



Содержание

Предисловие	10
Для кого предназначена эта книга	10
Графические выделения.....	10
О примерах кода	11
Как с нами связаться	11
Благодарности	12
Глава 1. Все, что нужно знать о Ruby	14
Интерактивная оболочка Ruby	14
Значения	15
Простые данные	15
Структуры данных	16
Процедуры	17
Поток управления.....	18
Объекты и методы	18
Классы и модули	20
Прочее	21
Локальные переменные и присваивание.....	22
Строковая интерполяция	22
Инспектирование объектов.....	22
Печать строк	23
Методы с переменным числом аргументов	23
Блоки.....	24
Модуль Enumerable	25
Класс Struct	26
Партизанское латание	27
Определение констант.....	28
Удаление констант	28

Часть I. ПРОГРАММЫ И МАШИНЫ	30
Глава 2. Семантика программ	32
В чем смысл слова «смысл»?	33
Синтаксис	35
Операционная семантика.....	36
Семантика мелких шагов.....	37
Выражения	39
Предложения.....	50
Корректность.....	60
Приложения.....	61
Семантика крупных шагов	62
Выражения	63
Предложения.....	65
Приложения.....	68
Денотационная семантика	70
Выражения	71
Предложения.....	75
Сравнение способов определения семантики	76
Приложения.....	77
Формальная семантика на практике.....	79
Формализм	79
Поиск смысла.....	80
Альтернативы	81
Реализация синтаксических анализаторов.....	82
Глава 3. Простейшие компьютеры	88
Детерминированные конечные автоматы.....	88
Состояния, правила и входной поток	89
Вывод.....	90
Детерминированность.....	91
Моделирование	92
Недетерминированные конечные автоматы	96
Недетерминированность	96
Свободные переходы.....	104
Регулярные выражения	108
Синтаксис.....	109
Семантика	112
Синтаксический анализ	122
Эквивалентность.....	124
Минимизация ДКА	134

Глава 4. Кое-что помощнее	136
Детерминированные автоматы с магазинной памятью.....	140
Память.....	140
Правила.....	142
Детерминированность.....	144
Моделирование	145
Недетерминированные автоматы с магазинной памятью	152
Моделирование	156
Неэквивалентность.....	159
Разбор с помощью автоматов с магазинной памятью.....	160
Лексический анализ	161
Синтаксический анализ	163
Применение на практике	168
Насколько мощнее?	169
Глава 5. Окончательная машина	172
Детерминированные машины Тьюринга	172
Память.....	173
Правила.....	176
Детерминированность.....	180
Моделирование	180
Недетерминированные машины Тьюринга	187
Максимальная мощность	188
Внутренняя память	189
Подпрограммы	192
Несколько лент	194
Многомерная лента	195
Машины общего назначения	196
Кодирование	198
Моделирование	200
Часть II. ВЫЧИСЛЕНИЯ И ВЫЧИСЛИМОСТЬ	201
Глава 6. Программирование на пустом месте	203
Имитация лямбда-исчисления	204
Работа с процедурами	205
Задача	207
Числа.....	209
Булевы значения.....	213

Предикаты	217
Пары.....	218
Операции над числами	219
Списки.....	228
Строки.....	231
Решение	234
Более сложные приемы программирования	238
Реализация лямбда-исчисления	245
Синтаксис.....	245
Семантика	247
Синтаксический разбор	253
Глава 7. Универсальность повсюду	256
Лямбда-исчисление	257
Частично рекурсивные функции	260
SKI-исчисление	266
Iota	276
Таг-системы	280
Циклические таг-системы	289
Игра «Жизнь» Конвея.....	300
Правило 110.....	303
Вольфрамова 2,3 машина Тьюринга	307
Глава 8. Невозможные программы.....	308
Факты как они есть	309
Универсальные системы могут выполнять алгоритмы.....	309
Программы могут замещать машины Тьюринга	313
Код – это данные	314
Универсальные системы могут заикликоваться.....	316
Программы могут ссылаться сами на себя.....	323
Разрешимость.....	329
Проблема остановки	331
Построение анализатора остановки	331
Это никогда работать не будет	334
Другие неразрешимые проблемы	339
Печальные следствия.....	342
Почему так происходит?.....	345
Жизнь в условиях невычислимости	346

Глава 9. Программирование в игрушечной стране	349
Абстрактная интерпретация	350
Планирование маршрута	351
Абстракция: умножение знаков.....	352
Аппроксимация и безопасность: сложение знаков.....	356
Статическая семантика	361
Реализация.....	363
Достоинства и ограничения	371
Приложения	374
Послесловие	376
Предметный указатель	378



Предисловие

Для кого предназначена эта книга

Это книга для программистов, интересующихся языками программирования и теорией вычислений, в особенности для тех, у кого нет формальной подготовки в области математики или информатики.

Если вам интересно расширить кругозор, познакомившись с разделами информатики, в которых изучаются программы, языки и машины, но пугает математический формализм, часто сопутствующий изложению этих тем, то эта книга для вас. Вместо сложной нотации мы будем использовать код для объяснения теоретических идей, превратив их тем самым в интерактивные инструменты, с которыми вы можете экспериментировать в удобном для себя темпе.

Предполагается, что вы знаете хотя бы один современный язык программирования, например: Ruby, Python, JavaScript, Java или C#. Все примеры написаны на Ruby, но если вы знакомы с любым другим языком, то все равно сможете понять код. Однако эта книга *не является* руководством ни по правильному написанию программ на Ruby, ни по объектно-ориентированному проектированию. Я стремился, чтобы код был кратким и ясным, но необязательно удобным для сопровождения; задача состояла в том, чтобы с помощью Ruby объяснить информатику, а не наоборот. Это также не учебник и не энциклопедия, поэтому вместо формальных рассуждений и строгих доказательств я попытаюсь раскрыть некоторые интересные идеи и побудить вас к более углубленному изучению.

Графические выделения

В книге применяются следующие графические выделения:

Курсив обозначает новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

Моноширинный шрифт так набраны листинги программ, а также элементы программ внутри основного текста, например, имена переменных и функций, типы данных, переменные окружения, предложения и ключевые слова языка.

Моноширинный полужирный команды и иной текст, который пользователь должен вводить буквально.

Моноширинный курсив текст, вместо которого нужно подставить значения, вводимые пользователем или определяемые контекстом.



Таким значком обозначаются советы, предложения и замечания общего характера.



Таким значком обозначаются предупреждения и предостережения.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Understanding Computation by Tom Stuart(O'Reilly). Copyright 2013 Tom Stuart, 978-1-4493-2927-3».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу permissions@oreilly.com.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North

Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой выкладываются списки замеченных ошибок, примеры и разного рода дополнительная информация. Адрес страницы:

<http://oreil.ly/understanding-computation>

Замечания и вопросы технического характера следует отправлять по адресу:

bookquestions@oreilly.com

Дополнительную информацию о наших книгах, конференциях, ресурсных центрах и сети O'Reilly Network можно найти по на сайте:

<http://www.oreilly.com>

Ищите нас на Facebook: <http://facebook.com/oreilly>.

Следуйте за нами на Twitter: <http://twitter.com/oreillymedia>.

Смотрите нас на YouTube: <http://www.youtube.com/oreillymedia>.

Благодарности

Я благодарен за гостеприимство компании Go Free Range, которая предоставила мне на время написания этой книги место в офисе, чашку чая и дружескую беседу. Без ее щедрой поддержки я бы точно пошел по стопам Джека Торренса¹.

Спасибо вам, Джеймс Адам (James Adam), Пол Баттли (Paul Battley), Джеймс Коглан (James Coglan), Питер Флетчер (Peter Fletcher), Крис Лоуис (Chris Lowis) и Мэррей Стил (Murray Steele), за отзывы на черновики и вам, Габриэль Кернейс (Gabriel Kerneis) и Алекс Стэнгл, за технические рецензии. Благодаря вашим глубоким замечаниям книга стала неизмеримо лучше. Хочу также поблагодарить Алана Майкрофта (Alan Mucroft) из Кэмбриджского университета за знания, которыми он щедро делился, и за ободрение.

Многие сотрудники издательства O'Reilly помогли довести этот проект до завершения, но особенно я благодарен Майку Лоукидесу (Mike Loukides) и Саймону Сен-Лорану (Simon St.Laurent) за

¹ Персонаж романа «Сияние» Стивена Кинга и одноименного фильма с Джеком Николсоном в главной роли. – *Прим. перев.*

энтузиазм на ранних этапах и веру в идею, Натану Джепсону (Nathan Jepson) за совет о том, как сделать из идеи книгу, и Сандерсу Клейнфельду (Sanders Kleinfeld), который с юмором относился к моим неустанным попыткам научиться правильно расставлять знаки препинания.

Спасибо моим родителям, которые дали неуугомонному дитяти возможность и подтолкнули его тратить все свое время на возню с компьютерами. И еще Лейле, которая напоминала, как низать на строчки чертовы слова, всякий раз, когда я забывал о работе. В конце концов я все-таки добрался до конца.



Глава 1. Все, что нужно знать о Ruby

Код в этой книге написан на Ruby, языке программирования, который был задуман простым и дружелюбным, чтобы работа с ним доставляла удовольствие. Я выбрал его за ясность и гибкость, но ничто в этой книге не зависит от особенностей, присущих только Ruby, поэтому можете переписать примеры на своем любимом языке, особенно если он динамический, как Python или JavaScript, – если это поможет усвоению идей.

Все примеры совместимы с версиями Ruby 2.0 и Ruby 1.9. Получить дополнительные сведения о Ruby и скачать официальную документацию можно на сайте Ruby по адресу <http://www.ruby-lang.org>.

Сейчас мы совершим небольшой экскурс в возможности Ruby. Нас будут интересовать в первую очередь те части языка, которые используются в этой книге; если хотите узнать больше, начните с книги «The Ruby Programming Language», вышедшей в издательстве O'Reilly¹.



Если вы уже знакомы с Ruby, можете, не опасаясь что-то пропустить, сразу переходить к главе 2.

Интерактивная оболочка Ruby

Одна из самых удобных черт Ruby – его интерактивная консоль *IRB*, которая позволяет вводить код и сразу же видеть результат его выполнения. В этой книге мы постоянно будем использовать *IRB*, чтобы интерактивно исследовать, как работает наш код.

¹ Д. Флэнаган, Ю. Мацумото. Язык программирования Ruby. – Питер, 2011. – Прим. перев.

Для запуска IRB на своей машине введите в командной строке слово `irb`. IRB выводит приглашение `>>`, когда ожидает ввод выражения Ruby. После того как вы введете выражение и нажмете клавишу Enter, код будет выполнен, и на экране появится результат после знака `=>`:

```
$ irb --simple-prompt
>> 1 + 2
=> 3
>> 'hello world'.length
=> 11
```

Встретив в книге приглашения `>>` и `=>`, знайте, что мы работаем с IRB. Чтобы длинные листинги было проще читать, мы показываем их без приглашений, но при этом предполагаем, что содержащийся в них код будет набран или скопирован в IRB. Так что, если в книге встречается код типа...

```
x = 2
y = 3
z = x + y
```

...то можно воспользоваться результатами его выполнения в IRB:

```
>> x * y * z
=> 30
```

Значения

Ruby – язык, ориентированный на выражения: любой допустимый фрагмент кода порождает при выполнении значение. Ниже дается краткий обзор различных видов значений в Ruby.

Простые данные

Как и следовало ожидать, Ruby поддерживает булевы значения, числа и строки, а также стандартные операции над ними:

```
>> (true && false) || true
=> true
>> (3 + 3) * (14 / 2)
=> 42
>> 'hello' + ' world'
=> "hello world"
>> 'hello world'.slice(6)
=> "w"
```

Символ в Ruby – это облегченное неизменяемое значение, представляющее имя. Символы широко используются в Ruby, поскольку они проще и потребляют меньше памяти, чем строки; чаще всего они встречаются в качестве ключей хешей (см. ниже раздел «Структуры данных»). Символьные литералы записываются с двоеточием в начале:

```
>> :my_symbol
=> :my_symbol
>> :my_symbol == :my_symbol
=> true
>> :my_symbol == :another_symbol
=> false
```

Специальное значение `nil` обозначает отсутствие полезного значения:

```
>> 'hello world'.slice(11)
=> nil
```

Структуры данных

Литеральные массивы в Ruby записываются в виде списка значений через запятую, заключенного в квадратные скобки:

```
>> numbers = ['zero', 'one', 'two']
=> ["zero", "one", "two"]
>> numbers[1]
=> "one"
>> numbers.push('three', 'four')
=> ["zero", "one", "two", "three", "four"]
>> numbers
=> ["zero", "one", "two", "three", "four"]
>> numbers.drop(2)
=> ["two", "three", "four"]
```

Диапазон – это коллекция значений между минимумом и максимумом. Диапазон обозначается крайними значениями, разделенными двумя точками:

```
>> ages = 18..30
=> 18..30
>> ages.entries
=> [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
>> ages.include?(25)
=> true
>> ages.include?(33)
=> false
```

Хеш – это коллекция, в которой каждое значение ассоциировано с ключом; в других языках программирования эта структура данных называется «словарем», «отображением» или «ассоциативным массивом». Литеральный хеш записывается в виде заключенного в фигурные скобки списка пар *ключ => значение* через запятую:

```
>> fruit = { 'a' => 'apple', 'b' => 'banana', 'c' => 'coconut' }
=> {"a"=>"apple", "b"=>"banana", "c"=>"coconut"}
>> fruit['b']
=> "banana"
>> fruit['d'] = 'date'
=> "date"
>> fruit
=> {"a"=>"apple", "b"=>"banana", "c"=>"coconut", "d"=>"date"}
```

В роли ключей хеша часто выступают символы, поэтому в Ruby имеется альтернативный синтаксис *key: value* для записи пары ключ-значение, в которой ключ является символом. Эта запись компактнее, чем *key => value* и выглядит точно так же, как популярный формат JSON для представления объектов в JavaScript:

```
>> dimensions = { width: 1000, height: 2250, depth: 250 }
=> {:width=>1000, :height=>2250, :depth=>250}
>> dimensions[:depth]
=> 250
```

Процедуры

Процедурой, или *proc-объектом* называется невыполненный фрагмент Ruby-кода, который можно передать в другое место программы и выполнить по запросу; в других языках такая конструкция называется «анонимной функцией» или «лямбдой». Существует несколько способов записать литеральную процедуру, из них самый компактный – синтаксис *-> arguments { body }*:

```
>> multiply = -> x, y { x * y }
=> #<Proc (lambda)>
>> multiply.call(6, 9)
=> 54
>> multiply.call(2, 3)
=> 6
```

Помимо синтаксиса *.call*, процедуру можно вызвать, передав аргументы в квадратных скобках:

```
>> multiply[3, 4]
=> 12
```

Поток управления

В Ruby имеются выражения `if`, `case` и `while`, которые работают привычным образом:

```
>> if 2 < 3
    'less'
  else
    'more'
  end
=> "less"
>> quantify =
  -> number {
    case number
    when 1
      'one'
    when 2
      'a couple'
    else
      'many'
    end
  }
=> #<Proc (lambda)>
>> quantify.call(2)
=> "a couple"
>> quantify.call(10)
=> "many"
>> x = 1
=> 1
>> while x < 1000
    x = x * 2
  end
=> nil
>> x
=> 1024
```

Объекты и методы

Ruby похож на другие динамические языки программирования, но обладает одной необычной особенностью: любое значение является *объектом*, и объекты общаются между собой, отправляя *сообщения*¹. У каждого объекта имеется свой набор *методов*, которые определяют его реакцию на различные сообщения.

Сообщение имеет имя и необязательные аргументы. Получив сообщение, объект выполняет соответствующий ему метод, передавая ему содержащиеся в сообщении аргументы. Именно так в Ruby

¹ Эта терминология заимствована из языка Smalltalk, который оказал самое непосредственное влияние на дизайн Ruby.

выполняется любая операция; даже запись `1 + 2` означает «отправить объекту 1 сообщение, которое называется `+`, с аргументом 2», а у объекта 1 есть метод `#+` для обработки такого сообщения.

Мы можем определять собственные методы с помощью ключевого слова `def`:

```
>> o = Object.new
=> #<Object>
>> def o.add(x, y)
      x + y
    end
=> nil
>> o.add(2, 3)
=> 5
```

Здесь мы создаем новый объект, посылая сообщение `new` специальному встроенному объекту `Object`; после того как объект создан, мы определяем для него метод `#add`, который складывает два аргумента и возвращает их сумму; явно употреблять ключевое слово `return` необязательно, поскольку метод автоматически возвращает значение последнего вычисленного выражения. Если послать этому объекту сообщение `add` с аргументами 2 и 3, то будет выполнен его метод `#add`, и в ответ мы получим ожидаемый результат.

Для отправки сообщения объекту обычно записывается объект-получатель и имя сообщения, разделенные точкой (например, `o.add`), но Ruby также хранит ссылку на *текущий объект* (она называется `self`) и позволяет отправить этому объекту сообщение, указав только его имя и не указывая явно получателя. Например, внутри определения метода текущим всегда является объект, получивший сообщение, в ответ на которое был вызван этот метод, поэтому из любого метода объекта мы можем отправлять другие сообщения тому же объекту, не указывая его явно:

```
>> def o.add_twice(x, y)
      add(x, y) + add(x, y)
    end
=> nil
>> o.add_twice(2, 3)
=> 10
```

Отметим, что для отправки сообщения `add` объекту `o` из метода `#add_twice` можно писать `add(x, y)` вместо `o.add(x, y)`, потому что `o` – именно тот объект, которому было отправлено сообщение `add_twice`.

Вне определения какого-либо метода текущим является специальный объект верхнего уровня, который называется `main`, ему доставляются любые сообщения, для которых не указан получатель. Аналогично, определения методов, в которых не указан объект, становятся доступны через `main`:

```
>> def multiply(a, b)
      a * b
    end
=> nil
>> multiply(2, 3)
=> 6
```

Классы и модули

Удобно, когда у нескольких объектов есть возможность пользоваться одним и тем же определением метода. В Ruby мы можем поместить определения методов в *класс*, а затем создавать объекты, посылая сообщение `new` этому классу. Возвращаемые в ответ объекты называются *экземплярами* класса и включают все методы этого класса. Например:

```
>> class Calculator
      def divide(x, y)
        x / y
      end
    end
=> nil
>> c = Calculator.new
=> #<Calculator>
>> c.class
=> Calculator
>> c.divide(10, 2)
=> 5
```

Отметим, что определение метода внутри определения `class` добавляет метод экземплярам этого класса, а не объекту `main`:

```
>> divide(10, 2)
NoMethodError: undefined method `divide' for main:Object
```

Один класс может «подтянуть» определения методов другого класса благодаря *наследованию*:

```
>> class MultiplyingCalculator < Calculator
      def multiply(x, y)
        x * y
      end
    end
```

```
    end
  end
=> nil
>> mc = MultiplyingCalculator.new
=> #<MultiplyingCalculator>
>> mc.class
=> MultiplyingCalculator
>> mc.class.superclass
=> Calculator
>> mc.multiply(10, 2)
=> 20
>> mc.divide(10, 2)
=> 5
```

Метод подкласса может вызвать одноименный метод своего суперкласса, воспользовавшись ключевым словом `super`:

```
>> class BinaryMultiplyingCalculator < MultiplyingCalculator
  def multiply(x, y)
    result = super(x, y)
    result.to_s(2)
  end
end
=> nil
>> bmc = BinaryMultiplyingCalculator.new
=> #<BinaryMultiplyingCalculator>
>> bmc.multiply(10, 2)
=> "10100"
```

Еще один способ обобществить определения методов – объявить их в *модуле*, который затем можно включить в любой класс:

```
>> module Addition
  def add(x, y)
    x + y
  end
end
=> nil
>> class AddingCalculator
  include Addition
end
=> AddingCalculator
>> ac = AddingCalculator.new
=> #<AddingCalculator>
>> ac.add(10, 2)
=> 12
```

Прочее

Ниже приводится сводка полезных возможностей Ruby, которые будут встречаться далее в примерах.

Локальные переменные и присваивание

Как мы уже видели, в Ruby можно объявить переменную, просто присвоив ей значение:

```
>> greeting = 'hello'
=> "hello"
>> greeting
=> "hello"
```

Можно также воспользоваться синтаксисом *параллельного присваивания* для одновременной записи значений в несколько переменных:

```
>> width, height, depth = [1000, 2250, 250]
=> [1000, 2250, 250]
>> height
=> 2250
```

Строковая интерполяция

Строки можно заключать в одиночные или двойные кавычки. Ruby автоматически производит *интерполяцию* в строках с двойными кавычками, то есть заменяет выражение `#{expression}` результатом его вычисления:

```
>> "hello #{'dlrow'.reverse}"
=> "hello world"
```

Если интерполированное выражение возвращает объект, не являющийся строкой, то этому объекту автоматически посылается сообщение `to_s` и ожидается, что оно вернет строку, которую можно использовать вместо объекта. Этим можно воспользоваться для управления представлением интерполированных объектов:

```
>> o = Object.new
=> #<Object>
>> def o.to_s
  'a new object'
end
=> nil
>> "here is #{o}"
=> "here is a new object"
```

Инспектирование объектов

Нечто подобное происходит, когда IRB должна отобразить объект: объекту посылается сообщение `inspect` и ожидается, что он

вернет свое строковое представление. Для всех объектов в Ruby по умолчанию определена разумная реализация метода `#inspect`, но предоставив собственное определение, мы сможем контролировать внешний вид объекта на консоли:

```
>> o = Object.new
=> #<Object>
>> def o.inspect
  'my object'
end
=> nil
>> o
=> [my object]
```

Печать строк

У каждого объекта в Ruby (в том числе у `main`) имеется метод `#puts`, который можно использовать для печати строк на стандартный вывод:

```
>> x = 128
=> 128
>> while x < 1000
  puts "x is #{x}"
  x = x * 2
end
x is 128
x is 256
x is 512
=> nil
```

Методы с переменным числом аргументов

В определении метода можно указать оператор `*`, означающий, что метод поддерживает переменное число аргументов:

```
>> def join_with_commas(*words)
  words.join(', ')
end
=> nil
>> join_with_commas('one', 'two', 'three')
=> "one, two, three"
```

В определении метода не может быть более одного параметра переменной длины, но обычные параметры могут находиться по обе стороны от него:

```
>> def join_with_commas(before, *words, after)
  before + words.join(', ') + after
```

```
end
=> nil
>> join_with_commas('Testing: ', 'one', 'two', 'three', '.')
=> "Testing: one, two, three."
```

Оператор `*` можно использовать также для того, чтобы передавать каждый элемент массива как отдельный аргумент при отправке сообщения:

```
>> arguments = ['Testing: ', 'one', 'two', 'three', '.']
=> ["Testing: ", "one", "two", "three", "."]
>> join_with_commas(*arguments)
=> "Testing: one, two, three."
```

И наконец, оператор `*` работает совместно с параллельным присваиванием:

```
>> before, *words, after = ['Testing: ', 'one', 'two', 'three', '.']
=> ["Testing: ", "one", "two", "three", "."]
>> before
=> "Testing: "
>> words
=> ["one", "two", "three"]
>> after
=> "."
```

Блоки

Блоком называется фрагмент Ruby-кода, заключенный в операторные скобки `do/end` или в фигурные скобки. Методы могут принимать в качестве аргумента неявный блок и вызывать содержащийся в нем код с помощью ключевого слова `yield`:

```
>> def do_three_times
  yield
  yield
  yield
end
=> nil
>> do_three_times { puts 'hello' }
hello
hello
hello
=> nil
```

Блок может принимать аргументы:

```
>> def do_three_times
  yield('first')
```

```
        yield('second')
        yield('third')
    end
=> nil
>> do_three_times { |n| puts "#{n}: hello" }
first: hello
second: hello
third: hello
=> nil
```

Предложение `yield` возвращает результат выполнения блока:

```
>> def number_names
    [yield('one'), yield('two'), yield('three')].join(', ')
end
=> nil
>> number_names { |name| name.upcase.reverse }
=> "ENO, OWT, EERHT"
```

Модуль *Enumerable*

В Ruby имеется встроенный модуль `Enumerable`, который включает классы `Array`, `Hash`, `Range` и другие, представляющие коллекции значений. Этот модуль содержит полезные методы для обхода, поиска и сортировки коллекций, причем многие методы ожидают на входе блок. Обычно код в переданном блоке выполняется для некоторых или всех значений в коллекции в зависимости от того, что делает метод. Например:

```
>> (1..10).count { |number| number.even? }
=> 5
>> (1..10).select { |number| number.even? }
=> [2, 4, 6, 8, 10]
>> (1..10).any? { |number| number < 8 }
=> true
>> (1..10).all? { |number| number < 8 }
=> false
>> (1..5).each do |number|
    if number.even?
        puts "#{number} is even"
    else
        puts "#{number} is odd"
    end
end
1 is odd
2 is even
3 is odd
4 is even
5 is odd
=> 1..5
>> (1..10).map { |number| number * 3 }
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```
