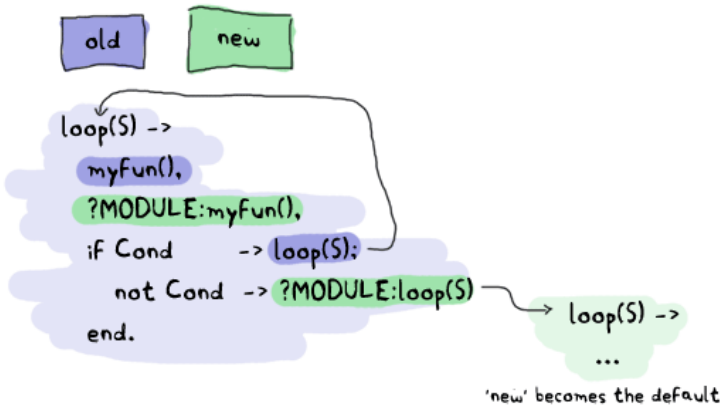
15.6 Горячая любовь к коду

Для горячей загрузки кода в Erlang есть такая штука, которая называется *сервер кода*. Сервер кода это в сущности процесс виртуальной машины, который управляет таблицей ETS (таблица базы данных, которая располагается в памяти – является частью VM.) Сервер кода может держать в памяти две версии одного модуля, и обе эти версии могут одновременно исполняться. Новая версия модуля автоматически загружается после её компиляции командой `c(Module)`, загрузки через `l(Module)` или загрузки посредством одной из множества функций модуля для работы с кодом.

Нужно понимать принцип, согласно которому в Erlang существуют *локальные (local)* и *внешние (external)* вызовы. Локальными считаются вызовы функций, которые не будут проэкспортированы. Формат их записи:

`Atom(Args)`. А внешний (external) вызов, напротив, может быть исполнен только с экспортированными функциями, и имеет вид `Module:Function(Args)`.

Когда в виртуальную машину загружены две версии модуля, все локальные вызовы производятся через версию, которая запущена внутри процесса в текущий момент. Но внешние вызовы **всегда** выполняются к самой новой версии кода, которая доступна через сервер кода. В дальнейшем, локальные вызовы, которые исполняются внутри внешнего вызова, используют новую версию кода.



Учитывая то, что каждому процессу/актору в Erlang для изменения состояния необходимо выполнить рекурсивный вызов, существует возможность загружать совершенно новые версии актора, исполняя внешний рекурсивный вызов.

> **Замечание:** если вы загрузите третью версию модуля в тот момент, когда процесс всё ещё исполняется с использованием первой, этот процесс будет убит виртуальной машиной. Она считает, что это был процесс-сирота, который выполнялся без участия супервизора, или без возможности себя модернизировать (upgrade). Если самая старая версия никем не используется, её просто выбрасывают, а вместо неё остаются более новые.

Существуют средства, позволяющие связаться с системным модулем, который будет отсылать сообщения при каждой загрузке новой версии модуля. Вы сможете запускать перезагрузку модуля только при получении такого сообщения, и всегда осуществлять её, используя функцию модернизации кода, скажем `MyModule:Upgrade(CurrentState)`, которая сможет привести структуру данных состояния к требованиям, которые предъявляет новая версия. Такая обработка с «подпиской» автоматически производится фреймворком OTP, который мы скоро начнём изучать. Для приложения-напоминалки мы не будем использовать сервер кода, а вместо этого будем посылать

из оболочки пользовательское сообщение `code_change`, после чего будет выполняться очень простая перезагрузка кода. Вот собственно и все знания, которые нужны для проведения горячей загрузки кода. Тем не менее, я приведу более общий пример:

```
-module(hotload).
-export([server/1, upgrade/1]).

server(State) ->
    receive
        update ->
            NewState = ?MODULE:upgrade(State),
            ?MODULE:server(NewState);  %% loop in the new version
                of the module
        SomeMessage ->
            %% do something here
            server(State)  %% stay in the same version no matter
                what.
    end.

upgrade(OldState) ->
    %% transform and return the state here.
```

Как видите, наша функция `?MODULE:loop(S)` вписывается в этот шаблон.

15.7 Я сказал, прячьте свои сообщения

Сокрытие сообщений! Если хотите, чтобы другие люди могли брать за основу ваш код и процессы, нужно научиться прятать сообщения за функциями интерфейса. Вот как это сделано в модуле `evserv`:

```
start() ->
    register(?MODULE, Pid=spawn(?MODULE, init, [])),
    Pid.

start_link() ->
    register(?MODULE, Pid=spawn_link(?MODULE, init, [])),
    Pid.

terminate() ->
    ?MODULE ! shutdown.
```

Я решил зарегистрировать серверный модуль, чтобы в каждый момент времени был запущен только один его экземпляр. Если бы вам захотелось расширить напоминатель, чтобы его смогли использовать несколько пользователей, было бы неплохо зарегистрировать имена в глобальном

модуле, или с помощью библиотеки gproc. Для нужд этого демонстрационного приложения, таких средств будет достаточно.

Давайте попробуем