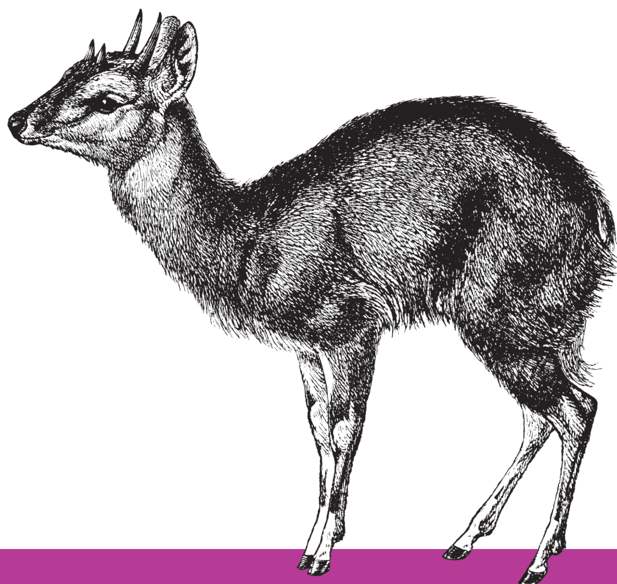
O'REILLY®



Введение в Elixir

Симон Сенлорен
Дэвид Эйзенберг

ДМК пресс
издательство

УДК 004.438Elixir
ББК 32.973.3
С31

Сенлорен С., Эйзенберг Д.

С31 Введение в Elixir: введение в функциональное программирование / пер. с анг. А. Н. Киселева – М.: ДМК Пресс, 2017. – 262 с.: ил.

ISBN 978-5-97060-518-9

Красивый, мощный и компактный, язык программирования Elixir отлично подходит для изучения функционального программирования, и это практическое руководство покажет, насколько широкими возможностями он обладает. В книге рассказано, как Elixir сочетает в себе надежность языка функционального программирования Erlang с подходом, свойственным языку Ruby, а также мощную поддержку макросов для метапрограммирования.

В итоге вы поймете, почему на Elixir так просто писать параллельные, надежные и отказоустойчивые программы, которые легко масштабируются как вверх, так и вниз!

УДК 004.438Elixir
ББК 32.973.3

Authorized Russian translation of the English edition of Introducing Elixir, 2nd Edition, ISBN 9781491956779 © 2017 Simon St. Laurent, J. David Eisenberg. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

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

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

ISBN 978-1-491-95677-9 (анг.)
ISBN 978-5-97060-518-9 (рус.)

© 2017 Simon St. Laurent and J. David Eisenberg
© Оформление, издание, ДМК Пресс, 2017

Содержание

Предисловие ... 10

Глава 1. Устраиваемся поудобнее ... 19

Установка ... 19

Установка Erlang ... 19

Установка Elixir ... 20

Запуск ... 21

Первые шаги ... 22

Навигация по тексту и истории команд ... 23

Навигация по файлам ... 23

Сделаем что-нибудь ... 24

Вызов функций ... 25

Числа в Elixir ... 26

Работа с переменными в оболочке ... 28

Глава 2. Функции и модули ... 31

Игры с fn ... 31

И & ... 33

Определение модулей ... 34

От модулей к свободным функциям ... 38

Деление кода на модули ... 38

Комбинирование функций с помощью оператора конвейера ... 40

Импортирование функций ... 41

Значения по умолчанию для аргументов ... 42

Документирование кода ... 43

Документирование функций ... 44

Документирование модулей ... 46

Глава 3. Атомы, кортежи и сопоставление с образцом ... 48

Атомы ... 48

Сопоставление с образцом и атомы ... 49

Логические атомы ... 50

Ограничители 52
Обозначайте подчеркиванием все, что не важно 55
Структуры данных: кортежи 57
Сопоставление с образцом и кортежи 58
Обработка кортежей 59

Глава 4. Логика и рекурсия 62
Логика внутри функций 62
Конструкция case 62
Конструкция cond 65
if или else 66
Присваивание значений переменным в конструкциях case и if 68
Самый желательный побочный эффект: IO.puts 69
Простая рекурсия 71
Обратный отсчет 71
Прямой отсчет 73
Рекурсия с возвратом значения 74

Глава 5. Взаимодействие с человеком 79
Строки 79
Многострочные строки 82
Юникод 82
Списки символов 83
Строковые метки 84
Запрос информации у пользователя 85
Ввод символов 85
Чтение строк текста 86

Глава 6. Списки 91
Основы списков 91
Деление списков на головы и хвосты 93
Обработка содержимого списков 94
Создание списка из головы и хвоста 96
Смешивание списков и кортежей 99
Создание списка списков 99

Глава 7. Пары имя/значение 103
Списки ключей 103
Списки кортежей с несколькими ключами 105

Словари ... 106
От списков к отображениям ... 107
Создание отображений ... 108
Изменение отображений ... 108
Чтение отображений ... 109
От отображений к структурам ... 109
Объявление структур ... 110
Создание и чтение экземпляров структур ... 110
Использование структур в сопоставлениях с образцом ... 111
Использование структур в функциях ... 112
Добавление поведения в структуры ... 114
Расширение существующих протоколов ... 116

Глава 8. Функции высшего порядка и генераторы списков ... 118
Простые функции высшего порядка ... 118
Создание новых списков с помощью функций высшего порядка ... 121
Получение информации о списке ... 121
Обработка элементов списка с помощью функций ... 122
Фильтрация значений в списках ... 123
За пределами возможностей генераторов списков ... 124
Проверка списков ... 124
Разбиение списков ... 125
Свертка списков ... 126

Глава 9. Процессы ... 129
Интерактивная оболочка – это процесс ... 129
Порождение процессов из модулей ... 132
Легковесные процессы ... 135
Регистрация процесса ... 136
Когда процесс останавливается ... 137
Взаимодействие между процессами ... 138
Наблюдение за процессами ... 141
Наблюдение за движением сообщений между процессами ... 143
Разрыв и установка связей между процессами ... 145

Глава 10. Исключения, ошибки и отладка ... 152
Виды ошибок ... 152
Восстановление работоспособности после ошибок времени выполнения ... 153

Журналирование результатов выполнения и ошибок ... 156
Трассировка сообщений ... 157
Трассировка вызовов функций ... 159

Глава 11. Статический анализ, спецификации типов и тестирование ... 161
Статический анализ ... 161
Спецификации типов ... 164
Модульные тесты ... 167
Настройка тестов ... 170
Встраивание тестов в документацию ... 172

Глава 12. Хранение структурированных данных ... 173
Записи: структурирование данных до появления структур ... 173
- Определение записей ... 174
- Создание и чтение записей ... 175
- Использование записей в функциях ... 177

Сохранение данных в долговременном хранилище Erlang ... 179
- Создание и заполнение таблицы ... 181
- Простые запросы ... 187
- Изменение значений ... 187
- Таблицы ETS и процессы ... 188
- Следующие шаги ... 190

Хранение записей в Mnesia ... 191
- Настройка базы данных Mnesia ... 191
- Создание таблиц ... 192
- Чтение данных ... 196

Глава 13. Основы OTP ... 198
Создание служб с помощью GenServer ... 199
Простой супервизор ... 204
Упаковка приложения с помощью Mix ... 209

Глава 14. Расширение языка Elixir с помощью макросов ... 213
Функции и макросы ... 213
Простой макрос ... 214
Создание новой логики ... 216
Программное создание функций ... 217
Когда (не) следует использовать макросы ... 219

Глава 15. Phoenix 221
Установка базовых компонентов фреймворка 221
Структура простого Phoenix-приложения 224
Представление страницы 224
  Маршрутизация 225
  Простой контроллер 227
  Простое представление 228
Вычисления 230
Продвижение Elixir 237

Приложение A. Каталог элементов языка Elixir 239
Команды интерактивной оболочки 239
Зарезервированные слова 240
Операторы 241
Ограничители 243
Часто используемые функции 244

Приложение B. Создание документации с помощью ExDoc 247
Использование ExDoc вместе с Mix 247

Предметный указатель 251

Об авторах 259

Колофон 260

Глава 1

Устраиваемся поудобнее

Изучение Elixir удобнее всего начинать в интерактивной оболочке Interactive Elixir (IEx). Этот интерфейс командной строки – лучшее средство, чтобы выяснить, что возможно и что невозможно в Elixir. Он поможет избавиться от головной боли в будущем, поэтому устраивайтесь поудобнее!

Установка

Elixir действует поверх Erlang, поэтому для начала необходимо установить Erlang в системе, а затем Elixir.

Установка Erlang

Установка Erlang в Windows выполняется очень просто (http://www.erlang.org/downloads). Загрузите двоичный файл для Windows, запустите установку – и все! Если вы – храбрый новичок, только начинающий осваивать программирование, для вас такой вариант станет лучшим выбором.

Пользователи Linux или macOS могут загрузить исходный код и скомпилировать его. В macOS придется распаковать архив с исходным кодом, затем перейти в каталог, созданный в процессе распаковки, и выполнить последовательность команд: `./configure`, `make` и `sudo make install`. Однако эта простая последовательность действий увенчается успехом, только если в системе установлены все необходимые инструменты и библиотеки, в противном случае есть риск получить множество малопонятных ошибок. В частности,

в новейших версиях XCode компания Apple использует компилятор LLVM вместо GCC, соответственно, велика вероятность отсутствия компилятора GCC в новейших версиях системы macOS, необходимого для сборки Erlang.

Вы можете спокойно игнорировать ошибки, вызванные отсутствием пакета FOP, который Erlang использует для создания документации в формате PDF – вы сможете загрузить ее отдельно. Кроме того, в новейших версиях macOS в конце может появиться ошибка, сообщающая, что wxWidgets не работает в 64-битных версиях системы. Ее тоже можно безопасно игнорировать, пока.

Если по каким-то причинам компиляция из исходного кода невозможна или нежелательна, на сайте Erlang Solutions (https://www.erlang-solutions.com/resources/download.html) можно найти большое количество установочных пакетов для наиболее распространенных версий дистрибутивов. Кроме того, многие диспетчеры пакетов (Debian, Ubuntu, MacPorts, Homebrew и др.) включают Erlang. Это может быть не самая последняя версия, но наличие любой действующей версии Erlang намного лучше ее отсутствия. Обычно распространители дистрибутивов стремятся включать в свои репозитории самые свежие версии пакетов, поэтому если у вас не заладилось с компиляцией, обратите внимание на диспетчеры пакетов.

Во многих системах Erlang все чаще включается в установку по умолчанию, в том числе в Ubuntu, в основном благодаря широкому распространению CouchDB.

Установка Elixir

Завершив установку Erlang, загрузите скомпилированную версию Elixir (https://github.com/elixir-lang/elixir/releases) или исходный код (https://github.com/elixir-lang/elixir/tags) из репозитория GitHub. Некоторые диспетчеры пакетов уже включают Elixir, в том числе Homebrew. Примеры, представленные в этой книге, должны без проблем выполняться в Elixir 1.3.0.

Затем настройте переменную окружения `PATH`, добавив в нее путь к каталогу *elixir/bin*.

Инструкции по настройке Elixir можно найти в руководстве, по адресу: http://elixir-lang.org/getting-started/introduction.html[1].

---

[1] На русском языке: http://elexir-lang.ru/install. – *Прим. перев.*

Запуск

Перейдите в командную строку (например, откройте окно терминала) и введите команду **mix new first_app**. Она запустит инструмент Mix (https://hexdocs.pm/mix/Mix.html), который «реализует задачи создания, компиляции и тестирования проектов на языке Elixir, управления их зависимостями и многие другие». В данном случае введенная команда создаст новый, пустой проект в каталоге с именем *first_app*:

```
$ mix new first_app
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/first_app.ex
* creating test
* creating test/test_helper.exs
* creating test/first_app_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd first_app
    mix test

Run "mix help" for more commands.
$
```

Не забудьте добавить путь к каталогу с выполняемым файлом Elixir в переменную окружения PATH, чтобы инструмент Mix смог найти его.

Не будем тратить время на компиляцию и тестирование пустого проекта, а сразу перейдем в каталог *first_app* и запустим оболочку IEx, выполнив следующие команды:

```
$ cd first_app
$ iex -S mix
```

Вы увидите на экране примерно следующие строки, возможно, с курсором в строке приглашения `iex(1)>`. Обратите внимание, что везде, где это необходимо, длинные строки переформатированы, чтобы уместить их по ширине страницы:

```
$ cd first_app
[david@localhost first_app]$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
```

```
 [kernel-poll:false]

Compiling 1 file (.ex)
Generated first_app app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Вот мы и в Elixir! (Первая строка с упоминанием Erlang подчеркивает, что Elixir действует внутри Erlang. Но не беспокойтесь об этом!)

Первые шаги

Прежде чем окунуться в захватывающий процесс программирования на языке Elixir, всегда имеет смысл научиться выходить из оболочки. Нажатие комбинации **Ctrl+C** приведет к появлению меню. Если в этом меню ввести «a», IEx завершит работу, и вы увидите приглашение к вводу командной оболочки системы, в которой вы запускали IEx:

```
iex(1)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
$
```

В оболочке `iex` можно также попросить вывести справку (после повторного запуска), введя `h()` или просто `h`[1]:

```
iex(1)> h()
# IEx.Helpers

                                 IEx.Helpers

Добро пожаловать в интерактивную болочку Interactive Elixir. Сейчас вы
смотрите документацию для модуля IEx.Helpers, включающего множество
вспомогательных функций,делающих работу с интерактивной оболочкой Elixir
более удобной.

Это сообщение сгенерировано вызовом вспомогательной функции h(), которую
обычно обозначают как h/0 (поскольку она принимает 0 аргументов).

Функцию h можно использовать для вызова документации к любому модулю Elixir
или функции:
```

[1] Здесь и далее в книге справочный текст приводится в переводе на русский язык, но имейте в виду, что в действительности вся документация в Elixir на английском языке. – *Прим. перев.*

```
| h Enum
| h Enum.map
| h Enum.reverse/1

Для исследования любого значения, созданного в интерактивной оболочке, можно
также использовать функцию i:

| i "hello"

Существует большое количество других вспомогательных функций:
...
:ok
```

Что, собственно, произошло? Мы выполнили команду `iex`, вызвали вспомогательную функцию `h()`, которая вывела перед вами некоторую справочную информацию. Она выводит на экран большой объем текста и завершает вывод, возвращая `:ok`.

Навигация по тексту и истории команд

Занявшись исследованием интерактивной оболочки, вы обнаружите, что своим действием она во многом напоминает другие командные оболочки или Emacs. Клавиши со стрелками влево и вправо перемещают курсор назад и вперед по редактируемой строке. Некоторые комбинации клавиш повторяют действие этих же комбинаций в текстовом редакторе Emacs. Комбинация **Ctrl**+**A** перенесет курсор в начало строки, а **Ctrl**+**E** – в конец. Если вы по ошибке ввели два символа не в той последовательности, нажмите **Ctrl**+**T**, чтобы поменять их местами.

Кроме того, после ввода закрывающей круглой, квадратной или фигурной скобки будет подсвечена парная ей открывающая скобка.

Клавиши со стрелками вверх и вниз выполняют навигацию по истории команд, упрощая их повторный запуск. На конкретную команду в истории можно также сослаться как `v(`*`N`*`)`, где *`N`* – номер строки.

Навигация по файлам

Оболочка IEx поддерживает некоторые простые операции с файловой системой, потому что вам может потребоваться переключаться между файлами, составляющими вашу программу. Команды имеют те же имена, что и команды Unix, но записываются как вызовы функций. Первоначально текущим рабочим каталогом оболочки IEx является каталог, в котором вы ее запустили. Узнать путь к этому каталогу можно с помощью `pwd()`:

```
iex(1)> pwd()
/Users/elixir/code/first_app
:ok
```

Для перехода в другой каталог используйте функцию `cd()`, но вы обязательно должны заключить ее аргумент в двойные кавычки:

```
iex(2)> cd ".."
/Users/elixir/code
:ok
iex(3)> cd "first_app"
/Users/elixir/code/first_app
:ok
```

Получить список с содержимым каталога можно с помощью команды `ls()`, которая в случае вызова без аргумента вернет список содержимого текущего каталога, а с аргументом – список содержимого указанного каталога.

Сделаем что-нибудь

Проще всего начать игру с Elixir – использовать интерактивную оболочку как калькулятор. В отличие от обычной командной строки, она позволяет вводить арифметические выражения и получать результаты:

```
iex(1)> 2 + 2
4
iex(2)> 27 - 14
13
iex(3)> 35 * 42023943
1470838005
iex(4)> 4 * (3 + 5)
32
iex(5)> 200 / 15
13.333333333333334
```

Первые три оператора – сложение (`+`), вычитание (`-`) и умножение (`*`) – действуют как обычно при работе с целыми или вещественными числами. Круглые скобки позволяют изменять порядок выполнения операторов, как показано в строке 4. (Порядок выполнения операций приводится в приложении А.) Четвертый оператор, `/`, возвращает вещественный результат (число с дробной частью), как показано в строке 5. Операторы необязательно окружать пробелами, но они делают код более читаемым.

Вызов функций

Функции – это блоки логики, которые принимают аргументы и возвращают значение. Проще всего начать исследование с математических функций. Например, если требуется получить целочисленный результат деления (целочисленных аргументов), используйте функцию `div()` вместо оператора / и функцию `rem()`, чтобы получить остаток, как показано в строках 6 и 7:

```
iex(6)> div(200,15)
13
iex(7)> rem(200, 15)
5
iex(8)> rem 200, 15
5
```

Строка 8 демонстрирует особенность синтаксиса Elixir: необязательность заключения аргументов в круглые скобки. Если вы считаете, что круглые скобки улучшают читаемость кода, используйте их. Если ввод круглых скобок кажется вам лишней работой, просто не вводите их. Elixir интерпретирует пробел, следующий за именем функции, как открывающую круглую скобку, автоматически закрывающуюся в конце строки. Когда подобный стиль приводит к неожиданным результатам, Elixir может попросить вас «не вставлять пробелы между именем функции и открывающей круглой скобкой».

Elixir допускает передачу целых чисел вместо вещественных, но вещественные числа не всегда можно использовать вместо целых. Если вам потребуется преобразовать вещественное число в целое, используйте встроенную функцию `round()`:

```
iex(9)> round 200/15
13
```

Функция `round()` отбрасывает дробную часть числа. Если дробная часть больше или равна .5, целая часть увеличивается на 1, округляя число вверх. Если вам требуется просто отбросить дробную часть, оставив целую, используйте функцию `trunc()`.

Сослаться на предыдущую команду можно по номеру строки, с помощью функции `v()`. Например:

```
iex(10)> 4*v(9)
52
```

Команда в строке 9 вернула результат 13, соответственно, выражение `4*13` дает в результате `52`.

Если вам интересно, для ссылки на предыдущие команды можно использовать отрицательные числа. `v(-1)` вернет предыдущий результат, `v(-2)` – результат, полученный перед предыдущим, и так далее.

Для выполнения более сложных вычислений Elixir позволяет использовать модуль `math` языка Erlang, содержащий классический набор функций, поддерживаемых научным калькулятором. Эти функции возвращают вещественные значения. Константа «пи» доступна как функция `:math.pi()`. Доступны также тригонометрические, логарифмические, экспоненциальные функции, функция квадратного корня и (исключая Windows) даже функция гауссовой ошибки. (Тригонометрические функции принимают аргументы в радианах, а не в градусах, поэтому при необходимости вам придется выполнять соответствующие преобразования.) При использовании этих функций вам придется также вводить префикс `:math.`, но это не слишком высокая плата за удобство.

Например, ниже показано, как получить синус от нуля радиан:

```
iex(11)> :math.sin(0)
0.0
```

Обратите внимание, что в результате получилось число `0.0`, а не просто `0`. Это указывает, что результат является вещественным числом. (И да, вы можете то же самое вычисление выполнить командой `:math.sin 0` без круглых скобок.)

Ниже показано, как получить косинус от π и 2π радиан:

```
iex(12)> :math.cos(:math.pi())
-1.0
iex(13)> :math.cos(2 * :math.pi())
1.0
```

Чтобы вычислить 2 в степени 16, используйте функцию:

```
iex(14)> :math.pow(2,16)
65536.0
```

Полный перечень функций, имеющихся в модуле `math` и доступных в Elixir, вы найдете в приложении A.

Числа в Elixir

Elixir распознает два вида чисел: целые и вещественные (их также часто называют числами с плавающей точкой). В отличие от целых чисел, не имеющих дробной части, вещественные числа включают

десятичную точку и некоторое значение правее ее – дробную часть. `1` – это целое число, а `1.0` – вещественное.

Однако в действительности все немного сложнее. Elixir хранит целые и вещественные числа в совершенно разных форматах. Elixir позволяет хранить огромные целые числа, но какими большими или маленькими они бы не были, они всегда точны. Вам не придется волноваться об их точности.

Вещественные числа, напротив, хотя и охватывают широкий диапазон значений, но имеют ограниченную точность. В Elixir используется 64-битное представление IEEE 754-1985 «с двойной точностью». То есть этот формат хранит примерно 15 цифр плюс экспоненту. Этот формат может представлять огромные значения – число 10 в степени от –308 до 308 – но из-за ограниченного количества цифр результат иногда может отличаться от точного значения, что особенно ярко проявляется при выполнении сравнения:

```
iex(1)> 3487598347598347598437583475893475843749245.0
3.4875983475983474e42
iex(2)> 2343243.345435893850234543339545
2343243.3454358936
iex(3)> 0.00000000000000000000000000023432432432432234232324
2.3432432432432235e-30
```

Как видите, некоторые цифры отсекаются, и в полном представлении число включает экспоненту.

При вводе вещественных чисел всегда указывайте хотя бы одну цифру слева от десятичной точки, даже если это цифра «ноль». Иначе Elixir сообщит о синтаксической ошибке – он не понимает, чего вы хотите:

```
iex(4)> .00000000000000000000000000023432432432432234232324
** (SyntaxError) iex:4: syntax error before: '.'
```

Вещественные числа можно также вводить в экспоненциальной нотации:

```
iex(4)> 2.923e127
2.923e127
iex(5)> 7.6345435e-231
7.6345435e-231
```

Из-за недостаточной точности применение вещественных чисел может приводить к аномальным результатам. Например, синус нуля равен нулю, и синус числа π также равен нулю. Но, производя вычисления в Elixir, вы не получите нулевого результата при использовании вещественной аппроксимации π:

```
iex(6)> :math.sin(0)
0.0
iex(7)> :math.sin(:math.pi())
1.2246467991473532e-16
```

Если бы в Elixir использовалось более точное представление числа π и вычисления с вещественными числами выполнялись бы с большей точностью, результат в строке 7 был бы ближе к нулю.

Для финансовых вычислений лучше всего использовать целые числа. Используйте наименьшую единицу измерения – например, копейки – и помните, что одна копейка – это 1/100 рубля. (Финансовые сделки могут выполняться в более мелких единицах, но и в этом случае желательно использовать целые числа и при необходимости применять известный множитель.) Однако для более сложных вычислений вполне можно использовать вещественные числа, просто помните, что результат получается не совсем точным.

Elixir поддерживает операции с целыми числами еще в нескольких системах счисления, кроме десятичной. Например, если понадобится, двоичное значение 1010111 можно записать как:

```
iex(8)> 0b01010111
87
```

Elixir вернет значение в виде десятичного числа. Аналогично можно указать шестнадцатеричное (в системе счисления с основанием 16) число, используя префикс `x` вместо `b`:

```
iex(9)> 0xcafe
51966
```

Чтобы ввести отрицательное число, просто добавьте знак минус (`-`) перед ним. Этот прием можно использовать с целыми числами, а также шестнадцатеричными, двоичными и вещественными числами:

```
iex(10)> -1234
-1234
iex(11)> -0xcafe
-51966
iex(12)> -2.045234324e6
-2045234.324
```

Работа с переменными в оболочке

Функция `v()` позволяет ссылаться на результаты предыдущих вычислений, но это не самый удобный способ хранения числовых ре-

зультатов, и к тому же функция v() работает только в интерактивной оболочке – это не универсальный механизм. Намного разумнее хранить значения в памяти, давая им текстовые имена, то есть создавая переменные.

Имена переменных в Elixir начинаются с буквы в нижнем регистре или символа подчеркивания. Обычные переменные начинаются с буквы в нижнем регистре, а «необязательные» – с символа подчеркивания (см. раздел «Обозначайте подчеркиванием все, что не важно» в главе 3). Но пока давайте придерживаться обычных переменных. Синтаксис присваивания значений переменным должен быть вам знаком по школьному курсу алгебры или по другим языкам программирования, например:

```
iex(1)> n=1
1
```

Чтобы узнать значение переменной, просто введите ее имя:

```
iex(2)> n
1
```

Elixir, в отличие от многих других функциональных языков программирования (включая Erlang), позволяет присваивать переменным новые значения:

```
iex(3)> n=2
2
iex(4)> n=n+1
3
```

Elixir сначала выполняет выражение справа, затем оператор = сопоставляет результат с левой стороной. Он присвоит переменной n новое значение, если попросить его об этом, и даже будет использовать прежнее значение n в выражении справа, чтобы получить новое значение. Инструкция n=n+1 означает: «присвоить значение выражения n+1, которое равно 3, переменной n».

В операции присваивания все вычисления должны производиться справа от знака «равно». Даже если известно, что m имеет значение 6, Elixir не сможет выполнить инструкцию 2*m = 3*4:

```
iex(5)> 2*m=3*4
** (CompileError) iex:12: illegal pattern
```

Оболочка IEx будет помнить переменные, пока вы не попросите ее забыть их.

Кроме того, в одну строку можно включить несколько инструкций, разделив их точкой с запятой (`;`). Синтаксически она действует подобно окончанию строки:

```
iex(6)> distance = 20; gravity = 9.8
9.8
iex(7)> distance
20
iex(8)> gravity
9.8
```

IEx сообщит только значение последней инструкции, но, как можно видеть в строках 7 и 8, все переменные получили соответствующие значения.

Если в оболочке накопилось много ненужного, вызовите `clear`. Эта функция просто очистит экран.

Прежде чем перейти к следующей главе, где рассказывается о модулях и функциях, поиграйте немного в IEx. Полученный опыт, даже такой простой, поможет вам увереннее двигаться вперед. Попробуйте использовать переменные и посмотрите, что происходит с большими целыми числами. Elixir прекрасно справляется с поддержкой даже гигантских чисел. Попробуйте смешивать в вычислениях целые и вещественные числа и посмотрите, что из этого получится. Все это должно быть для вас не очень сложно.

Глава 2

Функции и модули

Подобно большинству языков программирования, Elixir позволяет определять функции, помогающие представлять повторяющиеся вычисления. Функции Elixir могут быть очень сложными, но в своей основе это очень простые элементы программ.

Игры с fn

Создать функцию в оболочке IEx можно с помощью ключевого слова `fn`. Например, ниже показано, как создать функцию, вычисляющую скорость падающего объекта на определенном расстоянии от точки сброса, выраженном в метрах:

```
iex(1)> fall_velocity = fn (distance) -> :math.sqrt(2 * 9.8 * distance) end
#Function<6.6.111823515/1 in :erl_eval.expr/5>
```

Эта инструкция свяжет переменную `fall_velocity` с функцией, принимающей аргумент `distance`. (Круглые скобки, окружающие аргумент, можно опустить.) Функция возвращает (нам больше нравится читать стрелку `->` как «производит») квадратный корень из удвоенного произведения постоянной ускорения свободного падения для Земли (9,8 м/с^2) на расстояние (в метрах). Определение функции завершается ключевым словом `end`.

Возвращаемое значение в интерактивной оболочке, `Function <6.6.111823515/1 in :erl_eval.expr/5>`, само по себе не имеет особого смысла, но оно сообщает, что функция создана и никаких ошибок не возникло. (Точный формат этого возвращаемого значения зависит от версии Elixir, поэтому у вас оно может выглядеть несколько иначе.)

Связывание функции с переменной `fall_velocity` позволяет использовать эту переменную для вычисления скорости падающего объекта на Земле: