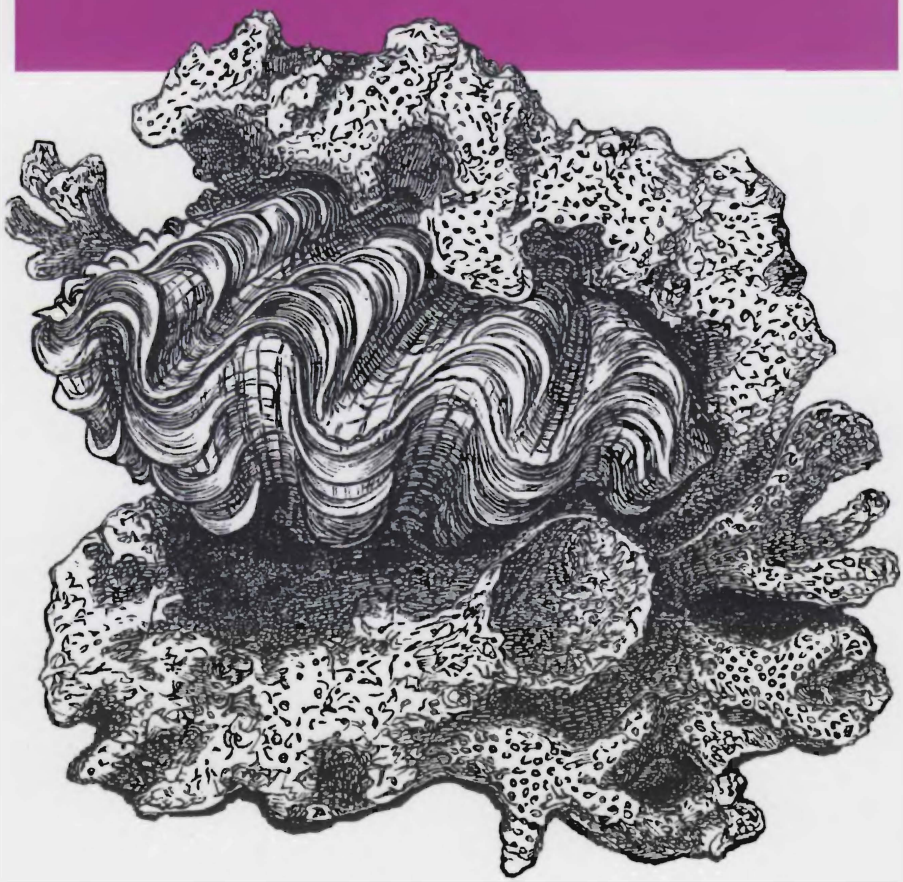
Теория вычислений для программистов



Том Стюарт

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

O'REILLY®

Том Стюарт

Теория вычислений для программистов

Tom Stuart

Understanding Computation

O'REILLY®

Том Стюарт

Теория вычислений для программистов

Москва, 2014

УДК 004.421.2
ББК 32.973-018
С759

Стюарт Т.

С759 Теория вычислений для программистов / Пер. с анг. А. А. Слинкин. – М.: ДМК Пресс, 2014. – 384 с.: ил.

ISBN 978-5-94074-979-0

Наконец-то появился увлекательный и практичный способ изучать теорию вычислений и проектирование языков программирования!

В этой книге теоретическая информатика излагается в хорошо знакомом вам контексте, что поможет оценить, почему ее идеи важны и как они отражаются на том, чем программист изо дня в день занимается на работе.

Вместо математической нотации или незнакомого академичного языка программирования типа Haskell или Lisp в этой книге для объяснения формальной семантики, теории автоматов и функционального программирования вкупе с лямбда-исчислением применяется язык Ruby, сведенный к минимуму.

Издание предназначено для программистов любой квалификации, знакомых хотя бы с одним из современных языков, но не имеющих формальной подготовки в информатике.

УДК 004.421.2
ББК 32.973-018

ISBN 978-1-449-32927-3 (анг.)
ISBN 978-5-94074-979-0 (рус.)

Содержание

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 Торренса[1].

Спасибо вам, Джеймс Адам (James Adam), Пол Баттли (Paul Battley), Джеймс Коглан (James Coglan), Питер Флетчер (Peter Fletcher), Крис Лоуис (Chris Lowis) и Маррей 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 Mycroft) из Кэмбриджского университета за знания, которыми он щедро делился, и за ободрение.

Многие сотрудники издательства O'Reilly помогали 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 неугомонному 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'Reilly[1].

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

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

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

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

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

Часто бывает, что блок принимает один аргумент и посылает ему одно сообщение без аргументов, поэтому Ruby предлагает нотацию `&:message` для сокращенной записи блока `{ |object| object.message }`:

```
>> (1..10).select(&:even?)
=> [2, 4, 6, 8, 10]
>> ['one', 'two', 'three'].map(&:upcase)
=> ["ONE", "TWO", "THREE"]
```

Один из методов `Enumerable`, `#flat_map`, можно использовать для того, чтобы вычислить порождающий массив блок для каждого значения в коллекции и конкатенировать результаты:

```
>> ['one', 'two', 'three'].map(&:chars)
=> [["o", "n", "e"], ["t", "w", "o"], ["t", "h", "r", "e", "e"]]
>> ['one', 'two', 'three'].flat_map(&:chars)
=> ["o", "n", "e", "t", "w", "o", "t", "h", "r", "e", "e"]
```

Еще один полезный метод, `#inject`, вычисляет блок для каждого значения в коллекции и аккумулирует результаты:

```
>> (1..10).inject(0) { |result, number| result + number }
=> 55
>> (1..10).inject(1) { |result, number| result + number }
=> 3628800
>> ['one', 'two', 'three'].inject('Words:') { |result, word| "#{result} #{word}" }
=> "Words: one two three"
```

Класс Struct

`Struct` – это специальный класс Ruby, задача которого – генерировать другие классы. Класс, сгенерированный с помощью `Struct`, содержит методы чтения и установки для атрибутов с именами, переданными методу `Struct.new`. Обычно создается подкласс сгенерированного `Struct` класса; этому подклассу можно присвоить имя и поместить в него дополнительные методы. Например, чтобы создать класс `Point` с атрибутами `x` и `y`, мы можем написать:

```
class Point < Struct.new(:x, :y)
  def +(other_point)
    Point.new(x + other_point.x, y + other_point.y)
  end

  def inspect
   "#<Point (#{x}, #{y})>"
  end
end
```

Теперь можно создавать экземпляры класса `Point`, инспектировать их в IRB и посылать им сообщения:

```
>> a = Point.new(2, 3)
=> #<Point (2, 3)>
>> b = Point.new(10, 20)
=> #<Point (10, 20)>
>> a + b
=> #<Point (12, 23)>
```

Помимо определенных нами методов, экземпляр класса `Point` отвечает на сообщения `x` и `x=`, означающие чтение и установку атрибута `x`, и аналогично – на сообщения `y` и `y=`:

```
>> a.x
=> 2
>> a.x = 35
=> 35
>> a + b
=> #<Point (45, 23)>
```

У классов, сгенерированных методом `Struct.new`, есть и другая полезная функциональность, например, реализация метода `#==`, который сравнивает на равенство атрибуты двух экземпляров:

```
>> Point.new(4, 5) == Point.new(4, 5)
=> true
>> Point.new(4, 5) == Point.new(6, 7)
=> false
```

Партизанское латание

В любой момент в существующий класс или модуль можно добавить новые методы. Эта весьма мощная техника, которую обычно называют *партизанским латанием* (monkey patching[1]), позволяет расширять поведение существующих классов:

```
>> class Point
     def -(other_point)
       Point.new(x - other_point.x, y - other_point.y)
     end
   end
=> nil
```

[1] Первоначально эта техника называлась *guerilla patch*, но затем *guerilla* по созвучию трансформировалось в *gorilla*, а оттуда уже недалеко до *monkey* (обезьяна). Но поскольку слово «партизанский» лучше отражает смысл техники, мы будем использовать его. – *Прим. перев.*

```
>> Point.new(10, 15) - Point.new(1, 1)
=> #<Point (9, 14)>
```

Можно даже накладывать партизанские заплаты на встроенные в Ruby классы:

```
>> class String
     def shout
       upcase + '!!!'
     end
   end
=> nil
>> 'hello world'.shout
=> "HELLO WORLD!!!"
```

Определение констант

Ruby поддерживает особый вид переменных, называемых *константами*, потому что им не следует присваивать новое значение после создания (Ruby не запрещает присваивать значения константам, но выдает предупреждение, дабы программист знал, что делает нечто неправильное). Любая переменная, имя которой начинается с заглавной буквы, считается константой. Новые константы можно определять как на верхнем уровне, так и внутри класса или модуля:

```
>> NUMBERS = [4, 8, 15, 16, 23, 42]
=> [4, 8, 15, 16, 23, 42]
>> class Greetings
     ENGLISH = 'hello'
     FRENCH = 'bonjour'
     GERMAN = 'guten Tag'
   end
=> "guten Tag"
>> NUMBERS.last
=> 42
>> Greetings::FRENCH
=> "bonjour"
```

Имена классов и модулей всегда начинаются с заглавной буквы, поэтому также являются константами.

Удаление констант

Исследуя некую идею в IRB, иногда полезно попросить Ruby забыть о ранее определенной константе, особенно если это имя класса или модуля, который мы хотим переопределить заново, а не латать существующее определение по-партизански. Для удаления

константы верхнего уровня нужно послать сообщение `remove_const` классу `Object`, передав имя константы в виде символа:

```
>> NUMBERS.last
 => 42
>> Object.send(:remove_const, :NUMBERS)
 => [4, 8, 15, 16, 23, 42]
>> NUMBERS.last
NameError: uninitialized constant NUMBERS
>> Greetings::GERMAN
 => "guten Tag"
>> Object.send(:remove_const, :Greetings)
 => Greetings
>> Greetings::GERMAN
NameError: uninitialized constant Greetings
```

Мы должны использовать синтаксис `Object.send(:remove_const, :NAME)`, а не просто `Object.remove_const(:NAME)`, потому что `remove_const` – закрытый метод, который в обычных обстоятельствах можно вызывать, только отправив сообщение изнутри самого класса `Object`; использование `Object.send` позволяет временно обойти это ограничение.

Часть I. ПРОГРАММЫ И МАШИНЫ

Что такое *вычисление*? Разные люди понимают под этим словом разные вещи, но все согласятся, что когда компьютер считывает программу, выполняет программу, читает исходные данные и в конечном итоге порождает результат, определенно имеет место некое вычисление. Это даст нам отправную точку: вычисление – это обозначение того, *что делает компьютер*.

Чтобы создать среду, в которой могут происходить вычисления этого знакомого вида, нам понадобятся три ингредиента:

- *Машина*, способная производить вычисления.
- *Язык* для написания команд, понятных машине.
- *Программа*, написанная на этом языке, которая точно описывает, какое вычисление должна произвести машина.

Поэтому в первой части этой книги речь пойдет о машинах, языках и программах – что это такое, как они себя ведут, как можно их моделировать и изучать и как воспользоваться ими для совершения полезной работы. Изучив эти три ингредиента, мы сможем выработать более отчетливое интуитивное понимание того, что такое вычисление и как оно происходит.

В главе 2 мы спроектируем и реализуем игрушечный язык программирования и изучим несколько способов специфицировать его семантику. Именно понимание семантики языка позволяет взять безжизненный кусок кода и превратить его в динамичный исполняемый процесс; каждый способ специфицирования дает конкретную стратегию исполнения программы, и в конечном итоге у нас образуется несколько вариантов реализации одного и того же языка.

Мы увидим, что программирование – это искусство собрать точно определенную конструкцию, которую машина можно разобрать, проанализировать и затем интерпретировать, породив тем самым

вычисление. И что еще более важно, мы обнаружим, что реализация языка программирования – простое и увлекательное дело; хотя синтаксический разбор, интерпретация и компиляция поначалу могут вселить трепет, на самом деле ничего сложного в них нет, и возиться с этим – сплошное удовольствие.

Программы бесполезны без машин, способных их выполнить, поэтому в главе 3 мы спроектируем несколько очень простых машин, умеющих выполнять несложные, жестко зашитые задачи. На этом скромном фундаменте мы построим более изощренные машины в главе 4, а в главе 5 увидим, как конструируется универсальное вычислительное устройство, которым можно управлять программно.

Добравшись до части II, мы уже будем представлять себе весь спектр вычислительной мощности: одни машины с очень ограниченными возможностями, другие – более полезные, но все еще раздражающе стесненные, и, наконец, самые мощные, которые мы хотя бы теоретически сумеем построить.

Глава 2. Семантика программ

Не думай, чувствуй! Это как указывать пальцем на Луну. Не концентрируйся на пальце, или пропустишь эту божественную красоту.

– Брюс Ли

Языки программирования и программы, которые на них пишутся, – основа работы программиста. Мы используем их, чтобы прояснить для себя сложные идеи, чтобы довести свои идеи до сведения коллег, и, самое главное, чтобы реализовать эти идеи на компьютере. Как человеческое общество не смогло бы существовать без естественных языков, так и глобальное сообщество программистов зависит от языков программирования для передачи и воплощения в жизнь своих идей. При этом каждая успешная программа становится частью фундамента, на котором можно возвести следующий уровень идей.

Программисты – существа практичные и прагматичные. Мы часто изучаем новый язык программирования, читая документацию, выполняя примеры из пособий, изучая существующие программы, экспериментируя с собственными простенькими программами и не особенно задумываясь о том, что эти программы *означают*. Иногда в процессе обучения мы продвигаемся вперед методом проб и ошибок: пытаемся понять какую-то часть языка, глядя на примеры и в документацию, затем пробуем написать нечто подобное, после этого все ломается, и мы должны возвращаться и начинать все заново, пока не сваяем нечто, работающее в большинстве случаев. По мере того как компьютеры и исполняемые на них системы становятся все сложнее, возникает искушение считать программы непостижимыми заклинаниями, которые представляют только самих себя и работают лишь по счастливой случайности.

По суть компьютерного программирования в действительности составляют не *программы*, а *идеи*. Программа – это лишь застывший слепок идеи, мгновенный снимок конструкции, когда-то существовавшей в воображении программиста. Программы стоит писать лишь потому, что у них есть *смысл*. Так что же связывает код с его смыслом и как высказать о смысле программы суждение более конкретное, чем «она делает то, что делает»? В этой главе мы рассмотрим несколько способов поймать ускользающий смысл компьютерной программы и увидим, как вернуть мертвые слепки к жизни.

В чем смысл слова «смысл»?

В лингвистике *семантикой* называют изучение связи между словами и их смыслом: слово «собака» – лишь сочетание штрихов на печатной странице или последовательность колебаний воздуха, создаваемых чьими-то голосовыми связками; ни то, ни другое не имеет никакого отношения к реальной собаке или общей идее собаки. Семантику интересует, как эти конкретные означающие соотносятся со своими абстрактными значениями, а равно фундаментальная природа самих абстрактных значений.

В информатике есть область *формальной семантики*, занимающаяся поиском способов ухватить трудноуловимый смысл программ и использовать его для выявления или доказательства интересных фактов, относящихся к языкам программирования. У формальной семантики имеется широкий спектр применений – от весьма конкретных, типа специфицирования новых языков программирования и придумывания оптимизаций, выполняемых компилятором, до более абстрактных, например, построения математических доказательств правильности программ.

Полная спецификация языка программирования состоит из двух частей: *синтаксиса*, то есть описания того, как может выглядеть программа, и семантики, то есть описания того, что она означает.

Существует множество языков, не имеющих официальной спецификации, а лишь работающий интерпретатор или компилятор. Ruby как раз попадает в эту категорию «спецификации путем реализации»: хотя имеется немало книг и пособий на тему того, как должен работать Ruby, единственным авторитетным источником всей информации является Мацевский Интерпретатор Ruby (Matz's Ruby Interpreter – MRI) – эталонная реализация языка. Если какая-то

документация отличается от фактического поведения MRI, то врет документация; любая сторонняя реализация Ruby, будь то JRuby, Rubinius или MacRuby, обязана в точности имитировать поведение MRI, только тогда она может претендовать на совместимость с языком Ruby. Такой же подход к определению языка на основе первенства реализации характерен и для других языков, например PHP и Perl 5.

Другой способ описать язык программирования – предложить официальную спецификацию, обычно на английском языке. Примерами такого подхода могут служить C++, Java и ECMAScript (стандартизованная версия JavaScript); для этих языков существует не зависящий от реализации стандарт, написанный комитетом, в который входят известные специалисты, и много совместимых со стандартом реализаций. Специфицирование языка посредством официального документа – способ, более строгий, чем следование эталонной реализации, поскольку проектные решения с большей вероятностью являются результатом обдуманного, рационального выбора, а не случайных свойств конкретной реализации. Однако спецификацию обычно трудно читать и очень сложно сказать, есть ли в ней какие-то противоречия, упущения или неоднозначности. В частности, не существует формального способа рассуждать о спецификации на английском языке; мы просто должны внимательно прочитать ее, долго осмысливать и надеяться, что правильно поняли все изложенное.

Текстовая спецификация версии Ruby 1.8.7 существует и даже принята в качестве стандарта ISO (ISO/IEC 30170)[1]. MRI по-прежнему считается канонической спецификацией языка Ruby посредством реализации, хотя в проекте mruby (*https:/github.com/mruby/mruby*) сделана попытка создать облегченную, встраиваемую реализацию Ruby, которая была бы совместима именно со стандартом ISO, а не с MRI.

Третий вариант – воспользоваться математической техникой формальной семантики, чтобы точно описать смысл языка программирования. Цели здесь две: добиться полного отсутствия неоднозначностей и представить спецификацию в виде, допускающем систематический или даже *автоматизированный* анализ, в ходе которого

[1] Доступ к документу ISO/IEC 30170 платный, но его предварительную версию можно бесплатно скачать по адресу *http://www.ipa.go.jp/osc/english/ruby/*.

можно провести исчерпывающую проверку на согласованность, непротиворечивость и отсутствие упущений. Мы еще рассмотрим эти формальные подходы к специфицированию семантики, после того как ближе познакомимся с синтаксисом.

Синтаксис

Традиционная компьютерная программа – это длинная цепочка символов. Любой язык программирования содержит набор правил, описывающих, какие цепочки символов являются допустимыми программами на этом языке; эти правила и составляют *синтаксис* языка.

Синтаксические правила языка позволяют отличить потенциально допустимые программы типа `y = x + 1` от бессмысленного набора знаков типа `>/;x:1@4`. Они также сообщают полезную информацию о том, как читать неоднозначные программы: например, правила приоритета операторов говорят, что выражение `1 + 2 * 3` следует трактовать так, будто оно записано в виде `1 + (2 * 3)`, а не `(1 + 2) * 3`.

Разумеется, компьютерная программа предназначена, прежде всего, для чтения компьютером, и для ее чтения необходим *синтаксический анализатор* – программа, которая умеет читать цепочку символов, представляющую программу, проверять, согласуется ли она с синтаксическими правилами, и преобразовывать в структурированную форму, пригодную для дальнейшей обработки.

Существует целый ряд инструментов, которые могут на основе синтаксических правил языка автоматически породить синтаксический анализатор для него. Детали задания правил и техника получения из них полезных анализаторов в этой главе не рассматриваются (краткий обзор см. в разделе «Реализация синтаксических анализаторов» на стр. 82), но, вообще говоря, анализатор должен прочитать строку типа `y = x + 1` и преобразовать ее в абстрактное синтаксическое дерево (АСД) – представление исходного кода, в котором отброшены несущественные детали, например пробелы, и основное внимание уделено иерархической структуре программы.

Но синтаксис определяет лишь внешнюю сторону программ, а не их семантику. Программа может быть синтаксически правильной, но абсолютно бесполезной; например, программа `y = x + 1` сама по себе не имеет особого смысла, потому что неизвестно, чему было равно `x` в начале, а программа `z = true + 1`, скорее всего, даст неправильные результаты во время выполнения, потому что пытается прибавить

число к булеву значению (разумеется, ее поведение зависит и от других свойств конкретного языка).

Как и следовало ожидать, не существует «единственно верного способа» объяснить, как синтаксис языка программирования соотносится со стоящей за ним семантикой. На самом деле, есть несколько способов конкретно сказать, что означает программа, – с разными компромиссами между формальностью, абстрактностью, выразительностью и практической эффективностью. В следующих разделах мы рассмотрим основные формальные подходы и соотношения между ними.

Операционная семантика

Самый практически полезный способ рассуждать о смысле программы – задать вопрос «*что она делает?*»; что мы ожидаем получить в результате запуска программы? Как различные конструкции языка программирования ведут себя во время выполнения и что происходит, когда мы объединяем их для получения более крупной программы?

Это область *операционной семантики* – способа описать смысл языка программирования путем определения правил исполнения написанных на нем программ устройствами определенного вида. В роли такого устройства часто выступает *абстрактная машина* – воображаемый идеализированный компьютер, предназначенный для одной-единственной цели: объяснить, как будут выполняться программы на некотором языке. Для языков программирования разных типов обычно требуются разные абстрактные машины, позволяющие точно описать поведение во время выполнения.

Применив операционную семантику, мы можем строго и точно описать назначение различных конструкций языка. В отличие от написанной по-английски спецификации языка, в которой могут содержаться скрытые неоднозначности и не рассматриваться важные граничные случаи, формальная операционная спецификация обязана быть явной и однозначной, только тогда она сможет убедительно описать поведение языка.

Семантика мелких шагов

Итак, как спроектировать абстрактную машину и использовать ее для специфицирования операционной семантики языка программирования? Один из возможных способов – представить себе машину, которая вычисляет программу, действуя прямо по синтаксическим правилам и мелкими шагами *сворачивая* программу, так что на каждом шаге она становится ближе к конечному результату, что бы это ни значило.

Эти мелкие шаги свертки похожи на способ вычисления алгебраических выражений, который мы проходили в школе. Например, чтобы вычислить выражение `(1 × 2) + (3 × 4)`, мы должны:

1. Выполнить умножение в скобках слева (`1 × 2` равно `2`) и свернуть выражение в `2 + (3 × 4)`.
2. Выполнить умножение в скобках справа (`3 × 4` равно `12`) и свернуть выражение в `2 + 12`.
3. Выполнить сложение (`2 + 12` равно `14`), получив окончательный результат – `14`.

Мы можем назвать число 14 результатом, потому что дальше оно не сворачивается, – мы понимаем, что 14 – особый вид алгебраического выражения, *значение*, у которого есть самостоятельная семантика, так что с нашей стороны никакой дополнительной работы не требуется.

Эту неформальную процедуру можно превратить в операционную семантику, выписав формальные правила выполнения каждого шага свертки. Эти правила сами должны быть написаны на некотором языке (*метаязыке*), который обычно представляет собой математическую нотацию.

В этой главе мы исследуем семантику игрушечного языка программирования, назовем его SIMPLE[1].

Математическое описание семантики мелких шагов для языка SIMPLE выглядит следующим образом:

---

[1] Если хотите, можете считать это аббревиатурой «simple imperative language» (простой императивный язык).

$$\frac{\langle e_1, \sigma\rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ + } e_2, \sigma\rangle \rightsquigarrow_e e_1' \texttt{ + } e_2} \qquad \frac{\langle e_2, \sigma\rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ + } e_2, \sigma\rangle \rightsquigarrow_e v_1 \texttt{ + } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ + } n_2, \sigma\rangle \rightsquigarrow_e n} \text{ if } n = n_1 + n_2$$

$$\frac{\langle e_1, \sigma\rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ * } e_2, \sigma\rangle \rightsquigarrow_e e_1' \texttt{ * } e_2} \qquad \frac{\langle e_2, \sigma\rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ * } e_2, \sigma\rangle \rightsquigarrow_e v_1 \texttt{ * } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ * } n_2, \sigma\rangle \rightsquigarrow_e n} \text{ if } n = n_1 \times n_2$$

$$\frac{\langle e_1, \sigma\rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ < } e_2, \sigma\rangle \rightsquigarrow_e e_1' \texttt{ < } e_2} \qquad \frac{\langle e_2, \sigma\rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ < } e_2, \sigma\rangle \rightsquigarrow_e v_1 \texttt{ < } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ < } n_2, \sigma\rangle \rightsquigarrow_e \texttt{true}} \text{ if } n_1 < n_2 \qquad \frac{}{\langle n_1 \texttt{ < } n_2, \sigma\rangle \rightsquigarrow_e \texttt{false}} \text{ if } n_1 \geq n_2$$

$$\frac{}{\langle x, \sigma\rangle \rightsquigarrow_e \sigma(x)} \text{ if } x \in \mathrm{dom}(\sigma)$$

$$\frac{\langle e, \sigma\rangle \rightsquigarrow_e e'}{\langle x \texttt{ = } e, \sigma\rangle \rightsquigarrow_s \langle x \texttt{ = } e', \sigma\rangle} \qquad \frac{}{\langle x \texttt{ = } v, \sigma\rangle \rightsquigarrow_s \langle \texttt{do-nothing}, \sigma[x \mapsto v]\rangle}$$

$$\frac{\langle e, \sigma\rangle \rightsquigarrow_e e'}{\langle \texttt{if (}e\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle \texttt{if (}e'\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle}$$

$$\frac{}{\langle \texttt{if (true) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle s_1, \sigma\rangle} \qquad \frac{}{\langle \texttt{if (false) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle s_2, \sigma\rangle}$$

$$\frac{\langle s_1, \sigma\rangle \rightsquigarrow_s \langle s_1', \sigma'\rangle}{\langle s_1\texttt{; }s_2, \sigma\rangle \rightsquigarrow_s \langle s_1'\texttt{; }s_2, \sigma'\rangle} \qquad \frac{}{\langle \texttt{do-nothing; }s_2, \sigma\rangle \rightsquigarrow_s \langle s_2, \sigma\rangle}$$

$$\frac{}{\langle \texttt{while (}e\texttt{) \{ }s\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle \texttt{if (}e\texttt{) \{ }s\texttt{; while (}e\texttt{) \{ }s\texttt{ \} \} else \{ do-nothing \}}, \sigma\rangle}$$

Математик сказал бы, что это множество *правил вывода*, которое определяет *отношение свертки* на абстрактных синтаксических деревьях SIMPLE. Ну а с точки зрения практика это нагромождение странных символов, которые не говорят ничего внятного о смысле компьютерных программ.

Мы оставим попытки разобраться в этой формальной нотации напрямую, а посмотрим, как записать те же самые правила вывода на Ruby. Ruby в качестве метаязыка понятнее программисту, и к тому же мы получаем дополнительное преимущество – возможность исполнять правила и смотреть, как они работают.

Мы *не* пытаемся описать семантику SIMPLE в виде «спецификации путем реализации». Основная причина, почему мы выбрали для описания семантики Ruby, а не математическую нотацию, – помочь разобраться в ней человеку. А то, что мы попутно получили исполняемую реализацию языка, – просто приятный бонус.

Большой недостаток использования Ruby заключается в том, что мы объясняем простой язык с использованием более сложного, что, вероятно, подрывает философские основы обучения. Следует помнить, что именно математические правила являются авторитетным описанием семантики, а Ruby лишь помогает понять, что эти правила означают.

Выражения

Начнем с семантики выражений SIMPLE. Правила оперируют абстрактными синтаксическими деревьями таких выражений, поэтому мы должны уметь представлять выражения SIMPLE в виде объектов Ruby. Сделать это можно, например, определив классы Ruby для каждого синтаксического элемента SIMPLE – чисел, операций сложения, умножения и т. д., а затем представив выражение в виде дерева, состоящего из экземпляров этих классов.

Вот, например, как выглядят определения классов `Number`, `Add` и `Multiply`:

```
class Number < Struct.new(:value)
end

class Add < Struct.new(:left, :right)
end

class Multiply < Struct.new(:left, :right)
end
```

Мы можем создать экземпляры этих классов и построить из них дерево вручную:

```
irb> Add.new(
       Multiply.new(Number.new(1), Number.new(2)),
       Multiply.new(Number.new(3), Number.new(4))
     )
=> #<struct Add
     left=#<struct Multiply
       left=#<struct Number value=1>,
       right=#<struct Number value=2>
     >,
     right=#<struct Multiply
       left=#<struct Number value=3>,
```

```
        right=#<struct Number value=4>
      >
    >
```

Конечно, мы хотим, чтобы в конечном итоге эти деревья автоматически создавались синтаксическим анализатором. Как это делается, мы увидим в разделе «Реализация синтаксических анализаторов» ниже на стр. 82.

Классы `Number`, `Add` и `Multiply` наследуют обобщенное определение метода `#inspect` от класса `Struct`, поэтому строковые представления их экземпляров в оболочке IRB содержат много ненужных деталей. Чтобы содержимое абстрактного синтаксического дерева выглядело в IRB понятнее, мы переопределим метод `#inspect` в каждом классе[1] и будем возвращать специализированное строковое представление:

```
class Number
    def to_s
        value.to_s
    end

    def inspect
        "«#{self}»"
    end
end

class Add
    def to_s
        "#{left} + #{right}"
    end

    def inspect
        "«#{self}»"
    end
end

class Multiply
    def to_s
        "#{left} + #{right}"
    end

    def inspect
        "«#{self}»"
    end
end
```

[1] Для простоты мы воспротивились искушению вынести общий код в суперкласс или модуль.

Теперь любое абстрактное синтаксическое дерево выглядит в IRB как короткая строка, содержащая исходный код на языке SIMPLE, заключенный в «шевроны», чтобы отличить его от обычного значения в смысле Ruby:

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
   )
=> «1 * 2 + 3 * 4»
>> Number.new(5)
=> «5»
```

В наших примитивных реализациях метода `#to_s` не принимается во внимание приоритет операторов, поэтому иногда результат оказывается неправильным с точки зрения традиционных правил предшествования операций (например, приоритет * выше, чем у +). Возьмем, к примеру, такое абстрактное синтаксическое дерево:

```
>> Multiply.new(
     Number.new(1),
     Multiply.new(
       Add.new(Number.new(2), Number.new(3)),
       Number.new(4)
     )
   )
=> «1 * 2 + 3 * 4»
```

Это дерево соответствует выражению «1 * (2 + 3) * 4», отличающемуся от «1 * 2 + 3 * 4», но из строкового представления это не видно.

Проблема серьезная, но имеющая лишь косвенное отношение к теме семантики. Чтобы не усложнять изложение, мы временно проигнорируем ее и просто не будем создавать выражения, для которых получается неправильно строковое представление. А корректное решение – уже для другого языка – будет приведено в разделе «Синтаксис» на стр. 109.

Теперь можно приступить к реализации операционной семантики мелких шагов, для чего нужно будет определить методы, выполняющие свертку абстрактных синтаксических деревьев. Каждый такой метод будет принимать абстрактное синтаксическое дерево в качестве параметра, сворачивать его тем или иным способом и возвращать получившееся в результате дерево.

Но прежде чем реализовывать саму свертку, мы должны научиться отличать выражения, допускающие свертку, от не допускающих. Выражения `Add` и `Multiply` всегда можно свернуть – оба они представляют операции и могут быть преобразованы в результат этой операции путем соответствующего вычисления. Но выражение `Number` представляет значение, которое ни во что свернуть нельзя.

В принципе, различить эти два вида выражений можно было бы с помощью предиката `#reducible?`, который возвращает `true` или `false` в зависимости от класса своего аргумента:

```
def reducible?(expression)
  case expression
  when Number
    false
  when Add, Multiply
    true
  end
end
```

В Ruby при выполнении предложения `case` управляющее выражение сопоставляется с ветвями путем вызова метода `#===` значения в каждой ветви, которому в качестве аргумента передается значение управляющего выражения. Реализация метода `#===` для классовых объектов проверяет, является ли аргумент экземпляром этого класса или какого-либо его подкласса, поэтому мы можем воспользоваться синтаксической конструкцией `case object when classname` для сопоставления объекта с классом.

Однако в общем случае такой код в объектно-ориентированном языке считается дурным тоном[1]; если поведение некоторой операции зависит от класса ее аргумента, то стандартный подход состоит в том, чтобы реализовать это поведение в методе экземпляра этого класса и дать языку возможность самостоятельно решить, какой метод вызывать, а не использовать для этой цели явное предложение `case`.

Поэтому давайте напишем методы `#reducible?` в каждом из классов `Number`, `Add` и `Multiply`:

```
class Number
  def reducible?
    false
  end
```

[1] Хотя именно так мы написали бы метод `#reducible?` в функциональном языке типа Haskell или ML.

```
end

class Add
  def reducible?
    true
  end
end

class Multiply
  def reducible?
    true
  end
end
```

Это дает нужное нам поведение:

```
irb> Number.new(1).reducible?
=> false
irb> Add.new(Number.new(1), Number.new(2)).reducible?
=> true
```

Теперь мы можем реализовать свертку выражений; для этого точно так же определим метод `#reduce` в классах `Add` и `Multiply`. Определять метод `Number#reduce` необязательно, потому что числа не сворачиваются, так что мы должны лишь следить за тем, чтобы не вызывать `#reduce` для выражений, не допускающих свертку.

Итак, по каким правилам сворачивается выражение сложения? Если левый и правый аргумент – числа, то достаточно просто сложить их, но что, если один или оба аргумента сами нуждаются в сворачивании? Поскольку мы говорим о мелких шагах, то должны решить, какой аргумент сворачивать первым, если свертку допускают оба[1]. Обычная стратегия – сворачивать аргументы слева направо, то есть правила формулируются так.

- Если левый аргумент операции сложения допускает свертку, свернуть левый аргумент.
- Если левый аргумент операции сложения не допускает свертку, а правый допускает, свернуть правый аргумент.
- Если ни один аргумент не допускает свертку, то оба должны быть числами, поэтому сложить их.

Структура этих правил характерна для операционной семантики мелких шагов. В каждом правиле постулируется общий вид выражения, к которому оно применяется – сложение со сворачиваемым

[1] В данный момент не важно, какой именно порядок мы выберем, но вовсе избежать этого решения не удастся.

левым аргументом, со сворачиваемым правым аргументом и с двумя несворачиваемыми аргументами соответственно – и описывается, как в этом случае построить новое свернутое выражение. Выбрав данные конкретные правила, мы специфицировали, что в языке SIMPLE при вычислении выражения сложения аргументы сворачиваются слева направо, а, кроме того, определили, как объединить аргументы после выполнения свертки.

Эти правила можно непосредственно транслировать в реализацию метода `Add#reduce`, и почти так же будет выглядеть код метода `Multiply#reduce` (с тем отличием, что аргументы нужно перемножать, а не складывать).

```
class Add
  def reduce
    if left.reducible?
      Add.new(left.reduce, right)
    elsif right.reducible?
      Add.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end

class Multiply
  def reduce
    if left.reducible?
      Multiply.new(left.reduce, right)
    elsif right.reducible?
      Multiply.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end
```

Метод `#reduce` всегда строит новое выражение, а не модифицирует существующее.

Реализовав метод `#reduce` для этих видов выражений, мы можем, многократно применяя его, вычислить полное выражение путем последовательного выполнения мелких шагов.

```
>> expression =
       Add.new(
         Multiply.new(Number.new(1), Number.new(2)),
         Multiply.new(Number.new(3), Number.new(4))
       )
```

```
=> «1 * 2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 12»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «14»
>> expression.reducible?
=> false
```

Отметим, что метод #reduce всегда преобразует одно выражение в другое в полном соответствии с принципом работы правил операционной семантики мелких шагов. В частности, `Add.new(Number.new(2), Number.new(12)).reduce` возвращает `Number.new(14)`, то есть представление выражения языка SIMPLE, а не 14 – число в смысле Ruby.

Такое разделение между языком SIMPLE, семантику которого мы *специфицируем*, и метаязыком Ruby, на котором *записывается спецификация*, проще поддерживать, когда различия между обоими языками абсолютно очевидны – как в случае, когда метаязыком является математический формализм, а не язык программирования. Нам же приходится быть более внимательными, потому что языки выглядят очень похоже.

Запоминая текущее выражение в качестве состояния программы и в цикле вызывая для него методы `#reducible?` и `#reduce`, пока не получится значение, мы моделируем работу абстрактной машины для вычисления выражений. Чтобы избавить себя от лишнего труда и сделать идею абстрактной машины более конкретной, мы легко можем написать на Ruby код, который делает за нас всю работу. Обернем код и состояние в класс, который назовем *виртуальной машиной*:

```
class Machine < Struct.new(:expression)
  def step
    self.expression = expression.reduce
  end

  def run
    while expression.reducible?
      puts expression
```

```
      step
    end
    puts expression
  end
end
```

Теперь мы можем создать экземпляр виртуальной машины, передав ему выражение, вызвать его метод `#run` и наблюдать, как происходит сворачивание:

```
>> Machine.new(
       Add.new(
         Multiply.new(Number.new(1), Number.new(2)),
         Multiply.new(Number.new(3), Number.new(4))
       )
     ).run
1 * 2 + 3 * 4
2 + 3 * 4
2 + 12
14
=> nil
```

Нетрудно обобщить эту реализацию на другие простые значения и операции: вычитание и деление, булевы значения `true` и `false`, логические операции `and`, `or` и `not`, операторы сравнения чисел, возвращающие булевы значения, и т. д. Вот, например, как выглядят реализации булевых значений и оператора «меньше»:

```
class Boolean < Struct.new(:value)
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    false
  end
end

class LessThan < Struct.new(:left, :right)
  def to_s
    "#{left} < #{right}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
```

```
    true
  end

  def reduce
    if left.reducible?
      LessThan.new(left.reduce, right)
    elsif right.reducible?
      LessThan.new(left, right.reduce)
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

Как и раньше, это позволяет свернуть булево выражение мелкими шагами:

```
>> Machine.new(
     LessThan.new(Number.new(5), Add.new(Number.new(2), Number.new(2)))
   ).run
5 < 2 + 2
5 < 4
false
=> nil
```

До сих пор все было просто: мы начали специфицировать операционную семантику языка, реализовав виртуальную машину, которая может вычислять выражения этого языка. В настоящий момент состояние виртуальной машины включает всего лишь текущее выражение, а поведение описывается набором правил, которые определяют, как изменяется состояние в ходе работы машины. Мы реализовали машину в виде программы, которая запоминает текущее выражение и продолжает сворачивать его, изменяя на каждом шаге, пока еще есть возможность произвести свертку.

Но язык простых алгебраических выражений не очень интересен и не обладает многими чертами, которые мы ожидаем даже от простейшего языка программирования. Поэтому давайте сделаем на его основе что-нибудь более совершенное, больше напоминающее язык, на котором можно было бы писать полезные программы.

Первое, что бросается в глаза, – отсутствие в языке SIMPLE переменных. От любого сколько-нибудь полезного языка мы ожидаем возможности наделять значения осмысленными именами, а не работать только с литералами. Имена вводят уровень косвенности, который позволяет использовать один и тот же код для обработки различных значений, в том числе поступающих из внешнего мира и, следовательно, неизвестных во время написания программы.

Введем новый класс выражений, `Variable`, для представления переменных в SIMPLE:

```
class Variable < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end
end
```

Чтобы свернуть переменную, абстрактная машина должна хранить соответствие между именами и значениями переменных – *окружение* – в дополнение к текущему выражению. В Ruby для хранения такого соответствия можно использовать хеш, в котором ключами являются символы, а значениями – объекты выражений. Например, хеш `{ x: Number.new(2), y: Boolean.new(false) }` представляет окружение, в котором переменные `x` и `y` ассоциированы соответственно с числом и булевым значением SIMPLE.

В данном языке мы хотим, чтобы окружение позволяло сопоставлять имена переменных только с несвертываемыми значениями типа `Number.new(2)`, а не с произвольными допускающими свертку выражениями, например `Add.new(Number.new(1), Number.new(2))`. Мы обеспечим это ограничение позже, когда будем писать правила, изменяющие содержимое окружения.

Имея окружение, мы легко можем реализовать метод `Variable #reduce`: достаточно найти имя переменной в окружении и вернуть соответствующее значение.

```
class Variable
  def reduce(environment)
    environment[name]
  end
end
```

Отметим, что теперь мы передаем методу `#reduce` аргумент `environment`, поэтому должны изменить реализации `#reduce` в других классах выражений, так чтобы этот аргумент правильно принимался и передавался.

```
class Add
  def reduce(environment)
    if left.reducible?
      Add.new(left.reduce(environment), right)
    elsif right.reducible?
      Add.new(left, right.reduce(environment))
    else
      Number.new(left.value + right.value)
    end
  end
end

class Multiply
  def reduce(environment)
    if left.reducible?
      Multiply.new(left.reduce(environment), right)
    elsif right.reducible?
      Multiply.new(left, right.reduce(environment))
    else
      Number.new(left.value * right.value)
    end
  end
end

class LessThan
  def reduce(environment)
    if left.reducible?
      LessThan.new(left.reduce(environment), right)
    elsif right.reducible?
      LessThan.new(left, right.reduce(environment))
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

Поддержав работу с окружением во всех реализациях `#reduce`, мы должны изменить еще и саму виртуальную машину, так чтобы она запоминала окружение и передавала его методу `#reduce`:

```
Object.send(:remove_const, :Machine) # забыть старый класс Machine

class Machine < Struct.new(:expression, :environment)
  def step
    self.expression = expression.reduce(environment)
  end

  def run
    while expression.reducible?
      puts expression
      step
    end
    puts expression
  end
end
```

Метод `#run` остался прежним, но у машины появился новый атрибут `environment`, который используется в новой реализации метода `#step`.

Теперь мы можем применять свертку к выражениям, содержащим переменные, при условии, что передается окружение, в котором хранятся значения переменных:

```
>> Machine.new(
       Add.new(Variable.new(:x), Variable.new(:y)),
       { x: Number.new(3), y: Number.new(4) }
     ).run
x + y
3 + y
3 + 4
7
=> nil
```

После добавления окружения нашу операционную семантику выражений можно считать законченной. Мы спроектировали абстрактную машину, которая начинает работу с начального выражения и окружения, а затем использует текущее выражение и окружение для порождения нового выражения на каждом мелком шаге сворачивания; окружение при этом остается неизменным.

Предложения

Теперь можно рассмотреть реализацию другой программной конструкции: *предложения*. Смысл выражения в том, чтобы в результате вычисления породить другое выражение; результатом же вычисления предложения является изменение состояния абстрактной машины. Единственным состоянием нашей машины (если не считать текущего выражения) является окружение, поэтому мы разрешим предложениям языка SIMPLE порождать новое окружение, заменяющее текущее.

Простейшее из всех возможных предложений не делает вообще ничего: его нельзя свернуть, поэтому оно не может хоть как-то повлиять на окружение. Реализация тривиальна:

```
class DoNothing    ❶
  def to_s
    'do-nothing'
  end

  def inspect
    "«#{self}»"
  end
```

```
  def ==(other_statement) ❷
    other_statement.instance_of?(DoNothing)
  end

  def reducible?
    false
  end
end
```

❶ До сих пор все наши синтаксические классы наследовали классу `Struct`, однако `DoNothing` не наследует ничему. Дело в том, что у `DoNothing` нет атрибутов, а метод `Struct.new`, к сожалению, не позволяет передать пустой список имен атрибутов.

❷ Мы хотим иметь возможность сравнивать предложения на равенство. Другие синтаксические классы наследуют реализацию `#==` от `Struct`, но в `DoNothing` мы вынуждены определить ее самостоятельно.

Предложение, которое ничего не делает, на первый взгляд кажется бессмысленным, однако очень удобно иметь специальное предложение, которое представляет программу, исполнение которой успешно завершилось. Мы сделаем так, что все остальные предложения будут сворачиваться в `«do-nothing»`, завершив свою работу.

Простейший пример предложения, которое делает что-то полезное, — *присваивание* вида `«x = x + 1»`, но прежде чем его реализовывать, необходимо решить, как для него должны выглядеть правила свертки.

Предложение присваивания состоит из имени переменной (`x`), знака равенства и выражения (`«x + 1»`). Если выражение допускает свертку, то его можно свернуть по общим правилам сворачивания выражений и породить новое предложение присваивания, содержащее свернутое выражение. Например, свертка `«x = x + 1»` в окружении, где переменная `x` имеет значение `«2»`, должно дать предложение `«x = 2 + 1»`, свертка которого дает предложение `«x = 3»`.

Но что потом? Если выражением уже является значение, например `«3»`, то мы должны просто выполнить присваивание, то есть изменить окружение, связав это значение с соответствующей переменной. Поэтому свертка предложения порождает не только новое предложение, но и новое окружение, которое иногда будет отличаться от окружения, в котором производилась свертка.

В нашей реализации для изменения окружения мы используем метод `Hash#merge`, который создает новый хеш, не изменяя старый:

```
>> old_environment = { y: Number.new(5) }
=> {:y=>«5»}
>> new_environment = old_environment.merge({ x: Number.new(3) })
=> {:y=>«5», :x=>«3»}
>> old_environment
=> {:y=>«5»}
```

Мы могли бы модифицировать текущее окружение, а не создавать новое, однако, избегая деструктивных обновлений, мы заставляем себя строить программу так, чтобы извещения о последствиях работы `#reduce` были явными и недвусмысленными. Если `#reduce` хочет изменить текущее окружение, он должен сообщить об этом, вернув измененное окружение вызывающей программе; напротив, если он не возвращает окружение, то есть уверенность, что он не произвел никаких изменений.

Это ограничение помогает подчеркнуть различие между выражениями и предложениями. В случае выражений мы передаем окружение в `#reduce` и получаем назад свернутое выражение; новое окружение не возвращается, откуда ясно, что свертка выражения не изменяет окружения. В случае предложений мы передаем `#reduce` текущее окружение и получаем назад новое окружение, а это означает, что свертка предложения может оказывать влияние на окружение. (Иными словами, структура семантики мелких шагов для языка SIMPLE показывает, что выражения в нем *чистые*, а предложения – *нечистые*.)

Итак, сворачивание «`x = 3`» в пустом окружении должно породить новое окружение `{ x: Number.new(3) }`, но мы также ожидаем, что и само предложение будет как-то свернуто, иначе наша абстрактная машина будет бесконечно присваивать переменной `x` значение `3`. Вот для этого и нужно предложение «`do-nothing`»: завершившееся присваивание сворачивается в «`do-nothing`», показывая, что свертка закончилась и получившееся новое окружение можно считать результатом присваивания.

Подведем итог, выписав правила свертки для присваивания:

- Если выражение в предложении присваивания допускает свертку, то свернуть его и получить в результате свернутое предложение присваивания и неизменившееся окружение.
- Если выражение в предложении присваивания не допускает свертку, то изменить окружение, связав это выражение с переменной в левой части присваивания, и вернуть предложение «`do-nothing`» и новое окружение.

Этой информации достаточно для реализации класса `Assign`. Единственная трудность состоит в том, что метод `Assign#reduce` должен возвращать как предложение, так и окружение, а методы в Ruby могут возвращать только один объект. Но мы можем создать иллюзию возврата двух объектов, поместив их в массив из двух элементов и вернув этот массив.

```
class Assign < Struct.new(:name, :expression)
  def to_s
    "#{name} = #{expression}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if expression.reducible?
      [Assign.new(name, expression.reduce(environment)), environment]
    else
      [DoNothing.new, environment.merge({ name => expression })]
    end
  end
end
```

Как мы и обещали, правила свертки для класса `Assign` гарантируют, что в окружение добавляются только выражения, не допускающие свертку (то есть значения).

Как и в случае выражений, мы можем вручную вычислить предложение присваивания, сворачивая его до тех пор, пока это возможно:

```
>> statement = Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
=> «x = x + 1»
>> environment = { x: Number.new(2) }
=> {:x=>«2»}
>> statement.reducible?
=> true
>> statement, environment = statement.reduce(environment)
=> [«x = 2 + 1», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«x = 3», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«do-nothing», {:x=>«3»}]
>> statement.reducible?
=> false
```

Это еще более трудоемкий процесс, чем ручная свертка выражений, поэтому изменим реализацию нашей виртуальной машины, научив ее обрабатывать предложения, и на каждом шаге свертки будем показывать текущее предложение и окружение:

```
Object.send(:remove_const, :Machine)

class Machine < Struct.new(:statement, :environment)
  def step
    self.statement, self.environment = statement.reduce(environment)
  end

  def run
    while statement.reducible?
      puts "#{statement}, #{environment}"
      step
    end
    puts "#{statement}, #{environment}"
  end
end
```

Теперь всю работу может сделать машина:

```
>> Machine.new(
       Assign.new(:x, Add.new(Variable.new(:x), Number.new(1))),
         { x: Number.new(2) }
       ).run
x = x + 1, {:x=>«2»}
x = 2 + 1, {:x=>«2»}
x = 3, {:x=>«2»}
do-nothing, {:x=>«3»}
=> nil
```

Как видим, машина по-прежнему выполняет шаги свертки выражений (`«x + 1»` в `«2 + 1»` и затем в `«3»`), но теперь они производятся внутри предложения, а не на верхнем уровне синтаксического дерева.

Зная, как работает свертка предложения, мы можем обобщить этот механизм на другие виды предложений. Начнем с условного предложения вида `«if (x) { y = 1 } else { y = 2 }»`, которое содержит выражение, называемое *условием* (`«x»`), и два предложения, которые мы будем называть *следствием* (`«y = 1»`) и *альтернативой* (`«y = 2»`)[1]. Правила свертки условных предложений очевидны:

[1] Это условное предложение отличается от конструкции `if` в Ruby. В Ruby `if` – выражение, которое возвращает значение, а в SIMPLE – предложение, которое выбирает, какое из двух предложений вычислять, и его единственный результат – это воздействие на текущее окружение.

- Если условие допускает свертку, то свернуть его и вернуть свернутое условное предложение и неизменившееся окружение.
- Если условием является выражение «`true`», то результатом свертки является предложение-следствие и неизменившееся окружение.
- Если условием является выражение «`false`», то результатом свертки является предложение-альтернатива и неизменившееся окружение.

В этом случае ни одно правило не изменяет окружение – свертка выражения условия в первом правило порождает лишь новое выражение, но не новое окружение.

Ниже показано, как эти правила транслируются в класс `If`:

```
class If < Struct.new(:condition, :consequence, :alternative)
  def to_s
    "if (#{condition}) { #{consequence} } else { #{alternative} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if condition.reducible?
      [If.new(condition.reduce(environment), consequence, alternative),
                environment]
    else
      case condition
      when Boolean.new(true)
        [consequence, environment]
      when Boolean.new(false)
        [alternative, environment]
      end
    end
  end
end
```

А вот как выглядят шаги свертки:

```
>> Machine.new(
       If.new(
         Variable.new(:x),
         Assign.new(:y, Number.new(1)),
         Assign.new(:y, Number.new(2))
       ),
       { x: Boolean.new(true) }
```

```
    ).run
if (x) { y = 1 } else { y = 2 }, {:x=>«true»}
if (true) { y = 1 } else { y = 2 }, {:x=>«true»}
y = 1, {:x=>«true»}
do-nothing, {:x=>«true», :y=>«1»}
=> nil
```

Все работает, как и положено, но хорошо было бы поддержать еще условные предложения без ветви «`else`», например «`if (x) { y = 1 }`». К счастью, мы уже можем это сделать, написав предложение вида «`if (x) { y = 1 } else { do-nothing }`», которое ведет себя, как если бы ветви «`else`» не было:

```
>> Machine.new(
       If.new(Variable.new(:x), Assign.new(:y, Number.new(1)), DoNothing.new),
       { x: Boolean.new(false) }
    ).run
if (x) { y = 1 } else { do-nothing }, {:x=>«false»}
if (false) { y = 1 } else { do-nothing }, {:x=>«false»}
do-nothing, {:x=>«false»}
=> nil
```

Итак, мы реализовали предложения присваивания, условные предложения и выражения и теперь имеем строительные блоки, из которых можно создавать программы, способные производить реальные вычисления и принимать решения. Вот только пока мы еще не умеем соединять эти блоки вместе: не существует способа присвоить значение более чем одной переменной или выполнить более одного условного предложения. И это резко ограничивает полезность нашего языка.

Ситуацию можно исправить, определив еще один вид предложения, *последовательность*, которое соединит два предложения, например «`x = 1 + 1`» и «`y = x + 3`» в одно более крупное: «`x = 1 + 1; y = x + 3`». Имея предложения последовательности, мы можем составлять из них сколь угодно длинные предложения; например, последовательность «`x = 1 + 1; y = x + 3`» и присваивание «`z = y + 5`» можно объединить, получив новую последовательность «`x = 1 + 1; y = x + 3; z = y + 5`»[1].

Шаги свертки для последовательности определяются довольно хитро.

---

[1] Для наших целей безразлично, было построено это предложение как «`(x = 1 + 1; y = x + 3); z = y + 5`» или как «`x = 1 + 1; (y = x + 3; z = y + 5)`». Выбор повлиял бы на порядок шагов свертки при запуске, но конечный результат был бы одинаков.

- Если первое предложение – «do-nothing», результатом свертки является второе предложение и исходное окружение.
- Если первое предложение – не «do-nothing», свернуть его, получив в результате новую последовательность (свернутое первое предложение, за которым следует второе) и результат свертки окружения.

Код поможет прояснить эти правила:

```
class Sequence < Struct.new(:first, :second)
  def to_s
    "#{first}; #{second}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    case first
    when DoNothing.new
      [second, environment]
    else
      reduced_first, reduced_environment = first.reduce(environment)
      [Sequence.new(reduced_first, second), reduced_environment]
    end
  end
end
```

Смысл этих правил в том, что при повторном сворачивании последовательности мы всякий раз сворачиваем первое из входящих в нее предложений, пока не получим «do-nothing», а затем возвращаем в качестве результата свертки второе предложение. Понаблюдать за тем, как это происходит можно, подав последовательность на вход виртуальной машины:

```
>> Machine.new(
       Sequence.new(
         Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
         Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
       ),
       {}
     ).run
x = 1 + 1; y = x + 3, {}
x = 2; y = x + 3, {}
do-nothing; y = x + 3, {:x=>«2»}
y = x + 3, {:x=>«2»}
```

```
y = 2 + 3, {:x=>«2»}
y = 5, {:x=>«2»}
do-nothing, {:x=>«2», :y=>«5»}
=> nil
```

Единственно, чего по-настоящему не хватает в SIMPLE, так это какой-нибудь циклической конструкции, поэтому завершим эту тему добавлением предложения «while», чтобы программа могла повторять вычисление произвольное число раз[1]. Предложение вида «while (x < 5) { x = x * 3 }» содержит выражение, которое называется *условием* («x < 5»), и предложение, называемое *телом* («x = x * 3»).

Корректно написать правила свертки для предложения «while» не вполне тривиально. Можно было бы подойти к этому, как в предложении «if»: свернуть условие, если возможно, а если нет, вернуть в качестве результата свертки либо тело, либо «do-nothing» – в зависимости от того, чему равно условие: «true» или «false». Но что делать после того, как абстрактная машина завершит сворачивание тела? Условие уже свернуто в значение и отброшено, а тело свернуто в «do-nothing»; и как нам теперь выполнить еще одну итерацию цикла? Каждый шаг свертки может обмениваться информацией с последующими шагами, только порождая новое предложение и окружение, при этом просто негде «запомнить» исходное условие и тело для использования на следующей итерации.

Решение на основе семантики мелких шагов[2] состоит в том, чтобы, воспользовавшись предложением последовательности, *раскрутить* один уровень цикла «while», свернув его в предложение «if», которое выполняет одну итерацию цикла, а затем повторяет исходный цикл «while». При таком подходе нам нужно только одно правило свертки.

- Свернуть «while (condition) { body }» в «if (condition) { body; while (condition) { body } } else { do-nothing }», оставив окружение без изменения.

---

[1] У нас уже есть средства зашить в код фиксированное количество повторений, воспользовавшись последовательностями, но это не позволяет управлять числом повторений во время выполнения.

[2] Возникает искушение встроить итеративное поведение «while» непосредственно в его правило свертки, а не ломать голову над тем, как это сделать с помощью абстрактной машины, но это противоречит принципу семантики мелких шагов. В разделе «Семантика крупных шагов» на стр. 62 описана семантика, которая позволяет выполнять какую-то работу внутри правил.

Это правило легко реализовать на Ruby:

```
class While < Struct.new(:condition, :body)
  def to_s
    "while (#{condition}) { #{body} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    [If.new(condition, Sequence.new(body, self), DoNothing.new), environment]
  end
end
```

Это дает виртуальной машине возможность вычислять условие и тело столько раз, сколько нужно:

```
>> Machine.new(
       While.new(
         LessThan.new(Variable.new(:x), Number.new(5)),
         Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
       ),
       { x: Number.new(1) }
     ).run
while (x < 5) { x = x * 3 }, {:x=>«1»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
if (1 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 1 * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«3»}
while (x < 5) { x = x * 3 }, {:x=>«3»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
if (3 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 3 * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 9; while (x < 5) { x = x * 3 }, {:x=>«3»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«9»}
while (x < 5) { x = x * 3 }, {:x=>«9»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«9»}
if (9 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«9»}
if (false) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«9»}
do-nothing, {:x=>«9»}
=> nil
```

Быть может, вам это правило напоминает своего рода увиливание – мы вроде бы постоянно откладываем сворачивание «while»

на потом, но это «потом» так никогда и не наступает. Но, с другой стороны, такой способ прекрасно объясняет, что именно означает предложение «while»: проверить условие, вычислить тело и начать заново. Любопытно, что сворачивание «while» порождает синтаксически более длинную программу, включающую предложение последовательности и условное предложение, а не просто сводится к сворачиванию условия или тела; одна из причин, почему полезно иметь техническую инфраструктуру для спецификации формальной семантики языка, заключается в том, чтобы иметь возможность увидеть, как различные части языка взаимосвязаны, и это как раз пример такого рода.

Корректность

Мы полностью проигнорировали вопрос о том, что случится, если попытаться выполнить синтаксически правильную, но в остальном некорректную программу в соответствии с описанной выше семантикой. Предложение «x = true; x = x + 1» – синтаксически правильная конструкция языка SIMPLE – нет сомнения, что мы сможем построить представляющее ее абстрактное синтаксическое дерево; но при попытке свернуть его произойдет ошибка, потому что абстрактная машина на каком-то этапе попробует прибавить «1» к «true»:

```
>> Machine.new(
       Sequence.new(
         Assign.new(:x, Boolean.new(true)),
         Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
       ),
       {}
     ).run
x = true; x = x + 1, {}
do-nothing; x = x + 1, {:x=>«true»}
x = x + 1, {:x=>«true»}
x = true + 1, {:x=>«true»}
NoMethodError: undefined method `+' for true:TrueClass
```

Один из способов решить эту проблему – ввести ограничения на условия, при которых возможна свертка выражений. Мы допускаем, что вычисление может *застрять*, не пытаясь во что бы то ни стало дойти до значения (и, возможно, по ходу дела «сломаться»). Можно было бы реализовать метод Add#reducible?, так что значение true возвращается лишь в случае, когда оба аргумента «+» либо допускают свертку, либо являются экземплярами класса Number; тогда

вычисление выражения «`true + 1`» застрянет и никогда не даст значения.

В конечном итоге нам понадобится более мощный инструмент, чем синтаксис, нечто такое, что может «видеть будущее» и предотвращать попытки выполнить программу, которая могла бы завершиться фатальной ошибкой или застрять. Настоящая глава посвящена *динамической семантике* – что программа делает при выполнении – однако это не единственный вид семантики, которой может обладать программа. В главе 9 при изучении *статической семантики* мы увидим, как можно решить, имеет ли синтаксически правильная программа какой-то полезный смысл в соответствии с динамической семантикой языка.

Приложения

Специфицированный нами язык программирования очень прост, но тем не менее при формулировании правил свертки мы должны были принять и недвусмысленно выразить ряд проектных решений. Например, в отличие от Ruby, в языке SIMPLE имеется разница между выражением, которое возвращает значение, и предложением, которое не возвращает значение. В SIMPLE, как и в Ruby, выражения вычисляются слева направо, а окружение позволяет ассоциировать переменные только с полностью свернутыми значениями, а не с выражениями общего вида, вычисление которых еще не завершено[1]. Мы могли бы изменить любое из этих решений, предложив другую семантику мелких шагов, определяющую новый язык с таким же синтаксисом, но иным поведением во время выполнения. Если бы мы захотели добавить в язык более развитые возможности – структуры данных, вызовы процедур, исключения и систему объектов, – то пришлось бы принять дополнительные проектные решения и однозначно выразить их в виде определения семантики.

Подробный, ориентированный на выполнение стиль семантики мелких шагов хорошо приспособлен к задаче однозначного специфицирования реальных языков программирования. Например, в последнем стандарте R6RS (*http://www.r6rs.org/final/html/r6rs/*

---

[1] Процедуры в Ruby позволяют в некотором смысле присваивать переменным сложные выражения, но процедура все равно является значением: она не может выполнять вычисления сама по себе, а должна быть свернута как часть объемлющего выражения, которое содержит и другие значения.

r6rs-Z-H-15.html) языка Scheme семантика мелких шагов применяется для описания его выполнения; там же имеется эталонная реализация этой семантики (*http://www.r6rs.org/refimpl/*), написанная на языке PLT Redex (*http://redex.racket-lang.org/*), «предметно-ориентированном языке, предназначенном для специфицирования и отладки операционной семантики». Для языка OCaml, который построен в виде последовательности слоев поверх более простого языка Core ML, также имеется определение семантики мелких шагов (*http://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#htoc5*), описывающее поведение базового языка во время выполнения.

В разделе «Семантика» на стр. 247 приведен еще один пример использования операционной семантики мелких шагов для специфицирования смысла выражений в еще более простом языке, который называется лямбда-исчислением.

Семантика крупных шагов

Мы только что видели, как устроена операционная семантика мелких шагов: мы проектируем абстрактную машину, которая хранит некоторое состояние выполнения, а затем определяем правила свертки, которые говорят, каким образом каждая программная конструкция может постепенно продвигаться к результату вычисления. Отметим, в частности, что характер семантики мелких шагов по преимуществу *итеративный*, то есть абстрактная машина должна раз за разом повторять шаги свертки (в программе на Ruby это делается в цикле `while` в методе `Machine#run`), которые построены так, что на выходе порождают информацию такого же вида, какую требуют на входе, – это и позволяет конструировать из них итеративное приложение[1].

У подхода на основе мелких шагов есть важное преимущество: сложная задача выполнения всей программы разбивается на более мелкие части, которые проще объяснить и проанализировать. Но при этом создается ощущение захода издалека: вместо того чтобы прямо объяснить, как работает программная конструкция, мы лишь показываем, как ее можно немножко свернуть. Почему бы не объяснить предложение более непосредственно, сразу описав, что про-

---

[1] Сворачивание выражения и окружения даст новое выражение, и в следующий раз мы можем повторно использовать старое окружение; сворачивание предложения и окружения даст новое выражение и новое окружение.

исходит при его выполнении? Это возможно и является основой *семантики крупных шагов*.

Идея семантики крупных шагов состоит в том, чтобы специфицировать, как перейти от выражения или предложения сразу к его результату. Это заставляет рассуждать о выполнении программы как о *рекурсивном*, а не итеративном процессе: семантика крупных шагов говорит, что для вычисления большого выражения мы должны вычислить все его подвыражения, а затем объединить частичные результаты в окончательный ответ.

Во многих отношениях это выглядит более естественно, чем подход на основе мелких шагов, хотя здесь и недостает характерного для последнего внимания к деталям. Например, в семантике мелких шагов явно указывается порядок выполнения операций, потому что в любой точке определено, каким должен быть следующий шаг свертки. Семантика же крупных шагов записывается в более свободном стиле и определяет лишь, какие подвычисления необходимо выполнить, не обязательно указывая их порядок[1]. Кроме того, семантика мелких шагов дает простой способ наблюдать за промежуточными стадиями вычисления, тогда как семантика крупных шагов лишь возвращает результат, ничего не говоря о том, как он был получен.

Чтобы лучше разобраться в природе этого компромисса, заново реализуем на Ruby некоторые языковые конструкции, воспользовавшись семантикой крупных шагов. Для семантики мелких шагов нам был нужен класс `Machine`, в котором запоминалось состояние и выполнялись повторные операции свертки. Сейчас он нам не понадобится; правила семантики крупных шагов описывают, как вычислить результат всей программы, обойдя его абстрактное синтаксическое дерево в один заход, поэтому ни состояния, ни повторения не будет. Мы просто определим метод `#evaluate` во всех классах выражений и предложений и будем вызывать его напрямую.

Выражения

В случае семантики мелких шагов мы должны были различать выражения, допускающие свертку (например, «`1 + 2`») и не

---

[1] Наша реализация семантики крупных шагов на Ruby не страдает такой неоднозначностью, потому что Ruby сам принимает решения о порядке вычислений, но когда семантика крупных шагов специфицируется математически, точное задание стратегии вычислений можно опускать.

допускающие (например, «3»), чтобы правила свертки знали, когда подвыражение может использоваться как часть объемлющего вычисления. В случае же семантики крупных шагов вычислить можно любое выражение. Единственное различие (если так уж необходимо обозначить его) заключается в том, что некоторые выражения вычисляются немедленно и в результате дают сами себя, тогда как другие требуют дополнительных действий и дают в результате другое выражение.

Цель семантики крупных шагов – смоделировать точно такое же поведение во время выполнения, как и в случае семантики мелких шагов, то есть мы ожидаем, что правила крупных шагов для любой программной конструкции будут согласованы с тем, что в конечном итоге даст повторное применение правил мелких шагов (это утверждение можно формально доказать, если определить операционную семантику математически). Правила мелких шагов для значений типа `Number` и `Boolean` говорят, что их вообще нельзя свернуть, поэтому соответствующие правила крупных шагов очень просты: результатом вычисления значения является оно само.

```
class Number
  def evaluate(environment)
    self
  end
end

class Boolean
  def evaluate(environment)
    self
  end
end
```

Выражения типа `Variable` уникальны тем, что в соответствии с семантикой мелких шагов их можно свернуть ровно один раз, после чего они становятся значениями, поэтому правила крупных и мелких шагов совпадают: найти имя переменной в окружении и вернуть ее значение.

```
class Variable
  def evaluate(environment)
    environment[name]
  end
end
```

Бинарные выражения `Add`, `Multiply` и `LessThan` несколько интереснее, потому что требуют рекурсивного вычисления левого и

правого подвыражения с последующим объединением результатов с помощью подходящего оператора Ruby:

```
class Add
  def evaluate(environment)
    Number.new(left.evaluate(environment).value + right.evaluate(environment).value)
  end
end

class Multiply
  def evaluate(environment)
    Number.new(left.evaluate(environment).value * right.evaluate(environment).value)
  end
end

class LessThan
  def evaluate(environment)
    Boolean.new(left.evaluate(environment).value < right.evaluate(environment).value)
  end
end
```

Чтобы убедиться в правильности семантики крупных шагов для выражения, покажем, как выглядят результаты в оболочке Ruby:

```
>> Number.new(23).evaluate({})
=> «23»
>> Variable.new(:x).evaluate({ x: Number.new(23) })
=> «23»
>> LessThan.new(
       Add.new(Variable.new(:x), Number.new(2)),
       Variable.new(:y)
     ).evaluate({ x: Number.new(2), y: Number.new(5) })
=> «true»
```

Предложения

Достоинства такого стиля специфицирования семантики особенно ярко проявляются при задании поведения предложений. В случае семантики мелких шагов выражения сворачиваются в другие выражения, тогда как предложения сворачиваются в «do-nothing», оставляя за собой модифицированное окружение. Мы можем представлять себе вычисление предложения в соответствии с семантикой крупных шагов как процесс, который преобразует это предложение и начальное окружение в конечное окружение, – без осложнений, связанных с необходимостью иметь дело еще и с промежуточным предложением, генерируемым методом #reduce. Например, при вычислении предложения присваивания крупными шагами нужно полностью вычислить выражение и вернуть обновленное окружение, содержащее результат этого вычисления:

```
class Assign
  def evaluate(environment)
    environment.merge({ name => expression.evaluate(environment) })
  end
end
```

Аналогично, метод `DoNothing#evaluate`, очевидно, возвращает немодифицированное окружение, а метод `If#evaluate` должен проделать вполне понятную работу: вычислить условие, а затем вернуть окружение, получающееся в результате вычисления следствия или альтернативы.

```
class DoNothing
  def evaluate(environment)
    environment
  end
end

class If
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
      consequence.evaluate(environment)
    when Boolean.new(false)
      alternative.evaluate(environment)
    end
  end
end
```

Интерес представляют предложения последовательности и цикла «`while`». В случае последовательности нужно просто вычислить оба предложения, «протащив» начальное окружение, так чтобы результат вычисления первого предложения стал окружением, в котором вычисляется второе предложение. На Ruby это можно записать, передав результат первого вычисления в качестве аргумента для вычисления второго:

```
class Sequence
  def evaluate(environment)
    second.evaluate(first.evaluate(environment))
  end
end
```

Такое протаскивание окружения важно, когда нужно, чтобы предшествующие предложения готовили переменные для последующих:

```
>> statement =
       Sequence.new(
```

```
        Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
        Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
      )
=> «x = 1 + 1; y = x + 3»
>> statement.evaluate({})
=> {:x=>«2», :y=>«5»}
```

Для предложений «while» мы должны продумать все стадии полного вычисления цикла:

- вычислить условие и получить «true» или «false»;
- если условие равно «true», вычислить тело цикла и получить новое окружение, после чего повторить цикл в новом окружении (то есть вычислить все предложение «while» заново) и вернуть результирующее окружение;
- если условие равно «false», вернуть неизмененное окружение.

Это рекурсивное объяснение того, как должно вести себя предложение «while». Как и в случае последовательностей, важно, чтобы измененное окружение, порожденное телом цикла, использовалось на следующей итерации; в противном случае условие всегда будет равно «true» и цикл не сможет завершиться[1].

Зная, как должна вести себя семантика крупных шагов для предложения «while», мы можем реализовать метод While#evaluate:

```
class While
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
      evaluate(body.evaluate(environment)) ❶
    when Boolean.new(false)
      environment
    end
  end
end
```

❶ Именно здесь реализуется собственно цикл: метод body.evaluate(environment) вычисляет тело цикла для получения нового окружения, а затем передает это окружение *текущему методу*, чтобы начать новую итерацию. Это означает, что в стеке может скопиться много вложенных вызовов While#evaluate, пока наконец условие не станет равно «false» и не будет возвращено конечное окружение.

[1] Разумеется, ничто не мешает программисту на SIMPLE написать предложение «while», в котором условие никогда не обращается в «false», но тут уж «что просил, то и получил».

Как и в случае любого рекурсивного кода, есть опасность, что стек вызовов Ruby переполнится, если количество вложенных вызовов окажется слишком большим. В некоторых реализациях Ruby имеется экспериментальная поддержка *оптимизации хвостовой рекурсии*, техники, которая снижает риск переполнения за счет повторного использования одного и того же кадра стека в тех случаях, когда это возможно. В официальной реализации Ruby (MRI) для включения оптимизации хвостовой рекурсии нужно написать:

```
RubyVM::InstructionSequence.compile_option = {
    tailcall_optimization: true,
    trace_instruction: false
}
```

Чтобы убедиться в правильности работы, попробуем вычислить то же предложение «while», которое использовалось при проверке семантики мелких шагов:

```
>> statement =
       While.new(
         LessThan.new(Variable.new(:x), Number.new(5)),
         Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
       )
=> «while (x < 5) { x = x * 3 }»
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«9»}
```

Результат такой же, как при вычислении мелкими шагами, так что, похоже, метод `While#evaluate` работает правильно.

Приложения

В нашей ранней реализации семантики мелких шагов стек вызовов Ruby использовался довольно скромно: при вызове `#reduce` для большой программы могли иметь место вложенные вызовы `#reduce` на пути сообщения вниз по абстрактному синтаксическому дереву, пока не будет достигнут код, который можно свернуть[1]. Но

[1] Существует другой стиль операционной семантики, называемый редукционной семантикой, в которой фазы «что сворачивать дальше?» и «как это сворачивать?» явным образом разделены за счет введения так называемых редукционных контекстов. Контекст представляет собой просто образец, который лаконично описывает те места в программе, в которых может происходить свертка, а это значит, что нам остается только выписать правила свертки, которые выполняют реальные вычисления, исключив тем самым некоторую стереотипную часть из семантических определений больших языков.

общий ход вычисления отслеживает виртуальная машина, которая запоминает текущую программу и окружение по мере выполнения мелких сверток; таким образом, глубина стека ограничена глубиной синтаксического дерева программы, так как вложенные вызовы используются лишь для обхода дерева в поисках того, что свернуть дальше, а не для выполнения самой свертки.

Напротив, реализация с крупными шагами пользуется стеком куда активнее, полагаясь исключительно на него для запоминания текущего места вычисления, для выполнения мелких вычислений в составе более крупного и для слежения за тем, какая часть вычисления еще не выполнена. То, что на поверхности выглядит, как одно обращение к `#evaluate`, на деле оказывается последовательностью рекурсивных вызовов, каждый из которых вычисляет часть программы, расположенную глубже в синтаксическом дереве.

Это различие проясняет назначение обоих подходов. Семантика мелких шагов ориентирована на простую абстрактную машину, способную выполнять небольшие операции, и потому явно и детально описывает, как порождать полезные промежуточные результаты. Семантика крупных шагов возлагает груз сборки всего вычисления на исполняющую его машину или человека, заставляя отслеживать многочисленные промежуточные подцели по мере преобразования программы в конечный результат за одну операцию. В зависимости от того, чего мы хотим от операционной семантики языка, – построения эффективной реализации, доказательства каких-то свойств программы или выполнения оптимизирующих преобразований – может оказаться предпочтительным тот или другой подход.

Наиболее известное применение семантики крупных шагов для специфицирования реального языка программирования – глава 6 оригинального определения языка Standard ML (*http://www.lfcs.inf.ed.ac.uk/reports/87/ECS-LFCS-87-36/*), в которой все поведение ML во время выполнения объясняется в стиле крупных шагов. Следуя этому примеру, в описание базового языка OCaml, помимо определения детальной семантики мелких шагов, включена также семантика крупных шагов (*http://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#htoc7*).

Операционная семантика крупных шагов используется также консорциумом W3C: в спецификации XQuery 1.0 и XPath 2.0 (*http://www.w3.org/TR/xquery-semantics/*) способ вычисления языка описан в терминах математических правил вывода, а в спецификацию XQuery и XPath Full Text 3.0 (*http://www.w3.org/TR/xpath-full-text-30/*) включена семантика крупных шагов, написанная на XQuery.

От вашего внимания, наверное, не ускользнул тот факт, что при записи семантики мелких и крупных шагов языка SIMPLE на Ruby вместо математического формализма мы реализовали два разных *интерпретатора*. Именно в этом и состоит существо операционной семантики: объяснения смысла языка путем описания его интерпретатора. Обычно это описание формулируется с помощью простой математической нотации, что делает его ясным и однозначным – при условии, что мы способны его понять. Но за это приходится платить высоким уровнем абстракции и удаленностью от конкретики компьютеров. У использования Ruby есть существенный недостаток: привнесение дополнительной сложности реального языка программирования (классы, объекты, вызовы методов и т. д.) в то, что по идее должно было быть упрощенным объяснением. Но если мы уже знаем Ruby, то такой подход, пожалуй, позволяет лучше видеть, что происходит, а возможность исполнять описание является приятным дополнительным бонусом.

Денотационная семантика

До сих пор мы изучали смысл языков программирования с операционной точки зрения, то есть объясняли, что программа означает, показывая, что происходит при ее исполнении. Другой подход, *денотационная семантика*, подразумевает трансляцию программы с языка, на котором она написана, в другое представление.

При таком стиле задания семантики мы вообще не задаемся вопросом об исполнении программы, по крайней мере, напрямую. Вместо этого мы хотим воспользоваться уже установленным смыслом другого языка – более низкого уровня, более формального или хотя бы лучше понятого, чем описываемый, – чтобы объяснить новый язык.

Денотационная семантика принципиально является более абстрактным подходом по сравнению с операционной, потому что просто заменяет один язык другим вместо превращения языка в реальное поведение. Например, если бы нам нужно было объяснить значение глагола «ходить» человеку, с которым мы не говорим на одном языкс, то мы могли сообщить его смысл операционно – вышагивая взад-вперед. С другой стороны, объяснить французу, что значит «ходить», можно было бы и денотационно – сообщив французский эквивалент «*marcher*». Безусловно, второй вариант представляет собой более высокую форму коммуникации, не требующую физических упражнений.

Неудивительно, что денотационная семантика обычно применяется для преобразования программ в математические объекты, которые можно изучать с помощью математического инструментария, но составить общее представление об этом подходе можно, попробовав денотировать программу на SIMPLE в другой форме.

Давайте опишем денотационную семантику SIMPLE, транслировав его на Ruby[1]. На практике это означает преобразование абстрактного синтаксического дерева в строку, содержащую Ruby-код, который каким-то образом передает предполагаемый смысл этого дерева.

Но что такое «предполагаемый смысл»? Как должна выглядеть денотация наших выражений и предложений на Ruby? С операционной точки зрения, мы уже видели, что выражение принимает окружение и преобразует его в значение; один из способов выразить это на Ruby заключается в использовании процедуры, которая принимает аргумент, представляющий окружение, и возвращает объект Ruby, представляющий значение. Для простых константных выражений типа «5» или «false» мы вообще не будем использовать окружение, так что нужно лишь позаботиться о представлении конечного результата в виде объекта Ruby. По счастью, в Ruby уже есть объекты, специально предназначенные для представления таких значений: мы можем использовать значение Ruby `5` как результат выражения SIMPLE «5» и значение Ruby `false` как результат выражения «false».

Выражения

Мы можем воспользоваться этой идеей и написать реализации метода `#to_ruby` для классов `Number` и `Boolean`:

```
class Number
  def to_ruby
    "-> e { #{value.inspect} }"
  end
end

class Boolean
  def to_ruby
    "-> e { #{value.inspect} }"
  end
end
```

[1] Это означает, что мы собираемся писать Ruby-код, который генерирует Ruby-код, но один и тот же язык выбран в качестве денотационного и метаязыка реализации только для простоты. Мы вполне могли бы написать, к примеру, Ruby-код, который генерирует строки, содержащие JavaScript-код.

Вот как их работа отражается на консоли:

```
>> Number.new(5).to_ruby
=> "-> e { 5 }"
>> Boolean.new(false).to_ruby
=> "-> e { false }"
```

Оба метода порождают строку, содержащую Ruby-код, а, поскольку смысл языка Ruby мы уже понимаем, то легко видеть, чтобы обе строки – это программы, которые строят процедуры. Каждая процедура принимает окружение в аргументе `e`, игнорирует этот аргумент и возвращает значение Ruby.

Поскольку денотации являются строками исходного кода на Ruby, мы можем проверить их поведение в IRB, воспользовавшись методом `Kernel#eval` для преобразования в настоящие, допускающие вызов объекты `Proc`[1]:

```
>> proc = eval(Number.new(5).to_ruby)
=> #<Proc (lambda)>
>> proc.call({})
=> 5
>> proc = eval(Boolean.new(false).to_ruby)
=> #<Proc (lambda)>
>> proc.call({})
=> false
```

В этот момент возникает соблазн вообще избавиться от процедур и реализовать метод `#to_ruby` проще – `Number.new(5)` будет преобразовываться в строку `'5'`, а не в `'-> e { 5 }'` и аналогично в остальных случаях. Однако идея построения денотационной семантики, по крайней мере отчасти, состоит в том, чтобы передать смысл конструкций исходного языка, и в данном случае мы сообщаем, что выражение, *вообще говоря*, нуждается в окружении, пусть даже в этих конкретных выражениях оно не используется.

Для денотации выражений, в которых окружение используется, нам нужно определить, как окружение будет представлено в Ruby. Мы уже встречались с окружениями в операционной семантике, и тогда они были реализованы на Ruby. Теперь мы можем просто

[1] Это можно сделать только потому, что Ruby является одновременно языком реализации и денотации. Если бы для денотации мы выбрали исходный код на JavaScript, то исполнять его пришлось бы на консоли JavaScript.

воспользоваться той же идеей и представить окружение в виде хеша. Но детали придется изменить, поэтому имейте в виду тонкое различие: в нашей операционной семантике окружение находилось внутри виртуальной машины и применялось для связывания имен переменных с абстрактными синтаксическими деревьями SIMPLE вида `Number.new(5)`. В денотационной же семантике окружение существует в языке, на который транслируется программа, поэтому оно должно иметь смысл в том мире, а не во «внешнем мире» виртуальной машины.

В частности, это означает, что наше денотационное окружение должно ассоциировать имена переменных со значениями в смысле Ruby, например `5`, а не с объектами, представляющими синтаксические конструкции SIMPLE. Мы можем считать, что операционное окружение, например `{ x: Number.new(5) }`, имеет денотацию `'{ x: 5 }'` в языке, на который транслируется программа, нужно только не запутаться, потому что и метаязык реализации, и денотационный язык – это Ruby.

Теперь, зная, что окружение будет хешем, мы можем реализовать метод `Variable#to_ruby`:

```
class Variable
  def to_ruby
    "-> e { e[#{name.inspect}] }"
  end
end
```

Таким образом, выражение-переменная транслируется в исходный код процедуры Ruby, которая ищет нужное значение в хеше окружения:

```
>> expression = Variable.new(:x)
=> «x»
>> expression.to_ruby
=> "-> e { e[:x] }"
>> proc = eval(expression.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 7 })
=> 7
```

Важный аспект денотационной семантики – ее *композиционность*: денотация программы состоит из денотаций ее частей. Понять, как выглядит композиционность на практике, мы сможем, когда перейдем к денотации более сложных выражений, например `Add`, `Multiply` и `LessThan`:

```
class Add
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) + (#{right.to_ruby}).call(e) }"
  end
end

class Multiply
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) * (#{right.to_ruby}).call(e) }"
  end
end

class LessThan
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) < (#{right.to_ruby}).call(e) }"
  end
end
```

Здесь мы пользуемся конкатенацией строк, чтобы скомпоновать денотацию выражения из денотаций составляющих его подвыражений. Мы знаем, что каждое подвыражение денотируется исходным Ruby-кодом процедуры, поэтому можем воспользоваться ими для построения большего Ruby-кода, который вызывает эти процедуры, передавая им окружение, после чего производит то или иное вычисление с возвращенными значениями. Вот как выглядят результирующие денотации:

```
>> Add.new(Variable.new(:x), Number.new(1)).to_ruby
=> "-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) }"
>> LessThan.new(Add.new(Variable.new(:x), Number.new(1)), Number.new(3)).to_ruby
=> "-> e { (-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) }).↵
call(e) < (-> e { 3 }).call(e) }"
```

Эти денотации уже настолько сложны, что трудно с уверенностью сказать, то ли они делают, что нужно. Протестируем их, чтобы убедиться:

```
>> environment = { x: 3 }
=> {:x=>3}
>> proc = eval(Add.new(Variable.new(:x), Number.new(1)).to_ruby)
=> #<Proc (lambda)>
>> proc.call(environment)
=> 4
>> proc = eval(
     LessThan.new(Add.new(Variable.new(:x), Number.new(1)),
     Number.new(3)).to_ruby
    )
=> #<Proc (lambda)>
>> proc.call(environment)
=> false
```

Предложения

Специфицировать денотационную семантику предложений можно аналогично, только нужно помнить, что в случае операционной семантики вычисление предложения порождает новое окружение, а не значение. Это означает, что метод `Assign#to_ruby` должен порождать код процедуры, результатом которой является обновленный хеш окружения:

```
class Assign
  def to_ruby
    "-> e { e.merge({ #{name.inspect} => (#{expression.to_ruby}).call(e) }) }"
  end
end
```

И снова проверим на консоли:

```
>> statement = Assign.new(:y, Add.new(Variable.new(:x), Number.new(1)))
=> «y = x + 1»
>> statement.to_ruby
=> "-> e { e.merge({ :y => (-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).↵
call(e) }).call(e) }) }"
>> proc = eval(statement.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 3 })
=> {:x=>3, :y=>4}
```

Как всегда, семантика `DoNothing` очень проста:

```
class DoNothing
  def to_ruby
    '-> e { e }'
  end
end
```

В случае условных предложений мы можем транслировать конструкцию Simple «`if (…) { … } else { … }`» в предложение Ruby `if … then … else … end`, не забыв передать окружение во все места, где оно необходимо:

```
class If
  def to_ruby
    "-> e { if (#{condition.to_ruby}).call(e)" +
      " then (#{condition.to_ruby}).call(e)" +
      " else (#{alternative.to_ruby}).call(e)" +
      " end }"
  end
end
```

Как и в операционной семантике крупных шагов, специфицировать предложение последовательности нужно аккуратно: результат вычисления первого предложения становится окружением для вычисления второго.

```
class Sequence
  def to_ruby
    "-> e { (#{second.to_ruby}).call((#{first.to_ruby}).call(e)) }"
  end
end
```

И наконец, как и для условных предложений, мы можем транслировать предложение «while» в процедуру, где в Ruby-цикле while повторно вычисляется тело, а затем возвращается конечное окружение:

```
class While
  def to_ruby
    "-> e {" +
      " while (#{condition.to_ruby}).call(e);" +
      "     e = (#{body.to_ruby}).call(e); end;" +
      " e" +
      " }"
  end
end
```

Даже совсем простое предложение «while» может породить длинную денотацию, поэтому имеет смысл поручить интерпретатору Ruby проверку правильности семантики:

```
>> statement =
       While.new(
         LessThan.new(Variable.new(:x), Number.new(5)),
         Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
       )
=> «while (x < 5) { x = x * 3 }»
>> statement.to_ruby
=> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).↵
call(e) }).call(e);
e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) *↵
(-> e { 3 }).call(e)}).call(e) }) }).call(e); end; e }"
>> proc = eval(statement.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 1 })
=> {:x=>9}
```

Сравнение способов определения семантики

Предложение «while» – удачный пример, на котором наглядно проявляется различие между разными семантиками: мелких шагов, крупных шагов и денотационной.

Семантика мелких шагов для «while» записывается в виде правила свертки для абстрактной машины. Циклическое поведение как таковое не является частью действия правила – свертка всего лишь преобразует предложение «while» в предложение «if» – однако оно проявляется косвенно в виде будущих сверток, выполняемых машиной. Чтобы понять, что же делает «while», мы должны изучить все мелкие шаги и разобраться, как они взаимодействуют в ходе выполнения программы на Simple.

Операционная семантика крупных шагов для «while» записывается в виде правила вычисления, которое показывает, как вычислить конечное окружение непосредственно. Правило содержит рекурсивный вызов себя же, так что имеется явное указание на то, что вычисление «while» подразумевает какой-то цикл, однако это не тот цикл, который мог бы легко распознать автор программы на Simple. Правила крупных шагов записываются рекурсивно и описывают полное вычисление выражения или предложения в терминах вычисления других синтаксических конструкций; данное конкретное правило говорит нам, что результат вычисления предложения «while» может зависеть от результата вычисления того же предложения в другом окружении. Однако чтобы связать эту идею с итеративным поведением, присущим «while», необходимо интуитивное озарение. К счастью, больших усилий для этого не требуется: несложное математическое рассуждение показывает, что эти два вида цикла принципиально эквивалентны, а если метаязык поддерживает оптимизацию хвостовой рекурсии, то они эквивалентны и практически.

Денотационная семантика «while» говорит, как переписать это предложение на Ruby, – воспользовавшись ключевым словом Ruby while. Это куда более прямая трансляция: в Ruby уже имеется поддержка итеративных циклов, а денотационное правило показывает, что «while» можно реализовать с помощью этой конструкции. Чтобы понять, как оба цикла связаны между собой, не требуется никаких озарений: если мы понимаем, как работают циклы while в Ruby, то знаем все и о циклах «while» в Simple. Конечно, это означает, что мы заменили задачу понимания Simple задачей понимания денотационного языка, и это серьезный недостаток в случае, когда этот язык так сложен и плохо специфицирован, как Ruby, но становится достоинством, если денотации записываются на небольшом математическом языке.

Приложения

Итак, чего же мы достигли с помощью описанной выше денотационной семантики? Ее основное назначение – показать, как трансли-

ровать SIMPLE на Ruby, используя последний в качестве инструмента для объяснения смысла различных языковых конструкций. Попутно мы получили способ выполнить программу на SIMPLE – потому что правила денотационной семантики написаны на исполняемом языке Ruby и результатом их применения тоже является исполняемый код на Ruby – но это всего лишь счастливое совпадение, мы могли бы с тем же успехом написать правила на обычном английском языке, а для денотации воспользоваться каким-нибудь формальным математическим языком. По-настоящему важно то, что мы взяли произвольный язык собственного изобретения и преобразовали его в язык, понятный кому-то или чему-то.

Чтобы придать этой трансляции объяснительную силу, полезно вытащить части смысла языка на поверхность, а не оставлять их неявными. Например, в этой семантике окружение присутствует явно, будучи представлено вполне осязаемым объектом Ruby – хешем, который передается в процедуры и возвращается ими, – вместо того чтобы обозначать переменные как настоящие переменные Ruby и полагаться на тонкие правила областей видимости в Ruby для специфицирования доступа к переменным. В этом отношении семантика не просто перекладывает весь груз объяснений на Ruby; она использует Ruby в качестве простой основы, но проделывает дополнительную работу, чтобы точно показать, как окружение используется и изменяется различными программными конструкциями.

Ранее мы видели, что назначение операционной семантики – объяснить смысл языка, сконструировав для него интерпретатор. Напротив, трансляция с одного языка на другой, составляющая суть денотационной семантики, – аналог *компилятора*: в данном случае наши реализации метода `#to_ruby` компилируют SIMPLE на Ruby. Ни один из этих семантических стилей ничего не говорит о том, как *эффективно* реализовать интерпретатор или компилятор языка, но зато они дают официальный эталон, на основании которого можно судить о правильности любой эффективной реализации.

Денотационные определения встречаются и в реальном мире. В старых вариантах стандарта Scheme денотационная семантика использовалась для специфицирования базового языка (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-10.html#%25_sec_7.2) (в текущем стандарте применяется операционная семантика мелких шагов), а при разработке языка преобразования документов XSLT использовались написанные Филиппом Уодлером (Philip Wadler) денотационные определения образцов XSLT

(http://homepages.inf.ed.ac.uk/wadler/topics/xml.html#xsl-semantics) и выражений XPath (http://homepages.inf.ed.ac.uk/wadler/topics/xml.html#xpath-semantics).

Практический пример использования денотационной семантики для специфицирования смысла регулярных выражений см. в разделе «Семантика» на стр. 112.

Формальная семантика на практике

В этой главе показано несколько подходов к задаче придания смысла компьютерным программам. Во всех случаях мы обошлись без математических деталей и попытались рассказать об их назначении с помощью Ruby. Однако обычно формальная семантика записывается с помощью математической нотации.

Формализм

Наш обзор формальной семантики был не особенно формальным. Мы не уделяли серьезного внимания математической нотации, а использование Ruby в качестве метаязыка означает, что мы сосредоточили внимание не столько на понимании смысла программ, сколько на различных способах их исполнения. Истинное назначение денотационной семантики в том, чтобы докопаться до глубинного смысла программ путем преобразования их в корректно определенные математические объекты, а не идти по пути лицемерного представления цикла «`while`» в SIMPLE циклом `while` в Ruby.

Чтобы предоставить определения и объекты, полезные для денотационной семантики, был разработан специальный раздел математики – *теория доменов*. Это дало возможность построить модель вычислений, основанную на неподвижных точках (*http://ru.wikipedia.org/wiki/Неподвижная_точка*) монотонных функций (*http://ru.wikipedia.org/wiki/Монотонная_функция*) на частично упорядоченных множествах (*http://ru.wikipedia.org/wiki/Частично_упорядоченное_множество*). Чтобы понять программу, ее необходимо «компилировать» в математическую функцию, а для доказательства свойств этих функций использовать методы теории доменов.

С другой стороны, если денотационную семантику мы лишь расплывчато обрисовали, то наш подход к операционной семантике ближе по духу к ее формальному представлению: наши определения

методов `#reduce` и `#evaluate` в действительности являются трансляцией на Ruby математических правил вывода.

Поиск смысла

Важное применение формальной семантики – однозначно специфицировать смысл языка программирования, не полагаясь на менее формальные подходы, как, например, спецификации, написанные на естественном языке, или «спецификации путем реализации». У формальной спецификации есть и другие применения, например доказательство свойств языка вообще и конкретных программ в частности, а также изучение способов безопасного преобразования программ с целью повысить их эффективность, не изменяя поведение.

Например, поскольку операционная семантика очень близко соответствует реализации интерпретатора, специалист по информатике может рассматривать подходящий интерпретатор как операционную семантику языка и затем доказать ее корректность относительно денотационной семантики того же языка. Это означает, что существует разумная связь между смыслами, придаваемыми программе интерпретатором и денотационной семантикой.

Преимущество денотационной семантики в том, что она более абстрактна, чем операционная, поскольку игнорирует детали того, как программа выполняется, а акцентирует внимание на ее преобразовании в другое представление. Например, это позволяет сравнить две программы, написанные на разных языках, при условии, что для обоих существует денотационная семантика, транслирующая их в общее представление.

Из-за столь высокой степени абстракции денотационная семантика может показаться окольным путем. Если задача состоит в том, чтобы объяснить смысл языка программирования, то как трансляция одного языка на другой приближает нас к ее решению? Денотация хороша лишь настолько, насколько хорош смысл; в частности, денотационная семантика приближает нас к возможности фактически выполнить программу лишь в том случае, когда у денотационного языка имеется какой-то *операционный* смысл, собственная семантика, которая показывает, как его можно выполнить, а не транслировать еще на один язык.

В формальной денотационной семантике используются абстрактные математические объекты, обычно функции, для обозначения таких конструкций языка программирования, как выражения и пред-

ложения, а поскольку в математике имеются соглашения о том, как вычислять функции, мы получаем прямой способ рассуждать о денотации в операционном смысле. Мы приняли менее формальный подход, при котором денотационная семантика мыслится как компилятор с одного языка на другой, и в действительности именно так в конечном итоге выполняется большинство языков программирования: программа на Java компилируется в байт-код с помощью компилятора `javac`, байт-код затем JIT-компилируется в команды процессора x86 виртуальной машиной Java, после чего процессор декодирует команды x86 в микрокоманды из RISC-подобного набора, предназначенные для исполнения ядром... конец когда-нибудь будет? Или на пути вниз мы будем встречать только компиляторы да виртуальные машины?

Разумеется, в конечном итоге программа исполняется, потому что семантическая пирамида все-таки возведена на фундаменте *реальной* машины: электронов, бегающих в полупроводниках, подчиняясь законам физики[1]. Компьютер – это устройство для поддержания шаткой конструкции, состоящей из многочисленных сложных уровней интерпретации, водруженных один на другой; он позволяет постепенно переводить человеческие представления, например жесты несколькими пальцами или циклы `while`, в физическую вселенную, где царят кремний и электрические сигналы.

Альтернативы

Семантические стили, с которыми мы познакомились в этой главе, известны и под другими именами. Семантику мелких шагов также называют *структурной операционной семантикой* или *семантикой переходов*; семантика крупных шагов больше известна под названиями *естественная семантика* или *реляционная семантика*. Наконец, денотационную семантику называют еще *семантикой неподвижной точки* или *математической семантикой*.

Существуют и другие стили формальной семантики. Например, *аксиоматическая семантика* описывает смысл предложения, формулируя утверждения о состоянии абстрактной машины до и после его выполнения: если одно утверждение (*предусловие*) верно до вы-

---

[1] Или в случае механического компьютера типа «аналитической машины», изобретенной Чарльзом Бэббиджем в 1837 году, зубчатое колесо и бумага, подчиняющиеся законам физики.

полнения предложения, то другое утверждение (*постусловие*) будет верно после его выполнения. Аксиоматическая семантика полезна для проверки правильности программ: когда предложения объединяются для построения большей программы, соответствующие им утверждения можно объединить в большее утверждение, а цель состоит в том, чтобы показать, что совокупное утверждение о программе соответствует ее задуманной спецификации.

Хотя детали несколько отличаются, стиль аксиоматической семантики лучше всего характеризует проект RubySpec (*http://rubyspec.org/*) – «исполняемую спецификацию для языка программирования Ruby», в котором утверждения в стиле RSpec используются для описания поведения встроенных в Ruby языковых конструкций, а также базовой и стандартных библиотек. Вот, например, фрагмент описания метода `Array#<<` в виде RubySpec-спецификации:

```
describe "Array#<<" do
  it "correctly resizes the Array" do
    a = []
    a.size.should == 0
    a << :foo
    a.size.should == 1
    a << :bar << :baz
    a.size.should == 3
    a = [1, 2, 3]
    a.shift
    a.shift
    a.shift
    a << :foo
    a.should == [:foo]
  end
end
```

Реализация синтаксических анализаторов

В этой главе мы строили абстрактные синтаксические деревья программ на Simple вручную – выписывая Ruby-выражения типа `Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))` – вместо того чтобы просто написать исходный код на Simple, например `'x = x + 1'`, и потом запустить анализатор, который автоматически преобразует его в синтаксическое дерево.

Реализация анализатора Simple с нуля потребовала бы погрузиться в многочисленные детали и увела бы нас далеко в сторону от обсуждения формальной семантики. Однако возиться с игрушеч-

ными языками программирования интересно, а благодаря наличию инструментов и библиотек для синтаксического разбора не так уж трудно построить анализатор, опираясь на уже проделанную кем-то работу. Ниже вкратце описывается, как это сделать.

Один из лучших инструментов разбора для Ruby – Treetop (*http://treetop.rubyforge.org/*), предметно-ориентированный язык описания синтаксиса способом, позволяющим автоматически генерировать синтаксические анализаторы. Описание синтаксиса языка на Treetop представляется в виде *грамматики, разбирающей выражения* (parsing expression grammar – PEG), набора правил в форме, напоминающей регулярные выражения, простых для записи и понимания. Но самое главное – эти правила можно аннотировать определениями методов, так что объекты Ruby, генерируемые в процессе разбора, можно наделить поведением. Способность определять одновременно синтаксическую структуру и исходный Ruby-код, оперирующей этой структурой, делает Treetop идеальным средством, чтобы вчерне набросать синтаксис языка и придать ему семантику исполнения.

Чтобы почувствовать, как это работает, мы приводим сокращенный вариант грамматики SIMPLE, записанной на Treetop и содержащей только правила, достаточные для разбора строки `'while (x < 5) { x = x * 3 }'`:

```
grammar Simple
  rule statement
    while / assign
  end

  rule while
    'while (' condition:expression ') {   body:statement ' }' {
      def to_ast
        While.new(condition.to_ast, body.to_ast)
      end
    }
  end

  rule assign
    name:[a-z]+ ' = ' expression {
      def to_ast
        Assign.new(name.text_value.to_sym, expression.to_ast)
      end
    }
  end

  rule expression
    less_than
  end
```

```
  rule less_than
    left:multiply ' < ' right:less_than {
      def to_ast
        LessThan.new(left.to_ast, right.to_ast)
      end
    }
    /
    multiply
  end

  rule multiply
    left:term ' * ' right:multiply {
      def to_ast
        Multiply.new(left.to_ast, right.to_ast)
      end
    }
    /
    term
  end

  rule term
    number / variable
  end

  rule number
    [0-9]+ {
      def to_ast
        Number.new(text_value.to_i)
      end
    }
  end

  rule variable
    [a-z]+ {
      def to_ast
        Variable.new(text_value.to_sym)
      end
    }
  end
end
```

Этот язык немного напоминает Ruby, но сходство лишь поверхностное; грамматики записываются на специальном языке Treetop. Ключевое слово `rule` начинает новое правила разбора синтаксической конструкции, а выражения внутри правила описывают структуру распознаваемых им строк. Одни правила могут рекурсивно вызывать другие, так правило `while` вызывает правила `expression` и `statement`. Разбор начинается с первого правила, в данной грамматике это `statement`.

Порядок, в котором правила разбора выражений вызывают друг друга, отражает приоритеты операторов языка SIMPLE. Правило `expression` вызывает `less_than`, которое сразу же вызывает прави-

ло `multiply`, чтобы дать ему шанс сопоставить оператор * где-то в строке, до того как `less_than` получит возможность провести сопоставление с оператором <, имеющем более низкий приоритет. Тем самым гарантируется, что выражение `'1 * 2 < 3'` разбирается как «`(1 * 2) < 3`», а не как «`1 * (2 < 3)`».

Чтобы избежать дополнительных сложностей, в этой грамматике не делается попытки наложить ограничения на то, какие выражения могут присутствовать внутри других выражений, а это значит, что анализатор будет принимать заведомо некорректные программы.

Например, у нас есть два правила для бинарных выражений – `less_than` и `multiply` – но единственная причина разделить их состоит в том, чтобы задать приоритеты, поэтому каждое правило требует лишь, чтобы его левый операнд был сопоставлен с правилом более высокого приоритета, а правый операнд – с правилом такого же или более высокого приоритета. Таким образом, строка `'1 < 2 < 3'` успешно разбирается, хотя семантика SIMPLE не придает получающемуся в результате выражению никакого смысла.

Некоторые проблемы такого рода можно разрешить, подкорректировав грамматику, но обязательно останутся случаи, которые анализатор не сможет корректно обработать. Мы разделим обязанности, оставив анализатор максимально либеральным, а для распознавания недопустимых программ применим другую технику в главе 9.

Большинство правил грамматики аннотировано Ruby-кодом, заключенным в фигурные скобки. В нем всегда определен метод `#to_ast`, который будет присутствовать в синтаксических объектах, которые Treetop генерирует при разборе программы на SIMPLE.

Если мы сохраним эту грамматику в файле *simple.treetop*, то впоследствии сможем загрузить его с помощью Treetop и сгенерировать класс `SimpleParser`. Этот анализатор преобразует строку исходного кода на SIMPLE в представление, построенное из объектов `SyntaxNode`, генерируемых Treetop:

```
>> require 'treetop'
=> true
>> Treetop.load('simple')
SimpleParser
>> parse_tree = SimpleParser.new.parse('while (x < 5) { x = x * 3 }')
=> SyntaxNode+While1+While0 offset=0, "…5) { x = x * 3 }" (to_ast,condition,body):
       SyntaxNode offset=0, "while ("
       SyntaxNode+LessThan1+LessThan0 offset=7, "x < 5" (to_ast,left,right):
```

```
    SyntaxNode+Variable0 offset=7, "x" (to_ast):
      SyntaxNode offset=7, "x"
    SyntaxNode offset=8, " < "
    SyntaxNode+Number0 offset=11, "5" (to_ast):
      SyntaxNode offset=11, "5"
  SyntaxNode offset=12, ") { "
  SyntaxNode+Assign1+Assign0 offset=16, "x = x * 3" (to_ast,name,expression):
    SyntaxNode offset=16, "x":
      SyntaxNode offset=16, "x"
    SyntaxNode offset=17, " = "
    SyntaxNode+Multiply1+Multiply0 offset=20, "x * 3" (to_ast,left,right):
      SyntaxNode+Variable0 offset=20, "x" (to_ast):
        SyntaxNode offset=20, "x"
      SyntaxNode offset=21, " * "
      SyntaxNode+Number0 offset=24, "3" (to_ast):
        SyntaxNode offset=24, "3"
  SyntaxNode offset=25, " }"
```

Эта структура из узлов `SyntaxNode` представляет собой *конкретное синтаксическое дерево*: она предназначена специально для манипулирования анализатором Treetop и содержит много посторонней информации о том, как узлы соотносятся с исходным кодом, из которого порождены. Вот что говорится по этому поводу в документации Treetop (*http://treetop.rubyforge.org/using_in_ruby.html*):

> Не пытайтесь обойти синтаксическое дерево самостоятельно и не используйте его как удобную структуру для хранения своих данных. Оно содержит гораздо больше узлов, чем нужно вашему приложению, и даже для одного входного символа нередко бывает больше одного узла.
>
> Вместо этого добавьте в корневое правило методы, которые возвращают нужную вам информацию в осмысленном виде. Каждое правило может вызывать свои подправила, и такой способ обхода синтаксического дерева намного предпочтительнее попытки обойти его извне.

Именно так мы и поступили. Вместо того чтобы манипулировать перегруженным информацией деревом напрямую, мы воспользовались аннотациями и определили в каждом узле дерева метод `#to_ast`. Если вызвать этот метод для корневого узла, то он построит абстрактное синтаксическое дерево, составленное из синтаксических объектов Simple:

```
>> statement = parse_tree.to_ast
=> «while (x < 5) { x = x * 3 }»
```

Таким образом, мы автоматически преобразовали исходный код в абстрактное синтаксическое дерево и теперь можем им воспользоваться для изучения смысла программы, как обычно:

```
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«9»}
>> statement.to_ruby
=> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).↵
call(e) }).call(e);
e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) *↵
(-> e { 3 }).call(e)}).call(e) }) }).call(e); end; e }"
```

Еще один недостаток этого анализатора и Treetop вообще – тот факт, что он генерирует *правоассоциативное* конкретное синтаксическое дерево. Это означает, что строка `'1 * 2 * 3 * 4'` генерируется, как будто она записана в виде `'1 * (2 * (3 * 4))'`:

```
>> expression = SimpleParser.new.parse('1 * 2 * 3 * 4', root:
:expression).to_ast
=> «1 * 2 * 3 * 4»
>> expression.left
=> «1»
>> expression.right
=> «2 * 3 * 4»
```

Но умножение традиционно считается *левоассоцитивной* операцией: запись `'1 * 2 * 3 * 4'` на самом деле означает `'((1 * 2) * 3) * 4'`, то есть числа группируются с левого, а не с правого конца выражения. В случае умножения это несущественно – результат в обоих случаях получается одинаковый, но для вычитания и деления возникают проблемы, поскольку вычисление «`((1 - 2) - 3) - 4`» и «`1 - (2 - (3 - 4))`» дает разные результаты.

Чтобы исправить это, нужно было бы несколько усложнить правила и реализации метода `#to_ast`. В разделе «Синтаксический разбор» на стр. 253 приведена грамматика Treetop, которая строит левоассоциативное AST.

Конечно, удобно разбирать SIMPLE-программы таким образом, но при этом всю сложную работу делает за нас Treetop, и мы ничего не узнали о том, как на самом деле работает синтаксический анализатор. В разделе «Синтаксический анализ с помощью автоматов с магазинной памятью» на стр. 160 мы покажем, как реализовать анализатор самостоятельно.

Глава 3. Простейшие компьютеры

За какие-то несколько недолгих лет мы оказались в плотном окружении компьютеров. Раньше они были надежно скрыты от глаз в военных научно-исследовательских центрах и университетских лабораториях, а теперь находятся повсюду: на столах, в карманах, под капотом автомобилей, имплантированы в наши тела. Мы, программисты, работаем со сложнейшими вычислительными устройствами повседневно, но хорошо ли мы понимаем, как они работают?

Мощь современных компьютеров неотделима от их сложности. Разобраться во всех деталях многочисленных подсистем компьютера трудно, но еще труднее понять, как эти подсистемы взаимодействуют, порождая единую систему. Из-за этой сложности практически бессмысленно рассуждать непосредственно о возможностях и поведении реальных компьютеров, поэтому было бы полезно построить упрощенную модель компьютера, которая, обладая некоторыми интересными чертами настоящей машины, все же может быть понята во всей полноте.

В этой главе мы оставим от идеи вычислительной машины только самое необходимое, посмотрим, для чего ее можно использовать, и изучим пределы доступного такому примитивному компьютеру.

Детерминированные конечные автоматы

Реальные компьютеры обычно оснащаются большим объемом оперативной (RAM) и энергонезависимой памяти (жесткие или твердотельные диски), многочисленными устройствами ввода-вывода и несколькими процессорными ядрами, способными одновременно выполнять несколько команд. *Конечный автомат* (КА) – это до предела упрощенная модель компьютера, которая жертвует всеми

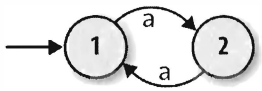
этими особенностями в обмен на простоту понимания, удобство рассуждения и легкость программной или аппаратной реализации.

Состояния, правила и входной поток

У конечного автомата нет постоянной памяти и почти нет оперативной памяти. Это небольшая машина с конечным числом возможных *состояний*, которая умеет запоминать, в каком состоянии находится в данный момент; можете считать, что это компьютер с оперативной памятью для хранения всего одного значения. У конечного автомата нет также клавиатуры, мыши или сетевого интерфейса для получения входных данных, а есть один внешний поток входных символов, которые он может читать по одному.

У конечного автомата нет универсального процессора, способного выполнять произвольные программы, зато есть встроенный набор *правил*, которые определяют, как следует переходить из одного состояния в другое в ответ на прочитанный входной символ. Автомат начинает работу в некотором состоянии, читает символы из входного потока и выполняет соответствующие правила.

Ниже наглядно представлен пример структуры конечного автомата:



Окружности представляют два состояния автомата, 1 и 2, стрелка извне означает, что автомат всегда начинает работу в состоянии 1 – *начальном состоянии*. Стрелки между состояниями соответствуют правилам автомата, а именно:

- если автомат находится в состоянии 1 и прочитан символ a, то перейти в состояние 2;
- если автомат находится в состоянии 2 и прочитан символ a, то перейти в состояние 1.

Этой информации достаточно, чтобы исследовать, как автомат обрабатывает входной поток.

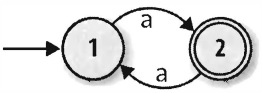
- Автомат начинает работу в состоянии 1.
- У автомата есть только правила для чтения символа a из входного потока, поэтому ничего другого произойти не может. Прочитав a, автомат переходит из состояния 1 в состояние 2.

- Прочитав еще один символ a, автомат возвращается в состояние 1.

Оказавшись снова в состоянии 1, автомат повторяет весь цикл сначала, этим поведение данного конкретного автомата и исчерпывается. Предполагается, что информация о текущем состоянии – внутреннее свойство автомата; он работает как «черный ящик», не раскрывающий своего внутреннего устройства. Таким образом, его поведение не только неинтересно, но и бесполезно, так как нет никакого наблюдаемого результата. Внешний наблюдатель не видит, что происходит какая-то деятельность, пока автомат переходит из одного состояния в другое и обратно, так что в данном случае мы вполне могли бы считать, что есть всего одно состояние, и вообще не думать о внутренней структуре.

Вывод

Для решения этой проблемы конечный автомат располагает рудиментарными средствами вывода. Конечно, речь не идет о сложных устройствах вывода настоящих компьютеров; мы всего лишь помечаем некоторые состояния как специальные и говорим, что автомат способен вывести один бит информации: находится он в данный момент в специальном состоянии или нет. Для данного автомата сделаем специальным состояние 2 и покажем его на диаграмме двойной окружностью.

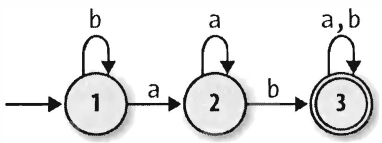


Такие специальные состояния обычно называются *заключительными* (или *допускающими*, имея в виду, что автомат может *допускать* или *отвергать* цепочки символов). Если рассматриваемый автомат начинает работу в состоянии 1 и читает один символ a, то он остается в заключительном состоянии 2, поэтому можно сказать, что автомат допускает цепочку `'a'`. С другой стороны, если автомат прочтет два символа a подряд, то он останется в состоянии 1, которое заключительным не является, и, значит, цепочка `'aa'` отвергается. Вообще, как легко видеть, этот автомат допускает любую цепочку символов a нечетной длины – `'a'`, `'aaa'`, `'aaaaa'` – и отвергает все цепочки четной длины – `'aa'`, `'aaaa'` и в том числе `''` (пустая цепочка).

Вот теперь мы получили нечто чуть более полезное: автомат, который умеет читать цепочку символов и возвращать «да» или «нет» в зависимости от того, допущена она или нет. Вполне можно сказать, что этот КА выполняет вычисление, потому что мы можем задать вопрос «является ли длина цепочки нечетным числом?» и получить осмысленный ответ. Пожалуй, этого уже достаточно, чтобы назвать его простым компьютером, а в таблице ниже его свойства сравниваются со свойствами настоящего компьютера.

	Настоящий компьютер	Конечный автомат
Постоянная память	Жесткий или твердотельный диск	Нет
Временная память	ОЗУ	Текущее состояние
Ввод	Клавиатура, мышь, сеть и т. д.	Поток символов
Вывод	Дисплей, динамики, сеть и т. д.	Является ли текущее состояние заключительным (да/нет)
Процессор	Ядра ЦП, способные исполнять произвольную программу	Правила изменения состояния в ответ на прочитанный входной символ

Конечно, этот автомат не делает ничего особо интересного или полезного, но можно построить и более сложные автоматы с большим числом состояний, способные читать не только один символ. Вот пример автомата с тремя состояниями, который может читать символы `a` и `b`:



Этот автомат допускает цепочки `'ab'`, `'baba'` и `'aaaab'` и отвергает цепочки `'a'`, `'baa'` и `'bbbba'`. Немного поэкспериментировав, мы поймем, что он допускает лишь цепочки, содержащие в любом месте подцепочку `'ab'`. Это тоже не бог весть как полезно, но все-таки демонстрирует некоторую гибкость. Ниже мы увидим приложения, обладающие большей практической ценностью.

Детерминированность

Автомат описанного выше вида называется *детерминированным*: в каком бы состоянии он ни находился и какой бы символ ни про-

чел, точно известно, в каком состоянии он окажется далее. Эта уверенность гарантируется при соблюдении двух ограничений.

- Непротиворечивость: не существует состояний, переход из которых неоднозначен из-за наличия конфликтующих правил (это означает, что ни для какого состояния не должно быть более одного правила с одним и тем же входным символом).
- Полнота: не существует состояний, переход из которых не определен из-за отсутствия правила (это означает, что для любого состояния должно существовать хотя бы одно правило для каждого возможного входного символа).

В совокупности эти ограничения означают, что для любой комбинации состояния и входного символа должно быть ровно одно правило. Автомат, который удовлетворяет таким ограничениям, называется *детерминированным конечным автоматом* (ДКА).

Моделирование

Детерминированные конечные автоматы предназначены для построения абстрактных моделей вычислений. Мы можем рисовать диаграммы таких автоматов (два примера было приведено выше) и рассуждать об их поведении, но физически таких машин не существует, поэтому подать им на вход данные и посмотреть, как они будут себя вести, невозможно. Впрочем, ДКА настолько просты, что мы без труда сможем построить *модель* автомата на Ruby и взаимодействовать с ней непосредственно.

Начнем построение модели с реализации набора правил, который мы будем называть *сводом правил* (rulebook):

```
class FARule < Struct.new(:state, :character, :next_state)
  def applies_to?(state, character)
    self.state == state && self.character == character
  end

  def follow
    next_state
  end

  def inspect
    "#<FARule #{state.inspect} --#{character}--> #{next_state.inspect}>"
  end
end

class DFARulebook < Struct.new(:rules)
  def next_state(state, character)
    rule_for(state, character).follow
```

```
  end

  def rule_for(state, character)
    rules.detect { |rule| rule.applies_to?(state, character) }
  end
end
```

Этот код определяет простой API для правил: у любого правила имеется метод `#applies_to?`, который возвращает `true` или `false` в зависимости от того, применимо правило в данной ситуации или нет, и метод `#follow`, сообщающий, как должно измениться состояние автомата при следовании этому правилу[1]. В методе `DFARulebook#next_state` оба эти метода используются для поиска подходящего правила и определения следующего состояния ДКА.

Использование метода `Enumerable#detect` в реализации `DFARulebook#next_state` подразумевает, что существует ровно одно правило для каждой комбинации состояния и входного символа. Если применимых правил больше, то выбирается первое, а все остальные игнорируются; если нет ни одного применимого правила, то `#detect` вернет `nil` и модель аварийно завершится, попытавшись выполнить `nil.follow`.

Именно поэтому мы назвали класс `DFARulebook`, а не просто `FARulebook`: он работает только при условии, что выполняются ограничения детерминированности.

Свод правил позволяет обернуть несколько правил одним объектом и спрашивать у него, каким будет следующее состояние:

```
>> rulebook = DFARulebook.new([
       FARule.new(1, 'a', 2), FARule.new(1, 'b', 1),
       FARule.new(2, 'a', 2), FARule.new(2, 'b', 3),
       FARule.new(3, 'a', 3), FARule.new(3, 'b', 3)
    ])
=> #<struct DFARulebook …>
>> rulebook.next_state(1, 'a')
=> 2
>> rulebook.next_state(1, 'b')
=> 1
>> rulebook.next_state(2, 'b')
=> 3
```

Мы могли выбрать разные способы представления состояния автомата в виде значений Ruby. Важно лишь, чтобы мож-

[1] Это достаточно общий дизайн, применимый к различным видам машин и правил, поэтому мы сможем повторно использовать его и позже, когда будем рассматривать более сложные вещи.

но было отличить одно состояние от другого: наша реализация `DFARulebook#next_state` должна иметь возможность сравнить два состояния на равенство, но в остальном ей безразлично, являются ли эти объекты числами, символами, строками, хешами или обезличенными экземплярами класса `Object`.

В данном случае обычные числа Ruby подходят больше всего – они точно соответствуют пронумерованным состояниям на диаграмме – поэтому на них мы и остановились.

Теперь мы можем воспользоваться сводом правил для построения объекта `DFA`, который будет хранить свое текущее состояние и сообщать, находится он в заключительном состоянии или нет.

```
class DFA < Struct.new(:current_state, :accept_states, :rulebook)
  def accepting?
    accept_states.include?(current_state)
  end
end

>> DFA.new(1, [1, 3], rulebook).accepting?
=> true
>> DFA.new(1, [3], rulebook).accepting?
=> false
```

Далее можно написать метод, который читает входной символ, справляется со сводом правил и нужным образом изменяет состояние:

```
class DFA
  def read_character(character)
    self.current_state = rulebook.next_state(current_state, character)
  end
end
```

Это дает нам возможность подавать символы на вход ДКА и наблюдать, как меняется его выход:

```
>> dfa = DFA.new(1, [3], rulebook); dfa.accepting?
=> false
>> dfa.read_character('b'); dfa.accepting?
=> false
>> 3.times do dfa.read_character('a') end; dfa.accepting?
=> false
>> dfa.read_character('b'); dfa.accepting?
=> true
```

Подавать ДКА по одному символу не очень удобно, поэтому добавим вспомогательный метод, который читает целую входную цепочку:

```
class DFA
  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end
end
```

Теперь мы можем подать ДКА сразу всю цепочку входных символов:

```
>> dfa = DFA.new(1, [3], rulebook); dfa.accepting?
=> false
>> dfa.read_string('baaab'); dfa.accepting?
=> true
```

Прочитав какие-то символы из входного потока, ДКА, скорее всего, не будет находиться в начальном состоянии, поэтому мы не можем без опаски повторно воспользоваться им для проверки новой входной цепочки. Следовательно, его нужно создавать заново – с одними и теми же начальным состоянием, заключительными состояниями и сводом правил – всякий раз, как мы захотим проверить, допускает ли автомат новую цепочку. Но можно не делать это вручную, а обернуть аргументы конструктора объектом, который представляет *структуру* конкретного ДКА, и воспользоваться этим объектом для автоматического создания экземпляров ДКА с целью проверки новой цепочки.

```
class DFADesign < Struct.new(:start_state, :accept_states, :rulebook)
  def to_dfa
    DFA.new(start_state, accept_states, rulebook)
  end

  def accepts?(string)
    to_dfa.tap { |dfa| dfa.read_string(string) }.accepting?
  end
end
```

Метод #tap вычисляет блок и возвращает объект, от имени которого был вызван.

Метод `DFADesign#accepts?` вызывает метод `DFADesign#to_dfa`, чтобы создать новый экземпляр ДКА, а затем метод `#read_string?`, чтобы перевести этот экземпляр в допускающее или отвергающее состояние:

```
>> dfa_design = DFADesign.new(1, [3], rulebook)
=> #<struct DFADesign …>
>> dfa_design.accepts?('a')
=> false
>> dfa_design.accepts?('baa')
=> false
>> dfa_design.accepts?('baba')
=> true
```

Недетерминированные конечные автоматы

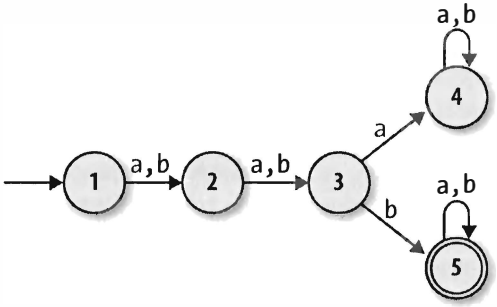
ДКА просты для понимания и реализации, но это потому, что они очень похожи на машины, с которыми мы уже знакомы. Абстрагировавшись от сложности настоящего компьютера, мы получили возможность экспериментировать с менее обыденными идеями, которые уведут нас в сторону от привычных машин, но при этом избавили себя от трудностей, связанных с воплощением этих идей в реальной системе.

Одно из возможных направлений исследования – ослабить принятые допущения и ограничения. Ну, например – ограничения детерминированности представляются чрезмерно строгими: быть может, не во всех состояниях нас интересуют все возможные символы, так почему бы тогда не опустить правила для безразличных нам символов и не разрешить автомату переход в обобщенное состояние отказа, если происходит что-то неожиданное? Или такой экзотический вопрос – как можно было бы интерпретировать наличие противоречивых правил, то есть нескольких путей исполнения? Далее, мы предполагали, что изменение состояния может происходить только в ответ на прочитанный символ; а что, если бы автомат мог менять свое состояние, ничего не читая?

В этом разделе мы исследуем эти идеи и посмотрим, какие новые возможности открываются в результате изменения свойств конечного автомата.

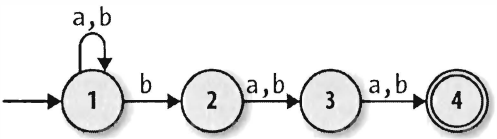
Недетерминированность

Допустим, нам нужно, чтобы конечный автомат мог допускать лишь такие цепочки символов `a` и `b`, в которых на третьем месте находится `b`. Соответствующая структура ДКА несложна:



Но что если нужен автомат, который допускал бы только цепочки, в которых на *третьем от конца* месте находится символ b? Как это можно было бы сделать? Похоже, задачка оказалась посложнее: гарантируется, что показанный выше ДКА находится в состоянии 3 после чтения третьего символа, но автомат не может знать, *в какой момент* он читает третий от конца символ, потому что длина цепочки неизвестна, пока она прочитана целиком. Не вполне понятно, возможно ли вообще построить такой ДКА.

Однако если ослабить ограничения детерминированности и разрешить включение в свод нескольких правил (или ни одного) для данной комбинации состояния и входного символа, то построить требуемый автомат удастся:



Такой конечный автомат называется *недетерминированным* (НКА), в нем путь выполнения для каждой входной цепочки уже необязательно единственный. Находясь в состоянии 1 и прочитав символ b, этот автомат может выбрать как правило, которое оставляет его в состоянии 1, так и правило, которое переводит его в состояние 2. С другой стороны, в состоянии 4 никаких правил нет, поэтому, оказавшись в нем, автомат не сможет продолжать чтение. В случае ДКА следующее состояние однозначно определено текущим состоянием и прочитанным символом, а в случае НКА иногда бывает несколько вариантов перехода в следующее состояние, а иногда ни одного.

ДКА допускает цепочку, если в результате чтения символов и слепого следования правилам он оказывается в заключительном состоянии, но что означает допустить или отвергнуть цепочку в случае НКА? Естественный ответ звучит так: цепочка допускается, если существует *хотя бы один* способ попасть в заключительное состояние, следуя каким-то правилам, то есть если завершение в заключительном состоянии *возможно*, пусть даже оно не является неизбежным.

Например, показанный выше НКА допускает цепочку `'baa'`, поскольку начав работу в состоянии 1, он может следовать таким правилам: прочитать `b` и перейти в состояние 2, затем прочитать `a` и перейти в состояние 3 и, наконец, прочитать еще один `a` и остановиться в состоянии 4, которое является заключительным. Он допускает также цепочку `'bbbbb'`, потому что может в самом начале выбрать другое правило и оставаться в состоянии 1 после чтения первых двух `b`, а только потом – после чтения третьего `b` – воспользоваться правилом для перехода в состояние 2, что позволит прочесть остаток цепочки и остановиться в состоянии 4, как и прежде.

С другой стороны, не существует способа прочитать цепочку `'abb'` и остановиться в состоянии 4; в зависимости от выбираемых правил данный НКА может оказаться в состояниях 1, 2 или 3, поэтому цепочка `'abb'` отвергается. Точно так же отвергается цепочка `'bbabb'`, которая не позволяет продвинуться дальше состояния 3; если перейти в состояние 2 сразу после чтения первого `b`, то НКА окажется в состоянии 4 слишком рано – ему нужно прочитать еще два символа, а никаких правил уже нет.

Множество всех цепочек, допускаемых конкретным автоматом, называется *языком*, мы говорим, что автомат *распознает* этот язык. Но не для всякого возможного языка существует распознающий его ДКА или НКА (дополнительные сведения см. в главе 4). Языки, распознаваемые каким-то конечным автоматом, называются *регулярными*.

Ослабив ограничения детерминированности, мы получили воображаемую машину, сильно отличающуюся от привычных нам реальных детерминированных компьютеров. НКА имеет дело с возможными, а не с достоверными событиями; при обсуждении его поведения мы говорим о том, что *может* произойти, а не о том, что *наверняка* произойдет. Вроде бы потенциал огромный, но как такая машина могла бы работать в реальном мире? На первый взгляд кажется, что НКА должен обладать пророческим даром, чтобы знать,

какую из нескольких возможностей выбрать при чтении входного потока; чтобы иметь шанс допустить цепочку, НКА из нашего примера должен был бы оставаться в состоянии 1, пока не прочтет третий от конца символ, но он же не знает, сколько всего символов получит. Как можно смоделировать такую удивительную машину на скучном детерминированном Ruby?

Чтобы смоделировать НКА на детерминированном компьютере, необходимо каким-то образом исследовать *все возможные пути выполнения* автомата. Такой полный перебор позволяет обойтись без потустороннего предвидения, которое потребовалось бы, чтобы смоделировать только один из возможных путей, каким-то образом принимая на каждом шаге правильные решения. После чтения очередного символа у НКА есть лишь конечное число возможностей сделать следующий переход, поэтому мы можем смоделировать недетерминированность, перебрав их все и посмотрев, какие в конечном итоге позволят прийти в заключительное состояние.

Перебор можно организовать рекурсивно: всякий раз, как после чтения символа оказывается несколько применимых правил, выбрать одно из них и попытаться прочитать остаток входной цепочки; если при этом автомат не попадает в заключительное состояние, вернуться к предыдущему состоянию, перемотать входную цепочку назад до позиции, соответствующей этому состоянию, и повторить попытку, выбрав другое правило. Повторять эти шаги, пока не будет найдена последовательность правил, переводящая автомат в заключительное состояние, или пока не будут безуспешно перепробованы все возможные последовательности.

Другая стратегия состоит в том, чтобы моделировать все возможности параллельно, запуская новые потоки всякий раз, как у автомата есть более одного правила на выбор. При этом мы по существу клонируем моделируемый НКА, давая каждой копии возможность попробовать свое правило и посмотреть, к чему это приведет. Все потоки можно запустить сразу, передав каждому отдельную копию входной цепочки; если какой-то поток прочитает все символы и остановится в заключительном состоянии, то мы сможем сказать, что цепочка допущена.

Обе описанные реализации осуществимы, но довольно сложны и неэффективны. Наша модель ДКА была простой и могла читать символы по отдельности, сообщая по ходу дела, находится ли автомат в заключительном состоянии или нет. Хорошо было бы построить модель НКА столь же простую и прозрачную.

По счастью, существует простой способ смоделировать НКА без возвратов назад, без запуска потоков и без знания всех входных символов наперед. Действительно, если в модели ДКА мы хранили текущее состояние, то для моделирования НКА нужно будет хранить множество всех *возможных* текущих состояний. Это проще и эффективнее, чем моделировать несколько НКА, расходящихся в разных направлениях, а конечный результат оказывается тем же самым. Если бы мы моделировали много отдельных автоматов, то интересовало бы нас только состояние каждого из них, но автоматы, находящиеся в одном и том же состоянии, неразличимы[1], поэтому мы ничего не потеряем, если вместо этого свернем все возможности в один-единственный автомат и зададимся вопросом «в каких состояниях он *мог бы* находиться в текущий момент?».

Например, проследим, что происходит с нашим НКА, когда он читает цепочку `'bab'`:

- До чтения первого символа НКА определенно находится в своем начальном состоянии 1.
- Он читает первый символ – `b`. В состоянии 1 для символа `b` существует правило, которое оставляет НКА в состоянии 1, и другое правило, которое переводит его в состояние 2, поэтому мы знаем, что в результате НКА окажется в состоянии 1 или 2. Ни одно из них не является заключительным, а значит, НКА никаким способом не может перейти в заключительное состояние, прочитав цепочку `'b'`.
- Он читает второй символ – `a`. Если он находится в состоянии 1, то проследовать он может по единственному правилу для символа `a`, которое оставит его в состоянии 1; если же он находится в состоянии 2, то должен будет проследовать по правилу для символа `a`, которое переведет его в состояние 3. Он должен оказаться в состоянии 1 или 3, и опять-таки ни одно из них не является заключительным, следовательно, и цепочка `'ba'` этим автоматом не допускается.
- Он читает первый символ – `b`. Если он находится в состоянии 1, то, как и раньше, может либо остаться в состоянии 1, либо перейти в состояние 2. Если же он находится в состоянии, то обязательно перейдет в состояние 4.

---

[1] Конечный автомат не хранит свою историю и вообще не имеет никакой памяти, кроме текущего состояния, поэтому с любой точки зрения два одинаковых автомата, находящихся в одном и том же состоянии, взаимозаменяемы.

- Теперь мы знаем, что после чтения всей цепочки НКА может оказаться в состоянии 1, 2 или 4. Состояние 4 заключительное, и таким образом моделирование показывает, что существует *хотя бы один* способ достичь состояния 4, прочитав цепочку `'bab'`, а, значит, данный НКА ее допускает.

Эту стратегию моделирования легко воплотить в код. Для начала нам понадобится свод правил, пригодный для хранения правил НКА. Свод правил ДКА всегда возвращает единственное состояние, когда мы спрашиваем, куда должен перейти ДКА после чтения определенного символа в определенном состоянии. Но свод правил НКА должен отвечать на другой вопрос: если НКА может находиться в одном из нескольких состояний и прочтет определенный символ, то в каких состояниях он сможет оказаться после этого? Реализация выглядит следующим образом:

```
require 'set'

class NFARulebook < Struct.new(:rules)
  def next_states(states, character)
    states.flat_map { |state| follow_rules_for(state, character) }.to_set
  end

  def follow_rules_for(state, character)
    rules_for(state, character).map(&:follow)
  end

  def rules_for(state, character)
    rules.select { |rule| rule.applies_to?(state, character) }
  end
end
```

Здесь мы пользуемся классом `Set` из стандартной библиотеки Ruby для хранения множества возможных состояний, возвращенных методом `#next_states`. Можно было бы воспользоваться классом `Array`, но у `Set` есть три полезных свойства:

1. Он автоматически удаляет дубликаты. `Set[1, 2, 2, 3, 3, 3]` – то же самое, что `Set[1, 2, 3]`.
2. Он игнорирует порядок элементов. `Set[3, 2, 1]` – то же самое, что `Set[1, 2, 3]`.
3. Он предоставляет стандартные операции над множествами: пересечение (`#&`), объединение (`#+`) и проверку того, является одно множество подмножеством другого (`#subset?`).

Первое свойство полезно, потому что утверждение «НКА находится в состоянии 3 или в состоянии 3» бессмысленно, а возврат экземпляра `Set` гарантирует отсутствие дубликатов. В чем состоит полезность двух других свойств, выяснится позже.

Мы можем создать свод правил для недетерминированного конечного автомата и задать ему вопросы:

```
>> rulebook = NFARulebook.new([
     FARule.new(1, 'a', 1), FARule.new(1, 'b', 1), FARule.new(1, 'b', 2),
     FARule.new(2, 'a', 3), FARule.new(2, 'b', 3),
     FARule.new(3, 'a', 4), FARule.new(3, 'b', 4)
   ])
=> #<struct NFARulebook rules=[…]>
>> rulebook.next_states(Set[1], 'b')
=> #<Set: {1, 2}>
>> rulebook.next_states(Set[1, 2], 'a')
=> #<Set: {1, 3}>
>> rulebook.next_states(Set[1, 3], 'b')
=> #<Set: {1, 2, 4}>
```

Далее мы должны реализовать класс NFA для представления моделируемого автомата:

```
class NFA < Struct.new(:current_states, :accept_states, :rulebook)
  def accepting?
    (current_states & accept_states).any?
  end
end
```

Метод NFA#accepting? проверяет, существует ли хотя бы одно состояние в пересечении множеств current_states и accept_states, иными словами, является ли хотя бы одно из возможных текущих состояний также и заключительным.

Класс NFA очень похож на рассмотренный выше класс DFA. Разница в том, что в нем хранится множество возможных текущих состояний current_states, а не единственное состояние current_state, поэтому нахождение в заключительном состоянии означает, что хотя бы одно из состояний, входящих в current_states, является заключительным.

```
>> NFA.new(Set[1], [4], rulebook).accepting?
=> false
>> NFA.new(Set[1, 2, 4], [4], rulebook).accepting?
=> true
```

Как и в классе DFA, мы можем реализовать метод #read_character для чтения одного символа и метод #read_string для чтения цепочки символов:

```
class NFA
  def read_character(character)
    self.current_states = rulebook.next_states(current_states, character)
  end

  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end
end
```

Оба почти не отличаются от одноименных методов в классе `DFA`, только вместо `current_state` и `next_state` в методе `#read_character` мы пишем `current_states` и `next_states`.

На этом трудная часть закончена. Теперь можно запустить модель НКА, передать ей символы и спросить, допускается ли поступившая на вход цепочка:

```
>> nfa = NFA.new(Set[1], [4], rulebook); nfa.accepting?
=> false
>> nfa.read_character('b'); nfa.accepting?
=> false
>> nfa.read_character('a'); nfa.accepting?
=> false
>> nfa.read_character('b'); nfa.accepting?
=> true
>> nfa = NFA.new(Set[1], [4], rulebook)
=> #<struct NFA current_states=#<Set: {1}>, accept_states=[4], rulebook=…>
>> nfa.accepting?
=> false
>> nfa.read_string('bbbbb'); nfa.accepting?
=> true
```

Как и для класса `DFA`, удобно не создавать новые экземпляры `NFA` вручную, а воспользоваться объектом `NFADesign`, который изготавливает их автоматически:

```
class NFADesign < Struct.new(:start_state, :accept_states, :rulebook)
  def accepts?(string)
    to_nfa.tap { |nfa| nfa.read_string(string) }.accepting?
  end

  def to_nfa
    NFA.new(Set[start_state], accept_states, rulebook)
  end
end
```

Теперь проверять различные цепочки на одном и том же НКА стало проще:

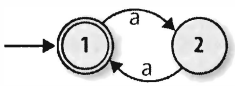
```
>> nfa_design = NFADesign.new(1, [4], rulebook)
=> #<struct NFADesign start_state=1, accept_states=[4], rulebook=…>
>> nfa_design.accepts?('bab')
=> true
>> nfa_design.accepts?('bbbbb')
=> true
>> nfa_design.accepts?('bbabb')
=> false
```

Вот, собственно, и все: мы построили простую реализацию необычной недетерминированной машины, смоделировав все возможные пути выполнения. Недетерминированность – удобный инструмент для проектирования изощренных конечных автоматов, поэтому весьма удачно, что НКА можно использовать на практике, а не только как теоретическую диковинку.

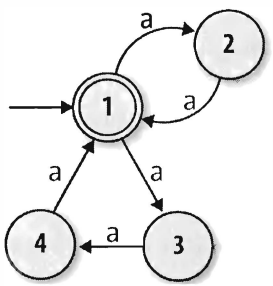
Свободные переходы

Мы видели, как ослабление ограничений детерминированности позволяет по-новому проектировать машины, не жертвуя возможностью реализовать их на практике. Что еще можно ослабить, чтобы получить еще больше свободы при проектировании?

Легко построить ДКА, который допускает только цепочки символов `a` четной длины: (`'aa'`, `'aaaa'`...):

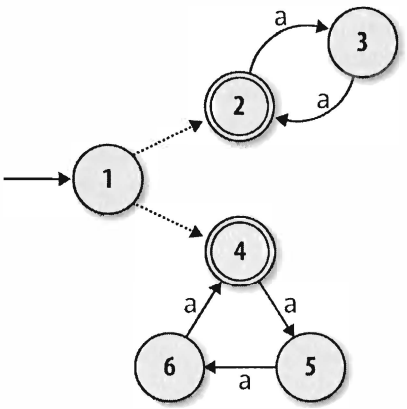


Но как построить автомат, допускающий цепочки, длина которых кратна *двум или трем*? Мы знаем, что недетерминированность наделяет машину возможностью выбора одного из нескольких путей выполнения, так, быть может, удастся построить НКА, у которого есть путь «кратна двум» и путь «кратна трем»? Наивная попытка могла бы выглядеть так:



Идея в том, чтобы заставить НКА переходить между состояниями 1 и 2 для допущения цепочек вида `'aa'` и `'aaaa'` и между состояниями 1, 3, 4 – для допущения цепочек вида `'aaa'` и `'aaaaaaaaa'`. С этим-то все хорошо, но беда в том, что такой автомат допускает также цепочку `'aaaaa'`, потому что может перейти из состояния 1 в состояние 2 и обратно в состояние 1, прочитав первые два символа, а затем, читая следующие три, посетить состояния 3, 4 и снова 1. И закончит он работу в заключительном состоянии, хотя длина цепочки не кратна ни 2, ни 3[1].

И на этот раз сразу не очевидно, может ли вообще НКА справиться с такой задачей, но мы можем решить проблему, наделив автомат еще одним свойством – *свободными переходами*. Это правила, которым автомат может следовать спонтанно, без чтения символов из входного потока. В данном случае они помогают, потому что в самом начале дают НКА выбор между двумя непересекающимися группами состояний:



Свободные переходы показаны пунктирными непомеченными стрелками, ведущими из состояния 1 в состояния 2 и 4. Этот автомат все еще может допустить цепочку `'aaaa'`, спонтанно перейдя в состояние 2, а затем переходя между состояниями 2 и 3 по мере чтения входных символов. Допускается и цепочка `'aaaaaaaaa'`, только для этого нужно сначала совершить свободный переход в состояние 4. Однако допустить цепочку `'aaaaa'` теперь невозможно: любое

[1] Этот НКА на самом деле допускает любую цепочку символов, кроме состоящей из одного символа 'a'.

возможное выполнение должно начинаться со свободного перехода в состояние 2 или 4, и обратного пути уже нет. Оказавшись в состоянии 2, автомат может допустить только цепочку с длиной, кратной 2, а оказавшись в состоянии 4 – цепочку с длиной, кратной 3.

Как поддержать свободные переходы в нашей модели НКА на Ruby? На самом деле, выбор между тремя вариантами – остаться в состоянии 1, спонтанно перейти в состояние 2 или спонтанно перейти в состояние 3 – ничем не хуже уже имеющейся недетерминированности, и обработать его можно аналогично. Мы уже реализовали идею о том, что моделируемый автомат может иметь сразу много возможных состояний, так теперь остается только расширить множество возможных состояний, включив в него те, что достижимы за один или несколько свободных переходов. В таком случае, автомат, начинающий работу в состоянии 1, может до чтения первого символа оказаться в любом из состояний 1, 2 или 4.

Сначала нужно придумать, как представить свободные переходы в Ruby. Проще всего взять обычные экземпляры класса `FARule` с `nil` вместо входного символа. Текущая реализация `NFARulebook` будет трактовать `nil`, как любой другой символ, поэтому мы можем задать вопрос «какие состояния достижимы из состояния 1 за один свободный переход?» (вместо «... в результате чтения одного символа?»).

```
>> rulebook = NFARulebook.new([
       FARule.new(1, nil, 2), FARule.new(1, nil, 4),
       FARule.new(2, 'a', 3),
       FARule.new(3, 'a', 2),
       FARule.new(4, 'a', 5),
       FARule.new(5, 'a', 6),
       FARule.new(6, 'a', 4)
    ])
=> #<struct NFARulebook rules=[…]>
>> rulebook.next_states(Set[1], nil)
=> #<Set: {2, 4}>
```

Далее нам нужен вспомогательный код для поиска всех состояний, достижимых в результате свободных переходов из заданного множества состояний. Этот код должен будет повторно совершать свободные переходы, потому что НКА может спонтанно изменять состояние столько раз, сколько захочет, пока имеются свободные переходы из текущего состояния. Поместить соответствующий метод удобно в класс `NFARulebook`:

```
class NFARulebook
  def follow_free_moves(states)
```

```
    more_states = next_states(states, nil)
    if more_states.subset?(states)
      states
    else
      follow_free_moves(states + more_states)
    end
  end
end
```

Метод `NFARulebook#follow_free_moves` рекурсивно ищет все новые и новые состояния, которых можно достичь из заданного множества состояний, совершая свободные переходы. Когда больше ничего найти не удается – потому что любое состояние, найденное методом `next_states(states, nil)`, уже находится в множестве `states` – он возвращает все состояния, которые сумел найти[1].

Следующий код корректно находит все состояния, в которых может оказаться наш НКА до чтения первого символа:

```
>> rulebook.follow_free_moves(Set[1])
=> #<Set: {1, 2, 4}>
```

Теперь можно включить поддержку свободных переходов в класс `NFA`, переопределив существующую реализацию метода `NFA#current_states` (предоставляемого классом `Struct`). Новая реализация просто вызывает метод `NFARulebook#follow_free_moves`, гарантируя, что множество возможных текущих состояний автомата обязательно включает все состояния, достижимые с помощью свободных переходов:

```
class NFA
  def current_states
    rulebook.follow_free_moves(super)
  end
end
```

Все остальные методы класса `NFA` получают доступ к возможным текущим состояниям, вызывая метод `#current_states`, и, следовательно, для поддержки свободных переходов больше ничего менять в этом классе не нужно.

Вот так. Теперь модель поддерживает свободные переходы, и мы можем посмотреть, какие цепочки допускает наш НКА:

---

[1] Строго говоря, в результате этой процедуры вычисляется неподвижная точка функции «добавить еще состояния, достижимые в результате свободных переходов».

```
>> nfa_design = NFADesign.new(1, [2, 4], rulebook)
=> #<struct NFADesign …>
>> nfa_design.accepts?('aa')
=> true
>> nfa_design.accepts?('aaa')
=> true
>> nfa_design.accepts?('aaaa')
=> false
>> nfa_design.accepts?('aaaaa')
=> true
```

Таким образом, реализовать свободные переходы оказалось совсем нетрудно, и мы получили дополнительную свободу дизайна сверх той, что уже предоставляет недетерминированность.

Терминология, употребляемая в этой главе, не всегда традиционна. В частности, набор правил, составляющих конечный автомат, называется *функцией переходов* (для НКА иногда – *отношением переходов*). Поскольку в математике пустая цепочка обозначается греческой буквой ε (эпсилон), то НКА со свободными переходами называется НКА-ε, а сами свободные переходы – ε-*переходами*.

Регулярные выражения

Мы видели, что недетерминированность и свободные переходы позволяют сделать конечные автоматы более выразительными без ущерба для возможности моделирования. В этом разделе мы рассмотрим одно практически важное применение этого аппарата: сопоставление с регулярными выражениями.

Регулярные выражения образуют язык для записи текстовых *образцов*, с которыми можно сопоставлять строки. Приведем несколько примеров:

- регулярное выражение `hello` сопоставляется только со строкой `'hello'`;
- регулярное выражение `hello|goodbye` сопоставляется со строками `'hello'` и `'goodbye'`;
- регулярное выражение `(hello)*` сопоставляется со строками `'hello'`, `'hellohello'`, `'hellohellohello'` и т. д., а также с пустой строкой.

В этой главе мы говорим только о сопоставлении регулярного выражения со *всей* строкой. В конкретных реализациях регулярных выражений обычно допускается сопоставление с *частью*

строки, а для сопоставления со всей строкой необходим специальный синтаксис.

Например, на Ruby наше регулярное выражение `hello|goodbye` следовало бы записать в виде `/\A(hello|goodbye)\z/`, тогда соответствие было бы привязано к началу (`\A`) и к концу (`\z`) строки.

Если даны регулярное выражение и строка, то как написать программу для определения того, соответствует строка выражению или нет? В большинство языков программирования, включая и Ruby, поддержка регулярных выражений уже встроена, но как она работает? Как мы могли бы реализовать регулярные выражения в Ruby, если бы язык их не поддерживал?

Оказывается, конечные автоматы отлично подходят для этой цели. Как мы увидим, любому регулярному выражению можно поставить в соответствие эквивалентный НКА – всякая строка, соответствующая данному регулярному выражению, допускается НКА и наоборот – а затем для установления соответствия строки регулярному выражению подать эту строку на вход модели НКА и посмотреть, допущена она или нет. В терминах главы 2 можно сказать, что мы предлагаем некую денотационную семантику регулярных выражений: возможно, мы не знаем, как выполнить регулярное выражение напрямую, зато умеем денотировать ее в виде НКА, а поскольку для НКА у нас есть операционная семантика («изменять состояние путем чтения символов и следования правилам»), то для получения нужного результата достаточно выполнить денотацию.

Синтаксис

Давайте точно определим, что понимается под «регулярным выражением». В качестве отправной точки возьмем два предельно простых регулярных выражения, которые уже невозможно упростить далее.

- Пустое регулярное выражение. Ему соответствует пустая строка и ничего больше.
- Регулярное выражение, состоящее из одного литерального символа. Например, `a` и `b` – регулярные выражения, которым соответствуют только строки `'a'` и `'b'` соответственно.

Из этих двух простых образцов можно тремя способами строить более сложные выражения.

- Конкатенация двух образцов. Конкатенировав регулярные выражения `a` и `b`, мы получаем регулярное выражение `ab`, которому соответствует только строка `'ab'`.

- Выбор одного из двух образцов, соединенных оператором `|`. Путем соединения регулярных выражений `a` и `b`, мы получаем регулярное выражение `a|b`, которому соответствуют строки `'a'` и `'b'`.
- Повторение образца нуль или более раз. Поместив после регулярного выражения `a` оператор `*`, мы получаем выражение `a*`, которому соответствуют строки `'a'`, `'aa'`, `'aaa'` и т. д., а также пустая строка (нуль повторений).

Практические реализации регулярных выражений, в том числе в Ruby, поддерживают дополнительные возможности, многие из которых, строго говоря, избыточны и предоставлены только для удобства. Простоты ради мы не будем пытаться их реализовать.

Например, отказавшись от операторов `?` и `+`, мы не теряем ничего существенного, потому что их действие – «повторить нуль или один раз» и «повторить один или более раз» соответственно – легко реализовать с помощью того, что у нас уже есть: регулярное выражение `ab?` можно переписать в виде `ab|a`, а образцу `ab+` соответствуют те же строки, что образцу `abb*`. Это относится и к другим вспомогательным возможностям, в частности к повторению со счетчиком (например, `a{2,5}`) и к классам символов (например, `[abc]`).

Более сложные средства, в том числе запоминаемые группы, обратные ссылки и утверждения с заглядыванием вперед или оглядыванием назад, выходят за рамки этой главы.

Чтобы реализовать такой синтаксис на Ruby, мы можем определить классы для каждого вида регулярных выражений и с помощью экземпляров этих классов представить абстрактное синтаксическое дерево произвольного регулярного выражения, как делали это для выражений языка SIMPLE в главе 2.

```
module Pattern
  def bracket(outer_precedence)
    if precedence < outer_precedence
      '(' + to_s + ')'
    else
      to_s
    end
  end

  def inspect
    "/#{self}/"
  end
end

class Empty
```

```
  include Pattern

  def to_s
    ''
  end

  def precedence
    3
  end
end

class Literal < Struct.new(:character)
  include Pattern

  def to_s
    character
  end

  def precedence
    3
  end
end

class Concatenate < Struct.new(:first, :second)
  include Pattern

  def to_s
    [first,second].map { |pattern| pattern.bracket(precedence) }.join
  end

  def precedence
    1
  end
end

class Choose < Struct.new(:first, :second)
  include Pattern

  def to_s
    [first,second].map { |pattern| pattern.bracket(precedence) }.join('|')
  end

  def precedence
    0
  end
end

class Repeat < Struct.new(:pattern)
  include Pattern

  def to_s
    pattern.bracket(precedence) + '*'
  end

  def precedence
    2
  end
end
```

В арифметических выражениях приоритет умножения выше, чем сложения ($1 + 2 \times 3$ равно 7, а не 9), и точно так же для регулярных выражений действует соглашение о том, что приоритет оператора * выше, чем у конкатенации, а у той – выше, чем у оператора |. Например, в регулярном выражении abc* звездочка * относится к c (`'abc'`, `'abcc'`, `'abccc'`...), а чтобы применить ее ко всему выражению abc (`'abc'`, `'abcabc'`...), нужно добавить скобки, то есть написать (abc)*.

Реализации метода `#to_s` в синтаксических классах в сочетании с методом `Pattern#bracket` автоматически вставляют скобки в нужные места, чтобы можно было видеть простое строковое представление абстрактного синтаксического дерева без потери информации о его структуре.

С помощью этих классов мы можем вручную строить деревья, представляющие регулярные выражения:

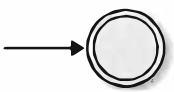
```
>> pattern =
       Repeat.new(
         Choose.new(
           Concatenate.new(Literal.new('a'), Literal.new('b')),
           Literal.new('a')
         )
       )
=> /(ab|a)*/
```

Разумеется, в настоящей реализации мы воспользовались бы для этой цели синтаксическим анализатором; указания на этот счет см. в разделе «Синтаксический анализ» на стр. 122.

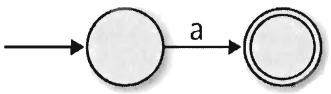
Семантика

Итак, представлять синтаксис регулярного выражения в виде дерева объектов Ruby мы научились, остается понять, как из этого дерева получить НКА.

Нам нужно решить, как преобразовывать в НКА экземпляры каждого синтаксического класса. Самым простым является класс `Empty`, который преобразуется в НКА с одним состоянием, допускающий только пустую цепочку:



Аналогично образец с единственным литеральным символом преобразуется в НКА, который допускает только цепочку, состоящую из одного этого символа. Вот как выглядит НКА, соответствующий образцу a:



Не слишком сложно реализовать в классах `Empty` и `Literal` методы `#to_nfa_design`, порождающие такие НКА:

```
class Empty
  def to_nfa_design
    start_state = Object.new
    accept_states = [start_state]
    rulebook = NFARulebook.new([])

    NFADesign.new(start_state, accept_states, rulebook)
  end
end

class Literal
  def to_nfa_design
    start_state = Object.new
    accept_state = Object.new
    rule = FARule.new(start_state, character, accept_state)
    rulebook = NFARulebook.new([rule])

    NFADesign.new(start_state, [accept_state], rulebook)
  end
end
```

Как уже отмечалось в разделе «Моделирование» на стр. 92, состояния автомата следует реализовывать в виде объектов Ruby, которые можно отличить друг от друга. Здесь мы представляем состояния не числами (то есть экземплярами класса `Fixnum`), а уникальными экземплярами класса `Object`.

Так сделано для того, чтобы у каждого НКА были собственные уникальные состояния, что даст нам возможность объединять маленькие автоматы в более крупные, не опасаясь случайного совпадения состояний. Например, если бы в двух разных НКА в качестве состояния использовался один и тот же объект `Fixnum` со значением 1, то их нельзя было бы соединить, сохранив уникальность состояний. Однако нам такая возможность необходима для составления сложных регулярных выражений из более простых.

По той же причине мы не помечаем состояния числами на диаграммах, чтобы не пришлось их перенумеровывать при соединении диаграмм.

Проверим, что НКА, порожденные из регулярных выражений `Empty` и `Literal`, действительно допускают ожидаемые цепочки:

```
>> nfa_design = Empty.new.to_nfa_design
=> #<struct NFADesign …>
>> nfa_design.accepts?('')
=> true
>> nfa_design.accepts?('a')
=> false
>> nfa_design = Literal.new('a').to_nfa_design
=> #<struct NFADesign …>
>> nfa_design.accepts?('')
=> false
>> nfa_design.accepts?('a')
=> true
>> nfa_design.accepts?('b')
=> false
```

Мы можем обернуть метод `#to_nfa_design` методом `#matches?`, получив при этом более изящный интерфейс:

```
module Pattern
  def matches?(string)
    to_nfa_design.accepts?(string)
  end
end
```

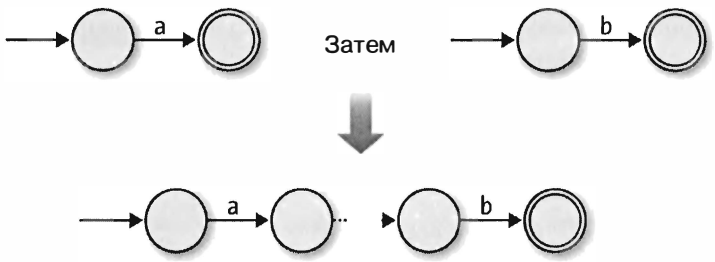
Это позволяет сопоставлять образцы со строками непосредственно:

```
>> Empty.new.matches?('a')
=> false
>> Literal.new('a').matches?('a')
=> true
```

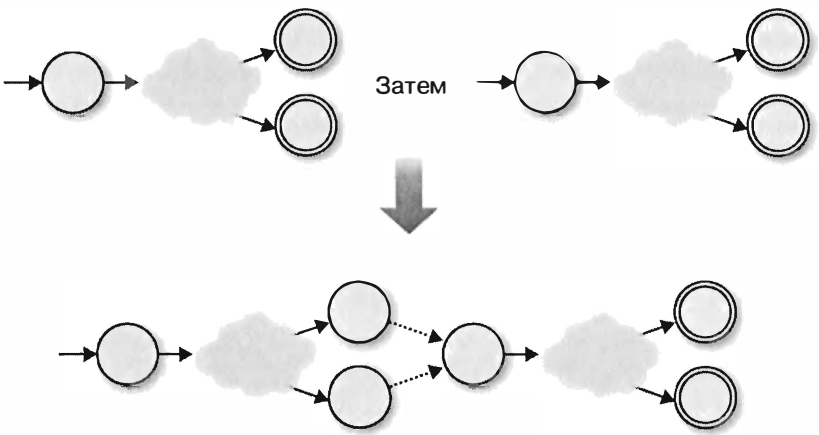
Теперь, зная, как преобразовать простые регулярные выражения `Empty` и `Literal` в НКА, мы должны сделать нечто подобное для классов `Concatenate`, `Choose` и `Repeat`.

Начнем с `Concatenate`: если есть два регулярных выражения, для которых НКА уже известны, то как построить НКА, представляющий их конкатенацию? Например, имея НКА для односимвольных регулярных выражений `a` и `b`, как получить НКА для выражения `ab`?

В случае выражения ab мы можем соединить оба НКА последовательно, связав их свободным переходом и сохранив только заключительное состояние второго НКА:



Эта техника работает и в других случаях. Любые два НКА можно конкатенировать, сделав все заключительные состояния первого НКА обычными и присоединив их к начальному состоянию второго НКА свободными переходами. После того как конкатенированный автомат прочитает входную цепочку, которая перевела бы первый НКА в заключительное состояние, он сможет спонтанно перейти в состояние, соответствующее начальному состоянию второго НКА, а затем достичь заключительного состояния, прочитав входную цепочку, которую этот второй НКА допустил бы.



Таким образом, результирующий автомат состоит из следующих частей:

- начальное состояние первого НКА;
- заключительные состояния второго НКА;
- все правила обоих НКА;
- дополнительные свободные переходы, соединяющие бывшие заключительные состояния первого НКА с бывшим начальным состоянием второго НКА.

Эту идею можно воплотить в реализацию метода `Concatenate#to_nfa_design`:

```
class Concatenate
  def to_nfa_design
    first_nfa_design = first.to_nfa_design
    second_nfa_design = second.to_nfa_design

    start_state = first_nfa_design.start_state
    accept_states = second_nfa_design.accept_states
    rules = first_nfa_design.rulebook.rules + second_nfa_design.rulebook.rules
    extra_rules = first_nfa_design.accept_states.map { |state|
      FARule.new(state, nil, second_nfa_design.start_state)
    }
    rulebook = NFARulebook.new(rules + extra_rules)

    NFADesign.new(start_state, accept_states, rulebook)
  end
end
```

Здесь мы сначала преобразуем первое и второе регулярное выражения в объекты `NFADesign`, а затем объединяем их состояния и правила указанным выше образом для получения нового объекта `NFADesign`. Все работает, как и в случае простого выражения `ab`:

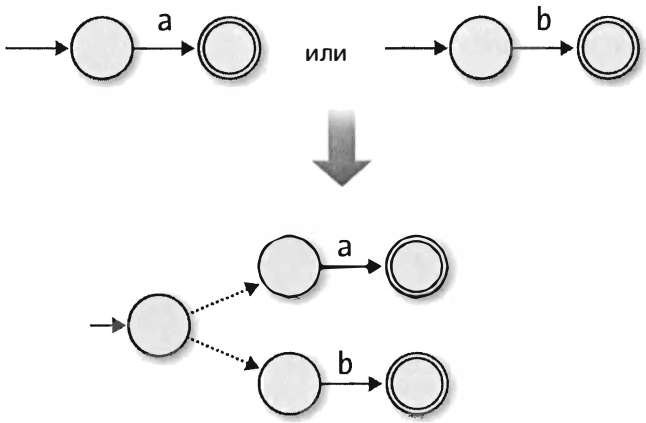
```
>> pattern = Concatenate.new(Literal.new('a'), Literal.new('b'))
=> /ab/
>> pattern.matches?('a')
=> false
>> pattern.matches?('ab')
=> true
>> pattern.matches?('abc')
=> false
```

Процедура преобразования рекурсивна – `Concatenate#to_nfa_design` вызывает метод `#to_nfa_design` других объектов – поэтому она будет работать и для регулярных выражений с большим уровнем вложенности, например для выражения `abc`, содержащего две операции конкатенации (`a` конкатенируется с `b`, затем результат конкатенируется с `c`):

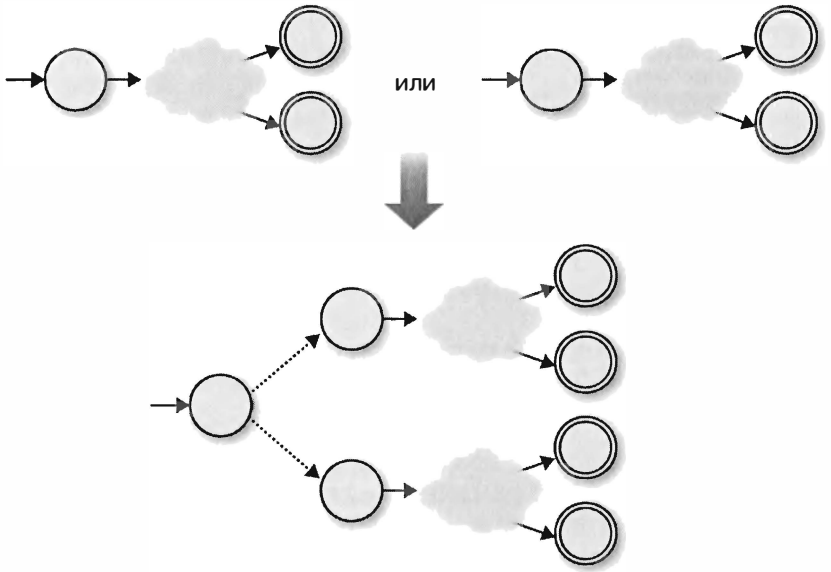
```
>> pattern =
       Concatenate.new(
         Literal.new('a'),
         Concatenate.new(Literal.new('b'), Literal.new('c'))
       )
=> /abc/
>> pattern.matches?('a')
=> false
>> pattern.matches?('ab')
=> false
>> pattern.matches?('abc')
=> true
```

Это еще один пример *композиционности* денотационной семантики: НКА, денотирующий составное регулярное выражение, состоит из денотаций его частей.

Аналогичную стратегию можно использовать для преобразования объекта `Choose` в НКА. В простейшем случае НКА регулярных выражений `a` и `b` можно объединить в НКА регулярного выражения `a|b`, добавив новое начальное состояние и соединив его свободными переходами с бывшими начальными состояниями исходных автоматов:



Перед тем как НКА, представляющий выражение `a|b`, начнет читать входные символы, он может спонтанно перейти в начальное состояние любого из двух исходных автоматов, а уже оттуда прочитать `'a'` или `'b'` для достижения заключительного состояния. Так же просто «склеить» два произвольных автомата – нужно лишь добавить новое начальное состояние и два свободных перехода:



В этом случае результирующий автомат состоит из следующих частей:

- новое начальное состояние;
- все заключительные состояния обоих НКА;
- все правила обоих НКА;
- два дополнительных свободных перехода, соединяющих новое начальное состояние с бывшими начальными состояниями обоих исходных НКА.

Все это без труда реализуется в методе `Choose#to_nfa_design`:

```
class Choose
  def to_nfa_design
    first_nfa_design = first.to_nfa_design
    second_nfa_design = second.to_nfa_design

    start_state = Object.new
    accept_states = first_nfa_design.accept_states + second_nfa_design.accept_states
    rules = first_nfa_design.rulebook.rules + second_nfa_design.rulebook.rules
    extra_rules = [first_nfa_design, second_nfa_design].map { |nfa_design|
      FARule.new(start_state, nil, nfa_design.start_state)
    }
    rulebook = NFARulebook.new(rules + extra_rules)

    NFADesign.new(start_state, accept_states, rulebook)
  end
end
```

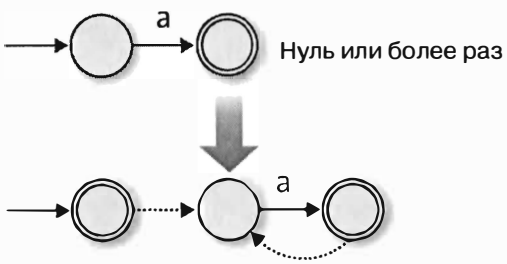
И прекрасно работает:

```
>> pattern = Choose.new(Literal.new('a'), Literal.new('b'))
=> /a|b/
>> pattern.matches?('a')
=> true
>> pattern.matches?('b')
=> true
>> pattern.matches?('c')
=> false
```

И наконец, повторение: как из НКА, который сопоставляется со строкой ровно один раз, получить НКА, сопоставляемый с той же строкой, повторенной нуль или более раз? Чтобы построить НКА для регулярного выражения `a*`, мы начнем с НКА для выражения `a` и добавим две вещи:

- свободный переход из заключительного состояния в начальное, чтобы можно было произвести сопоставление со строкой из нескольких символов `'a'`;
- новое заключительное начальное состояние со свободным переходом в бывшее начальное состояние, чтобы можно было провести сопоставление с пустой строкой.

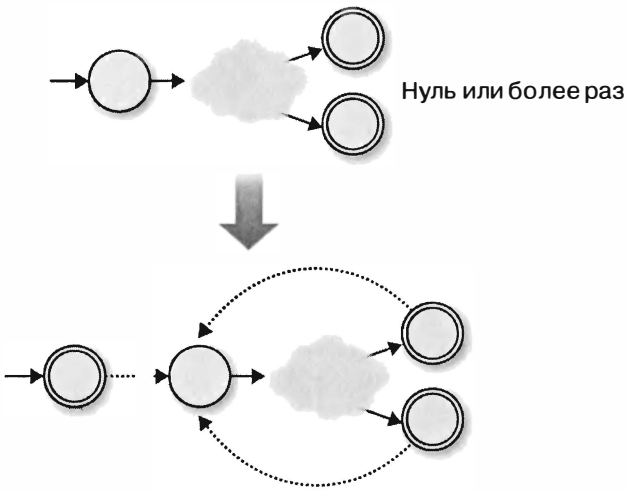
Вот как это выглядит:



Свободный переход из бывшего заключительного состояния в бывшее начальное состояние позволяет автомату проводить сопоставление несколько раз, а не только один (`'aa'`, `'aaa'` и т. д.), а новое начальное состояние – допускать пустую цепочку вне зависимости от того, какие еще цепочки допускаются[1]. То же самое

[1] В этом простом случае можно было бы обойтись одним лишь преобразованием исходного начального состояния в заключительное, а не добавлять новое, но в более сложных случаях (например, `(a*b)*`) такой подход мог бы породить автомат, допускающий нежелательные цепочки в дополнение к пустой.

можно проделать для любого НКА – нужно лишь соединить все бывшие заключительные состояния с бывшим начальным состоянием свободными переходами:



На этот раз нам понадобятся:

- новое начальное состояние, одновременно являющееся заключительным;
- все заключительные состояния исходного НКА;
- все правила исходного НКА;
- дополнительные свободные переходы, которые соединят заключительные состояния исходного НКА с его бывшим начальным состоянием;
- еще один свободный переход, который соединит новое начальное состояние с бывшим начальным состоянием.

Превратим это в код:

```
class Repeat
  def to_nfa_design
    pattern_nfa_design = pattern.to_nfa_design
    start_state = Object.new
    accept_states = pattern_nfa_design.accept_states + [start_state]
    rules = pattern_nfa_design.rulebook.rules
    extra_rules =
      pattern_nfa_design.accept_states.map { |accept_state|
        FARule.new(accept_state, nil, pattern_nfa_design.start_state)
      } +
```

```
      [FARule.new(start_state, nil, pattern_nfa_design.start_state)]
    rulebook = NFARulebook.new(rules + extra_rules)

    NFADesign.new(start_state, accept_states, rulebook)
  end
end
```

И проверим, что все работает:

```
>> pattern = Repeat.new(Literal.new('a'))
=> /a*/
>> pattern.matches?('')
=> true
>> pattern.matches?('a')
=> true
>> pattern.matches?('aaaa')
=> true
>> pattern.matches?('b')
=> false
```

Теперь, имея реализации метода `#to_nfa_design` во всех классах, описывающих синтаксис регулярных выражений, мы можем строить сложные образцы и сопоставлять с ними строки:

```
>> pattern =
     Repeat.new(
       Concatenate.new(
         Literal.new('a'),
         Choose.new(Empty.new, Literal.new('b'))
       )
     )
=> /(a(|b))*/
>> pattern.matches?('')
=> true
>> pattern.matches?('a')
=> true
>> pattern.matches?('ab')
=> true
>> pattern.matches?('aba')
=> true
>> pattern.matches?('abab')
=> true
>> pattern.matches?('abaab')
=> true
>> pattern.matches?('abba')
=> false
```

Отличный результат. Начав с синтаксиса образцов, мы теперь придали этому синтаксису семантику, показав, как сопоставить любому образцу НКА – вид абстрактной машины, для которой мы уже знаем правила выполнения. В сочетании с синтаксическим анали-

затором это дает нам практически применимый способ прочитать регулярное выражение и решить, соответствует ли ему заданная строка. Для преобразования регулярных выражений в НКА полезны оказались свободные переходы, поскольку они позволяют «склеивать» из простых автоматов более сложные, не нарушая поведение исходных компонентов.

Большинство существующих реализаций регулярных выражений, в частности библиотека Onigmo, применяемая в Ruby, не компилируют буквально образцы в конечные автоматы с последующим моделированием их выполнения. Хотя такой подход и дает быстрый и эффективный способ сопоставить регулярное выражение со строкой, он малопригоден для поддержки более сложных функций, в частности запоминаемых групп и утверждений с заглядыванием вперед или оглядыванием назад. Поэтому в большинстве библиотек используется тот или иной вариант *алгоритма с возвратом*, который работает с регулярными выражениями напрямую без преобразования их в конечные автоматы.

Библиотека RE2 (*https://code.google.com/p/re2/*), написанная Рассом Коксом (Russ Cox), – высококачественная реализация регулярных выражений на C++, в которой образцы таки компилируются в автоматы[1], а Пат Шогнесси (Pat Shaughnessy) в своем блоге подробно описал, как устроен алгоритм работы с регулярными выражениями в Ruby (*http://patshaughnessy.net/2012/4/3/exploring-rubys-regular-expression-algorithm*).

Синтаксический анализ

Мы почти завершили полную (хотя и простую) реализацию регулярных выражений. Не хватает только синтаксического анализатора для языка образцов: было бы гораздо удобнее написать просто `(a(|b))*`, чем строить абстрактное синтаксическое дерево вручную: `Repeat.new(Concatenate.new(Literal.new('a'), Choose.new(Empty.new, Literal.new ('b'))))`. В разделе «Реализация синтаксических анализаторов» на стр. 82 мы видели, что совсем нетрудно с помощью библиотеки Treetop сгенерировать анализатор, который умеет автоматически строить АСД по исходному коду; сделаем это, чтобы довести нашу реализацию до конца.

---

[1] В описании RE2 говорится, что это «эффективная, базирующаяся на твердых принципах библиотека регулярных выражений». С этим трудно спорить.

Ниже приведена грамматика простых регулярных выражений в нотации Treetop:

```
grammar Pattern
  rule choose
    first:concatenate_or_empty '|' rest:choose {
      def to_ast
        Choose.new(first.to_ast, rest.to_ast)
      end
    }
    /
    concatenate_or_empty
  end

  rule concatenate_or_empty
    concatenate / empty
  end

  rule concatenate
    first:repeat rest:concatenate {
      def to_ast
        Concatenate.new(first.to_ast, rest.to_ast)
      end
    }
    /
    repeat
  end

  rule empty
    '' {
      def to_ast
        Empty.new
      end
    }
  end

  rule repeat
    brackets '*' {
      def to_ast
        Repeat.new(brackets.to_ast)
      end
    }
    /
    brackets
  end

  rule brackets
    '(' choose ')' {
      def to_ast
        choose.to_ast
      end
    }
    /
    literal
  end

  rule literal
```

```
    [a-z] {
      def to_ast
        Literal.new(text_value)
      end
    }
  end
end
```

Как и раньше, порядок следования правил отражает приоритеты операторов: у оператора | самый низкий приоритет, поэтому правило choose поставлено первым; чем выше приоритет оператора, тем дальше от начала грамматики он находится.

Теперь у нас есть все необходимое, чтобы разобрать регулярное выражение, построить для него абстрактное синтаксическое дерево и воспользоваться им для сопоставления со строками:

```
>> require 'treetop'
=> true
>> Treetop.load('pattern')
=> PatternParser
>> parse_tree = PatternParser.new.parse('(a(|b))*')
=> SyntaxNode+Repeat1+Repeat0 offset=0, "(a(|b))*" (to_ast,brackets):
     SyntaxNode+Brackets1+Brackets0 offset=0, "(a(|b))" (to_ast,choose):
       SyntaxNode offset=0, "("
       SyntaxNode+Concatenate1+Concatenate0 offset=1, "a(|b)" (to_ast,first,rest):
         SyntaxNode+Literal0 offset=1, "a" (to_ast)
         SyntaxNode+Brackets1+Brackets0 offset=2, "(|b)" (to_ast,choose):
           SyntaxNode offset=2, "("
           SyntaxNode+Choose1+Choose0 offset=3, "|b" (to_ast,first,rest):
             SyntaxNode+Empty0 offset=3, "" (to_ast)
             SyntaxNode offset=3, "|"
             SyntaxNode+Literal0 offset=4, "b" (to_ast)
           SyntaxNode offset=5, ")"
       SyntaxNode offset=6, ")"
     SyntaxNode offset=7, "*"
>> pattern = parse_tree.to_ast
=> /(a(|b))*/
>> pattern.matches?('abaab')
=> true
>> pattern.matches?('abba')
=> false
```

Эквивалентность

В этой главе была представлена идея детерминированного конечного автомата, к которому мы затем добавили ряд дополнительных свойств: сначала недетерминированность, в результате чего мы смогли проектировать машины, имеющие несколько возможных путей выполнения, а не один-единственный, а затем свободные переходы,

позволяющие недетерминированному автомату изменять состояние, не читая входной поток.

Благодаря недетерминированности и свободным переходам стало проще проектировать конечные автоматы для решения конкретных задач – мы видели, насколько они полезны для преобразования регулярных выражений в конечные автоматы, – но действительно ли они позволяют делать нечто такое, для чего стандартного ДКА недостаточно?

На самом деле, оказывается, что любой недетерминированный конечный автомат можно преобразовать в детерминированный, который будет допускать в точности те же цепочки. Учитывая ограничения ДКА, это может показаться удивительным, однако становится понятным, если задуматься о том, каким образом мы моделировали выполнение обоих видов автоматов.

Допустим, у нас имеется ДКА, поведение которого мы хотим смоделировать. Модель этого гипотетического ДКА, читающего некоторую цепочку символов, может выполнить, к примеру, следующие действия.

- До начала чтения входного потока автомат находится в состоянии 1.
- Автомат читает символ `'a'` и переходит в состояние 2.
- Автомат читает символ `'b'` и переходит в состояние 3.
- Больше символов нет, а состояние 3 является заключительным, поэтому цепочка `'ab'` допускается.

Здесь имеется тонкий нюанс: модель, которая в нашем случае является Ruby-программой, исполняемой на реальном компьютере, воссоздает поведение ДКА – абстрактной машины, которая вообще не может исполняться, потому что не существует в природе. Всякий раз, как воображаемый ДКА изменяет состояние, то же самое происходит с исполняемой моделью – именно в этом и состоит существо моделирования.

Это отличие трудно уловить, потому что ДКА и модель детерминированные и их состояния в точности соответствуют друг другу: если ДКА находится в состоянии 2, то модель находится в состоянии, которое означает, что «ДКА находится в состоянии 2». В нашей модели на Ruby это *состояние модели* по существу является значением атрибута `current_state` экземпляра класса `DFA`.

Несмотря на дополнительные осложнения, связанные с недетерминированностью и свободными переходами, модель гипотетического НКА, читающего символы, отличается не так уж сильно.

- До того как автомат прочел первый входной символ, он может находиться в состоянии 1 или 3[1].
- Автомат читает символ `c` и теперь может оказаться в любом из состояний 1, 3 или 4.
- Автомат читает символ `d` и теперь может оказаться в состоянии 2 или 5.
- Больше символов нет, а состояние 5 является заключительным, поэтому цепочка `'cd'` допускается.

Теперь легче увидеть, что состояние модели – не то же самое, что состояние НКА. На самом деле, ни в одной точке моделирования мы не можем быть уверены, в каком состоянии находится НКА, однако сама модель тем не менее детерминирована, потому что в ее состояниях эта неопределенность учтена. Хотя НКА *может* находиться в любом из состояний 1, 3 или 4, мы знаем *наверняка*, что модель находится в одном определенном состоянии, которое означает «НКА находится в состоянии 1, 3 или 4».

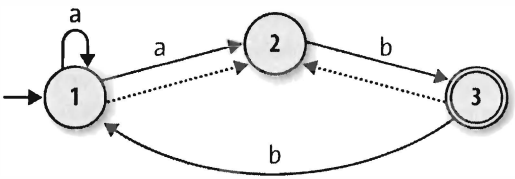
Единственное существенное различие между этими примерами заключается в том, что модель ДКА переходит из одного текущего состояния в другое, тогда как модель НКА переходит из текущего *множества возможных состояний* в другое множество. Хотя свод правил НКА может быть недетерминированным, решение о том, какие возможные состояния достижимы из текущего множества состояний для данного входного символа, полностью детерминировано.

Эта детерминированность означает, что мы всегда можем построить ДКА, который будет моделировать заданный НКА. У такого ДКА должно быть по одному состоянию для каждого множества возможных состояний НКА, а правила перехода между этими состояниями будут соответствовать способам перехода между множествами возможных состояний в детерминированной модели НКА. Получившийся ДКА сможет полностью моделировать поведение НКА и при условии правильного выбора заключительных состояний ДКА – в случае нашей реализации на Ruby это будут состояния, которые соответствуют утверждению, что НКА *возможно* находится в заключительном состоянии, – он будет допускать в точности такие же цепочки.

Проделаем такое преобразование для показанного ниже НКА.

---

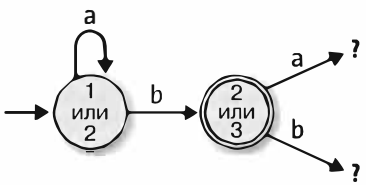
[1] У НКА есть только одно начальное состояние, но благодаря наличию свободных переходов он может оказаться в одном из нескольких состояний до начала чтения.



До начала чтения этот НКА может находиться в состоянии 1 или 2 (состояние 1 начальное, а состояние 2 достижимо из него с помощью свободного перехода), поэтому модель может начать работу в состоянии, которое мы назовем «1 или 2». Начиная с этого момента, модель может оказаться в различных состояниях в зависимости от того, прочтет она символ a или b.

- Если прочитан символ a, то модель остается в состоянии «1 или 2»: если НКА находится в состоянии 1, то он может прочитать a и выбрать либо правило, оставляющее его в состоянии 1, либо правило, переводящее его в состояние 2; если же он находится в состоянии 2, то прочитать символ a он вообще не может.
- Если прочитан символ b, то НКА может находиться в состоянии 2 или 3 – в состоянии 1 прочитать символ b вообще невозможно, а из состояния 2 можно перейти в состояние 3 и потенциально совершить свободный переход обратно в состояние 2. Поэтому мы будем говорить, что после чтения символа b модель переходит в состояние «2 или 3».

Размышляя о поведении модели НКА, мы на самом деле начали строить автомат для его моделирования:



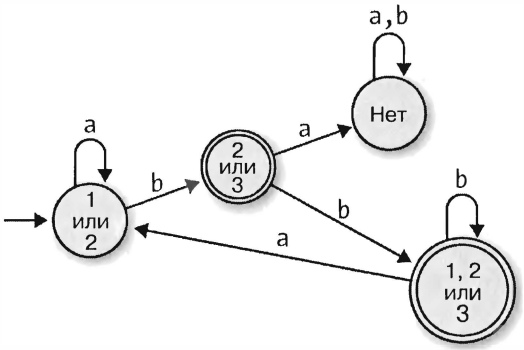
«2 или 3» – заключительное состояние модели, потому что 3 – заключительное состояние НКА.

Мы можем продолжать процедуру выявления состояний модели, пока на очередном шаге не окажется, что новых состояний нет, а это произойдет обязательно, потому что количество возможных

комбинаций состояний НКА конечно[1]. Повторяя это процедуру для рассматриваемого в примере НКА, мы обнаружим, что имеется всего четыре различных комбинации состояний, в которых может оказаться модель, начав работу в состоянии «1 или 2» и читая цепочки символов a и b.

Если НКА находится в состоянии(ях)...	и прочитан символ...	то он может оказаться в состоянии(ях)...
1 или 2	a b	1 или 2 2 или 3
2 или 3	a b	нет 1, 2 или 3
нет	a b	нет нет
1, 2 или 3	a b	1 или 2 1, 2 или 3

Эта таблица полностью описывает изображенный ниже ДКА, который допускает те же строки, что и исходный НКА.



У этого ДКА на одно состояние больше, чем у исходного НКА, а для некоторых НКА описанная процедура может даже порождать ДКА с *меньшим* числом состояний, чем у исходного автомата. Но в худшем случае НКА с n состояниями может соответствовать ДКА с 2^n состояниями, поскольку всего существует 2^n комбинаций n состояний (каждую комбинацию можно представить в виде n-битного числа, в котором i-ый бит равен 1, если состояние i входит в эту комбинацию), и, возможно, модели придется посетить их все.

---

[1] В худшем случае модель НКА с тремя состояниями может иметь состояния «1», «2», «3», «1 или 2», «1 или 3», «2 или 3», «1, 2 или 2».

Попробуем реализовать это преобразование НКА в ДКА на Ruby. Мы заведем новый класс `NFASimulation` для сбора всей информации о модели НКА, а затем на основе этой информации построим ДКА. Экземпляр `NFASimulation` будет создаваться по конкретному экземпляру `NFADesign` и предоставлять метод `#to_dfa_design` для преобразования в эквивалентный экземпляр `DFADesign`.

У нас уже есть класс `NFA`, который умеет моделировать НКА, поэтому класс `NFASimulation` может создавать экземпляры `NFA`, подавать им на вход все возможные входные символы и смотреть, как они реагируют. Но прежде чем приняться за `NFASimulation`, вернемся к классу `NFADesign` и добавим в метод `NFADesign#to_nfa` дополнительный параметр «текущие состояния», чтобы можно было построить экземпляр `NFA` с произвольным множеством текущих состояний, а не только с начальным состоянием, взятым из `NFADesign`:

```
class NFADesign
  def to_nfa(current_states = Set[start_state])
    NFA.new(current_states, accept_states, rulebook)
  end
end
```

Раньше модель НКА могла начать работу только в его начальном состоянии, а новый параметр позволяет начинать с любой точки:

```
>> rulebook = NFARulebook.new([
       FARule.new(1, 'a', 1), FARule.new(1, 'a', 2), FARule.new(1, nil, 2),
       FARule.new(2, 'b', 3),
       FARule.new(3, 'b', 1), FARule.new(3, nil, 2)
     ])
=> #<struct NFARulebook rules=[…]>
>> nfa_design = NFADesign.new(1, [3], rulebook)
=> #<struct NFADesign start_state=1, accept_states=[3], rulebook=…>
>> nfa_design.to_nfa.current_states
=> #<Set: {1, 2}>
>> nfa_design.to_nfa(Set[2]).current_states
=> #<Set: {2}>
>> nfa_design.to_nfa(Set[3]).current_states
=> #<Set: {3, 2}>
```

Класс `NFA` автоматически учитывает свободные переходы – мы видим, что если наш НКА запущен в состоянии 3, то до начала чтения он может оказаться в состоянии 2 или 3, – поэтому для их поддержки в классе `NFASimulation` специально ничего делать не нужно.

Теперь мы можем создать НКА с любым множеством возможных состояний, подать на вход символ и посмотреть, в каких состояниях он может оказаться, а это и есть самый важный шаг преобразования НКА в ДКА. Если наш НКА находится в состоянии 2 или 3 и читает символ b, то в каких состояниях он может оказаться?

```
>> nfa = nfa_design.to_nfa(Set[2, 3])
=> #<struct NFA current_states=#<Set: {2,3}>, accept_states=[3], rulebook=…>
>> nfa.read_character('b'); nfa.current_states
=> #<Set: {3, 1, 2}>
```

Ответ: в состояниях 1, 2 или 3, что мы определили раньше, когда выполняли это преобразование вручную (напомним, что порядок элементов в множестве `Set` не имеет значения).

Воспользуемся этой идеей – создадим класс `NFASimulation` и включим в него метод, который будет вычислять, как меняется состояние модели в ответ на заданный входной символ. Мы считаем, что состояние модели – это текущее множество возможных состояний НКА (например, «1, 2 или 3»), поэтому можем написать метод `#next_state`, который допускает состояние модели и символ, подает этот символ на вход НКА, находящегося в указанном состоянии, и возвращает новое состояние, получая его от результирующего НКА:

```
class NFASimulation < Struct.new(:nfa_design)
  def next_state(state, character)
    nfa_design.to_nfa(state).tap { |nfa|
      nfa.read_character(character)
    }.current_states
  end
end
```

В двух видах состояний, которые встретились в этом описании, легко запутаться. Одно состояние модели (параметр `state` метода `NFASimulation#next_state`) – это множество, состоящее из нескольких состояний НКА, именно поэтому мы можем передать его в качестве аргумента `current_states` метода `NFADesign#to_nfa`.

Это дает нам удобный способ исследовать состояния модели:

```
>> simulation = NFASimulation.new(nfa_design)
=> #<struct NFASimulation nfa_design=…>
>> simulation.next_state(Set[1, 2], 'a')
=> #<Set: {1, 2}>
>> simulation.next_state(Set[1, 2], 'b')
=> #<Set: {3, 2}>
>> simulation.next_state(Set[3, 2], 'b')
```

```
=> #<Set: {1, 3, 2}>
>> simulation.next_state(Set[1, 3, 2], 'b')
=> #<Set: {1, 3, 2}>
>> simulation.next_state(Set[1, 3, 2], 'a')
=> #<Set: {1, 2}>
```

Теперь нужно придумать, как систематически исследовать состояния модели и представить результат в виде состояний и правил ДКА. Мы хотим, чтобы состояния модели напрямую отображались на состояния ДКА, поэтому первым делом реализуем метод `NFASimulation#rules_for`, который строит все правила, ведущие из заданного состояния модели, для чего воспользуемся методом `#next_state`, который определит множество конечных состояний для каждого правила. Говоря «все правила», мы имеем в виду правила для всех возможных входных символов, поэтому определим также вспомогательный метод `NFARulebook#alphabet`, который будет сообщать, какие символы может читать исходный НКА.

```
class NFARulebook
  def alphabet
    rules.map(&:character).compact.uniq
  end
end

class NFASimulation
  def rules_for(state)
    nfa_design.rulebook.alphabet.map { |character|
      FARule.new(state, character, next_state(state, character))
    }
  end
end
```

Теперь мы, как и собирались, можем видеть, каким образом модель переходит из одного состояния в другое в ответ на различные входные символы:

```
>> rulebook.alphabet
=> ["a", "b"]
>> simulation.rules_for(Set[1, 2])
=>[
      #<FARule #<Set: {1, 2}> --a--> #<Set: {1, 2}>>,
      #<FARule #<Set: {1, 2}> --b--> #<Set: {3, 2}>>
    ]
>> simulation.rules_for(Set[3, 2])
=>[
      #<FARule #<Set: {3, 2}> --a--> #<Set: {}>>,
      #<FARule #<Set: {3, 2}> --b--> #<Set: {1, 3, 2}>>
    ]
```

Метод `#rules_for` позволяет начать с известного состояния модели и обнаружить новые состояния, а, повторяя это раз за разом, мы сможем найти все возможные состояния модели. Для этого напишем метод `NFASimulation#discover_states_and_rules`, который рекурсивно находит дополнительные состояния по аналогии с методом `NFARulebook#follow_free_moves`:

```
class NFASimulation
  def discover_states_and_rules(states)
    rules = states.flat_map { |state| rules_for(state) }
    more_states = rules.map(&:follow).to_set
    if more_states.subset?(states)
      [states, rules]
    else
      discover_states_and_rules(states + more_states)
    end
  end
end
```

Методу `#discover_states_and_rules` безразлична истинная структура состояния модели, ему важно лишь, чтобы состояние можно было передать в качестве аргумента методу `#rules_for`, зато программист может в очередной раз запутаться. Переменные `states` и `more_states` – это множества состояний модели, однако, как мы знаем, каждое состояние модели само по себе является множеством состояний НКА, так что `states` и `more_states` – на самом деле *множества множеств* состояний НКА.

Первоначально нам известно только одно состояние модели: множество возможных состояний нашего НКА, начавшего работу в начальном состоянии. Метод `#discover_states_and_rules` производит расширяющийся поиск от этой начальной точки и в конечном счете находит все четыре состояния и восемь правил модели.

```
>> start_state = nfa_design.to_nfa.current_states
=> #<Set: {1, 2}>
>> simulation.discover_states_and_rules(Set[start_state])
=> [
       #<Set: {
         #<Set: {1, 2}>,
         #<Set: {3, 2}>,
         #<Set: {}>,
         #<Set: {1, 3, 2}>
     }>,
     [
       #<FARule #<Set: {1, 2}> --a--> #<Set: {1, 2}>>,
       #<FARule #<Set: {1, 2}> --b--> #<Set: {3, 2}>>,
       #<FARule #<Set: {3, 2}> --a--> #<Set: {}>>,
```

```
      #<FARule #<Set: {3, 2}> --b--> #<Set: {1, 3, 2}>>,
      #<FARule #<Set: {}> --a--> #<Set: {}>>,
      #<FARule #<Set: {}> --b--> #<Set: {}>>,
      #<FARule #<Set: {1, 3, 2}> --a--> #<Set: {1, 2}>>,
      #<FARule #<Set: {1, 3, 2}> --b--> #<Set: {1, 3, 2}>>
    ]
  ]
```

И последнее, что нам нужно знать о каждом состоянии модели, – следует ли считать его заключительным. Но это легко проверить, задав НКА вопрос в этой точке моделирования:

```
>> nfa_design.to_nfa(Set[1, 2]).accepting?
=> false
>> nfa_design.to_nfa(Set[2, 3]).accepting?
=> true
```

Имея все составные части модельного ДКА, мы должны написать метод `NFASimulation#to_dfa_design`, который элегантно обернет их экземпляром класса `DFADesign`:

```
class NFASimulation
  def to_dfa_design
    start_state = nfa_design.to_nfa.current_states
    states, rules = discover_states_and_rules(Set[start_state])
    accept_states = states.select
       { |state| nfa_design.to_nfa(state).accepting? }

    DFADesign.new(start_state, accept_states, DFARulebook.new(rules))
  end
end
```

Вот теперь все. Мы можем построить экземпляр `NFASimulation` по любому НКА и преобразовать его в ДКА, который допускает точно такие же цепочки:

```
>> dfa_design = simulation.to_dfa_design
=> #<struct DFADesign …>
>> dfa_design.accepts?('aaa')
=> false
>> dfa_design.accepts?('aab')
=> true
>> dfa_design.accepts?('bbbabb')
=> true
```

Блестяще!

В начале этого раздела мы задались вопросом, обладает ли НКА какими-то свойствами, позволяющими делать нечто, чего не может ДКА. Теперь ясно, что ответ отрицательный, потому что любой

НКА можно преобразовать в ДКА, выполняющий точно такие же действия. НКА не дает ничего принципиально нового. Недетерминированность и свободные переходы – это лишь удобная упаковка для того, что ДКА и так умеет делать – как синтаксическая глазурь в языках программирования – не дающая никаких возможностей сверх того, что можно сделать в рамках ограничений детерминированности.

Тот факт, что кажущееся расширение возможностей простой машины не делает ее принципиально более мощной, представляет теоретический интерес, но это полезно и на практике, потому что моделировать ДКА проще, чем НКА: нужно следить всего за одним состоянием, и ДКА настолько прост, что может быть реализован аппаратно или в виде машинного кода, где роль состояний играют ячейки памяти, а роль правил – команды условного перехода. Это означает, что библиотека, реализующая механизм регулярных выражений, может сначала преобразовывать образец в НКА, а затем этот НКА в ДКА, так что в результате получится очень простой автомат, который можно смоделировать быстро и эффективно.

Минимизация ДКА

Некоторые ДКА обладают свойством *минимальности*, то есть не существует ДКА с меньшим числом состояний, который допускал бы в точности такие же цепочки. В результате преобразования НКА в ДКА могут порождаться неминимальные ДКА, содержащие избыточные состояния, но существует элегантный способ устранить избыточность, который называется *алгоритмом Бржозовского*.

1. Начать с неминимального ДКА.
2. Обратить все правила. Визуально это означает, что все стрелки на диаграмме автомата остаются, но меняют направление, а в коде каждый вызов `FARule.new(state, character, next_state)` заменяется вызовом `FARule.new(next_state, character, state)`. Обращение правил обычно приводит к нарушению ограничений детерминированности, так что теперь мы имеем НКА.
3. Поменять ролями начальное и заключительное состояние: начальное состояние становится заключительным, а каждое из заключительных – начальным. (Нельзя напрямую сделать все заключительные состояния начальными, потому что НКА может иметь только одно начальное состояние, но того же эф-

факта можно достичь, создав новое начальное состояние и соединив его с каждым из бывших заключительных состояний свободным переходом.)

4. Преобразовать обращенный НКА в ДКА обычным способом.

Как это ни удивительно, гарантируется, что результирующий ДКА минимален и не содержит избыточных состояний. Неприятность же состоит в том, что этот автомат допускает не те же цепочки, что исходный, а обращенные: если исходный ДКА допускал цепочки `'ab'`, `'aab'`, `'aaab'` и т. д., то минимизированный будет допускать цепочки вида `'ba'`, `'baa'` и `'baaa'`. Чтобы исправить это, нужно просто выполнить всю описанную процедуру второй раз, начав с обращенного ДКА и закончив дважды обращенным ДКА, который также минимален, но допускает те же самые цепочки, что и исходный автомат.

Иметь автоматический способ устранения избыточности само по себе приятно, но интересно, что минимизированный ДКА еще и *канонический*: любые два ДКА, допускающие одно и то же множество цепочек, после минимизации будут иметь одинаковую структуру, поэтому чтобы проверить, являются ли два ДКА эквивалентными, нужно минимизировать их и сравнить структуры получившихся автоматов[1]. Это в свою очередь дает красивый способ проверить эквивалентность двух регулярных выражений: если преобразовать два образца, сопоставляемых с одними и теми же строками (например, `ab(ab)*` и `a(ba)*b`) в НКА, затем эти НКА преобразовать в ДКА и минимизировать оба ДКА с помощью алгоритма Бржозовского, то получатся одинаковые автоматы.

---

[1] Алгоритм решения этой задачи об изоморфизме графов сам по себе непрост, но неформально достаточно взглянуть на диаграммы двух автоматов и решить, «одинаковы» ли они.

Глава 4. Кое-что помощнее

В главе 3 мы изучали конечные автоматы – воображаемые машины, которые устраняют сложность, присущую настоящему компьютеру, и сводят его к простейшей форме. Мы подробно исследовали поведение таких машин и видели, для чего они полезны; мы также выяснили, что несмотря на экзотический способ выполнения, недетерминированные конечные автоматы обладают не большей мощностью, чем более привычные детерминированные аналоги.

Тот факт, что конечный автомат не удается сделать более действенным путем добавления таких причудливых черт, как недетерминированность и свободные переходы, наводит на мысль, что мы застряли на плато – уровне вычислительной мощности, общем для всех таких простых машин, и что невозможно покинуть это плато, не внеся кардинальных изменений в способ работы машины. Так что же все-таки могут делать все эти машины? Не так уж много. Их применение очень ограничено – они умеют допускать или отвергать последовательности символов, но даже и в этом узком диапазоне легко построить язык, который такая машина не сможет распознать.

Допустим, к примеру, что требуется спроектировать машину с конечным числом состояний, которая должна читать цепочки открывающих и закрывающих скобок и допускать только такие, где скобки *сбалансированы*, то есть каждой закрывающей скобке можно поставить в соответствие ранее встретившуюся открывающую[1].

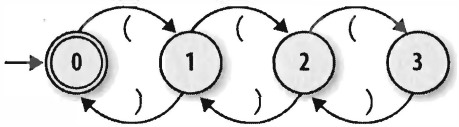
Общая стратегия решения этой задачи заключается в том, чтобы читать по одному символу и запоминать текущий *уровень вложенности*: чтение открывающей скобки увеличивает этот уровень, а чтение

---

[1] Это не то же самое, что допущение цепочек, содержащих одинаковое количество открывающих и закрывающих скобок. Так, в каждой из цепочек '()' и ')(' по одной открывающей и закрывающей скобке, но только цепочка '()' сбалансирована.

закрывающей – уменьшает. Если уровень вложенности равен нулю, то встретившиеся к этому моменту скобки сбалансированы – так как уровень столько же раз уменьшался, сколько увеличивался. Если же уровень вложенности хотя бы раз станет меньше нуля, значит, закрывающих скобок слишком много (например, '())'), и эта цепочка никогда не будет сбалансированной, сколько бы еще символов мы ни прочитали.

Попробуем спроектировать НКА для решения этой задачи. Вот, например, НКА с четырьмя состояниями:



Каждое состояние соответствует одному уровню вложенности, чтение открывающей или закрывающей скобки переводит автомат в состояние с большим или меньшим уровнем соответственно, причем состояние «нет вложенности» заключительное. Поскольку мы уже реализовали на Ruby все необходимое для моделирования НКА, давайте запустим его:

```
>> rulebook = NFARulebook.new([
       FARule.new(0, '(', 1), FARule.new(1, ')', 0),
       FARule.new(1, '(', 2), FARule.new(2, ')', 1),
       FARule.new(2, '(', 3), FARule.new(3, ')', 2)
    ])
=> #<struct NFARulebook rules=[…]>
>> nfa_design = NFADesign.new(0, [0], rulebook)
=> #<struct NFADesign start_state=0, accept_states=[0], rulebook=…>
```

На некоторых входных цепочках наш НКА работает. Он может сказать, что цепочки '(()' и '())' не сбалансированы, а цепочка '(())' сбалансирована. У него даже не возникает проблем с определением сбалансированности более сложной цепочки '((()(()))'':

```
>> nfa_design.accepts?('())')
=> false
>> nfa_design.accepts?('())')
=> false
>> nfa_design.accepts?('(())')
=> true
>> nfa_design.accepts?('((()(()))')
=> true
```

Но у этого автомата имеется серьезный дефект: он выходит из строя, если уровень вложенности больше трех. У него просто недостаточно состояний для отслеживания вложенности в строке типа '(((())))', поэтому он отвергает ее, хотя скобки и сбалансированы:

```
>> nfa_design.accepts?('(((())))')
=> false
```

Можно предложить паллиативное средство – увеличить количество состояний. НКА с пятью состояниями сможет распознать как цепочку '(((())))', так и любую другую, в которой меньше пяти уровней вложенности. При увеличении числа состояний до десяти, ста или тысячи растет и уровень вложенности распознаваемых цепочек. Но как спроектировать НКА, который сможет распознать *любую* сбалансированную цепочку, с произвольным уровнем вложенности? Оказывается, что это невозможно: у конечного автомата должно быть конечное число состояний, поэтому у любого автомата есть предел поддерживаемого им уровня вложенности, и, подав на вход цепочку с уровнем вложенности, на единицу большим, мы сможем этот автомат «сломать».

Истинная проблема в том, что у конечного автомата ограниченная память в виде фиксированного набора его состояний, поэтому не существует способа хранить *произвольный* объем информации. В задаче о сбалансированных скобках, автомат легко может подсчитать максимальный уровень вложенности, заложенный в его проект, но не способен поддержать входные данные произвольного размера[1]. Это несущественно для задач принципиально фиксированного размера, например сопоставления с литеральной строкой `'abc'`, или задач, в которых не нужно отслеживать количество повторений, например сопоставления с регулярным выражением `ab*c`, но тем не менее конечные автоматы непригодны для решения задач, в которых в ходе вычисления необходимо хранить заранее неизвестный объем информации и повторно использовать память в дальнейшем.

Регулярные выражения и вложенные строки

Мы видели, что конечные автоматы тесно связаны с регулярными выражениями. В разделе «Семантика» на стр. 112 было показано, как

[1] Это не означает, что входная цепочка непременно должна быть бесконечна. Достаточно того, что ее можно сделать конечной, но чересчур большой для данного автомата.

превратить любое регулярное выражение в НКА и существует алгоритм обратного преобразования НКА в регулярное выражение[1]. Таким образом, регулярные выражения и НКА эквивалентны и, стало быть, подвержены одним и тем же ограничениям, то есть невозможно с помощью регулярного выражения распознать сбалансированную строку скобок или любой другой язык, в определении которого встречаются пары объектов с произвольным уровнем вложенности.

Быть может, самый известный пример этого ограничения – невозможность установить допустимость HTML-разметки с помощью регулярных выражений (*http://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags/1732454#1732454*). В HTML многие элементы подразумевают наличие открывающего и закрывающего тега, причем такие элементы могут содержать другие элементы, и, следовательно, конечного автомата недостаточно, чтобы прочитать HTML-разметку, подсчитывая, сколько раз встречались незакрытые теги и как глубоко они вложены.

Однако на практике библиотеки «регулярных выражений» зачастую выходят за рамки того, на что технически способны настоящие регулярные выражения. Объекты Regexp в Ruby обладают многими свойствами, не входящими в формальное определение регулярных выражений, и эти дополнительные свойства позволяют распознавать больше языков.

Одно из расширений Regexp – возможность пометить подвыражение с помощью синтаксической конструкции `(?<name>)`, а затем «вызвать» его в другом месте с помощью конструкции `\g<name>`. Возможность сослаться на подвыражение позволяет объекту Regexp рекурсивно вызывать себя самого, а это делает возможным сопоставление пар с произвольной глубиной вложенности.

Например, механизм вызова подвыражений позволяет написать объект Regexp, который способен распознать сбалансированную строку скобок, хотя НКА (а, стало быть, и регулярное выражение в строгом смысле) этого делать не умеет. Вот как выглядит такой объект Regexp:

```
balanced =
  /
    \A                # сопоставляется с началом строки
    (?<brackets>      # начало подвыражения с именем «brackets»
      \(              # сопоставляется с символом открывающей скобки
```

[1] Если коротко, то этот алгоритм преобразует НКА в обобщенный недетерминированный конечный автомат (ОНКА), в котором каждое правило помечено регулярным выражением, а не одним символом, а затем в цикле объединяет состояния и правила ОНКА, пока не останется всего два состояния и одно правило. Регулярное выражение, которое помечает это оставшееся правило, сопоставляется с теми и только теми строками, которые распознает исходный НКА.

```
      \g<brackets>*   # сопоставляется с подвыражением «brackets» нуль
                      # или более раз
      \)              # сопоставляется с символом закрывающей скобки
    )                 # конец подвыражения
    *                 # повторить весь образец нуль или более раз
    \z                # сопоставляется с концом строки
  /x
```

Подвыражение (?<brackets>…) сопоставляется с одной парой скобок, но внутри этой пары может быть произвольное число рекурсивных сопоставлений с тем же подвыражением, поэтому образец в целом может корректно распознать скобки с любым уровнем вложенности:

```
>> ['(()', '())', '(())', '(()(()()))', '((((((((((()))))))))))'].grep(balanced)
=> ["(())", "(()(()()))", "((((((((((()))))))))))"]
```

Это работает только потому, что в механизме регулярных выражений в Ruby используется *стек вызовов* для отслеживания рекурсивных обращений к (?<brackets>…) – ничего подобного ни ДКА, ни НКА делать не умеют. В следующем разделе мы обобщим понятие конечного автомата, наделив его как раз такой возможностью.

Да, кстати, ту же идею можно использовать для написания объекта Regexp, который корректно распознает вложенные теги HTML, но это заведомо не самое полезное времяпрепровождение.

Ясно, что у этих машин есть ограничения. Если недетерминированности недостаточно для расширения возможностей конечного автомата, то что еще можно сделать? Наши проблемы проистекают из ограниченности памяти автомата, так давайте наделим его дополнительной памятью и посмотрим, что из этого выйдет.

Детерминированные автоматы с магазинной памятью

Проблему памяти можно решить, снабдив конечный автомат специальной временной областью, в которой можно хранить данные во время вычислений. Это наделяет машину чем-то вроде *внешней памяти* наряду с ограниченной внутренней памятью, предоставляемой ее состоянием. Как мы скоро увидим, наличие внешней памяти кардинальным образом отражается на вычислительной мощности машины.

Память

Простой способ добавить память конечному автомату – дать ему доступ к *стеку*, структуре данных, обслуживаемой по принци-

ну «последним пришел, первым ушел», – символы заталкиваются в стек, а затем выталкиваются обратно. Стек представляет собой простую структуру данных с ограниченным набором операций – в любой момент доступен только верхний элемент; чтобы узнать, что находится под ним, этот элемент необходимо убрать, и, если вы затолкнули в стек некоторую последовательность символов, то вытолкнуть их можно только в обратном порядке. Однако проблему ограниченной памяти стек тем не менее решает. Никакого встроенного ограничения на размер стека нет, поэтому теоретически он может расти бесконечно и вмещать столько данных, сколько необходимо[1].

Конечный автомат со встроенным стеком называется *автоматом с магазинной памятью* (АМП). Если правила этого автомата детерминированы, то мы будем называть его *детерминированным автоматом с магазинной памятью* (ДАМП). Наличие доступа к стеку открывает ряд новых возможностей; например, легко построить ДАМП, распознающий сбалансированную цепочку скобок. Вот как он работает.

- Заведем два состояния, 1 и 2, и пусть состояние 1 – заключительное.
- Пусть автомат начинает работу в состоянии 1 с пустым стеком.
- Если автомат находится в состоянии 1 и прочитана открывающая скобка, затолкнуть некоторый символ – пусть это будет b от слова «bracket» (скобка) – в стек и перейти в состояние 2.
- Если автомат находится в состоянии 2 и прочитана открывающая скобка, затолкнуть символ b в стек.
- Если автомат находится в состоянии 2 и прочитана закрывающая скобка, вытолкнуть символ b из стека.
- Если автомат находится в состоянии 2 и стек пуст, перейти в состояние 1.

В этом ДАМП размер стека используется для подсчета текущего числа незакрытых открывающих скобок. Если стек пуст, значит, все открывающие скобки закрыты, то есть цепочка сбалансирована. Посмотрим, как растет и сжимается стек при чтении цепочки '(() (()()))':

---

[1] Разумеется, всякая реализация стека ограничена размером оперативной памяти компьютера или объемом свободного места на диске или количеством атомов во вселенной, но в интересах мысленного эксперимента мы будем предполагать, что таких ограничений не существует.

Состояние	Заключительное?	Содержимое стека	Осталось прочитать	Действие
1	да		(()(()()))	читать (, затолкнуть b, перейти в состояние 2
2	нет	b	)(()()))	читать (, затолкнуть b
2	нет	bb	)()()))	читать), вытолкнуть b
2	нет	b	(()()))	читать (, затолкнуть b
2	нет	bb	()()))	читать (, затолкнуть b
2	нет	bbb	)()))	читать), вытолкнуть b
2	нет	bb	()()))	читать (, затолкнуть b
2	нет	bbb	)))	читать), вытолкнуть b
2	нет	bb	))	читать), вытолкнуть b
2	нет	b	)	читать), вытолкнуть b
2	нет			перейти в состояние 1
1	да			-

Правила

Идея, стоящая за ДАМП, распознающим сбалансированные скобки, проста, однако прежде чем реализовать ее, необходимо утрясти некоторые скучные технические детали. Прежде всего, нужно точно решить, как должны работать правила автомата с магазинной памятью. При этом необходимо ответить на такие вопросы:

- Должно ли каждое правило только модифицировать стек, только читать входные данные, только изменять состояние или выполнять все три операции?
- Следует ли предусмотреть разные виды правил для операций заталкивания и выталкивания?
- Нужно ли заводить специальный вид правил для изменения состояния, когда стек пуст?
- Допустимо ли изменять состояние без чтения входного символа, как в случае свободного перехода в НКА?
- Если ДАМП может спонтанно менять свое состояние, как описано выше, то что означает слово «детерминированный»?

На все эти вопросы можно ответить, выбрав единый стиль правил, достаточно гибкий, чтобы поддержать все, что нам нужно. Мы разобьем правило АМП на пять частей:

- текущее состояние машины;
- символ, который должен быть прочитан из входного потока (необязательно);
- следующее состояние машины;

- символ, который должен быть вытолкнут из стека;
- *последовательность* символов, которую нужно затолкнуть в стек после выталкивания символа с вершины.

Первые три части знакомы по работе с ДКА и НКА. Если правило не хочет изменять состояние машины, оно может сделать следующее состояние таким же, как текущее; если оно не хочет читать входной символ (то есть это свободный переход), то может этого не делать при условии, что автомат при этом не станет недетерминированным (см. раздел «Детерминированность» на стр. 144).

Оставшиеся две части – выталкиваемый символ и последовательность заталкиваемых символов – относятся только к АМП. Предполагается, что при выполнении *любого* правила АМП выталкивает символ с вершины стека, а затем заталкивает в стек какие-то другие символы. Каждое правило объявляет, какой символ оно хочет вытолкнуть, и применяется лишь в том случае, когда на вершине стека находится именно этот символ. Если правило хочет, чтобы символ остался в стеке, а не был вытолкнут, оно может включить его в последовательность заталкиваемых символов.

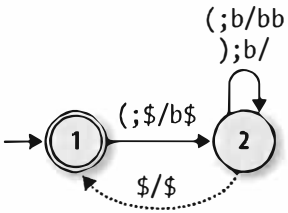
Этот пятичастный формат не позволяет написать правила, применяемые, когда стек пуст, но мы можем обойти эту трудность, выбрав специальный символ, помечающий дно стека (часто его обозначают знаком `$`), а затем сравнивать с этим символом, когда нужно определить, пуст ли стек. Применяя это соглашение, важно следить за тем, чтобы стек не стал действительно пустым, потому что в этом случае нельзя будет применить никакое правило. Автомат должен начинать работу, когда в стеке уже есть специальный символ, и любое правило, которое выталкивает этот символ, должно затем затолкнуть его обратно.

Несложно переписать в этом формате правила ДАМП, распознающего сбалансированные скобки.

- Если автомат находится в состоянии 1 и прочитана открывающая скобка, вытолкнуть символ `$`, затолкнуть символы `b$` и перейти в состояние 2.
- Если автомат находится в состоянии 2 и прочитана открывающая скобка, вытолкнуть символ `b`, затолкнуть символы `bb` и остаться в состоянии 2.
- Если автомат находится в состоянии 2 и прочитана закрывающая скобка, вытолкнуть символ `b`, ничего не заталкивать и остаться в состоянии 2.

- Если автомат находится в состоянии 2 (без чтения символов), вытолкнуть символ $, затолкнуть символы $ и перейти в состояние 1.

Можно показать эти правила на диаграмме автомата. Диаграмма ДАМП очень похожа на диаграмму НКА, только для каждой стрелки, представляющей правило, нужно ставить метку, которая содержит выталкиваемые символы, заталкиваемые символы и символ, прочитанный из входного потока. Если воспользоваться нотацией `a;b/cd` для пометки правила, которое читает входной символ `a`, выталкивает из стека `b` и заталкивает в стек `cd`, то диаграмма автомата будет выглядеть следующим образом:



Детерминированность

Теперь нужно преодолеть следующее препятствие – определить, что в случае АМП означает детерминированность. Для ДКА у нас было ограничение «непротиворечивости»: не должно быть состояний, для которых переход в следующее состояние неоднозначен из-за конфликтующих правил. Та же идея применима и к ДАМП; например, мы можем иметь только одно правило, применимое, когда автомат находится в состоянии 2, следующим символом является открывающая скобка и на вершине стека находится символ `b`. Можно даже написать правило свободного перехода, которое ничего не читает, при условии, что не существует никаких других правил для того же состояния и того же символа на вершине стека, поскольку в противном случае возникла бы неоднозначность: читать символ из потока или нет.

У ДКА имеется также ограничение «полноты» – должно существовать правило для любой возможной ситуации – однако для ДАМП это было бы слишком громоздко из-за большого числа возможных комбинаций состояния, входного символа и символа на вершине стека. Поэтому это ограничение обычно игнорируют, по-

зволяя ДАМП задавать только интересные правила, которые действительно нужны для работы, и предполагая, что ДАМП попадает в неявное *состояние заклинивания* (stuck state), если ни одно правило неприменимо. Именно это и происходит, когда наш ДАМП для распознавания сбалансированных скобок читает цепочку ')' или '())', потому что не существует правила для чтения закрывающей скобки в состоянии 1.

Моделирование

Разобравшись с техническими деталями, построим на Ruby модель детерминированного автомата с магазинной памятью, с которой сможем взаимодействовать. Самую сложную часть работы мы уже проделали, когда моделировали ДКА и НКА, теперь осталось только кое-что подрихтовать.

Главное, чего нам недостает, – это стек. Вот один из способов реализовать класс `Stack`:

```
class Stack < Struct.new(:contents)
  def push(character)
    Stack.new([character] + contents)
  end

  def pop
    Stack.new(contents.drop(1))
  end

  def top
    contents.first
  end

  def inspect
    "#<Stack (#{top})#{contents.drop(1).join}>"
  end
end
```

Объект класса `Stack` хранит свое содержимое во внутреннем массиве и раскрывает операции `#push` и `#pop` для выталкивания и заталкивания символов соответственно, а также операцию `#top` для чтения символа, находящегося на вершине стека:

```
>> stack = Stack.new(['a', 'b', 'c', 'd', 'e'])
=> #<Stack (a)bcde>
>> stack.top
=> "a"
>> stack.pop.pop.top
=> "c"
>> stack.push('x').push('y').top
```

```
=> "y"
>> stack.push('x').push('y').pop.top
=> "x"
```

Это *чисто функциональный* стек. Методы #push и #pop не деструктивные: каждый из них возвращает новый объект, а не модифицирует существующий. Создание нового объекта при каждой операции снижает эффективность реализации по сравнению со стеком, где операции #push и #pop деструктивны (если бы нам это было надо, можно было бы напрямую воспользоваться классом Array), но зато упрощает работу, потому что не нужно беспокоиться о последствиях модификации стека, используемого в нескольких местах.

В главе 3 мы видели, что для моделирования детерминированного конечного автомата нужно сохранить всего один элемент данных – текущее состояние ДКА – и обновлять его при чтении каждого символа, справляясь со сводом правил. Но об автомате с магазинной памятью мы должны на каждом шаге вычислений знать *две* важных вещи: текущее состояние самого автомата и текущее содержимое стека. Если воспользоваться словом *конфигурация* для обозначения комбинации состояния и стека, то можно сказать, что АМП переходит от одной конфигурации к другой при чтении каждого символа; это звучит проще, чем упоминание по отдельности состояния и стека. С этой точки зрения, ДАМП характеризуется только текущей конфигурацией, а свод правил говорит, как перейти от текущей конфигурации к следующей после чтения символа.

В приведенном ниже классе PDAConfiguration хранится конфигурация АМП – состояние и стек, а класс PDARule представляет одно правило из свода[1]:

```
class PDAConfiguration < Struct.new(:state, :stack)
end

class PDARule < Struct.new(:state, :character, :next_state,
                           :pop_character, :push_characters)
  def applies_to?(configuration, character)
    self.state == configuration.state &&
      self.pop_character == configuration.stack.top &&
      self.character == character
  end
end
```

[1] Имена классов начинаются с PDA, а не DPDA, потому что в их реализации нет никаких предположений о детерминированности. Поэтому они без изменения будут работать и при моделировании недетерминированного АМП.

Правило применимо только в случае, когда состояние автомата, символ на вершине стека и следующий прочитанный символ имеют ожидаемые значения:

```
>> rule = PDARule.new(1, '(', 2, '$', ['b', '$'])
=> #<struct PDARule
     state=1,
     character="(",
     next_state=2,
     pop_character="$",
     push_characters=["b", "$"]
   >
>> configuration = PDAConfiguration.new(1, Stack.new(['$']))
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
>> rule.applies_to?(configuration, '(')
=> true
```

Для конечного автомата следование правилу означает просто изменение состояния, тогда как в случае АМП правило изменяет не только состояние, но и содержимое стека, поэтому метод `PDARule#follow` должен принимать в качестве аргумента текущую конфигурацию автомата и возвращать следующую:

```
class PDARule
  def follow(configuration)
    PDAConfiguration.new(next_state, next_stack(configuration))
  end

  def next_stack(configuration)
    popped_stack = configuration.stack.pop

    push_characters.reverse.
    inject(popped_stack) { |stack, character| stack.push(character) }
  end
end
```

Если затолкнуть в стек несколько символов, а затем вытолкнуть их, то их порядок изменится на противоположный:

```
>> stack = Stack.new(['$']).push('x').push('y').push('z')
=> #<Stack (z)yx$>
>> stack.top
=> "z"
>> stack = stack.pop; stack.top
=> "y"
>> stack = stack.pop; stack.top
=> "x"
```

Зная об этом, метод `PDARule#next_stack` обращает массив `push_characters`, перед тем как затолкнуть хранящиеся в нем символы

в стек. Поэтому последний символ в массиве `push_characters` заталкивается в стек *первым*, так что после выталкивания он снова станет последним. Это сделано просто для удобства, чтобы свойство `push_characters` правила можно было читать как последовательность символов («в порядке выталкивания»), которая будет находиться на вершине стека после применения правила, не заботясь о том, как она туда попадает.

Итак, если у нас есть правило `PDARule`, которое применяется к `PDAConfiguration`, то мы можем последовать ему и узнать, каким будет следующее состояние и стек:

```
>> rule.follow(configuration)
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
```

Этого достаточно для реализации свода правил для ДАМП. Реализация очень похожа на класс `DFARulebook`, приведенный в разделе «Моделирование» на стр. 92.

```
class DPDARulebook < Struct.new(:rules)
  def next_configuration(configuration, character)
    rule_for(configuration, character).follow(configuration)
  end

  def rule_for(configuration, character)
    rules.detect { |rule| rule.applies_to?(configuration, character) }
  end
end
```

Теперь можно составить свод правил для ДАМП, распознающего сбалансированные скобки, и в пошаговом режиме посмотреть, как меняется конфигурация при чтении нескольких входных символов:

```
>> rulebook = DPDARulebook.new([
       PDARule.new(1, '(', 2, '$', ['b', '$']),
       PDARule.new(2, '(', 2, 'b', ['b', 'b']),
       PDARule.new(2, ')', 2, 'b', []),
       PDARule.new(2, nil, 1, '$', ['$'])
    ])
=> #<struct DPDARulebook rules=[…]>
>> configuration = rulebook.next_configuration(configuration, '(')
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
>> configuration = rulebook.next_configuration(configuration, '(')
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)b$>>
>> configuration = rulebook.next_configuration(configuration, ')')
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
```

Но чтобы не делать это вручную, воспользуемся сводом правил для построения объекта DPDA, который умеет запоминать текущую конфигурацию автомата при чтении символов из входного потока:

```
class DPDA < Struct.new(:current_configuration, :accept_states, :rulebook)
  def accepting?
    accept_states.include?(current_configuration.state)
  end

  def read_character(character)
    self.current_configuration =
      rulebook.next_configuration(current_configuration, character)
  end

  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end
end
```

Таким образом, мы можем создать объект DPDA, подать ему на вход цепочку символов и посмотреть, допускает он ее или нет:

```
>> dpda = DPDA.new(PDAConfiguration.new(1, Stack.new(['$'])), [1], rulebook)
=> #<struct DPDA ...>
>> dpda.accepting?
=> true
>> dpda.read_string('(()'); dpda.accepting?
=> false
>> dpda.current_configuration
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
```

Пока все хорошо, но в нашем своде правил имеется свободный переход, поэтому если мы хотим, чтобы модель работала правильно, она должна поддерживать свободные переходы. Добавим в класс DPDARulebook вспомогательный метод, аналогичный имеющемуся в классе NFARulebook (см. раздел «Свободные переходы» на стр. 104):

```
class DPDARulebook
  def applies_to?(configuration, character)
    !rule_for(configuration, character).nil?
  end

  def follow_free_moves(configuration)
    if applies_to?(configuration, nil)
      follow_free_moves(next_configuration(configuration, nil))
    else
      configuration
    end
  end
end
```

Метод `DPDARulebook#follow_free_moves` повторно следует по свободным переходам, применимым к текущей конфигурации, прекращая работу, когда таковых не останется:

```
>> configuration = PDAConfiguration.new(2, Stack.new(['$']))
=> #<struct PDAConfiguration state=2, stack=#<Stack ($)>>
>> rulebook.follow_free_moves(configuration)
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
```

Впервые в наших экспериментах с автоматами появилась возможность зацикливания модели. Бесконечный цикл возникает в случае, когда существует цепочка свободных переходов, которая начинается и заканчивается в одном и том же состоянии; простейший пример – наличие одного свободного перехода, не изменяющего конфигурацию вовсе:

```
>> DPDARulebook.new([PDARule.new(1, nil, 1, '$', ['$'])]).
     follow_free_moves(PDAConfiguration.new(1, Stack.new(['$'])))
SystemStackError: stack level too deep
```

Такие бесконечные циклы бесполезны, поэтому, проектируя автоматы с магазинной памятью, нужно их всячески избегать.

Мы должны также обернуть подразумеваемую по умолчанию реализацию метода `DPDA#current_configuration`, чтобы воспользоваться поддержкой свободных переходов в своде правил:

```
class DPDA
  def current_configuration
    rulebook.follow_free_moves(super)
  end
end
```

Теперь у нас есть модель ДАМП, мы можем запустить, подать на вход цепочку символов и проверить, допустима ли она:

```
>> dpda = DPDA.new(PDAConfiguration.new(1, Stack.new(['$'])), [1], rulebook)
=> #<struct DPDA …>
>> dpda.read_string('(()('); dpda.accepting?
=> false
>> dpda.current_configuration
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)b$>>
>> dpda.read_string('))()'); dpda.accepting?
=> true
>> dpda.current_configuration
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
```

Если, как обычно, обернуть эту модель классом `DPDADesign`, то можно будет без труда проверить сколько угодно цепочек:

```
class DPDADesign < Struct.new(:start_state, :bottom_character,
                              :accept_states, :rulebook)
  def accepts?(string)
    to_dpda.tap { |dpda| dpda.read_string(string) }.accepting?
  end

  def to_dpda
    start_stack = Stack.new([bottom_character])
    start_configuration = PDAConfiguration.new(start_state, start_stack)
    DPDA.new(start_configuration, accept_states, rulebook)
  end
end
```

Как и следовало ожидать, наш объект `DPDADesign` способен распознать сложные цепочки сбалансированных скобок с произвольной глубиной вложенности:

```
>> dpda_design = DPDADesign.new(1, '$', [1], rulebook)
=> #<struct DPDADesign …>
>> dpda_design.accepts?('((((((((((()))))))))))')
=> true
>> dpda_design.accepts?('()(())((()))(()(()))')
=> true
>> dpda_design.accepts?('((()(()()(()(()))))()')
=> false
```

Остался последний штрих. Наша модель отлично работает на входных цепочках, которые оставляют ДАМП в корректном состоянии, но «ломается», когда автомат заклинивает:

```
>> dpda_design.accepts?('())')
NoMethodError: undefined method `follow' for nil:NilClass
```

Так происходит, потому что метод `DPDARulebook#next_configuration` предполагает, что сможет найти применимое правило, поэтому не следует вызывать его, когда применимых правил нет. Чтобы исправить эту ошибку, изменим метод `DPDA#read_character`, так чтобы он проверял, есть ли хотя бы одно подходящее правило, и, если нет, то переводил ДАМП в специальное состояние заклинивания, из которого тот никогда не сможет выйти:

```
class PDAConfiguration
  STUCK_STATE = Object.new

  def stuck
```

```
    PDAConfiguration.new(STUCK_STATE, stack)
  end

  def stuck?
    state == STUCK_STATE
  end
end

class DPDA
  def next_configuration(character)
    if rulebook.applies_to?(current_configuration, character)
      rulebook.next_configuration(current_configuration, character)
    else
      current_configuration.stuck
    end
  end

  def stuck?
    current_configuration.stuck?
  end

  def read_character(character)
    self.current_configuration = next_configuration(character)
  end

  def read_string(string)
    string.chars.each do |character|
      read_character(character) unless stuck?
    end
  end
end
```

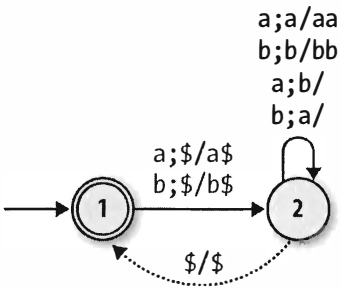
Теперь ДАМП не ломается, а только заклинивается:

```
>> dpda = DPDA.new(PDAConfiguration.new(1, Stack.new(['$'])), [1], rulebook)
=> #<struct DPDA ...>
>> dpda.read_string('())'); dpda.current_configuration
=> #<struct PDAConfiguration state=#<Object>, stack=#<Stack ($)>>
>> dpda.accepting?
=> false
>> dpda.stuck?
=> true
>> dpda_design.accepts?('())')
=> false
```

Недетерминированные автоматы с магазинной памятью

Хотя автомату, распознающему сбалансированные скобки, для работы нужен стек, использует он его всего лишь как счетчик, а при составлении правил нас интересовало только различие между «стек пуст» и «стек не пуст». Более сложные ДАМП'ы заталкивают в стек

различные символы и используют эту информацию при вычислениях. Простой пример – автомат для распознавания цепочек, содержащих одно и то же число двух разных символов, скажем a и b.



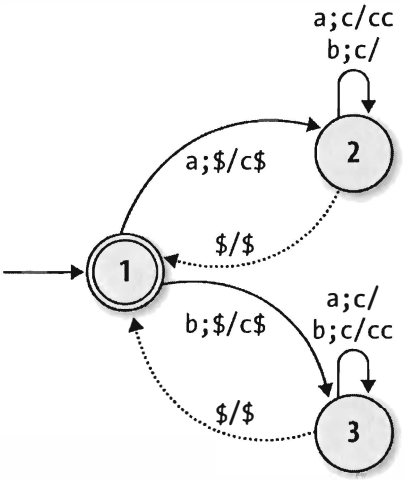
Наша модель показывает, что этот ДАМП справляется с задачей:

```
>> rulebook = DPDARulebook.new([
       PDARule.new(1, 'a', 2, '$', ['a', '$']),
       PDARule.new(1, 'b', 2, '$', ['b', '$']),
       PDARule.new(2, 'a', 2, 'a', ['a', 'a']),
       PDARule.new(2, 'b', 2, 'b', ['b', 'b']),
       PDARule.new(2, 'a', 2, 'b', []),
       PDARule.new(2, 'b', 2, 'a', []),
       PDARule.new(2, nil, 1, '$', ['$'])
    ])
=> #<struct DPDARulebook rules=[…]>
>> dpda_design = DPDADesign.new(1, '$', [1], rulebook)
=> #<struct DPDADesign …>
>> dpda_design.accepts?('ababab')
=> true
>> dpda_design.accepts?('bbbaaaab')
=> true
>> dpda_design.accepts?('baa')
=> false
```

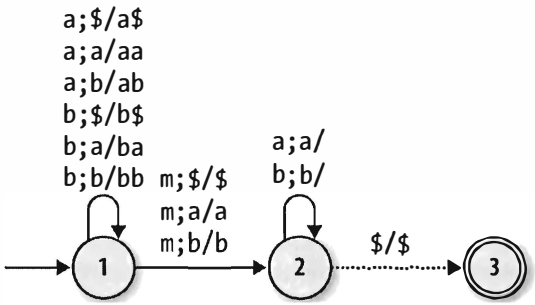
Данный автомат похож на автомат для распознавания сбалансированных скобок, однако его поведение зависит от того, какой символ находится на вершине стека. Если на вершине символ a, значит, автомат увидел избыток символов a, и все вновь прочитанные a будут накапливаться в стеке, а каждый прочитанный b будет выталкивать один a из стека. Наоборот, если на вершине стека находится b, то будут накапливаться b, тогда как чтение a будет их выталкивать.

Но даже этот ДАМП задействует стек не в полной мере. Ниже верхнего символа нет никакой интересной истории, а лишь невыразительная груда a или b; того же результата можно было достичь,

заталкивая в стек символы только одного вида (то есть снова используя стек как счетчик) и заведя два разных состояния, чтобы отличить «избыток a» от «избыток b»:



Чтобы использовать весь потенциал стека, нам необходима задачка посложнее, для которой требуется хранить структурированную информацию. Классический пример – распознавание палиндромов: читая символ за символом, мы должны запоминать, что уже видели, а дойдя до середины цепочки, – смотреть, совпадают ли новые символы с запомненными, только в обратном порядке. Ниже показан ДАМП, распознающий палиндромы, составленные из символов a и b, при условии, что в середине цепочки находится символ m:



Этот автомат начинает работу в состоянии 1, читает входной поток, и пока видит символы a и b, заталкивает их в стек. Прочитав символ m, он переходит в состояние 2, в котором продолжает чтение, но теперь выталкивает символы из стека. Если прочитанный символ из второй половины цепочки совпадает с вытолкнутым из стека, то автомат остается в состоянии 2 и в конце концов обнаружит символ $ на дне стека, после чего перейдет в состояние 3 и допустит входную цепочку. Если же символ, прочитанный в состоянии 2, не совпадает с тем, что находится на вершине стека, то применимого правила не найдется, поэтому автомат перейдет в состояние заклинивания и отвергнет цепочку.

Смоделируем этот ДАМП и проверим, правильно ли он работает:

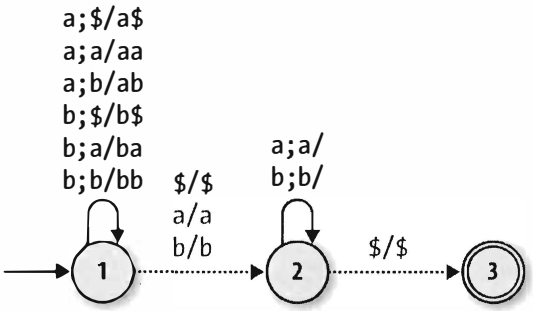
```
>> rulebook = DPDARulebook.new([
       PDARule.new(1, 'a', 1, '$', ['a', '$']),
       PDARule.new(1, 'a', 1, 'a', ['a', 'a']),
       PDARule.new(1, 'a', 1, 'b', ['a', 'b']),
       PDARule.new(1, 'b', 1, '$', ['b', '$']),
       PDARule.new(1, 'b', 1, 'a', ['b', 'a']),
       PDARule.new(1, 'b', 1, 'b', ['b', 'b']),
       PDARule.new(1, 'm', 2, '$', ['$']),
       PDARule.new(1, 'm', 2, 'a', ['a']),
       PDARule.new(1, 'm', 2, 'b', ['b']),
       PDARule.new(2, 'a', 2, 'a', []),
       PDARule.new(2, 'b', 2, 'b', []),
       PDARule.new(2, nil, 3, '$', ['$'])
    ])
=> #<struct DPDARulebook rules=[…]>
>> dpda_design = DPDADesign.new(1, '$', [3], rulebook)
=> #<struct DPDADesign …>
>> dpda_design.accepts?('abmba')
=> true
>> dpda_design.accepts?('babbamabbab')
=> true
>> dpda_design.accepts?('abmb')
=> false
>> dpda_design.accepts?('baambaa')
=> false
```

Отлично, только вот m в середине входной цепочки портит все дело. Почему нельзя спроектировать автомат, который просто будет распознавать палиндромы – aa, abba, babbaabbab и т. д. – не требуя специального маркера?

Такой автомат должен перейти из состояния 1 в состояние 2, как только дойдет до середины цепочки, но без маркера он не знает, когда это случится. Ранее, обсуждая НКА, мы видели, что задачи типа «как узнать, когда...?» можно решить, ослабив ограничения детерминированности и позволив автомату совершить необходимую

смену состояния в любой точке, так чтобы было *возможно* допустить палиндром, выбрав нужное правило в нужный момент.

Неудивительно, что автомат с магазинной памятью без ограничений детерминированности называется *недетерминированным автоматом с магазинной памятью*. Ниже приведен такой автомат для распознавания палиндромов с четным числом букв[1]:



Здесь все происходит так же, как в ДАМП, за исключением правил, ведущих из состояния 1 в состояние 2: в ДАМП они читают из потока `m`, а здесь это свободные переходы. Таким образом, мы даем НАМП возможность изменять состояние в любом месте входной цепочки безо всякого маркера.

Моделирование

Недетерминированный автомат моделировать труднее, чем детерминированный, но вся сложная работа для НКА уже проделана в разделе «Недетерминированность» на стр. 96, и те же идеи годятся и для НАМП. Нам понадобится класс `NPDARulebook` для хранения недетерминированного набора правил `PDARule`, его реализация почти не отличается от `NFARulebook`:

```
require 'set'

class NPDARulebook < Struct.new(:rules)
  def next_configurations(configurations, character)
```

[1] Ограничение «четное число букв» позволяет сделать автомат простым: палиндром длины `2n` можно допустить, сначала затолкнув в стек `n` символов, а затем столько же вытолкнув. Чтобы распознать любой палиндром, понадобится еще несколько правил, переводящих автомат из состояния 1 в состояние 2.

```
    configurations.flat_map { |config| follow_rules_for(config, character) }.to_set
  end

  def follow_rules_for(configuration, character)
    rules_for(configuration, character).map { |rule| rule.follow(configuration) }
  end

  def rules_for(configuration, character)
    rules.select { |rule| rule.applies_to?(configuration, character) }
  end
end
```

В разделе «Недетерминированность» на стр. 96 мы моделировали НКА, запоминая возможные состояния в объекте `Set`; в случае НАМП в `Set` запоминаются возможные *конфигурации*.

В своде правил необходимо поддержать свободные переходы, и это тоже делается практически так же, как в классе `NFARulebook`:

```
class NPDARulebook
  def follow_free_moves(configurations)
    more_configurations = next_configurations(configurations, nil)
    if more_configurations.subset?(configurations)
      configurations
    else
      follow_free_moves(configurations + more_configurations)
    end
  end
end
```

Еще нужен класс `NPDA`, чтобы обернуть свод правил и множество `Set` текущих конфигураций:

```
class NPDA < Struct.new(:current_configurations, :accept_states, :rulebook)
  def accepting?
    current_configurations.any?
     { |config| accept_states.include?(config.state) }
  end

  def read_character(character)
    self.current_configurations =
      rulebook.next_configurations(current_configurations, character)
  end

  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end

  def current_configurations
    rulebook.follow_free_moves(super)
  end
end
```

Теперь можно в пошаговом режиме смоделировать все возможные конфигурации НАМП при чтении очередного символа:

```
>> rulebook = NPDARulebook.new([
       PDARule.new(1, 'a', 1, '$', ['a', '$']),
       PDARule.new(1, 'a', 1, 'a', ['a', 'a']),
       PDARule.new(1, 'a', 1, 'b', ['a', 'b']),
       PDARule.new(1, 'b', 1, '$', ['b', '$']),
       PDARule.new(1, 'b', 1, 'a', ['b', 'a']),
       PDARule.new(1, 'b', 1, 'b', ['b', 'b']),
       PDARule.new(1, nil, 2, '$', ['$']),
       PDARule.new(1, nil, 2, 'a', ['a']),
       PDARule.new(1, nil, 2, 'b', ['b']),
       PDARule.new(2, 'a', 2, 'a', []),
       PDARule.new(2, 'b', 2, 'b', []),
       PDARule.new(2, nil, 3, '$', ['$'])
    ])
=> #<struct NPDARulebook rules=[…]>
>> configuration = PDAConfiguration.new(1, Stack.new(['$']))
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
>> npda = NPDA.new(Set[configuration], [3], rulebook)
=> #<struct NPDA …>
>> npda.accepting?
=> true
>> npda.current_configurations
=> #<Set: {
       #<struct PDAConfiguration state=1, stack=#<Stack ($)>>,
       #<struct PDAConfiguration state=2, stack=#<Stack ($)>>,
       #<struct PDAConfiguration state=3, stack=#<Stack ($)>>
    }>
>> npda.read_string('abb'); npda.accepting?
=> false
>> npda.current_configurations
=> #<Set: {
       #<struct PDAConfiguration state=1, stack=#<Stack (b)ba$>>,
       #<struct PDAConfiguration state=2, stack=#<Stack (a)$>>,
       #<struct PDAConfiguration state=2, stack=#<Stack (b)ba$>>
    }>
>> npda.read_character('a'); npda.accepting?
=> true
>> npda.current_configurations
=> #<Set: {
       #<struct PDAConfiguration state=1, stack=#<Stack (a)bba$>>,
       #<struct PDAConfiguration state=2, stack=#<Stack ($)>>,
       #<struct PDAConfiguration state=2, stack=#<Stack (a)bba$>>,
       #<struct PDAConfiguration state=3, stack=#<Stack ($)>>
    }>
```

И наконец, класс `NPDADesign` для непосредственной проверки цепочек:

```
class NPDADesign < Struct.new(:start_state, :bottom_character,
                              :accept_states, :rulebook)
  def accepts?(string)
```

```
    to_npda.tap { |npda| npda.read_string(string) }.accepting?
  end

  def to_npda
    start_stack = Stack.new([bottom_character])
    start_configuration = PDAConfiguration.new(start_state, start_stack)
    NPDA.new(Set[start_configuration], accept_states, rulebook)
  end
end
```

Теперь можно убедиться, что наш НАМП действительно распознает палиндромы:

```
>> npda_design = NPDADesign.new(1, '$', [3], rulebook)
=> #<struct NPDADesign …>
>> npda_design.accepts?('abba')
=> true
>> npda_design.accepts?('babbaabbab')
=> true
>> npda_design.accepts?('abb')
=> false
>> npda_design.accepts?('baabaa')
=> false
```

Вещь! Недетерминированность таки позволяет распознавать языки, неподвластные детерминированным машинам.

Неэквивалентность

Однако минуточку: в разделе «Эквивалентность» на стр. 124 мы видели, что недетерминированные автоматы без стека в точности эквивалентны детерминированным. Написанная на Ruby модель НКА вела себя как ДКА – переходила между конечным числом «состояний модели» по мере чтения входных символов – что позволило нам преобразовать НКА в ДКА, допускающий те же строки. Так дала ли сейчас недетерминированность какие-нибудь дополнительные возможности или Ruby-модель НАМП ведет себя так же, как ДАМП? Существует ли алгоритм преобразования любого недетерминированного автомата с магазинной памятью в детерминированный?

Нет, как выясняется, не существует. Трюк с преобразованием НКА в ДКА работал только потому, что одно состояние ДКА можно использовать для представления многих возможных состояний НКА. Чтобы смоделировать НКА, нам нужно только следить, в каких состояниях он мог бы оказаться в данный момент, а затем выбирать отличное множество возможных состояний при каждом чтении входного символа, и ДКА вполне может справиться с этой задачей.

Но для ДАМП это не проходит: невозможно придумать осмысленное представление нескольких конфигураций НАМП в виде одной конфигурации ДАМП. Проблема, естественно, в стеке. Модель НАМП должна знать все символы, которые в настоящий момент могли бы быть на вершине стека, и она должна уметь производить операции выталкивания и заталкивания одновременно над несколькими стеками. Невозможно объединить все возможные стеки в один, так чтобы ДАМП видел все верхние символы и мог получить доступ к каждому стеку в отдельности. Мы могли бы без труда написать на Ruby программу, которая все это делает, но ДАМП для этого не обладает достаточной мощью.

Итак, к сожалению, наша модель НАМП *не* ведет себя, как ДАМП, и алгоритма преобразования НАМП в ДАМП не существует. Задача о палиндроме без маркера – пример того, что НАМП сделать может, а ДАМП – нет, так что недетерминированные автоматы с магазинной памятью *действительно* мощнее детерминированных.

Разбор с помощью автоматов с магазинной памятью

В разделе «Регулярные выражения» на стр. 108 было показано, как можно использовать недетерминированные конечные автоматы для реализации сопоставления с регулярными выражениями. У автоматов с магазинной памятью также есть важное практическое применение: разбор языков программирования.

В разделе «Реализация синтаксических анализаторов» на стр. 82 мы видели, как с помощью библиотеки Treetop построить анализатор для части языка SIMPLE. В анализаторах, генерируемых Treetop, используется единая *грамматика, разбирающая выражения*, для описания полного синтаксиса разбираемого языка, но это сравнительно новая идея. Более традиционный подход заключается в разбиении процесса разбора на две фазы.

Лексический анализ

Прочитать строку символов и преобразовать ее в последовательность *лексем*. Каждая лексема представляет отдельный структурный элемент синтаксиса программы, например: «имя переменной», «открывающая скобка» или «ключевое слово `while`». Лексический анализатор применяет зависящий от языка набор правил, называемых *лексической грамматикой*, чтобы решить,

какие последовательности символов порождают какие лексемы. В этой фазе приходится разбираться с такими низкоуровневыми деталями, как правила образования имен переменных, комментарии и пробельные символы, чтобы в итоге получить чистую последовательность лексем, которую можно подать на вход следующей фазы.

Синтаксический анализ

Прочитать последовательность лексем и решить, представляют они корректную программу с точки зрения синтаксической грамматики разбираемого языка или нет. Если программа корректна, то синтаксический анализатор может породить дополнительную информацию о ее структуре (например, дерево разбора).

Лексический анализ

Фаза лексического анализа обычно довольна проста. Ее можно реализовать с помощью регулярных выражений (и, следовательно, НКА), потому что требуется всего лишь применять к линейной последовательности символов некоторые правила и смотреть, образуют символы нечто похожее на ключевое слово, имя переменной, оператор и т. д. Вот написанный на скорую руку Ruby-код, который разбивает SIMPLE-программу на лексемы:

```
class LexicalAnalyzer < Struct.new(:string)
  GRAMMAR = [
    { token: 'i', pattern: /if/          }, # ключевое слово if
    { token: 'e', pattern: /else/        }, # ключевое слово else
    { token: 'w', pattern: /while/       }, # ключевое слово while
    { token: 'd', pattern: /do-nothing/  }, # ключевое слово do-nothing
    { token: '(', pattern: /\(/          }, # открывающая скобка
    { token: ')', pattern: /\)/          }, # закрывающая скобка
    { token: '{', pattern: /\{/          }, # открывающая фигурная скобка
    { token: '}', pattern: /\}/          }, # закрывающая фигурная скобка
    { token: ';', pattern: /;/           }, # точка с запятой
    { token: '=', pattern: /=/           }, # знак равенства
    { token: '+', pattern: /\+/          }, # знак сложения
    { token: '*', pattern: /\*/          }, # знак умножения
    { token: '<', pattern: /</           }, # знак меньше
    { token: 'n', pattern: /[0-9]+/      }, # число
    { token: 'b', pattern: /true|false/  }, # булево значение
    { token: 'v', pattern: /[a-z]+/      }  # имя переменной
  ]

  def analyze
    [].tap do |tokens|
      while more_tokens?
        tokens.push(next_token)
```

```
      end
    end
  end

  def more_tokens?
    !string.empty?
  end

  def next_token
    rule, match = rule_matching(string)
    self.string = string_after(match)
    rule[:token]
  end

  def rule_matching(string)
    matches = GRAMMAR.map {|rule| match_at_beginning(rule[:pattern], string)}
    rules_with_matches = GRAMMAR.zip(matches).reject
      { |rule, match| match.nil? }
    rule_with_longest_match(rules_with_matches)
  end

  def match_at_beginning(pattern, string)
    /\A#{pattern}/.match(string)
  end

  def rule_with_longest_match(rules_with_matches)
    rules_with_matches.max_by { |rule, match| match.to_s.length }
  end

  def string_after(match)
    match.post_match.lstrip
  end
end
```

В этой реализации в качестве лексем используются одиночные символы – w означает «ключевое слово while», + – «знак сложения» и т. д. – поскольку мы собираемся подавать лексемы на вход АМП, а наша написанная на Ruby модель АМП ожидает именно символы.

Использование символов в роли лексем годится для простой демонстрации, когда не нужно сохранять имена переменных или значения литералов. Но в настоящем анализаторе следовало бы представлять лексемы подходящей структурой данных, которая позволяет передать больше информации, чем «какая-то переменная с неизвестным именем» или «некое булево значение».

Если создать экземпляр `LexicalAnalyzer`, передав конструктору строку кода на Simple, и вызвать его метод `#analyze`, то мы получим в ответ массив лексем, показывающий, как код разбивается на ключевые слова, операторы, знаки препинания и другие синтаксические элементы:

```
>> LexicalAnalyzer.new('y = x * 7').analyze
=> ["v", "=", "v", "*", "n"]
>> LexicalAnalyzer.new('while (x < 5) { x = x * 3 }').analyze
=> ["w", "(", "v", "<", "n", ")", "{", "v", "=", "v", "*", "n", "}"]
>> LexicalAnalyzer.new('if (x < 10) { y = true; x = 0 } else
    { do-nothing }').analyze
=> ["i", "(", "v", "<", "n", ")", "{", "v", "=", "b", ";", "v", "=", "n", ↵
"}", "e", "{", "d", "}"]
```

Лексический анализатор всегда выбирает правило, соответствующее самой длинной подстроке, и потому может обрабатывать переменные с именами, которые в противном случае могли бы быть приняты за ключевые слова:

```
>> LexicalAnalyzer.new('x = false').analyze
=> ["v", "=", "b"]
>> LexicalAnalyzer.new('x = falsehood').analyze
=> ["v", "=", "v"]
```

Эту проблему можно решать и по-другому. Один из вариантов – использовать в правилах более ограничительные регулярные выражения: если бы правило для булевых значений было записано в виде `/(true|false)(?![a-z])/`, то оно вообще не сопоставилось бы со строкой `'falsehood'`, так что выбирать самую длинную подстроку не пришлось бы.

Синтаксический анализ

Справившись с легкой работой – преобразованием строки в последовательность лексем, – мы можем приступить к более трудной: принятию решения о том, представляет ли эта последовательность синтаксически корректную Simple-программу. Применить регулярные выражения или НКА не получится – синтаксис Simple допускает скобки с произвольной глубиной вложенности, а мы уже знаем, что конечные автоматы не обладают достаточной мощностью для распознавания таких языков. Однако *возможно* использовать для распознавания допустимых последовательностей лексем автомат с магазинной памятью, и мы сейчас его построим.

Для начала нам необходима синтаксическая грамматика, которая описывает, как разрешено объединять лексемы в программу. Ниже показана часть грамматики Simple, основанная на структуре Treetop-грамматики, приведенной в разделе «Реализация синтаксических анализаторов» на стр. 82.

```
<statement>  ::= <while> | <assign>
<while>      ::= 'w' '(' <expression> ')' '{' <statement> '}'
```

```
<assign>     ::= 'v' '=' <expression>
<expression> ::= <less-than>
<less-than>  ::= <multiply> '<' <less-than> | <multiply>
<multiply>   ::= <term> '*' <multiply> | <term>
<term>       ::= 'n' | 'v'
```

Такая грамматика называется *контекстно-свободной* (КСГ)[1]. Каждое правило состоит из *символа* в левой части и одной или нескольких последовательностей символов в правой. Например, правило `<statement> ::= <while> | <assign>` означает, что предложение в SIMPLE – это либо цикл `while`, либо присваивание, а правило `<assign> ::= 'v' '=' <expression>` – что предложение присваивания состоит из имени переменной, за которым следует знак равенства и выражение.

КСГ представляет собой статическое описание структуры SIMPLE, но ее можно также рассматривать как набор правил для *генерации* SIMPLE-программ. Начав с символа `<statement>`, мы можем применять правила грамматики для рекурсивного раскрытия символов до тех пор, пока не останутся только лексемы. Вот один из многих способов полностью раскрыть `<statement>` согласно правилам:

```
<statement> → <assign>
            → 'v' '=' <expression>
            → 'v' '=' <less-than>
            → 'v' '=' <multiply>
            → 'v' '=' <term> '*' <multiply>
            → 'v' '=' 'v' '*' <multiply>
            → 'v' '=' 'v' '*' <term>
            → 'v' '=' 'v' '*' 'n'
```

Отсюда следует, что последовательность `'v' '=' 'v' '*' 'n'` представляет синтаксически корректную программу, но нас больше интересует обратная задача: *распознавать* корректные программы, а не генерировать их. Получив последовательность лексем от лексического анализатора, мы хотели бы знать, можно ли, применяя в каком-то порядке правила грамматики, раскрыть символ `<statement>`, так чтобы получилась именно такая последовательность. К счастью, су-

[1] Грамматика является «контекстно-свободной» в том смысле, что ее правила ничего не говорят о том, в каком контексте могут встречаться различные синтаксические элементы; присваивание всегда состоит из имени переменной, знака равенства и выражения вне зависимости от того, какие лексемы его окружают. Не все теоретически возможные языки можно описать такой грамматикой, но большинство языков программирования в эту категорию попадают.

ществует способ преобразовать контекстно-свободную грамматику в недетерминированный автомат с магазинной памятью, который умеет принимать такое решение.

Методика преобразования КСГ в АМП выглядит так.

1. Выбрать знак[1] для представления каждого символа грамматики. В данном случае будем использовать начальную букву символа в верхнем регистре: `S` вместо `<statement>`, `W` вместо `<while>` и т. д. – чтобы отличить от букв в нижнем регистре, которые обозначают лексемы.
2. Воспользоваться стеком АМП для хранения знаков, представляющих символы грамматики (`S`, `W`, `A`, `E`, ...) и лексемы (`w`, `v`, `=`, `*`, ...). В момент начала работы АМП должен сразу затолкнуть в стек символ, представляющий структуру, которую он пытается распознать. Мы хотим распознавать предложения SIMPLE, поэтому наш АМП в самом начале заталкивает в стек `S`:

```
>> start_rule = PDARule.new(1, nil, 2, '$', ['S', '$'])
=> #<struct PDARule …>
```

3. Сопоставим правилам грамматики правила АМП, которые раскрывают символы, находящиеся на вершине стека, не читая входной поток. Каждое правило грамматики описывает, как раскрыть один символ в последовательность других символов и лексем, и это описание можно преобразовать в правило АМП, которое выталкивает из стека знак, соответствующий символу, и заталкивает другие знаки:

```
>> symbol_rules = [
       # <statement> ::= <while> | <assign>
       PDARule.new(2, nil, 2, 'S', ['W']),
       PDARule.new(2, nil, 2, 'S', ['A']),

       # <while> ::= 'w' '(' <expression> ')' '{' <statement> '}'
       PDARule.new(2, nil, 2, 'W', ['W', '(', 'E', ')', '{', 'S', '}']),

       # <assign> ::= 'v' '=' <expression>
       PDARule.new(2, nil, 2, 'A', ['v', '=', 'E']),

       # <expression> ::= <less-than>
       PDARule.new(2, nil, 2, 'E', ['L']),
```

[1] К сожалению, русскоязычная терминология неоднозначна: слова character и symbol переводятся одним словом «символ». В большинстве случаев это не приводит к недоразумениям, но здесь character и symbol встречаются в одном контексте. Во избежание путаницы здесь – и только здесь – слово character переводится как «знак». – *Прим. перев.*

```
        # <less-than> ::= <multiply> '<' <less-than> | <multiply>
        PDARule.new(2, nil, 2, 'L', ['M', '<', 'L']),
        PDARule.new(2, nil, 2, 'L', ['M']),

        # <multiply> ::= <term> '*' <multiply> | <term>
        PDARule.new(2, nil, 2, 'M', ['T', '*', 'M']),
        PDARule.new(2, nil, 2, 'M', ['T']),

        # <term> ::= 'n' | 'v'
        PDARule.new(2, nil, 2, 'T', ['n']),
        PDARule.new(2, nil, 2, 'T', ['v'])
      ]
=> [#<struct PDARule …>, #<struct PDARule …>, …]
```

Например, правило для предложений присваивания говорит, что символ `<assign>` можно раскрыть в последовательность, состоящую из лексем `v` и `=`, за которыми следует символ `<expression>`, так что соответствующее правило АМП спонтанно выталкивает `A` из стека и заталкивает знаки `v=E`. Правило `<statement>` говорит, что символ `<statement>` можно заменить символом `<while>` или `<assign>`; мы сопоставим ему два правила АМП: первое выталкивает из стека `S` и заталкивает вместо него `W`, а второе – выталкивает `S` и заталкивает `A`.

4. Сопоставим каждой лексеме правило АМП, которое читает соответствующий ей знак из входного потока и выталкивает его из стека:

```
>> token_rules = LexicalAnalyzer::GRAMMAR.map do |rule|
     PDARule.new(2, rule[:token], 2, rule[:token], [])
   end
=> [#<struct PDARule …>, #<struct PDARule …>, …]
```

Правила для лексем работают противоположно правилам для символов. Правила для символов стремятся увеличить размер стека, так как иногда заталкивают несколько знаков вместо одного вытолкнутого; правила же для лексем всегда уменьшают размер стека, читая попутно знаки из входного потока.

5. Наконец, создадим правило АМП, которое позволяет автомату перейти в заключительное состояние, если стек оказывается пуст:

```
>> stop_rule = PDARule.new(2, nil, 3, '$', ['$'])
=> #<struct PDARule …>
```

Теперь мы можем построить АМП с такими правилами, подать ему на вход цепочку лексем и посмотреть, сумеет ли он ее распознать. Правила, созданные по грамматике Simple, недетерминирова-

ны – когда на вершине стека находится один из знаков S, L, M, T, мы можем применить несколько правил – и, следовательно, мы имеем НАМП:

```
>> rulebook = NPDARulebook.new([start_rule, stop_rule] + symbol_rules + token_rules)
=> #<struct NPDARulebook rules=[…]>
>> npda_design = NPDADesign.new(1, '$', [3], rulebook)
=> #<struct NPDADesign …>
>> token_string = LexicalAnalyzer.new('while (x < 5) { x = x * 3 }').analyze.join
=> "w(v<n){v=v*n}"
>> npda_design.accepts?(token_string)
=> true
>> npda_design.accepts?(LexicalAnalyzer.new('while (x < 5 x = x * }').analyze.join)
=> false
```

Чтобы было ясно, что происходит, приведем одно из возможных выполнений НАМП, когда на вход подается цепочка `'w(v<n){v=v*n}'`:

Состояние	Заключительное?	Содержимое стека	Осталось прочитать	Действие
1	нет	$	w(v<n){v=v*n}	затолкнуть S, перейти в состояние 2
2	нет	S$	w(v<n){v=v*n}	вытолкнуть S, затолкнуть W
2	нет	W$	w(v<n){v=v*n}	вытолкнуть W, затолкнуть w(E){S}
2	нет	w(E){S}$	w(v<n){v=v*n}	читать w, вытолкнуть w
2	нет	(E){S}$	(v<n){v=v*n}	читать (, вытолкнуть (
2	нет	E){S}$	v<n){v=v*n}	вытолкнуть E, затолкнуть L
2	нет	L){S}$	v<n){v=v*n}	вытолкнуть L, затолкнуть M<L
2	нет	M<L){S}$	v<n){v=v*n}	вытолкнуть M, затолкнуть T
2	нет	T<L){S}$	v<n){v=v*n}	вытолкнуть T, затолкнуть v
2	нет	v<L){S}$	v<n){v=v*n}	читать v, вытолкнуть v
2	нет	<L){S}$	<n){v=v*n}	читать <, вытолкнуть <
2	нет	L){S}$	n){v=v*n}	вытолкнуть L, затолкнуть M
2	нет	M){S}$	n){v=v*n}	вытолкнуть M, затолкнуть T
2	нет	T){S}$	n){v=v*n}	вытолкнуть T, затолкнуть n
2	нет	n){S}$	n){v=v*n}	читать n, вытолкнуть n
2	нет	){S}$	){v=v*n}	читать), вытолкнуть)
2	нет	{S}$	{v=v*n}	читать {, вытолкнуть {
2	нет	S}$	v=v*n}	вытолкнуть S, затолкнуть A
2	нет	A}$	v=v*n}	вытолкнуть A, затолкнуть v=E
2	нет	v=E}$	v=v*n}	читать v, вытолкнуть v
2	нет	=E}$	=v*n}	читать =, вытолкнуть =
2	нет	E}$	v*n}	вытолкнуть E, затолкнуть L
2	нет	L}$	v*n}	вытолкнуть L, затолкнуть M
2	нет	M}$	v*n}	вытолкнуть M, затолкнуть T*M

Состояние	Заключительное?	Содержимое стека	Осталось прочитать	Действие
2	нет	T*M}$	v*n}	вытолкнуть T, затолкнуть v
2	нет	v*M}$	v*n}	читать v, вытолкнуть v
2	нет	*M}$	*n}	читать *, вытолкнуть *
2	нет	M}$	n}	вытолкнуть M, затолкнуть T
2	нет	T}$	n}	вытолкнуть T, затолкнуть n
2	нет	n}$	n}	читать n, вытолкнуть n
2	нет	}$	}	читать }, вытолкнуть }
2	нет	$		перейти в состояние 3
2	да	$		–

Из этой трассировки видно, что автомат переключается между правилами для символов и для лексем: правила для символов раскрывают символ на вершине стека до тех пор, пока он не заменяется лексемой, затем правила для лексем потребляют знаки из стека (и из входного потока), пока на вершине снова не окажется символ. Если входная цепочка может быть порождена правилами грамматики, то в конце концов мы получим пустой стек[1].

Откуда АМП знает, какое правило выбирать на очередном шаге выполнения? Так в этом и состоит сила недетерминированности: наша модель НАМП пробует все возможные правила, поэтому если существует *какой-то* способ получить пустой стек, то мы его найдем.

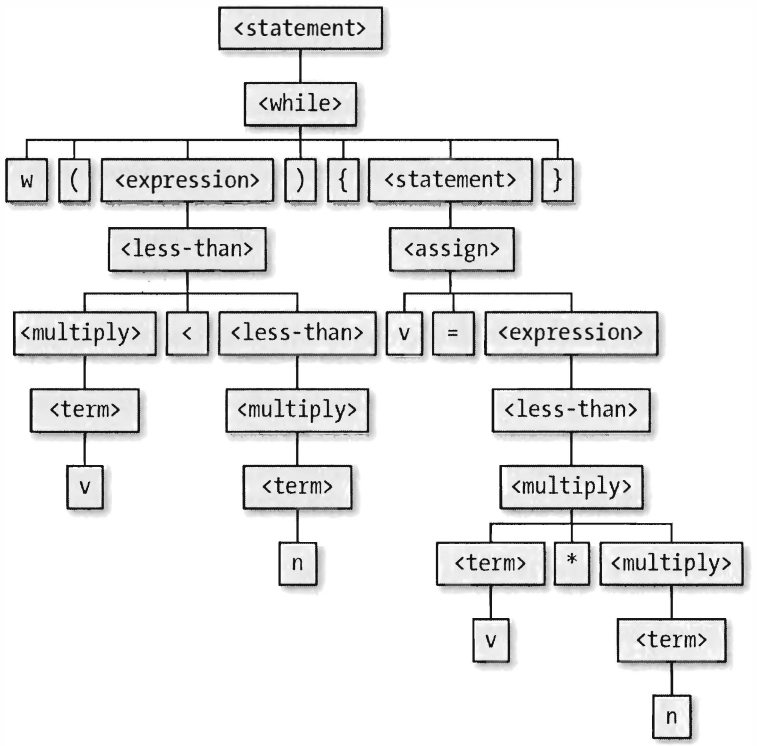
Применение на практике

Эта процедура разбора опирается на недетерминированность, но в реальных приложениях недетерминированности лучше избегать, потому что детерминированный АМП гораздо быстрее и проще для моделирования. По счастью, почти всегда можно устранить недетерминированность, используя сами входные лексемы для решения о том, какое правило для символа применять на каждом шаге (эта техника называется *заглядыванием вперед*), однако при этом трансляция КСГ в АМП усложняется.

Кроме того, просто *распознавать* корректные программы недостаточно. В разделе «Реализация синтаксических анализаторов» на стр. 82 мы видели, что весь смысл разбора программы состоит в том,

[1] Этот алгоритм называется *LL-разбором*. Первая буква L означает «left-to-right» (слева направо), поскольку входная цепочка читается именно в таком направлении, вторая L означает «left derivation» (левый вывод), потому что всегда раскрывается самый левый (то есть самый верхний) символ, находящийся в стеке.

чтобы получить структурированное представление, с которым можно сделать что-то полезное. На практике такое представление можно создать, дополнив модель АМП средствами для протоколирования последовательности правил, которые были выбраны для перехода в допускающее состояние, – этой информации достаточно, чтобы построить дерево разбора. Например, приведенная выше трассировка показывает, какие символы в стеке раскрывались для формирования требуемой последовательности лексем, а это, в свою очередь, определяет форму дерева разбора цепочки `'w(v<n){v=v*n}'`.



Насколько мощнее?

В этой главе мы рассмотрели еще два способа увеличить вычислительную мощность: ДАМП мощнее, чем ДКА и НКА, а НАМП еще мощнее. Похоже, доступ к стеку действительно наделяет автоматы

с магазинной памятью способностью решать более сложные задачи по сравнению с конечными автоматами.

Основное следствие наличия стека – возможность распознавать некоторые языки, которые конечные автоматы распознать не в состоянии, например палиндромы и цепочки сбалансированных скобок. Неограниченная память, предоставляемая стеком, позволяет АМП запоминать во время вычисления произвольный объем информации, а затем обращаться к ней.

В отличие от конечных автоматов, АМП может сколь угодно долго работать в цикле, ничего не читая из входного потока; это любопытно, хотя и не особенно полезно. ДКА способен изменить состояние, только прочитав входной символ, а НКА, хотя и может спонтанно изменить состояние, проследовав по свободному переходу, но лишь конечное число раз, перед тем как вернется туда, откуда начал. С другой стороны, АМП может, находясь в одном состоянии, бесконечно заталкивать в стек символы, никогда не повторяя одну и ту же конфигурацию.

Автоматы с магазинной памятью также могут до некоторой степени управлять своей работой. Существует обратная связь между правилами и стеком – содержимое стека влияет на то, какие правила может выбрать автомат, а выбор правила отражается на содержимом стека. И это позволяет АМП поместить в стек информацию, которая окажет влияние на его будущее выполнение. Конечные автоматы опираются на аналогичную обратную связь между правилами и текущим состоянием, но эта связь гораздо слабее, потому что после изменения прежнее состояние сразу забывается, тогда как заталкивание символов в стек сохраняет и его прежнее содержимое для последующего использования.

Ну хорошо, АМП несколько мощнее, но какие у них ограничения? Даже если нас интересуют только задачи сопоставления с образцом уже рассмотренных видов, автоматы с магазинной памятью все равно подвержены серьезным ограничениям из-за способа работы стека. Из стека нельзя выбрать произвольный элемент, доступен лишь элемент на вершине, поэтому если машина хочет прочитать символ где-то в середине стека, то ей придется сначала вытолкнуть все, что находится выше. Но вытолкнутые символы пропадают навсегда; мы построили АМП, умеющий распознавать цепочки с одинаковым числом `a` и `b`, но приспособить его для распознавания цепочек с одинаковым числом *трех* разных символов (`'abc'`, `'aabbcc'`,

`'aaabbbccc'`, ...) не получится, потому что информация о количестве символов `a` уничтожается в процессе подсчета символов `b`.

Но дисциплина обслуживания «последним пришел, первым пришел», характерная для стека, создает не только проблемы с тем, *сколько раз* можно использовать помещенные в стек символы, но и с *порядком* сохранения и извлечения информации. АМП может распознать палиндромы, но не смог бы распознать удвоенные цепочки типа `'abab'` и `'baaabaaa'`, потому что данные, помещенные в стек, могут быть извлечены только в обратном порядке.

Если отвлечься от конкретной задачи распознавания цепочек и попытаться трактовать эти машины как модель компьютера общего назначения, то выясняется, что ДКА, НКА и АМП еще далеко до настоящей полезности. Начать хотя бы с того, что ни у одной из этих машин нет нормального механизма вывода: они могут сообщить об успешном завершении путем перехода в заключительное состояние, но не в состоянии вывести хотя бы один символ (не говоря уж о цепочке символов) для возврата более содержательных сведений об исходе. Невозможность отправить информацию во внешний мир означает, что с их помощью нельзя реализовать даже простейший алгоритм сложения двух чисел. И подобно конечным автоматам, каждый АМП имеет фиксированную программу; не существует очевидного способа построить АМП, который мог бы как-то прочитать программу из входного потока и выполнить ее.

Принимая во внимание эти слабые места, мы приходим к выводу о необходимости более качественной модели вычислений, которая позволила бы исследовать, на что способны компьютеры. Именно этим мы и займемся в следующей главе.

Глава 5. Окончательная машина

В главах 3 и 4 мы исследовали возможности простых моделей вычислений. Мы видели, как распознавать цепочки возрастающей сложности, как производить сопоставление с регулярными выражениями и как разбирать языки программирования – все с помощью довольно примитивных машин.

Но мы также видели, что у этих машин – конечных автоматов и автоматов с магазинной памятью – имеются серьезные ограничения, которые не позволяют считать их полезными в качестве реалистичных моделей вычислений. Насколько более мощными должны стать наши игрушечные системы, чтобы избавиться от этих ограничений и делать все то, что может нормальный компьютер? Сколько еще сложности необходимо для моделирования поведения ОЗУ, жесткого диска или приемлемого механизма вывода? Что нужно для проектирования машины, которая сможет действительно *запускать программы*, а не просто выполнять одно и то же фиксированное задание?

В 1930-х годах Алан Тьюринг работал именно над этой проблемой. В то время словом «computer» называли вычислителя, то есть человека, обычно женщину, в обязанности которого входило выполнение длинной цепочки трудоемких математических операций. Тьюринг пытался понять и охарактеризовать работу человека-вычислителя, чтобы те же задачи можно было целиком поручить машине. В этой главе мы рассмотрим революционные идеи Тьюринга о том, как спроектировать простейшую «автоматическую машину», способную производить вычисления той же сложности, что и человек.

Детерминированные машины Тьюринга

В главе 1 мы смогли увеличить вычислительную мощность конечного автомата, снабдив его стеком, используемым как внешняя память. По сравнению с конечной внутренней памятью в виде со-

стояний машины у стека имеется несомненное преимущество – он может динамически расти и вмещать любой объем информации, поэтому автомат с магазинной памятью способен решать задачи, в которых требуется хранить произвольное количество данных.

Однако так организованная внешняя память налагает неудобные ограничения на способы использования сохраненных данных. Заменив стек более гибким механизмом хранения, мы сможем снять эти ограничения и еще больше расширить возможности машины.

Память

> Вычисление обычно производится путем записывания определенных символов на листе бумаге. Мы можем предположить, что этот лист разграфлен на квадратики, как школьная тетрадка по арифметике. В элементарной арифметике иногда используется двумерный характер листа. Но такого использования всегда можно избежать, и я думаю, все согласятся, что двухмерность листа не является неотъемлемым свойством вычислений. Я буду предполагать, что вычисление производится на одномерной бумаге, то есть на ленте, разделенной на квадраты.
>
> *– Алан Тьюринг*
> *«О вычислимых числах с приложением к проблеме разрешимости»*

Решение Тьюринга заключалось в том, чтобы снабдить машину чистой лентой неограниченной длины – по существу, одномерным массивом, который может расти в обе стороны по мере необходимости, – и разрешить ей читать и записывать символы в любом месте ленты. Одна и та же лента служит для ввода и хранения данных: ее можно заранее заполнить цепочкой входных символов, которые машина сможет в процессе выполнения читать и перезаписывать.

Машина с конечным числом состояний, имеющая доступ к бесконечно длинной ленте, называется *машиной Тьюринга* (МТ). Обычно этот термин применяется для машины с детерминированными правилами, но, чтобы избежать всякой двусмысленности, мы можем назвать такую машину *детерминированной машиной Тьюринга* (ДМТ).

Мы уже знаем, что автомат с магазинной памятью может получить доступ только к одной ячейке в своей внешней памяти – вершине стека – но для машины Тьюринга это представляется слишком

сильным ограничением. Сама идея ленты заключается в том, чтобы на ней можно было хранить данные любого объема и читать их в любом порядке. Но как сконструировать машину, которая сможет взаимодействовать со всей лентой целиком?

Один из вариантов – сделать ленту произвольно адресуемой, пометив каждый квадратик уникальным числовым адресом, как это делается в оперативной памяти компьютера; тогда машина сможет получить прямой доступ к любой ячейке для чтения или записи. Но это сложнее, чем в действительности необходимо, поскольку придется разрешить различные технические вопросы, например: как назначать адреса всем квадратикам на бесконечной ленте и как машина должна задавать адрес квадратика, к которому хочет обратиться.

Традиционная машина Тьюринга устроена проще: имеется *головка ленты*, которая указывает на конкретную ячейку и может читать или записывать символ в этой ячейке и никакой другой. Головку можно смещать влево или вправо на одну ячейку после каждого шага вычисления, то есть машина Тьюринга должна без устали перемещать головку взад-вперед по ленте, чтобы добраться до отдаленных ячеек. Медленность перемещения головки не мешает машине получить доступ к любым данным на ленте, а отражается только на времени доступа, однако это приемлемая плата за сохранение простоты.

Наличие доступа к ленте позволяет решать новые классы задач, не ограничиваясь допуском или отвержением цепочек. Например, можно построить ДМТ для инкремента двоичного числа по месту. Для этого нужно знать, как инкрементировать одну *цифру* двоичного числа, но это, к счастью, просто: если цифра – нуль, заменить ее единицей, а если единица – заменить ее нулем и точно так же инкрементировать цифру слева («перенос единицы»). Машина Тьюринга просто должна использовать эту процедуру для инкремента самой правой цифры двоичного числа, а затем вернуть головку ленты в исходное положение.

- Наделить машину тремя состояниями 1, 2 и 3, сделав состояние 3 заключительным.
- Начать работу в состоянии 1, установив головку ленты над самой правой цифрой двоичного числа.
- Если машина находится в состоянии 1 и прочитан нуль (или пустой квадратик), заменить нуль единицей, переместить головку вправо и перейти в состояние 2.

- Если машина находится в состоянии 1 и прочитана единица, заменить единицу нулем и переместить головку влево.
- Если машина находится в состоянии 2 и прочитан нуль или единица, переместить головку вправо.
- Если машина находится в состоянии 2 и прочитан пустой квадратик, переместить головку влево и перейти в состояние 3.

Эта машина находится в состоянии 1, когда пытается инкрементировать цифру, в состоянии 2, когда перемещается назад в исходное положение, и в состоянии 3, когда работа закончена. Ниже приведена трассировка выполнения для случая, когда лента исходно содержит цепочку `'1011'`; символ под головкой заключен в скобки, а подчеркиваниями обозначены пустые квадратики по обе стороны от входной цепочки.

Состояние	Заключительное?	Содержимое ленты	Действие
1	нет	_101(1)__	записать 0, сдвинуть влево
1	нет	__10(1)0_	записать 0, сдвинуть влево
1	нет	___1(0)00	записать 1, сдвинуть вправо, перейти в состояние 2
2	нет	__11(0)0_	сдвинуть вправо
2	нет	_110(0)__	сдвинуть вправо
2	нет	1100(_)__	сдвинуть влево, перейти в состояние 3
3	да	_110(0)__	–

Перемещать головку в исходное положение, строго говоря, необязательно – если бы мы сделали состояние 2 заключительным, то машина остановилась бы сразу после успешной замены нуля единицей, и лента все равно содержала бы правильный результат – но это поведение желательно, поскольку оставляет головку в положении, когда машину можно перезапустить, просто вернув ее в состояние 1. А запуская машину несколько раз подряд, мы можем последовательно инкрементировать записанное на ленте число. Эту функциональность можно было бы повторно использовать в составе более сложной машины, например, для сложения или умножения двух двоичных чисел.

Правила

> Представим себе, что операции, выполняемые вычислителем, разбиты на «простые операции», настолько элементарные, что трудно представить, как их можно еще разбить.[...] Фактически выполняемая операция определяется [...] состоянием мозга вычислителя и наблюдаемыми символами. В частности, они определяют состояние мозга вычислителя после выполнения операции. Теперь мы можем сконструировать машину, которая проделает работу вычислителя.
>
> – Алан Тьюринг
> «О вычислимых числах с приложением к проблеме разрешимости»

Существует несколько «простых операций», которые машина Тьюринга должна уметь выполнять на каждом шаге вычисления: прочитать символ в ячейке под головкой ленты, записать новый символ в эту ячейку, сдвинуть головку влево или вправо и изменить состояние. Вместо того чтобы задавать правила разного вида для каждого из этих действий, мы можем упростить себе жизнь и придумать единый достаточно гибкий формат правила, как это было сделано для автоматов с магазинной памятью.

В таком унифицированном формате будет пять частей:

- текущее состояние машины;
- символ, который должен находиться в ячейке под головкой ленты;
- следующее состояние машины;
- символ, который нужно записать в ячейку под головкой ленты;
- направление (влево или вправо) перемещения головки после записи на ленту.

Здесь предполагается, что машина Тьюринга изменяет состояние и записывает символ на ленту всякий раз, как следует какому-то правилу. Как обычно, мы можем сделать «следующее состояние» совпадающим с текущим, если хотим, чтобы правило не изменяло состояние; аналогично, если мы хотим, чтобы правило не изменяло содержимое ленты, то можем просто записать тот же символ, который прочитали.

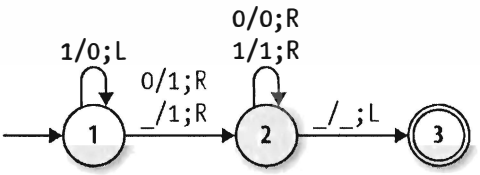
Мы также предполагаем, что головка ленты обязательно сдвигается после каждого шага. Поэтому невозможно написать одно правило, которое изменяет состояние машины или содержимое

ленты, не сдвигая головку, но того же эффекта можно достичь, написав два правила: одно производит требуемое изменение, а второе возвращает головку в исходное положение.

Машина Тьюринга для инкремента двоичного числа имеет шесть правил, если записывать их в таком формате:

- если находится в состоянии 1 и прочитан нуль, записать единицу, сдвинуть вправо и перейти в состояние 2;
- если находится в состоянии 1 и прочитана единица, записать нуль, сдвинуть влево и перейти в состояние 1;
- если находится в состоянии 1 и прочитан пустой квадратик, записать единицу, сдвинуть вправо и перейти в состояние 2;
- если находится в состоянии 2 и прочитан нуль, записать нуль, сдвинуть вправо и остаться в состояние 2;
- если находится в состоянии 2 и прочитана единица, записать единицу, сдвинуть вправо и остаться в состояние 2;
- если находится в состоянии 2 и прочитан пустой квадратик, записать пустой квадратик, сдвинуть влево и перейти в состояние 3.

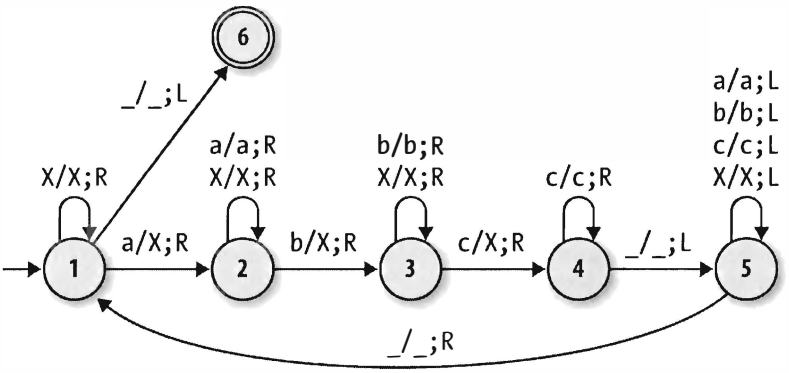
Мы можем изобразить состояния и правила этой машины на диаграмме, похожей на те, которыми пользовались для конечных автоматов и автоматов с магазинной памятью.



От диаграммы ДКА эта отличается только метками на стрелках. Метка вида `a/b;L` означает, что с ленты прочитан символ `a`, записан символ `b`, после чего головка сдвинулась на одну позицию влево. Правило, помеченное `a/b;R`, делает почти то же самое, только головка сдвигается вправо.

Теперь посмотрим, как воспользоваться машиной Тьюринга для решения задачи о распознавании, оказавшейся не по силам автомату с магазинной памятью: определить входные цепочки, содержащие один или несколько символов `a`, за которыми следует столько же символов `b`, а затем столько же символов `c` (например, `'aaabbbccc'`).

У машины Тьюринга, решающей эту задачу, имеется 6 состояний и 16 правил:



Работает она следующим образом.

1. Просматривать входную строку, повторно сдвигая головку вправо, пока не будет найден символ a, после чего вычеркнуть его, заменив символом X (состояние 1).
2. Сдвигая головку вправо, найти символ b и вычеркнуть его (состояние 2).
3. Сдвигая головку вправо, найти символ c и вычеркнуть его (состояние 3).
4. Сдвигая головку вправо, найти конец входной цепочки (состояние 4), а затем двигать головку влево, пока не будет найдено начало цепочки (состояние 5).
5. Повторять, начиная с шага 1, пока не будут вычеркнуты все символы.

Если входная строка содержит один или более символов a, за которыми следует столько же символов b и c, то машина совершит несколько проходов по цепочке, вычеркивая по одному символу каждого вида на каждом проходе, а затем – когда вся цепочка будет вычеркнута – перейдет в заключительное состояние. Ниже показана трассировка выполнения для входной цепочки `'aabbcc'`:

Состояние	Заключительное?	Содержимое ленты	Действие
1	нет	_____(a)abbcc_	записать X, сдвинуть вправо, перейти в состояние 2
2	нет	_____X(a)bbcc__	сдвинуть вправо

Состояние	Заключительное?	Содержимое ленты	Действие
2	нет	___Xa(b)bcc___	записать X, сдвинуть вправо, перейти в состояние 3
3	нет	___XaX(b)cc___	сдвинуть вправо
3	нет	__XaXb(c)c____	записать X, сдвинуть вправо, перейти в состояние 4
4	нет	_XaXbX(c)_____	сдвинуть вправо
4	нет	XaXbXc(_)_____	сдвинуть влево, перейти в состояние 5
5	нет	_XaXbX(c)_____	сдвинуть влево
5	нет	__XaXb(X)c____	сдвинуть влево
5	нет	___XaX(b)Xc___	сдвинуть влево
5	нет	____Xa(X)bXc___	сдвинуть влево
5	нет	_____X(a)XbXc__	сдвинуть влево
5	нет	______(X)aXbXc_	сдвинуть влево
5	нет	______(_)XaXbXc	сдвинуть вправо, перейти в состояние 1
1	нет	______(X)aXbXc_	сдвинуть вправо
1	нет	_____X(a)XbXc__	записать X, сдвинуть вправо, перейти в состояние 2
2	нет	____XX(X)bXc___	сдвинуть вправо
2	нет	___XXX(b)Xc___	записать X, сдвинуть вправо, перейти в состояние 3
3	нет	__XXXX(X)c____	сдвинуть вправо
3	нет	_XXXXX(c)_____	записать X, сдвинуть вправо, перейти в состояние 4
4	нет	XXXXXX(_)_____	сдвинуть влево, перейти в состояние 5
5	нет	_XXXXX(X)_____	сдвинуть влево
5	нет	__XXXX(X)X____	сдвинуть влево
5	нет	___XXX(X)XX___	сдвинуть влево
5	нет	____XX(X)XXX___	сдвинуть влево
5	нет	_____X(X)XXXX__	сдвинуть влево
5	нет	______(X)XXXXX_	сдвинуть влево
5	нет	______(_)XXXXXX	сдвинуть вправо, перейти в состояние 1
1	нет	______(X)XXXXX_	сдвинуть вправо
1	нет	_____X(X)XXXX__	сдвинуть вправо
1	нет	____XX(X)XXX___	сдвинуть вправо
1	нет	___XXX(X)XX____	сдвинуть вправо
1	нет	__XXXX(X)X_____	сдвинуть вправо
1	нет	_XXXXX(X)______	сдвинуть вправо
1	нет	XXXXXX(_)______	сдвинуть влево, перейти в состояние 6
6	да	_XXXXX(X)______	–

Эта машина работает благодаря точному выбору правил на стадиях просмотра. Например, пока машина находится в состоянии 3 –

просматривает ленту вправо и ищет символ c – она располагает только правилами для перемещения головки мимо символов b и X. Встретив какой-нибудь другой символ (например, неожиданный символ a), для которого у нее нет правила, машина перейдет в неявное состояние заклинивания и прекратит работу, то есть отвергнет входную цепочку.

Мы ввели упрощающее предположение о том, что входная цепочка может содержать только символы a, b и c. Если это не так, показанная машина будет работать неправильно; например, она допускает цепочку 'XaXXbXXXc', хотя должна была бы отвергнуть. Чтобы поправить дело, нужно было бы ввести дополнительные состояния и правила для просмотра всей цепочки на предмет отсутствия неожиданных символов еще до того, как машина начнет что-то вычеркивать.

Детерминированность

Чтобы машина Тьюринга была детерминированной, она должна подчиняться тем же ограничениям, что и детерминированный автомат с магазинной памятью (см. раздел «Детерминированность» на стр. 144), хотя на этот раз нам нет нужды беспокоиться о свободных переходах, так как в машинах Тьюринга их не бывает.

Следующее действие машина Тьюринга выбирает исходя из текущего состояния и символа, находящегося под головкой ленты, поэтому у детерминированной машины может быть только одно правило для каждой комбинации состояния и символа – ограничение «непротиворечивости» – чтобы предотвратить неоднозначность при выборе следующего действия. Для простоты мы ослабим ограничение «полноты», как делали и в случае ДАМП, и будем предполагать, что существует неявное состояние заклинивания, в которое машина может перейти, когда ни одно правило не применимо, не настаивая на наличии правила для каждой потенциально возможной ситуации.

Моделирование

Теперь, получив ясное представление о том, как должна работать детерминированная машина Тьюринга, построим на Ruby модель, чтобы понаблюдать за ней в действии.

Первым делом следует реализовать ленту машины Тьюринга. Очевидно, что мы должны хранить символы, записанные на ленту, но еще требуется запоминать текущую позицию головки, чтобы

модель машины могла прочитать текущий символ, записать новый символ в текущую ячейку и сдвинуть головку влево или вправо.

Элегантное решение заключается в том, чтобы разбить ленту на три участка – все символы слева от головки, один символ под головкой и все символы справа от головки – и хранить каждый участок отдельно. Тогда становится очень просто читать и записывать текущий символ, а сдвиг головки реализуется перемещением символов между тремя участками; например, сдвиг на одну позицию вправо означает, что текущий символ становится последним символом слева от головки, а первый символ справа от головки становится текущим.

Наша реализация должна поддерживать иллюзию, будто лента бесконечна и заполнена пустыми квадратиками, но, по счастью, нам не понадобится для этого бесконечно большая структура данных. В каждый момент времени читать можно только ячейку, находящуюся под головкой, поэтому достаточно просто сделать так, чтобы там оказался пустой квадратик, когда головка сдвинется за пределы области, занятой конечным числом непустых символов, уже записанных на ленту. Чтобы осуществить это, мы должны заранее договориться, какой символ представляет пустой квадратик и подсовывать его под головку, когда она попадет в еще неисследованную часть ленты.

Таким образом, базовая реализация ленты выглядит следующим образом:

```
class Tape < Struct.new(:left, :middle, :right, :blank)
  def inspect
    «#<Tape #{left.join}(#{middle})#{right.join}>»
  end
end
```

Это уже позволяет создать ленту и прочитать символ под головкой:

```
>> tape = Tape.new(['1', '0', '1'], '1', [], '_')
=> #<Tape 101(1)>
>> tape.middle
=> "1"
```

Добавим операции для записи символа в текущую ячейку и сдвига головки влево и вправо[1]:

[1] Класс `Tape`, как и `Stack`, чисто функциональный: запись на ленту и сдвиг головки – недеструктивные операции, то есть они возвращают новый объект `Tape`, а не модифицируют существующий.

```
class Tape
  def write(character)
    Tape.new(left, character, right, blank)
  end

  def move_head_left
    Tape.new(left[0..-2], left.last || blank, [middle] + right, blank)
  end

  def move_head_right
    Tape.new(left + [middle], right.first || blank, right.drop(1), blank)
  end
end
```

Теперь мы можем писать на ленту и двигать головку в обе стороны:

```
>> tape
=> #<Tape 101(1)>
>> tape.move_head_right
=> #<Tape 10(1)1>
>> tape.write('0')
=> #<Tape 101(0)>
>> tape.move_head_right
=> #<Tape 1011(_)>
>> tape.move_head_right.write('0')
=> #<Tape 1011(0)>
```

В главе 4 мы употребляли слово *конфигурация* для обозначения комбинации состояния и стека автомата с магазинной памятью, та же идея полезна и в данном случае. Мы можем сказать, что *конфигурацией машины Тьюринга* называется комбинация состояния и ленты, и реализовать правила машины Тьюринга, работающие непосредственно с такими конфигурациями:

```
class TMConfiguration < Struct.new(:state, :tape)
end

class TMRule < Struct.new(:state, :character, :next_state,
                          :write_character, :direction)
  def applies_to?(configuration)
    state == configuration.state && character == configuration.tape.middle
  end
end
```

Правило применимо только тогда, когда текущее состояние машины и символ под головкой совпадают с ожидаемыми:

```
>> rule = TMRule.new(1, '0', 2, '1', :right)
=> #<struct TMRule
```

```
        state=1,
        character="0",
        next_state=2,
        write_character="1",
        direction=:right
    >
>> rule.applies_to?(TMConfiguration.new(1, Tape.new([], '0', [], '_')))
=> true
>> rule.applies_to?(TMConfiguration.new(1, Tape.new([], '1', [], '_')))
=> false
>> rule.applies_to?(TMConfiguration.new(2, Tape.new([], '0', [], '_')))
=> false
```

Получив средства проверить, что правило применимо к данной конфигурации, мы должны научиться обновлять конфигурацию путем записи нового символа, сдвига головки и изменения состояния машины в соответствии с выбранным правилом.

```
class TMRule
  def follow(configuration)
    TMConfiguration.new(next_state, next_tape(configuration))
  end

  def next_tape(configuration)
    written_tape = configuration.tape.write(write_character)
    case direction
    when :left
      written_tape.move_head_left
    when :right
      written_tape.move_head_right
    end
  end
end
```

Этот код, похоже, работает правильно:

```
>> rule.follow(TMConfiguration.new(1, Tape.new([], '0', [], '_')))
=> #<struct TMConfiguration state=2, tape=#<Tape 1(_)>>
```

Класс `DTMRulebook` реализуется почти так же, как `DFARulebook` и `DPDARulebook` с тем исключением, что у метода `#next_configuration` нет аргумента `character`, так как никакого внешнего источника входных данных не существует (есть только лента, а она уже и так является частью конфигурации):

```
class DTMRulebook < Struct.new(:rules)
  def next_configuration(configuration)
    rule_for(configuration).follow(configuration)
  end

  def rule_for(configuration)
```

```
    rules.detect { |rule| rule.applies_to?(configuration) }
  end
end
```

Теперь можно создать объект `DTMRulebook` для машины Тьюринга по «инкременту двоичных чисел» и воспользоваться им для ручного прогона нескольких конфигураций:

```
>> rulebook = DTMRulebook.new([
     TMRule.new(1, '0', 2, '1', :right),
     TMRule.new(1, '1', 1, '0', :left),
     TMRule.new(1, '_', 2, '1', :right),
     TMRule.new(2, '0', 2, '0', :right),
     TMRule.new(2, '1', 2, '1', :right),
     TMRule.new(2, '_', 3, '_', :left)
   ])
=> #<struct DTMRulebook rules=[…]>
>> configuration = TMConfiguration.new(1, tape)
=> #<struct TMConfiguration state=1, tape=#<Tape 101(1)>>
>> configuration = rulebook.next_configuration(configuration)
=> #<struct TMConfiguration state=1, tape=#<Tape 10(1)0>>
>> configuration = rulebook.next_configuration(configuration)
=> #<struct TMConfiguration state=1, tape=#<Tape 1(0)00>>
>> configuration = rulebook.next_configuration(configuration)
=> #<struct TMConfiguration state=2, tape=#<Tape 11(0)0>>
```

Удобно обернуть все это в класс `DTM`, включив в него методы `#step` и `#run`, как мы поступали при реализации семантики мелких шагов в главе 2:

```
class DTM < Struct.new(:current_configuration, :accept_states, :rulebook)
  def accepting?
    accept_states.include?(current_configuration.state)
  end

  def step
    self.current_configuration =
      rulebook.next_configuration(current_configuration)
  end

  def run
    step until accepting?
  end
end
```

Вот теперь у нас имеется работающая модель детерминированной машины Тьюринга, так что можно подать ей что-нибудь на вход и посмотреть, что из этого выйдет:

```
>> dtm = DTM.new(TMConfiguration.new(1, tape), [3], rulebook)
=> #<struct DTM …>
>> dtm.current_configuration
```

```
=> #<struct TMConfiguration state=1, tape=#<Tape 101(1)>>
>> dtm.accepting?
=> false
>> dtm.step; dtm.current_configuration
=> #<struct TMConfiguration state=1, tape=#<Tape 10(1)0>>
>> dtm.accepting?
=> false
>> dtm.run
=> nil
>> dtm.current_configuration
=> #<struct TMConfiguration state=3, tape=#<Tape 110(0)_>>
>> dtm.accepting?
=> true
```

Как и в случае модели ДАМП, необходимо кое-что подправить для аккуратной обработки заклинивания машины Тьюринга:

```
>> tape = Tape.new(['1', '2', '1'], '1', [], '_')
=> #<Tape 121(1)>
>> dtm = DTM.new(TMConfiguration.new(1, tape), [3], rulebook)
=> #<struct DTM …>
>> dtm.run
NoMethodError: undefined method `follow' for nil:NilClass
```

На это раз нам не понадобится специальное представление состояния заклинивания. В отличие от АМП, у машины Тьюринга нет внешних входных данных, поэтому для того чтобы сказать, заклинило ее или нет, достаточно взглянуть на свод правил и текущую конфигурацию:

```
class DTMRulebook
  def applies_to?(configuration)
    !rule_for(configuration).nil?
  end
end

class DTM
  def stuck?
    !accepting? && !rulebook.applies_to?(current_configuration)
  end

  def run
    step until accepting? || stuck?
  end
end
```

Теперь модель обнаруживает, что машину заклинило и останавливается автоматически:

```
>> dtm = DTM.new(TMConfiguration.new(1, tape), [3], rulebook)
=> #<struct DTM …>
```

```
>> dtm.run
=> nil
>> dtm.current_configuration
=> #<struct TMConfiguration state=1, tape=#<Tape 1(2)00>>
>> dtm.accepting?
=> false
>> dtm.stuck?
=> true
```

Интереса ради мы приводим ниже уже рассмотренную выше машину Тьюринга для распознавания цепочек вида `'aaabbbccc'`:

```
>> rulebook = DTMRulebook.new([
      # состояние 1: просмотр вправо в поисках a
      TMRule.new(1, 'X', 1, 'X', :right),  # пропустить X
      TMRule.new(1, 'a', 2, 'X', :right),  # вычеркнуть a, перейти в сост. 2
      TMRule.new(1, '_', 6, '_', :left),   # найти пустой, перейти в сост. 6
                                           # (заключительное)

      # состояние 2: просмотр вправо в поисках b
      TMRule.new(2, 'a', 2, 'a', :right),  # пропустить a
      TMRule.new(2, 'X', 2, 'X', :right),  # пропустить X
      TMRule.new(2, 'b', 3, 'X', :right),  # вычеркнуть b, перейти в сост. 3

      # состояние 3: просмотр вправо в поисках c
      TMRule.new(3, 'b', 3, 'b', :right),  # пропустить b
      TMRule.new(3, 'X', 3, 'X', :right),  # пропустить X
      TMRule.new(3, 'c', 4, 'X', :right),  # вычеркнуть c, перейти в сост. 4

      # состояние 4: просмотр вправо в поисках конца цепочки
      TMRule.new(4, 'c', 4, 'c', :right),  # пропустить c
      TMRule.new(4, '_', 5, '_', :left),   # найти пустой, перейти в сост. 5

      # состояние 5: просмотр влево в поисках начала цепочки
      TMRule.new(5, 'a', 5, 'a', :left),   # пропустить a
      TMRule.new(5, 'b', 5, 'b', :left),   # пропустить b
      TMRule.new(5, 'c', 5, 'c', :left),   # пропустить c
      TMRule.new(5, 'X', 5, 'X', :left),   # пропустить X
      TMRule.new(5, '_', 1, '_', :right)   # найти пустой, перейти в сост. 1
    ])
=> #<struct DTMRulebook rules=[…]>
>> tape = Tape.new([], 'a', ['a', 'a', 'b', 'b', 'b', 'c', 'c', 'c'], '_')
=> #<Tape (a)aabbbccc>
>> dtm = DTM.new(TMConfiguration.new(1, tape), [6], rulebook)
=> #<struct DTM …>
>> 10.times { dtm.step }; dtm.current_configuration
=> #<struct TMConfiguration state=5, tape=#<Tape XaaXbbXc(c)_>>
>> 25.times { dtm.step }; dtm.current_configuration
=> #<struct TMConfiguration state=5, tape=#<Tape _XXa(X)XbXXc_>>
>> dtm.run; dtm.current_configuration
=> #<struct TMConfiguration state=6, tape=#<Tape _XXXXXXXX(X)_>>
```

Построить эту реализацию оказалось несложно – моделировать машину Тьюринга вообще просто при условии, что имеются струк-

туры данных для представления ленты и свода правил. Оно и понятно, ведь Алан Тьюринг и хотел сделать ее максимально простой, чтобы было легко сконструировать и рассуждать о поведении. Ниже мы увидим (в разделе «Универсальные машины» на стр. 196), что простота реализации – это важное свойство.

Недетерминированные машины Тьюринга

В разделе «Эквивалентность» на стр. 124 мы видели, что недетерминированность не расширяет возможности конечного автомата, а в разделе «Неэквивалентность» на стр. 159 – что недетерминированный автомат с магазинной памятью умеет больше, чем детерминированный. Тогда возникает очевидный вопрос о машинах Тьюринга: верно ли, что недетерминированность[1] увеличивает их вычислительную мощность?

В данном случае ответ отрицательный: недетерминированная машина Тьюринга может не больше, чем детерминированная. Автоматы с магазинной памятью в этом смысле являются исключением, так как и ДКА и ДМТ обладают достаточной мощностью для моделирования своих недетерминированных собратьев. С помощью единственного состояния конечного автомата можно представить сочетание многих состояний, а на одной ленте машины Тьюринга можно хранить содержимое многих лент. Однако стек автомата с магазинной памятью не позволяет представить несколько возможных стеков одновременно.

Итак, как и в случае конечных автоматов, детерминированная машина Тьюринга способна смоделировать недетерминированную. Для этого лента используется для хранения очереди подходящим образом закодированных конфигураций машины Тьюринга, каждая из которых содержит возможные текущее состояние и ленту моделируемой машины. На каждом шаге вычисления модель читает конфигурацию, находящуюся в начале очереди, находит все применимые к ней правила и использует каждое правило для порождения новой конфигурации, которая записывается на ленту в конец оче-

---

[1] Для машины Тьюринга «недетерминированность» означает, что для данной комбинации состояния и символа под головкой может существовать несколько правил, то есть из одной начальной конфигурации возможно несколько путей выполнения.

реди. После того как это проделано для всех применимых правил, конфигурация, находившаяся в начале очереди, стирается, и весь процесс начинается заново для следующей конфигурации в очереди. Этот шаг моделируемой машины повторяется, пока в начале очереди не окажется конфигурация, представляющая машину, которая достигла заключительного состояния.

Эта техника позволяет детерминированной машине Тьюринга исследовать все возможные конфигурации моделируемой машины, применяя поиск в ширину; если для недетерминированной машины существует хотя бы один способ перейти в заключительное состояние, то модель его найдет, даже если другие пути выполнения ведут к бесконечным циклам. Фактическая реализация модели в виде свода правил требует рассмотрения многочисленных деталей, поэтому мы опустим ее, но сам тот факт, что это возможно, означает, что не получится сделать машину Тьюринга более мощной одним лишь разрешением недетерминированности.

Максимальная мощность

Детерминированные машины Тьюринга знаменуют драматический переход от ограниченных вычислительных машин к полнофункциональным. На самом деле, любая попытка модернизировать определение машины Тьюринга, сделав ее более мощной, обречена на провал, потому они уже способны *смоделировать* любое потенциальное усовершенствование[1]. Хотя добавление некоторых черт может уменьшить размер машин Тьюринга или сделать их более эффективными, придать им принципиально новые возможности никак не получится.

Мы уже видели, почему это так в случае недетерминированности. Рассмотрим еще четыре расширения традиционных машин Тьюринга – внутренняя память, подпрограммы, несколько лент и многомерная лента – и покажем, почему ни одно из них не дает увеличения вычислительной мощности. Правда, некоторые алгоритмы моделирования сложны, но в конце концов это всего лишь вопрос программирования.

---

[1] Строго говоря, это утверждение справедливо только для усовершенствований, которые мы знаем, как реализовать. Машина Тьюринга *стала* бы более мощной, если бы мы наделили ее магической способностью мгновенно выводить ответы на вопросы, на которые ни одна традиционная машина Тьюринга ответить не может (см. главу 8), но на практике сделать это невозможно.

Внутренняя память

Построение свода правил для машины Тьюринга может оказаться утомительным делом из-за отсутствия внутренней памяти. Например, нам часто нужно, чтобы машина могла переместить головку ленты к конкретной ячейке, прочитать хранящийся в ней символ, затем переместиться в другую часть ленты и выполнить действие, зависящее от прочитанного ранее символа. На первый взгляд, это кажется невозможно, потому что машине негде «запомнить» символ; он, конечно, по-прежнему записан на ленте, и мы можем переместить головку назад и снова прочитать его, но как только головка сдвинется с этой ячейки, мы уже не сможем выбрать правило, исходя из ее содержимого.

Было бы удобнее, если бы у машины Тьюринга была какая-то временная внутренняя память – назовите ее «ОЗУ», «регистры», «локальные переменные» или еще как-нибудь – где можно было бы сохранить символ, прочитанный из ячейки на ленте, а позже обратиться к нему, даже если головка ленты сместилась совершенно в другое место. На самом деле, если бы машина Тьюринга обладала такой возможностью, то можно было бы не ограничивать ее хранением только прочитанных с ленты символов, а разрешить хранение любой относящейся к делу информации, например, промежуточных результатов вычислений. Это освободило бы нас от утомительной необходимости гонять головку туда-сюда, чтобы записать на ленту обрывки данных. Казалось бы, такая дополнительная гибкость должна позволить машине Тьюринга решать некоторые новые задачи.

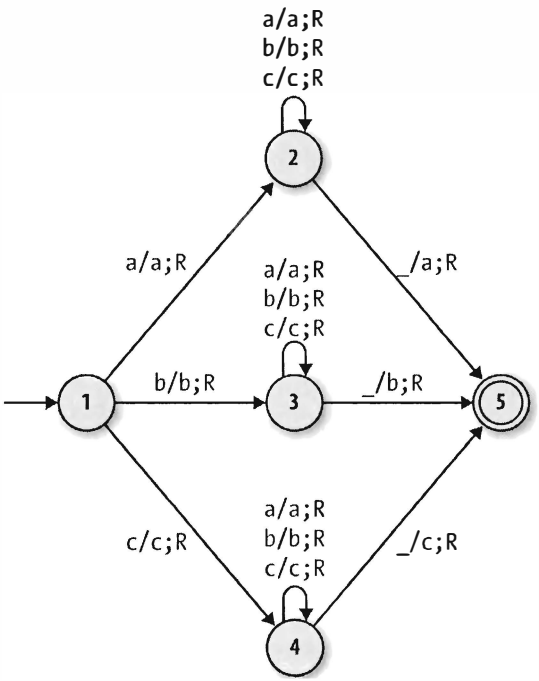
Но, как и в случае недетерминированности, оснащение машины Тьюринга дополнительной внутренней памятью, хотя и несомненно упрощает решение определенного вида задач, но не наделяет ее возможностью делать что-то такое, что было невозможно раньше. От желания хранить промежуточные результаты внутри машины, а не на ленте, можно сравнительно безболезненно отказаться, потому что для этой цели вполне можно задействовать ленту, пусть даже на перемещение головки к нужной ячейке и обратно затрачивается время. Однако к вопросу о запоминании символа стоит отнестись серьезно, потому что машина Тьюринга оказалась бы очень ограниченной в своих возможностях, если бы не могла использовать содержимое ячейки после перемещения головки в другое место.

По счастью, у машины Тьюринга уже имеется отличная внутренняя память – ее текущее состояние. Число состояний не ограничено

сверху, хотя для любого конкретного набора правил оно должно быть конечным и определено заранее, потому что не существует способа создать новое состояние во время вычисления. Если необходимо, мы можем построить машину с сотней, тысячей, даже миллиардом состояний и использовать текущее состояние для сохранения произвольного объема информации между шагами.

Это неизбежно означает, что некоторые правила будут дублироваться, подстраиваясь под различные состояния, назначение которых совпадает во всем, кроме информации, которую они «запоминают». Вместо одного состояния, которое означает «просмотр вправо в поисках пустого квадратика», машина может иметь такие состояния: «просмотр вправо в поисках пустого квадратика (памятуя о том, что раньше я прочла a)», «просмотр вправо в поисках пустого квадратика (памятуя о том, что раньше я прочла b)» и т. д. для всех возможных символов – впрочем, число символов тоже конечно, так что и у дублирования есть предел.

Вот простая машина Тьюринга, в которой эта техника применяется для копирования символа из начала цепочки в ее конец:



```
>> rulebook = DTMRulebook.new([
       # состояние 1: читать первый символ с ленты
       TMRule.new(1, 'a', 2, 'a', :right), # запомнить a
       TMRule.new(1, 'b', 3, 'b', :right), # запомнить b
       TMRule.new(1, 'c', 4, 'c', :right), # запомнить c

       # состояние 2: просмотр вправо в поисках конца строки (запомнив a)
       TMRule.new(2, 'a', 2, 'a', :right), # пропустить a
       TMRule.new(2, 'b', 2, 'b', :right), # пропустить b
       TMRule.new(2, 'c', 2, 'c', :right), # пропустить c
       TMRule.new(2, '_', 5, 'a', :right), # найти пустой, записать a

       # состояние 3: просмотр вправо в поисках конца строки (запомнив b)
       TMRule.new(3, 'a', 3, 'a', :right), # пропустить a
       TMRule.new(3, 'b', 3, 'b', :right), # пропустить b
       TMRule.new(3, 'c', 3, 'c', :right), # пропустить c
       TMRule.new(3, '_', 5, 'b', :right), # найти пустой, записать b

       # состояние 4: просмотр вправо в поисках конца строки (запомнив c)
       TMRule.new(4, 'a', 4, 'a', :right), # пропустить a
       TMRule.new(4, 'b', 4, 'b', :right), # пропустить b
       TMRule.new(4, 'c', 4, 'c', :right), # пропустить c
       TMRule.new(4, '_', 5, 'c', :right)  # найти пустой, записать c
    ])
=> #<struct DTMRulebook rules=[…]>
>> tape = Tape.new([], 'b', ['c', 'b', 'c', 'a'], '_')
=> #<Tape (b)cbca>
>> dtm = DTM.new(TMConfiguration.new(1, tape), [5], rulebook)
=> #<struct DTM …>
>> dtm.run; dtm.current_configuration.tape
=> #<Tape bcbcab(_)>
```

Состояния 2, 3 и 4 этой машины отличаются только тем, что соответствуют разным символам в начале строки и соответственно по достижении конца строки в них выполняются чуть различающиеся действия.

Эта машина работает только для цепочек, составленных из символов a, b и c; чтобы заставить ее работать для цепочек, содержащих *любые* буквы (или буквы и цифры, или еще больший набор символов), пришлось бы добавить еще кучу состояний – по одному для каждого запоминаемого символа – и соответствующие им правила.

Такое применение текущего состояния позволяет строить машины Тьюринга, которые могут запомнить любую конечную комбинацию фактов лишь с помощью перемещения головки ленты. По существу, мы получаем те же возможности, которые дает машина с явными «регистрами» для внутренней памяти, но ценой увеличения числа состояний.

Подпрограммы

Свод правил машины Тьюринга – это длинный фиксированный перечень инструкций очень низкого уровня; писать эти правила, не теряя из виду высокоуровневую задачу, стоящую перед машиной, трудно. Строить свод правил стало бы куда проще, умей мы каким-то образом вызывать *подпрограммы*: если бы в некоторой части машины можно было сохранить все правила, к примеру, для инкрементирования числа, то в своде правил мы могли бы просто сказать «а теперь инкрементируй число», вместо того чтобы вручную выписывать инструкции для этой цели. И, чем черт не шутит, вдруг эта дополнительная гибкость позволит конструировать машины с новыми возможностями.

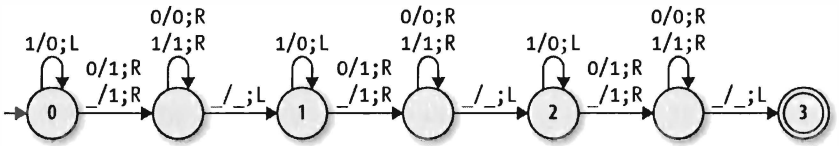
Увы, и эта функциональность – не более чем удобство, ничего принципиально нового она не дает. Как и конечные автоматы, реализующие отдельные фрагменты регулярного выражения (см. раздел «Семантика» на стр. 112), несколько небольших машин Тьюринга можно соединить в одну более крупную, в которой каждая внутренняя машина будет играть роль подпрограммы. Состояния и правила рассмотренной выше машины для инкрементирования двоичного числа можно погрузить в большую машину, которая будет складывать два двоичных числа, а сам этот сумматор встроить в еще большую машину для выполнения умножения.

Если меньшую машину нужно «вызывать» только из одного состояния большей, то организовать такое погружение легко: достаточно включить копию меньшей машины, объединив ее начальное и заключительные состояния с состояниями большей машины в том месте, где подпрограмма должна начинаться и заканчиваться. Именно так мы и ожидаем использовать инкрементирующую машину как часть суммирующей, потому что общая структура свода правил в этом случае – не что иное как повторение единственной операции «если первое число не равно нулю, то декрементировать первое число и инкрементировать второе» столько раз, сколько возможно. В машине есть всего одно место, где происходит инкремент, и всего одно место, где следует продолжить выполнение после завершения инкремента.

Единственная трудность возникает, когда мы хотим вызвать некую подпрограмму из нескольких мест объемлющей машины. В машине Тьюринга нет механизма для сохранения «адреса возврата», чтобы подпрограмма могла знать, в какое состояние вернуться по

завершении работы, так что, на первый взгляд, поддержать такой, более общий, вид повторного использования кода мы не можем. Однако проблему можно решить путем дублирования, как мы поступили в разделе «Внутренняя память» на стр. 189: вместо того чтобы встраивать единственную копию состояний и правил меньшей машины, мы можем построить много копий – по одной для каждого места, где она используется.

Например, простейший способ превратить машину «инкрементировать число» в машину «прибавить три к числу» состоит в том, чтобы соединить вместе три копии, получив тем самым конструкцию «инкрементировать число, затем инкрементировать число, затем инкрементировать число». Тогда в большей машине появится несколько промежуточных состояний, через которые она дойдет до конечной цели, при этом каждая часть «инкрементировать число» будет начинаться и заканчиваться в разных промежуточных состояниях:




```
>> def increment_rules(start_state, return_state)
       incrementing = start_state
       finishing = Object.new
       finished = return_state
       [
         TMRule.new(incrementing, '0', finishing,      '1', :right),
         TMRule.new(incrementing, '1', incrementing,   '0', :left),
         TMRule.new(incrementing, '_', finishing,      '1', :right),
         TMRule.new(finishing,    '0', finishing,      '0', :right),
         TMRule.new(finishing,    '1', finishing,      '1', :right),
         TMRule.new(finishing,    '_', finished,       '_', :left)
       ]
     end
=> nil
>> added_zero, added_one, added_two, added_three = 0, 1, 2, 3
=> [0, 1, 2, 3]
>> rulebook = DTMRulebook.new(
       increment_rules(added_zero, added_one) +
       increment_rules(added_one, added_two) +
       increment_rules(added_two, added_three)
     )
=> #<struct DTMRulebook rules=[…]>
>> rulebook.rules.length
=> 18
>> tape = Tape.new(['1', '0', '1'], '1', [], '_')
```

```
=> #<Tape 101(1)>
>> dtm = DTM.new(TMConfiguration.new(added_zero, tape), [added_three], rulebook)
=> #<struct DTM …>
>> dtm.run; dtm.current_configuration.tape
=> #<Tape 111(0)_>
```

Возможность таким образом объединять состояния и правила позволяет строить для машин Тьюринга сколь угодно большие и сложные своды правил, не требуя явной поддержки подпрограмм, если, конечно, мы готовы смириться с возрастанием размера машины.

Несколько лент

Иногда мощность машины можно увеличить, расширив ее внешнюю память. Например, автомат с магазинной памятью станет более мощным, если дать ему доступ ко второму стеку, потому что два стека можно использовать для моделирования бесконечной ленты: в одном стеке хранятся символы из левой, а в другом – из правой половины ленты, и АМП моделирует движение головки ленты, выталкивая символы из одного стека и заталкивая в другой – точно так же, как в нашей реализации класса `Tape` в разделе «Моделирование» на стр. 180. Любая машина с конечным числом состояний, имеющая доступ к бесконечной ленте, по существу является машиной Тьюринга, так что оснащение АМП дополнительным стеком делает его значительно более мощным.

Поэтому не без оснований можно ожидать, что машину Тьюринга удастся сделать мощнее, добавив одну или несколько лент, каждая со своей независимой головкой. Увы, и это не так. На одной ленте машины Тьюринга достаточно места для хранения содержимого любого числа лент, нужно лишь чередовать его: три ленты, содержащие `abc`, `def` и `ghi`, можно записать на одну в виде `adgbehcfi`. Если рядом с каждым из чередующихся символов оставить пустой квадратик, то у машины будет место для записи маркеров, показывающих, где находятся все моделируемые ленточные головки: используя символы `X` для обозначения текущего положения каждой головки, мы можем представить следующие состояния трех лент – `ab(c)`, `(d)ef`, `g(h)i` – на одной ленте в виде `a_dXg_b_e_hXcXf_i_`.

Запрограммировать машину Тьюринга для моделирования нескольких лент, трудно, но головоломные детали чтения, записи и перемещения головок можно обернуть специальными состояниями и правилами («подпрограммами»), так что основная логика машина не окажется чрезмерно запутанной. В любом случае, каким бы не-

удобным ни было программирование, одноленточная машина Тьюринга способна справиться с любой задачей, которую может решить многоленточная машина, поэтому наличие дополнительных лент не наделяет машину Тьюринга новыми возможностями.

Многомерная лента

Наконец, возникает искушение оснастить машину Тьюринга более «просторным» запоминающим устройством. Вместо линейной ленты мы могли бы предложить бесконечную двумерную квадратную сетку и разрешить головке перемещаться не только влево и вправо, но также вверх и вниз. Это было бы полезно в ситуации, когда требуется быстро обратиться к некоторому участку внешней памяти, не пробегая все ячейки на пути к нему. Кроме того, мы смогли бы оставлять неограниченное пустое пространство вокруг нескольких строк, так чтобы каждая могла расти в любую сторону, и при этом не пришлось бы вручную сдвигать информацию на ленте, чтобы освободить место для нового символа.

Однако сетку вполне можно смоделировать с помощью одномерной ленты. Проще всего использовать *две* одномерных ленты: основную – для хранения собственно данных – и дополнительную – как временную память. Каждая строка моделируемой сетки[1] сохраняется на основной ленте, верхняя строка сначала, и конец строки отмечается специальным символом-маркером.

Головка основной ленты позиционируется над текущим символом, как обычно, поэтому для сдвига влево и вправо по моделируемой сетке, машина просто перемещает головку влево и вправо. Если головка наткнулась на маркер конца строки, то используется подпрограмма, которая сдвигает все данные вдоль ленты, чтобы сделать сетку на одну позицию шире.

Для сдвига вверх и вниз по моделируемой сетке, ленточная головка должна сместиться на целую строку влево или вправо соответственно. Чтобы сделать это, машина сначала сдвигает головку в начало или в конец текущей строки, используя дополнительную ленту для записи пройденного расстояния, а затем сдвигает головку на то же расстояние в предыдущей или в следующей строке. Если головка выходит за верхний или нижний край моделируемой сетки,

---

[1] Хотя сама сетка бесконечна, в нее можно записать лишь конечное число символов, поэтому нам нужно хранить только прямоугольную область, содержащую все непустые ячейки.

то с помощью подпрограммы можно выделить новую пустую строку, в которой будет двигаться головка.

Для этой модели требуется машина с двумя лентами, но мы уже знаем, как ее смоделировать, так что в итоге мы получаем смоделированную сетку, хранящуюся на двух смоделированных лентах, которые сами хранятся на одной физической ленте. Для двух слоев модели требуется множество дополнительных правил и состояний, и исходной машине придется проделать немало шагов для выполнения одного шага моделируемой, но, несмотря на увеличившийся размер и замедление, мы все-таки сумеем сделать то, что хотели (рано или поздно).

Машины общего назначения

У всех машин, с которыми мы до сих пор встречались, есть один серьезный недостаток: правила жестко зашиты, то есть машина неспособна адаптироваться к разным задачам. ДКА, который допускает все цепочки, соответствующие некоторому регулярному выражению, нельзя научить допускать другой набор цепочек; НАМП, распознающий палиндромы, только это и умеет делать; машина Тьюринга, умеющая инкрементировать двоичное число, больше ни на что не годна.

Реальные компьютеры в большинстве своем работают иначе. Они проектируются не для выполнения какой-то одной задачи, а как устройства *общего назначения*, которые можно программировать для решения различных задач. Хотя набор команд и конструкция центрального процессора любого отдельно взятого программируемого компьютера фиксированы, он может использовать *программное обеспечение* для управления оборудованием и адаптации поведения к задаче, поставленной пользователем.

Способна ли на это какая-нибудь из рассмотренных нами простых машин? Можем ли мы не проектировать новую машину для решения каждой новой задачи, а построить *единственную* машину, которая сможет прочитать программу из входного потока и выполнить определяемую этой программой задачу?

Наверное, вас не удивит, что машина Тьюринга обладает достаточной мощностью, что прочитать со своей ленты описание простой машины – к примеру, конечного автомата – и, прогнав модель этой машины, выяснить, что она делает. А разделе «Моделирование» на стр. 180 мы написали на Ruby программу для моделирования ДКА

по его описанию; при некотором усилии представленные там идеи можно превратить в свод правил для машины Тьюринга, выполняющей точно такую же модель.

Существует важное различие между машиной Тьюринга, которая моделирует *конкретный* ДКА, и машиной, которая может смоделировать *любой* ДКА.

Построить машину Тьюринга, способную воспроизвести поведение конкретного ДКА, очень просто – в конце концов, машина Тьюринга – это просто детерминированный конечный автомат, оснащенный лентой. Любое правило из свода правил ДКА можно преобразовать непосредственно в правило эквивалентной машины Тьюринга; вместо того чтобы читать символы из внешнего входного потока, как ДКА, преобразованное правило читает символ с ленты и сдвигает головку к следующему квадратику. Но это не особенно интересно, потому что получающаяся машина Тьюринга ничуть не более полезна, чем исходный ДКА.

Более интересна машина Тьюринга, которая моделирует ДКА *общего вида*. Эта машина читает с ленты структуру ДКА – правила, начальное состояние и заключительные состояния – и пошагово исполняет ДКА, используя другой участок ленты для запоминания текущего состояния и оставшейся части входной цепочки моделируемой машины. Реализовать такое обобщенное моделирование гораздо труднее, но в итоге мы получаем единственную машину Тьюринга, которая может адаптироваться к любой задаче, доступной ДКА, – нужно лишь подать ей на вход описание этого ДКА.

То же самое относится и к детерминированным Ruby-моделям НКА, ДАМП и НАМП – каждую из них можно преобразовать в машину Тьюринга, способную смоделировать любой автомат данного типа. Но что особенно важно, та же идея применима и к моделированию самих машин Тьюринга: реализовав классы `Tape`, `TMRule`, `DTMRulebook` и `DTM` в виде правил машины Тьюринга, мы сможем построить машину, которая сможет смоделировать любую другую ДМТ, прочитав с ленты ее правила, заключительные состояния и начальную конфигурацию и организовав пошаговое выполнение – по существу, это будет интерпретатор свода правил машины Тьюринга. Машина, которая работает таким образом, называется *универсальной машиной Тьюринга* (УМТ).

Это возбуждает воображение, потому что максимальную вычислительную мощность машин Тьюринга оказывается возможно пред-

ставить в виде одного программируемого устройства. Мы можем записать программу – кодированное описание машины Тьюринга – на ленту, подать эту ленту на вход УМТ и выполнить программу, получив требуемое поведение. Конечные автоматы и автоматы с магазинной памятью неспособны моделировать себе подобных, поэтому машина Тьюринга знаменует собой не только переход от ограниченных вычислительных машин к мощным, но и от узкоспециализированных к полностью программируемым.

Рассмотрим вкратце, как работает универсальная машина Тьюринга. Ее построение сопряжено с многочисленными кропотливыми и малоинтересными техническими деталями, поэтому наше знакомство будет сравнительно поверхностным, но мы по крайней мере убедимся в том, что такая штука возможна.

Кодирование

Прежде чем строить свод правил УМТ, необходимо решить, как представить машину Тьюринга в виде последовательности символов на ленте. УМТ должна прочитать правила, заключительные состояния и начальную конфигурацию произвольной машины Тьюринга, а затем обновлять текущую конфигурацию моделируемой машины в ходе моделирования. Поэтому необходим какой-то удобный способ хранения всей этой информации, так чтобы УМТ могла с ней работать.

Одна из проблем состоит в том, что у любой машины Тьюринга есть лишь конечное число состояний и конечное число различных записываемых на ленту символов, причем то и другое заранее фиксируется в своде правил, – и УМТ в этом плане не исключение. Если мы построим УМТ, умеющую обрабатывать 10 разных символов на ленте, то как с ее помощью смоделировать машину, в правилах которой используется 11 символов? Если мы проявим большую щедрость и разрешим УМТ обрабатывать сто разных символов, что произойдет, когда мы захотим смоделировать машину, использующую тысячу? Сколько бы символов мы ни разрешили использовать на собственной ленте УМТ, этого все равно не хватит для непосредственного представления любой мыслимой машины Тьюринга.

Существует также опасность случайного конфликта между символами моделируемой машины и УМТ. Чтобы сохранить правила и конфигурации машины Тьюринга на ленте, мы должны уметь помечать их границы символами, имеющими специальный смысл для

УМТ, дабы та могла понять, где кончается одно правило и начинается другое. Но если мы выберем, к примеру, `X` в качестве разделителя правил, то возникнут проблемы, когда в каком-то из правил моделируемой машины встретится символ `X`. Даже если зарезервировать суперспециальный набор символов, которые разрешено использовать только универсальной машине Тьюринга, все равно не избежать проблем, если мы попытаемся смоделировать УМТ с помощью ее самой, – следовательно, получится, что машина не совсем универсальна. Поэтому напрашивается мысль применить какое-то экранирование, чтобы предотвратить ошибочную интерпретацию обычных символов моделируемой машины как специальных символов УМТ.

Обе проблемы можно решить, придумав схему, в которой для кодирования моделируемой машины на ленте используется ограниченный набор символов. Если в схеме кодирования употребляются лишь определенные символы, то мы можем быть уверены, что любые другие символы УМТ может безопасно применять для специальных целей, а если схема способна закодировать машину с любым числом состояний и символов, то можно не беспокоиться о размере и сложности моделируемой машины.

Точные детали схемы кодирования не так существенны, главное – чтобы она удовлетворяла этим требованиям. В качестве примера упомянем схему *унарного*[1] представления, в которой для кодирования различных значений применяются цепочки из одного символа (скажем, `1`), только разной длины: если в моделируемой машине употребляются символы `a`, `b`, `c`, то их можно закодировать как `1`, `11`, `111`. Еще один символ, скажем `0`, можно использовать в качестве маркера для разграничения унарных значений: цепочка `acbc` будет представлена в виде `101110110111`. Эта схема не очень эффективна с точки зрения расходования памяти, зато масштабируется под любое число подлежащих кодированию символов – просто на ленте будут храниться все более и более длинные цепочки.

Определившись с кодированием отдельных символов, мы должны придумать, как представлять правила моделируемой машины Тьюринга. Для этого можно закодировать части правила (состояние, символ, следующее состояние, записываемый символ, направление сдвига) и записать их на ленту подряд, используя там, где нужно, специальные символы-разделители. В нашей схеме кодирования

---

[1] В основе бинарного представления чего-либо лежит двойка, в основе унарного – единица.

состояния также можно представить в унарном виде: состояние 1 кодируется цепочкой 1, состояние 2 – цепочкой 11 и т. д., а для представления «лево» и «право» мы вправе использовать выделенные символы (скажем, L и R), так как знаем, что направлений только два.

Мы можем конкатенировать отдельные правила для представления всего свода. Аналогично текущую конфигурацию моделируемой машины можно закодировать, конкатенировав представление текущего состояния с представлением текущего содержимого ленты[1]. В результате мы получаем то, что хотели: полную машину Тьюринга, записанную в виде последовательности символов на ленте другой машины Тьюринга, и можем оживить ее с помощью моделирования.

Моделирование

В принципе, универсальная машина Тьюринга работает так же, как Ruby-модель, которую мы построили в разделе «Моделирование» на стр. 180, только самой работы больше. Описание моделируемой машины – ее свода правил, заключительных состояний и начальной конфигурации – хранится в закодированном виде на ленте УМТ. Для выполнения одного шага моделирования УМТ перемещает головку между правилами, текущим состоянием и лентой моделируемой машины в поисках правила, применимого к текущей конфигурации. Найдя его, УМТ обновляет ленту моделируемой машины в соответствии с символом и направлением сдвига, указанными в правиле, и переводит моделируемую машину в новое состояние.

Этот процесс повторяется, пока моделируемая машина не попадет в заключительное состояние или не заклинится, достигнув конфигурации, к которой неприменимо ни одно правило.

---

[1] Мы опустили вопрос о представлении ленты, но это тоже нетрудно, и к тому же у нас всегда есть возможность сохранить ее на второй моделируемой ленте, как описано в разделе «Несколько лент» на стр. 194.

Часть II. ВЫЧИСЛЕНИЯ И ВЫЧИСЛИМОСТЬ

В первой части этой книги предметом наших экспериментов были знакомые примеры вычислений; императивные языки программирования, машины с конечным числом состояний и универсальные компьютеры. На этих примерах мы убедились, что вычисление – это в той или иной степени процесс использования системы для манипулирования информацией и получения ответов на вопросы.

Во второй части мы пустимся в более рискованные приключения. Мы начнем с поиска вычислений в совершенно неожиданных местах и закончим исследованием фундаментальных ограничений на то, что может делать машина.

Будучи программистами, мы работаем с языками и машинами, которые по построению отвечают нашим умозрительным моделям мира, и ожидаем, что у них есть средства для перевода наших идей на язык реализации. Такая конструкция, ориентированная на человека, продиктована скорее удобством, чем необходимостью; даже простейшая конструкция машины Тьюринга напоминает нам о математике, рабочими инструментами которого служат карандаш и бумага.

Однако знакомые машины – не единственные места, в которых могут иметь место вычисления. Способностями к вычислениям могут обладать и не столь традиционные системы, пусть даже человеку не так просто понять их внутреннее устройство и управлять ими. Мы исследуем эту идею в главе 6, где попытаемся писать программы на совсем уж минимальном языке, который вроде бы не обладает никакими полезными свойствами, и продолжим эту тему в главе 7, где дадим обзор разнообразных простых систем и увидим, как они могут производить такие же вычисления, как более сложные машины.

Убедившись в том, что полноценные вычисления могут производиться во многих непохожих друг на друга системах, мы в главе

8 займемся вопросом о том, на что вообще способно вычисление. Естественно предположить, что компьютер может решить практически любую задачу при условии, что у него достаточно времени и что написана подходящая программа. Однако, как выясняется, на этом пути имеются теоретические ограничения: существуют задачи, которые не способен решить ни один компьютер, сколь угодно быстрый и эффективный.

К сожалению, некоторые из таких неразрешимых задач связаны с предсказанием поведения программ, то есть как раз с теми вопросами, в которых программисты ожидают помощи от компьютера. Мы рассмотрим различные стратегии преодоления этих пределов компьютерной вселенной и завершим обсуждение главой 9, изучив, как можно использовать абстракции для получения ответов на вопросы, ответов не имеющие.

Глава 6. Программирование на пустом месте

> Для того чтобы приготовить яблочный пирог с нуля, сначала придется придумать Вселенную.
> – *Карл Саган*

В этой книге мы пытаемся разобраться в природе вычислений, строя их модели. До сих пор мы моделировали вычисления, конструируя простые воображаемые машины с различными ограничениями и наблюдая за тем, как в зависимости от наложенных ограничений получаются системы с различной вычислительной мощностью.

Машины Тьюринга из главы 5 интересны тем, что могут реализовать сложное поведение, не опираясь на сложные механизмы. Имея лишь ленту, головку считывания/записи и фиксированный набор правил, машина Тьюринга обладает достаточной гибкостью для моделирования поведения машин с более развитыми возможностями хранения, недетерминированным поведением и вообще настолько причудливых, насколько мы захотим. Отсюда следует, что для полноценных вычислений не нужна неимоверно сложная машина, хватает лишь возможности сохранять значения, извлекать их и принимать на их основе простые решения.

Модели вычислений не обязаны выглядеть как машины; они вполне могут быть устроены по образцу языков программирования. Язык программирования SIMPLE из главы 2 безусловно может производить вычисления, хотя он и не так элегантен, как машина Тьюринга. В него уже встроено много синтаксических конструкций – числа, булевы значения, двоичные выражения, переменные, операторы присваивания, последовательности, условные предложения, циклы – а мы еще даже не начинали добавлять средства для написания реальных программ: строки, структуры данных, вызовы процедур и т. д.

Чтобы превратить SIMPLE в по-настоящему полезный язык программирования, потребуется усердно поработать, а результат будет содержать массу несущественных деталей, мало что говоря об истинной природе вычислений. Было бы интереснее начать с нуля и создать нечто минимальное – машину Тьюринга в мире языков программирования – и с ее помощью изучить, что действительно необходимо для вычисления, а что является случайным шумом.

В этой главе мы собираемся исследовать в высшей степени минимальный язык программирования, который называется *бестиповым лямбда-исчислением*. Сначала мы поэкспериментируем с написанием на диалекте Ruby программ, которые аппроксимируют лямбда-исчисление путем использования возможно меньшего подмножества языка; это по-прежнему будет программа на Ruby, но с искусственными ограничениями, которые позволят нам исследовать ограниченную семантику, не изучая совершенно новый язык. Затем, посмотрев, что можно сделать с такой крайне скромной функциональностью, мы реализуем автономный язык ровно с такими возможностями – с собственным анализатором, абстрактным синтаксисом и операционной семантикой – применяя знания, полученные в предыдущих главах.

Имитация лямбда-исчисления

Чтобы составить представление о программировании на минимальном языке, попробуем решить какую-нибудь задачу на Ruby, не используя большинство его полезных возможностей. Естественно, мы сразу исключаем gem-пакеты, стандартную библиотеку, модули, методы, классы, объекты, а заодно – уж минимальное так минимальное – управляющие конструкции, присваивание, массивы, строки, числа и булевы значения.

Конечно, если мы откажемся *вообще* от всех средств Ruby, то не останется языка, на котором можно программировать, поэтому сохраним следующим возможности:

- ссылка на переменные;
- создание процедур;
- вызов процедур.

Это означает, что можно писать только такой Ruby-код:

```
-> x { -> y { x.call(y) } }
```

Примерно так и выглядит программа в бестиповом лямбда-исчислении, поэтому для наших целей эта аппроксимация годится. Более подробно мы будем рассматривать лямбда-исчисление в разделе «Реализация лямбда-исчисления» на стр. 245.

Чтобы сделать код короче и понятнее, мы позволим себе использовать константы в качестве аббревиатур: создав сложное выражение, мы сможем сопоставить ему константу, то есть короткое имя, по которому его можно будет использовать впоследствии. Ссылка по имени ничем не отличается от повторной записи исходного выражения – просто код становится более лаконичным, – так что мы не вводим зависимость от механизма присваивания в Ruby. В любой момент мы можем отказаться от послаблений и отменить аббревиатуры, заменив каждую константу процедурой, на которую она ссылается, – смирившись с тем, что программы станут гораздо длиннее.

Работа с процедурами

Поскольку мы собираемся строить программы целиком из процедур, потратим некоторое время на изучение их свойств.

В этом разделе мы все еще применяем Ruby в полном объеме, чтобы проиллюстрировать общее поведение процедур. Ограничение мы наложим только тогда, когда начнем писать код в разделе «Задача» на стр. 205.

Трубопровод

Процедуры – это трубы, по которым значения передаются из одного места в другое. Рассмотрим, что происходит, когда мы вызываем процедуру:

```
-> x { x + 2 }.call(1)
```

Значение, переданное в качестве аргумента вызова, в данном случае 1, *втекает* в параметр блока, в данном случае x, а затем *вытекает* из этого параметра в тех местах, где используется, так что в результате Ruby вычисляет выражение 1 + 2. Фактическую работу делают другие средства языка, а процедуры всего лишь связывают части программы воедино и позволяют значениям перетекать туда, где они нужны.

Это не сулит ничего хорошего нашим экспериментам с минимальным Ruby. Если процедуры могут только перемещать значения между частями Ruby-программы, которые что-то делают с ними, то

как же построить полезную программу из одних лишь процедур? Мы ответим на этот вопрос, после того как изучим еще кое-какие свойства процедур.

Аргументы

Процедуры могут принимать несколько аргументов, но эта возможность не является существенной. Любую процедуру, принимающую несколько аргументов, например:

```
-> x, y {
  x + y
}.call(3, 4)
```

Можно переписать в виде вложенных процедур с одним аргументом:

```
-> x {
  -> y {
    x + y
  }
}.call(3).call(4)
```

Здесь внешняя процедура принимает один аргумент, `x`, и возвращает внутреннюю процедуру, также с одним аргументом, `y`. Мы можем вызвать внешнюю процедуру, передав значение `x`, а затем внутреннюю процедуру, передав значение `y`, и получим тот же результат, как при вызове процедуры с двумя аргументами[1].

Поскольку мы стремимся убрать столько возможностей Ruby, сколько сможем, ограничимся созданием и вызовом процедур *с одним аргументом*; хуже от этого не будет.

Равенство

Единственный способ что-то узнать о коде внутри процедуры – вызвать его, поэтому две процедуры являются взаимозаменяемыми, если при вызове с одними и теми же аргументами дают один и тот же результат, пусть даже внутри их код различается. Идея считать две сущности эквивалентными, основываясь на их наблюдаемом извне поведении, называется *экстенсиональным равенством.*

Пусть, например, имеется такая процедура `p`:

```
>> p = -> n { n * 2 }
=> #<Proc (lambda)>
```

[1] Это называется *каррированием*, и такое преобразование можно выполнить автоматически с помощью метода `Proc#curry`.

Мы можем написать другую процедуру q, которая принимает аргумент и вызывает p с этим аргументом:

```
>> q = -> x { p.call(x) }
=> #<Proc (lambda)>
```

Очевидно, что p и q – разные процедуры, но экстенсионально они равны, потому что делают в точности одно и то же для любого аргумента:

```
>> p.call(5)
=> 10
>> q.call(5)
=> 10
```

Знание того, что p эквивалентно `-> x { p.call(x) }`, открывает новые возможности для рефакторинга. Увидев в программе конструкцию `-> x { p.call(x) }`, мы можем либо устранить ее, заменив все выражение на p; и наоборот, при некоторых обстоятельствах (каких, узнаем позже) мы можем сделать обратное преобразование.

Синтаксис

В Ruby есть несколько синтаксических нотаций для создания и вызова процедур. Начиная с этого момента, мы будем использовать запись `-> arguments { body }` для создания процедуры и квадратные скобки для ее вызова:

```
>> -> x { x + 5 }[6]
=> 11
```

Так проще различить тело и аргумент процедуры без излишнего синтаксического «шума».

Задача

Наша цель – написать хорошо известную программу FizzBuzz:

> Написать программу, которая печатает числа от 1 до 100. Но вместо чисел, кратных трем, печатать «Fizz», а вместо чисел, кратных пяти, – «Buzz». Если число кратно одновременно трем и пяти, печать «FizzBuzz».
>
> – Имран Гори,
> *Using FizzBuzz to Find Developers who Grok Coding*[1]

[1] http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

Это намеренно очень простая задача, назначение которой – проверить на собеседовании, имеет ли кандидат хоть какие-то навыки программирования. Всякий, кто знает, как программировать, должен справиться с ней без труда.

Вот реализация FizzBuzz на полном Ruby:

```
(1..100).each do |n|
  if (n % 15).zero?
    puts 'FizzBuzz'
  elsif (n % 3).zero?
    puts 'Fizz'
  elsif (n % 5).zero?
    puts 'Buzz'
  else
    puts n.to_s
  end
end
```

Это не самая изобретательная реализация FizzBuzz – существует множество более хитроумных (*http://redd.it*) – но она прямолинейна, ее можно написать, особенно не задумываясь.

Однако в этой программе встречаются предложения `puts`, а у нас нет способа вывести текст на консоль, используя только процедуры[1], поэтому мы заменим ее приблизительно эквивалентной программой, которая возвращает массив строк, а не печатает их:

```
(1..100).map do |n|
  if (n % 15).zero?
    'FizzBuzz'
  elsif (n % 3).zero?
    'Fizz'
  elsif (n % 5).zero?
    'Buzz'
  else
    n.to_s
  end
end
```

Это решение задачи FizzBuzz по-прежнему осмысленно, и теперь есть шанс реализовать его только с помощью процедур.

Несмотря на простоту, в отсутствие необходимых средств языка программирования эта программа оказывается довольно амбициоз-

[1] Мы, конечно, могли бы *смоделировать* вывод на консоль, введя процедуру для представления стандартного ввода и вывода и приняв соглашение о том, как передавать ей текст, но это усложнило бы упражнение в направлении, не интересном для нас. Смысл программы FizzBuzz не в печати, а в арифметике и управлении потоком выполнения.

ной: она создает диапазон, строит его отображение на множество, вычисляет условное предложение с несколькими ветвями, производит арифметические операции по модулю, использует предикат `Fixnum#zero?`, включает строковые литералы и преобразует числа в строки с помощью метода `Fixnum#to_s`. Так что задействована немалая толика встроенной в Ruby функциональности, которую мы хотим исключить, заменив процедурами.

Числа

Мы начнем с чисел, которые встречаются в FizzBuzz. Как можно представить числа, не используя класс `Fixnum` и прочие типы данных, предоставляемые Ruby?

Если мы собираемся реализовать числа[1] с нуля, то надо бы иметь ясное понятие о том, что мы реализуем. Что такое *число*? Трудно дать конкретное определение, в которое не скрыты допущения о какой-то характеристике того, что мы пытаемся определить; например, формулировка «нечто, говорящее о том, сколько...» не особенно полезна, потому что «сколько» – просто иной способ сказать «число».

Существует один способ охарактеризовать числа: представим, что имеется мешок яблок и мешок апельсинов. Достанем одно яблоко и один апельсин и отложим их в сторону; продолжим доставать яблоки и апельсины одновременно, пока хотя бы один мешок не окажется пустым.

Если оба мешка опустеют одновременно, то мы узнаем кое-что интересное: хотя мешки содержали разные объекты, у них было некое общее свойство, означающее, что они стали пустыми в одно и то же время; в любой момент в ходе процедуры повторного доставания объекта из мешка оба мешка были либо одновременно пусты, либо одновременно не пусты. Это абстрактное свойство, общее для мешков, и есть то, что можно назвать числом (хотя мы не знаем, *каким именно* числом!), оно позволяет сравнить эти конкретные мешки с любыми другими и узнать, характеризуются ли те таким же «числом».

Таким образом, охарактеризовать числа можно повторением (или *итерацией*) некоторого действия, в данном случае доставанием объекта из мешка. Каждому числу соответствует свой способ повтора действия: числу один – простое совершение действия,

---

[1] Точнее, мы собираемся реализовать *неотрицательные целые числа*: нуль, один, два, три и т. д.

числу два – совершение действия и затем совершение его еще раз и т. д. Числу нуль, естественно, соответствует несовершение действия ни разу.

Поскольку создание и вызов процедур – единственные действия, которые может выполнять наша программа, то можно попытаться реализовать число `n` с помощью кода, который вызывает процедуру `n` раз.

Например, если бы было разрешено определять методы – это не разрешено, но помечтать-то можно, – то мы могли бы определить `#one` как метод, который принимает процедуру и некий произвольный второй аргумент, а затем один раз вызывает процедуру с этим аргументом:

```
def one(proc, x)
    proc[x]
end
```

Можно было бы также определить метод `#two`, который вызывает процедуру один раз, а затем еще раз, передавая в качестве аргумента результат первого вызова[1]:

```
def two(proc, x)
    proc[proc[x]]
end
```

И так далее:

```
def three(proc, x)
    proc[proc[proc[x]]]
end
```

Следуя этому образцу, естественно было бы определить `#zero` как метод, который принимает процедуру и еще какой-то аргумент, игнорирует процедуру (вызывает ее нуль раз) и возвращает второй аргумент без изменения:

```
def zero(proc, x)
    x
end
```

Все эти реализации можно преобразовать в представления, не содержащие никаких методов; например, метод `#one` можно заменить

[1] Это называется «итерирование функции».

процедурой, которая принимает два аргумента[1] и вызывает первый аргумент, передавая ему второй. Выглядит это так:

```
ZERO  = -> p { -> x {       x    } }
ONE   = -> p { -> x {     p[x]   } }
TWO   = -> p { -> x {   p[p[x]]  } }
THREE = -> p { -> x { p[p[p[x]]] } }
```

Это позволяет избежать использования неразрешенных средств, придав вместо этого процедурам имена путем присваивания их константам.

Эта техника представления данных в виде чистого кода называется *кодированием Чёрча* по имени Алонзо Чёрча, который изобрел лямбда-исчисление (*http://www.jstor.org/discover/10.2307/2371045?uid=2&uid=4&sid=21102188356963*). Закодированные таким образом числа называются *нумералами Чёрча*, а чуть ниже мы встретимся с *булевыми значениями Чёрча* и с *парами Чёрча*.

А теперь, хотя мы и избегаем использовать средства Ruby *внутри* нашего решения задачи FizzBuzz, было бы полезно транслировать эти загадочные представления чисел в обычные Ruby-значения, когда они используются *вне* нашего кода, – чтобы можно было взглянуть на них в оболочке, использовать в утверждениях при тестировании или хотя бы убедиться, что они действительно представляют числа.

К счастью, можно написать метод `#to_integer`, который выполняет такое преобразование:

```
def to_integer(proc)
    proc[-> n { n + 1 }][0]
end
```

Этот метод принимает процедуру, представляющую число, и вызывает ее, передавая в качестве аргумента другую процедуру (которая просто увеличивает свой аргумент на единицу), и обычное число

---

[1] На самом деле, говорить «принимает два аргумента» не вполне корректно, потому что мы ограничили себя процедурами с одним аргументом (см. раздел «Аргументы» на стр. 206). Технически правильно было бы сказать «принимает один аргумент и возвращает процедуру, принимающую один аргумент», но это слишком длинно, поэтому будем придерживаться сокращенной формы, но помнить, что имеется в виду.

0 в смысле Ruby. Если вызвать #to_integer с аргументом ZERO, то в силу определения ZERO инкрементирующая процедура не вызывается ни разу, и мы получаем исходное число 0:

```
>> to_integer(ZERO)
=> 0
```

А если вызвать #to_integer с аргументом THREE, то инкрементирующая процедура будет вызвана три раза, и мы получим в результате число 3:

```
>> to_integer(THREE)
=> 3
```

Таким образом, эти основанные на процедурах представления действительно кодируют числа, и мы в любой момент можем преобразовать их в более удобную на практике форму.

Для программы FizzBuzz нам нужны числа пять, пятнадцать и сто; все их можно представить описанным способом:

```
FIVE     = -> p { -> x { p[p[p[p[p[x]]]]] } }
FIFTEEN  = -> p { -> x { p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]] } }
HUNDRED  = -> p { -> x { p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p↵
[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p↵
[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]]]↵
]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]] } }
```

Определения, конечно, компактными не назовешь, но они работают, в чем легко убедиться с помощью #to_integer:

```
>> to_integer(FIVE)
=> 5
>> to_integer(FIFTEEN)
=> 15
>> to_integer(HUNDRED)
=> 100
```

Итак, возвращаясь к программе FizzBuzz, мы видим, что любое число Ruby можно заменить его реализацией на основе процедур:

```
(ONE..HUNDRED).map do |n|
  if (n % FIFTEEN).zero?
    'FizzBuzz'
  elsif (n % THREE).zero?
    'Fizz'
  elsif (n % FIVE).zero?
    'Buzz'
```

```
  else
    n.to_s
  end
end
```

Как и обещали, мы пишем ONE вместо `-> p { -> x { p[x] } }` и т. д., чтобы сделать код понятнее.

К сожалению, эта программа больше не работает, потому что мы пытаемся применить операции типа `..` и `%` к реализациям чисел на основе процедур. Поскольку Ruby не умеет трактовать подобные конструкции как числа, он просто сообщает об ошибке: `TypeError: can't iterate from Proc, NoMethodError: undefined method `%' for #<Proc (lambda)>` и т. д. Нам нужно заменить все эти операции такими, которые работали бы с нашими представлениями, – и при этом использовать только процедуры.

Однако прежде чем приступать к переделке операций, нужно реализовать `true` и `false`.

Булевы значения

Как представить булевы значения, пользуясь только процедурами? Отметим, что булевы значения предназначены для использования исключительно в условных выражениях, которые в общем случае выглядят так: «if булево значение then *это* else *то*».

```
>> success = true
=> true
>> if success then 'happy' else 'sad' end
=> "happy"
>> success = false
=> false
>> if success then 'happy' else 'sad' end
=> "sad"
```

Итак, истинный смысл булевых значений заключается в выборе одного из двух вариантов, и этим можно воспользоваться, представив булево значение в виде процедуры, которая выбирает одно из двух значений. Вместо того чтобы рассматривать булево значение как безжизненный элемент данных, который какой-то будущий код прочитает и на его основе решит, что делать дальше, мы реализуем его непосредственно в виде кода, который, будучи вызван с двумя аргументами, выбирает первый или второй.

В виде методов `#true` и `#false` можно было бы записать так:

```
def true(x, y)
    x
end

def false(x, y)
    y
end
```

Мстод `#true` принимает два аргумента и возвращает первый из них, а мстод `#false` тоже принимает два аргумента, но возвращает второй. Этого достаточно для получения грубого аналога условного поведения:

```
>> success = :true
=> :true
>> send(success, 'happy', 'sad')
=> "happy"
>> success = :false
=> :false
>> send(success, 'happy', 'sad')
=> "sad"
```

Как и раньше, несложно преобразовать эти методы в процедуры:

```
TRUE  = -> x { -> y { x } }
FALSE = -> x { -> y { y } }
```

И точно так же, как мы определили метод `#to_integer`, чтобы убедиться в возможности преобразовать числа, основанные на процедурах, в обычные числа Ruby, так можно определить и метод `#to_boolean`, который преобразует процедуры `TRUE` и `FALSE` в обычные Ruby-объекты `true` и `false`:

```
def to_boolean(proc)
    proc[true][false]
end
```

Этот метод принимает процедуру, представляющую булево значение, и вызывает ее с двумя аргументами: `true` и `false`. `TRUE` возвращает свой первый аргумент, поэтому `to_boolean(TRUE)` всегда возвращает `true`. Для `FALSE` все аналогично.

```
>> to_boolean(TRUE)
=> true
>> to_boolean(FALSE)
=> false
```

Таким образом, представить булевы значения с помощью процедур оказалось на удивление просто, но для программы FizzBuzz нужны не только булевы значения, но и основанная лишь на процедурах реализация предложения Ruby `if-elsif-else`. На самом деле, благодаря способу реализации булевых значений написать метод `#if` также нетрудно:

```
def if(proc, x, y)
    proc[x][y]
end
```

И столь же легко преобразовать его в процедуру:

```
IF =
  -> b {
    -> x {
      -> y {
        b[x][y]
      }
    }
  }
```

Очевидно, `IF` не должен делать никакой полезной работы, потому что булево значение само выбирает нужный аргумент (`IF` – не более чем синтаксическая глазурь), но выглядит он более естественно, чем обращение к булеву значению напрямую:

```
>> IF[TRUE]['happy']['sad']
=> "happy"
>> IF[FALSE]['happy']['sad']
=> "sad"
```

Кстати, это означает, что можно переписать определение метода `#to_boolean` с использованием `IF`:

```
def to_boolean(proc)
    IF[proc][true][false]
end
```

По ходу рефакторинга стоить отметить, что реализацию `IF` можно существенно улучшить, поскольку она содержит процедуры, эквивалентные более простым, как обсуждалось в разделе «Равенство» на стр. 206. Взгляните, например, на самую внутреннюю процедуру в `IF`:

```
-> y {
    b[x][y]
}
```

Этот код означает следующее.

1. Принять аргумент y.
2. Вызвать b с аргументом x, получив в ответ процедуру.
3. Вызвать эту процедуру с аргументом y.

Шаги 1 и 3 – пустая трата времени: когда мы вызываем эту процедуру с неким аргументом, она просто передает его другой процедуре. Поэтому все вместе эквивалентно единственному шагу 2, `b[x]`, и мы можем упростить реализацию `IF`, удалив из нее лишнее:

```
IF =
  -> b {
    -> x {
      b[x]
    }
  }
```

Такая же структура наблюдается в процедуре, которая теперь стала внутренней:

```
-> x {
    b[x]
}
```

По той же причине эта процедура эквивалентна просто `b`, так что мы можем и дальше упростить `IF`:

```
IF = -> b { b }
```

Сделать еще проще вряд ли получится.

`IF` не делает ничего полезного – вся работа ложится на `TRUE` и `FALSE` – поэтому ее таки *можно было бы* еще упростить, исключив вовсе. Но наша цель – преобразовать исходное решение задачи FizzBuzz в процедуру, минимально отклоняясь от оригинала, поэтому процедуру `IF` удобно оставить, чтобы она напоминала о том, где в оригинале находилось выражение `if-elsif-else`, пусть даже она является чисто декоративной.

Так или иначе, теперь у нас есть `IF`, и мы можем вернуться к программе FizzBuzz и заменить выражение `if-elsif-else` вложенными вызовами `IF`:

```
(ONE..HUNDRED).map do |n|
  IF[(n % FIFTEEN).zero?][
    'FizzBuzz'
  ][IF[(n % THREE).zero?][
```

```
    'Fizz'
  ][IF[(n % FIVE).zero?][
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

Предикаты

Наша следующая задача – заменить `Fixnum#zero?` основанной на процедурах реализацией, которая работает с основанными на процедурах числами. Для чисел Ruby алгоритм метода `#zero?` выглядит как-то так:

```
def zero?(n)
  if n == 0
    true
  else
    false
  end
end
```

(Его можно было бы записать короче, но тут ясно, что происходит: сравнить число с `0`; если они равны, вернуть `true`, иначе вернуть `false`.)

Как адаптировать это к процедурам вместо чисел Ruby? Взглянем еще раз на нашу реализацию чисел:

```
ZERO  = -> p { -> x {       x    } }
ONE   = -> p { -> x {     p[x]   } }
TWO   = -> p { -> x {   p[p[x]]  } }
THREE = -> p { -> x { p[p[p[x]]] } }
⋮
```

Обратите внимание, что процедура `ZERO` – единственное число, которая не вызывает `p` (она просто возвращает `x`), тогда как все остальные хотя бы один раз вызывают `p`. Этим можно воспользоваться: если вызвать неизвестное число, передав `TRUE` в качестве второго аргумента, то оно сразу вернет `TRUE`, если это число равно `ZERO`. Если же оно не равно `ZERO`, то будет возвращено то, что возвращает `p`, поэтому если в качестве `p` взять процедуру, которая всегда возвращает `FALSE`, то мы получим желаемое поведение:

```
def zero?(proc)
    proc[-> x { FALSE }][TRUE]
end
```

И снова нетрудно переписать этот метод в виде процедуры:

```
IS_ZERO = -> n { n[-> x { FALSE }][TRUE] }
```

Мы можем вызвать из оболочки метод `#to_boolean` и убедиться, что процедура работает:

```
>> to_boolean(IS_ZERO[ZERO])
=> true
>> to_boolean(IS_ZERO[THREE])
=> false
```

Все правильно, поэтому заменим в FizzBuzz все обращения к методу `#zero?` процедурой `IS_ZERO`:

```
(ONE..HUNDRED).map do |n|
  IF[IS_ZERO[n % FIFTEEN]][
    'FizzBuzz'
  ][IF[IS_ZERO[n % THREE]][
    'Fizz'
  ][IF[IS_ZERO[n % FIVE]][
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

Пары

Мы научились представлять и использовать числа и булевы значения, но у нас нет *структур* данных, в которых можно было бы хранить несколько значений. Вскоре какой-то вид структур нам понадобится для реализации более сложной функциональности, поэтому сделаем паузу и познакомимся с одним таким представлением.

Простейшая структура данных – это *пара*, аналог массива из двух элементов. Реализовать пары очень просто:

```
PAIR  = -> x { -> y { -> f { f[x][y] } } }
LEFT  = -> p { p[-> x { -> y { x } } ] }
RIGHT = -> p { p[-> x { -> y { y } } ] }
```

Назначение пары – хранить два значения и предоставлять их по запросу. Для конструирования пары мы вызываем процедуру `PAIR` с двумя аргументами, `x` и `y`, а она возвращает свою внутреннюю процедуру:

```
-> f { f[x][y] }
```

Если вызвать эту процедуру, передав в качестве аргумента другую процедуру f, то она вызовет последнюю с запомненными ранее значениями x и y в качестве аргументов. Операции LEFT и RIGHT выбирают из пары левый или правый элемент, передавая ей процедуру, которая возвращает свой первый или второй аргумент соответственно. Все работает довольно просто:

```
>> my_pair = PAIR[THREE][FIVE]
=> #<Proc (lambda)>
>> to_integer(LEFT[my_pair])
=> 3
>> to_integer(RIGHT[my_pair])
=> 5
```

Этой очень простой структуры данных для начала будет достаточно; мы воспользуемся парами позже в разделе «Списки» на стр. 228 как элементом для построения более сложных структур.

Операции над числами

Имея числа, булевы значения, условные предложения, предикаты и пары, мы почти готовы к тому, чтобы реализовать оператор деления по модулю.

Но прежде чем приниматься за такую амбициозную задачу, как вычисление остатка от деления одного числа на другое, мы должны научиться выполнять более простые операции, например, инкремент и декремент одного числа. С инкрементом особых проблем не возникает:

```
INCREMENT = -> n { -> p { -> x { p[n[p][x]] } } }
```

Посмотрим, как работает процедура INCREMENT: если вызвать ее, передав основанное на процедурах число n, то она вернет новую процедуру, которая принимает некую процедуру p и произвольный второй аргумент x, как и в случае чисел.

Что сделает эта новая процедура, если ее вызвать? Сначала она вызывает n с аргументами p и x; поскольку n – число, это означает «вызвать n раз p с аргументом x», как это сделало бы исходное число, а затем вызвать p еще раз, передав в качестве аргумента результат. Таким образом, это процедура, первый аргумент которой n + 1 раз вызывается со вторым аргументом, а это в точности и является представлением числа n + 1.

А как быть с декрементом? На первый взгляд, задача куда сложнее: если процедура уже вызывается `n` раз, то достаточно просто добавить еще один вызов, получив всего `n + 1` вызовов, но не видно способа «отменить» один вызов, оставив только `n - 1`.

Одно из возможных решений – придумать такую процедуру, которая, будучи вызвана `n` раз с некоторым начальным аргументом, возвращает число `n - 1`. По счастью, пары позволяют именно это и сделать. Подумайте, что делает такой метод Ruby:

```
def slide(pair)
    [pair.last, pair.last + 1]
end
```

Если вызвать `slide`, передав ему массив чисел с двумя элементами, то он вернет новый массив, в котором первым элементом будет второе число, а вторым – число, на единицу большее. Если входной массив содержит два *последовательных* числа, мы получаем узкое окошко, «скользящее» по числовой оси:

```
>> slide([3, 4])
=> [4, 5]
>> slide([8, 9])
=> [9, 10]
```

Нам это полезно, потому что, начав с окна, позиционированного на `-1`, мы получим ситуацию, когда первое число в массиве *на единицу меньше* количества вызовов `slide` для этого окна, несмотря на то что мы производили только операцию инкремента:

```
>> slide([-1, 0])
=> [0, 1]
>> slide(slide([-1, 0]))
=> [1, 2]
>> slide(slide(slide([-1, 0])))
=> [2, 3]
>> slide(slide(slide(slide([-1, 0]))))
=> [3, 4]
```

Мы не можем поступить в точности так для чисел, основанных на процедурах, потому что у нас нет способа представить `-1`, но у метода `slide` есть интересная особенность: он использует только второй элемент массива, поэтому вместо `-1` мы можем взять любое другое значение, например `0`, и получить точно такой же результат:

```
>> slide([0, 0])
=> [0, 1]
```

```
>> slide(slide([0, 0]))
=> [1, 2]
>> slide(slide(slide([0, 0])))
=> [2, 3]
>> slide(slide(slide(slide([0, 0]))))
=> [3, 4]
```

Это ключ к написанию процедуры DECREMENT: мы можем преобразовать slide в процедуру, воспользоваться процедурным представлением числа n для вызова slide n раз для пары, состоящей из двух ZERO, а затем с помощью LEFT извлечь левое число из получившейся пары.

```
SLIDE     = -> p { PAIR[RIGHT[p]][INCREMENT[RIGHT[p]]] }
DECREMENT = -> n { LEFT[n[SLIDE][PAIR[ZERO][ZERO]]] }
```

Вот демонстрация DECREMENT в действии:

```
>> to_integer(DECREMENT[FIVE])
=> 4
>> to_integer(DECREMENT[FIFTEEN])
=> 14
>> to_integer(DECREMENT[HUNDRED])
=> 99
>> to_integer(DECREMENT[ZERO])
=> 0
```

Результатом DECREMENT[ZERO] на самом деле является фиктивный левый элемент из начальной пары PAIR[ZERO][ZERO], и для его получения SLIDE вообще не вызывается. Поскольку отрицательных чисел у нас нет, то самым близким к истинному значением DECREMENT[ZERO] будет 0, так что использование ZERO в качестве фиктивного значения – удачная мысль.

Имея процедуры INCREMENT и DECREMENT, можно реализовать более полезные операции над числами: сложение, вычитание, умножение и возведение в степень:

```
ADD      = -> m { -> n { n[INCREMENT][m] } }
SUBTRACT = -> m { -> n { n[DECREMENT][m] } }
MULTIPLY = -> m { -> n { n[ADD[m]][ZERO] } }
POWER    = -> m { -> n { n[MULTIPLY[m]][ONE] } }
```

Эти реализации в общем-то не нуждаются в пояснениях. Чтобы сложить m и n, мы «начинаем с m и инкрементируем (INCREMENT) его n раз», с вычитанием все аналогично. Имея операцию ADD, мы можем перемножить m и n следующим образом: «начав с ZERO, прибa-

вить (ADD) n раз число m». Для возведения в степень нужно заменить в этой фразе ADD на MULTIPLY, а ZERO на ONE.

В разделе «Свертка выражений» на стр. 250 мы заставим Ruby мелкими шагами вычислить ADD[ONE][ONE] и посмотрим, как он порождает TWO.

Описанных арифметических операций достаточно, чтобы отправиться в путь, но прежде чем реализовывать оператор % с помощью процедур, нужно познакомиться с алгоритмом вычисления остатка. Показанный ниже код работает для чисел Ruby:

```
def mod(m, n)
  if n <= m
    mod(m - n, n)
  else
    m
  end
end
```

Например, чтобы вычислить 17 по модулю 5, нужно выполнить следующие действия:

- если 5 меньше или равно 17, а так оно и есть, то вычесть 5 из 17 и применить #mod к результату, то есть вычислить 12 по модулю 5;
- 5 меньше или равно 12, поэтому вычисляем 7 по модулю 5;
- 5 меньше или равно 7, поэтому вычисляем 2 по модулю 5;
- 5 *не* меньше или равно 2, поэтому возвращаем результат 2.

Но реализовать #mod с помощью процедур мы пока не можем, так как в коде выше используется оператор <=, для которого у нас еще нет реализации. Поэтому придется сделать краткое отступление и реализовать <= с помощью процедур.

Можно начать с бессмысленной на первый взгляд реализации метода #less_or_equal? для чисел Ruby:

```
def less_or_equal?(m, n)
    m - n <= 0
end
```

Это решение не очень полезно, потому что сводится все к тому же оператору <=, но теперь мы по крайней мере заменили исходную задачу двумя другими, которые уже рассматривали: вычитанием и сравнением с нулем. С вычитанием мы уже справились, а вот с нулем сравнивали только *на равенство*; но как же реализовать операцию *меньше или равно* нулю?

На самом деле, проблемы-то и нет, потому что нуль – наименьшее из чисел, которые мы умеем реализовывать (напомним, что наши основанные на процедурах числа целые и неотрицательные), поэтому в нашей числовой системе выражение «меньше нуля» не имеет смысла.

Если использовать процедуру SUBTRACT для вычитания большего числа из меньшего, то она вернет ZERO, потому что вернуть отрицательное число никак не может, а ZERO – ближайшая аппроксимация[1]:

```
>> to_integer(SUBTRACT[FIVE][THREE])
=> 2
>> to_integer(SUBTRACT[THREE][FIVE])
=> 0
```

Мы уже написали процедуру IS_ZERO, и поскольку SUBTRACT[m][n] возвращает ZERO, если m меньше или равно n (то есть если n по крайней мере столь же велико, как m), то у нас есть все необходимое для реализации предиката #less_or_equal? с помощью процедур:

```
def less_or_equal?(m, n)
  IS_ZERO[SUBTRACT[m][n]]
end
```

И теперь преобразуем метод в процедуру:

```
IS_LESS_OR_EQUAL =
  -> m { -> n {
    IS_ZERO[SUBTRACT[m][n]]
  } }
```

И как, работает?

```
>> to_boolean(IS_LESS_OR_EQUAL[ONE][TWO])
=> true
>> to_boolean(IS_LESS_OR_EQUAL[TWO][TWO])
=> true
>> to_boolean(IS_LESS_OR_EQUAL[THREE][TWO])
=> false
```

Вроде, все нормально.

---

[1] Вы можете возразить, что в ваших местах операция 3 - 5 = 0 не называется «вычитанием», и будете правы: в математике эта операция называется «монус» (*http://en.wikipedia.org/wiki/Monus*), поскольку неотрицательные числа образуют относительно операции сложения коммутативный моноид (*http://en.wikipedia.org/wiki/Monoid#Commutative_monoid*), а не абелеву группу (*http://en.wikipedia.org/wiki/Abelian_group*).

Только этого фрагмента и недоставало для реализации метода #mod, так что теперь мы можем переписать его через процедуры:

```
def mod(m, n)
  IF[IS_LESS_OR_EQUAL[n][m]][
    mod(SUBTRACT[m][n], n)
  ][
    m
  ]
end
```

И заменим определение метода процедурой:

```
MOD =
  -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      MOD[SUBTRACT[m][n]][n]
    ][
      m
    ]
  } }
```

Отлично! Но будет ли работать?

```
>> to_integer(MOD[THREE][TWO])
SystemStackError: stack level too deep
```

Нет.

Ruby погрузился в бесконечный рекурсивный цикл при вызове MOD, потому что, переводя встроенную функциональность на язык процедур, мы забыли одну важную вещь, касающуюся семантики условных предложений. В языках типа Ruby предложение `if-else` «ленивое»: если задать условие и два блока, то оно вычисляет условие и решает, какой блок вычислить и вернуть, – оба блока не вычисляются никогда.

Но наша реализация IF не может воспользоваться «ленивым» поведением, встроенным в предложение Ruby `if-else`; мы просто говорим «вызвать процедуру IF, передав ей две других процедуры». Соответственно Ruby очертя голову бросается вперед и вычисляет оба аргумента еще до того, как IF получает шанс решить, какой вернуть.

Взгляните еще раз на MOD:

```
MOD =
  -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      MOD[SUBTRACT[m][n]][n]
    ][
```

```
      m
    ]
  } }
```

Мы вызываем `MOD` со значениями для `m` и `n`, Ruby начинает вычислять тело внутренней процедуры, доходит до рекурсивного вызова `MOD[SUBTRACT[m][n]][n]` и сразу же начинает вычислять его как аргумент для передачи `IF` вне зависимости от значения `IS_LESS_OR_EQUAL[n][m]`: `TRUE` или `FALSE`. Второе обращение к `MOD` приводит к следующему безусловному рекурсивному вызову и так далее, отсюда и бесконечная рекурсия.

Чтобы исправить эту ошибку, нам нужно как-то сообщить Ruby, чтобы он отложил вычисление второго аргумента `IF` до момента, когда мы будем знать, что он действительно нужен. Вычисление любого выражения в Ruby можно отложить, обернув его процедурой, но оборачивание произвольного значения Ruby процедурой в общем случае изменяет его смысл (например, результат сложения `1 + 2` не равен `-> { 1 + 2 }`), поэтому нужно что-то более хитрое.

По счастью, ничего изобретать не надо, так как имеет место особый случай: мы знаем, что результатом вызова `MOD` является процедура с одним аргументом, потому что *все* наши значения – процедуры с одним аргументом, и еще мы знаем (из раздела «Равенство» на стр. 206), что обертывание любой процедуры `p` другой процедурой, которая принимает те же аргументы, что `p`, и сразу вызывает `p` с этими аргументами, дает значение, не отличимое от `p`, поэтому мы можем воспользоваться этим трюком, чтобы отложить рекурсивный вызов, не изменяя смысла значения, передаваемого `IF`:

```
MOD =
  -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      -> x {
        MOD[SUBTRACT[m][n]][n][x]
      }
    ][
      m
    ]
  } }
```

Теперь рекурсивный вызов `MOD` обернут процедурой `-> x { …[x] }`, чтобы отложить его выполнение; Ruby не станет пытаться вычислить тело процедуры при вызове `IF`, но если `IF` выберет эту процедуру и вернет ее в качестве результата, то получатель сможет вызвать

ее и наконец инициировать рекурсивный вызов MOD (уже заведомо необходимый).

А теперь MOD заработает?

```
>> to_integer(MOD[THREE][TWO])
=> 1
>> to_integer(MOD[
       POWER[THREE][THREE]
     ][
       ADD[THREE][TWO]
     ])
=> 2
```

Да! Ура!

Но праздновать победу рановато, так как имеется еще одна, более коварная, проблема: мы определяем константу MOD *через константу MOD*, так что это определение уже *не* является невинной аббревиатурой. На этот раз мы не просто присваиваем сложную процедуру константе, чтобы использовать ее впоследствии, а полагаемся на семантику присваивания в Ruby, в соответствии с которой несмотря на то, что MOD, очевидно, еще не определена, когда находится в процессе определения, мы тем не менее можем сослаться на нее в реализации MOD и ожидать, что она *станет* определена в момент, когда мы позже будем ее вычислять.

Это жульничество, потому что у нас должна оставаться принципиальная возможность отказаться от всех аббревиатур – «всюду, где написано MOD, на самом деле имеется в виду вот эта длинная процедура» – однако теперь это невозможно, потому что MOD определена через себя саму.

Эту проблему можно решить с помощью *Y-комбинатора*, широко известного вспомогательного кода, предназначенного именно для этой цели: определения рекурсивной функции без жульничества. Вот как он выглядит:

```
Y = -> f { -> x { f[x[x]] }[-> x { f[x[x]] }] }
```

Объяснить принцип работы Y-комбинатора, не даваясь в многочисленные детали, трудно, но попытаемся дать такое (технически не вполне точное) описание: когда мы вызываем Y-комбинатор, передавая ему процедуру, он вызывает эту процедуру, *передавая ее же в качестве первого аргумента*. Поэтому если написать процедуру, которая ожидает аргумент и затем вызвать Y-комбинатор с этой процедурой, то процедура получит себя же в качестве аргумента и,

следовательно, может использовать этот аргумент, когда захочет вызвать себя.

Увы, по той же причине, по какой MOD попадала в бесконечный цикл, Y-комбинатор в Ruby также зациклится, поэтому нужно его немного модифицировать. Проблема возникает из-за выражения x[x], и исправить ее можно, обернув, как и раньше, все вхождения этого выражения ничего не делающими процедурами -> y { ...[y] }, чтобы отложить вычисление:

```
Z = -> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }
```

Это *Z-комбинатор*, то есть просто Y-комбинатор, адаптированный к строгим языкам типа Ruby.

Наконец-то мы можем написать удовлетворительную реализацию MOD, передав ей дополнительный аргумент f, обернув его вызовом Z-комбинатора и вызывая f там, где мы раньше вызывали MOD:

```
MOD =
  Z[-> f { -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      -> x {
        f[SUBTRACT[m][n]][n][x]
      }
    ][
      m
    ]
  } } }]
```

Слава богу, эта честная версия MOD по-прежнему работает:

```
>> to_integer(MOD[THREE][TWO])
=> 1
>> to_integer(MOD[
       POWER[THREE][THREE]
     ][
       ADD[THREE][TWO]
     ])
=> 2
```

Теперь можно заменить все вхождения % в программу FizzBuzz обращениями к MOD:

```
(ONE..HUNDRED).map do |n|
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    'FizzBuzz'
  ][IF[IS_ZERO[MOD[n][THREE]]][
    'Fizz'
  ][IF[IS_ZERO[MOD[n][FIVE]]][
```

```
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

Списки

Для завершения FizzBuzz нам осталось реализовать еще несколько средств Ruby: диапазон, метод `#map`, строковые литералы и метод `Fixnum#to_s`. Реализацию предыдущих возможностей мы рассматривали во всех деталях, так что на оставшихся постараемся не задерживаться слишком долго (не переживайте, если не все поймете, важно лишь уяснить общий принцип).

Чтобы реализовать диапазоны и метод `#map`, нам понадобятся списки, и построить их проще всего с помощью пар. Наша реализация работает, как связанный список: в каждой паре хранится значение и указатель на следующую пару в списке, только в данном случае место указателей занимают вложенные пары. Стандартные операции со списком выглядят следующим образом:

```
EMPTY    = PAIR[TRUE][TRUE]
UNSHIFT  = -> l { -> x {
             PAIR[FALSE][PAIR[x][l]]
           } }
IS_EMPTY = LEFT
FIRST    = -> l { LEFT[RIGHT[l]] }
REST     = -> l { RIGHT[RIGHT[l]] }
```

А работают они так:

```
>> my_list =
       UNSHIFT[
         UNSHIFT[
           UNSHIFT[EMPTY][THREE]
         ][TWO]
       ][ONE]
=> #<Proc (lambda)>
>> to_integer(FIRST[my_list])
=> 1
>> to_integer(FIRST[REST[my_list]])
=> 2
>> to_integer(FIRST[REST[REST[my_list]]])
=> 3
>> to_boolean(IS_EMPTY[my_list])
=> false
>> to_boolean(IS_EMPTY[EMPTY])
=> true
```

При использовании FIRST и REST для извлечения отдельных элементов из списка получаются очень громоздкие конструкции, поэтому для вывода списка на консоль напишем вспомогательный метод #to_array, как поступали с числами и булевыми значениями:

```
def to_array(proc)
  array = []
  until to_boolean(IS_EMPTY[proc])
    array.push(FIRST[proc])
    proc = REST[proc]
  end
  array
end
```

Теперь инспектировать списки стало проще:

```
>> to_array(my_list)
=> [#<Proc (lambda)>, #<Proc (lambda)>, #<Proc (lambda)>]
>> to_array(my_list).map { |p| to_integer(p) }
=> [1, 2, 3]
```

А как быть с диапазонами? Вместо того чтобы думать, как явным образом представить диапазоны в виде процедур, давайте просто напишем процедуру, которая строит список всех элементов, принадлежащих диапазону. Для встроенных в Ruby чисел и «списков» (то есть массивов) это можно сделать следующим образом:

```
def range(m, n)
  if m <= n
    range(m + 1, n).unshift(m)
  else
    []
  end
end
```

Алгоритм несколько вычурный, поскольку мы памятуем о доступных операциях со списками, но смысл в нем есть: список всех чисел от m до n — то же самое, что список всех чисел от m + 1 до n, в начало которого добавлено число m; если m больше n, то список чисел пуст.

По счастью, у нас уже есть все необходимое, чтобы преобразовать этот метод в процедуру:

```
RANGE =
  Z[-> f {
    -> m { -> n {
      IF[IS_LESS_OR_EQUAL[m][n]][
        -> x {
          UNSHIFT[f[INCREMENT[m]][n]][m][x]
```

```
        }
      ][
        EMPTY
      ]
    } }
  }]
```

Обратите внимание на использование Z-комбинатора вместо рекурсии и откладывающую вычисление процедуру -> x { ... [x] } вокруг истинной ветви условного предложения.

Работает?

```
>> my_range = RANGE[ONE][FIVE]
=> #<Proc (lambda)>
>> to_array(my_range).map { |p| to_integer(p) }
=> [1, 2, 3, 4, 5]
```

Вполне, так что вставляем в FizzBuzz:

```
RANGE[ONE][HUNDRED].map do |n|
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    'FizzBuzz'
  ][IF[IS_ZERO[MOD[n][THREE]]][
    'Fizz'
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

Чтобы реализовать метод `#map`, воспользуемся вспомогательной процедурой `FOLD`, немного напоминающей метод Ruby `Enumerable#inject`:

```
FOLD =
  Z[-> f {
    -> l { -> x { -> g {
      IF[IS_EMPTY[l]][
        x
      ][
        -> y {
          g[f[REST[l]][x][g]][FIRST[l]][y]
        }
      ]
    } } }
  }]
```

.

`FOLD` упрощает написание процедур, которые обрабатывают каждый элемент списка:

```
>> to_integer(FOLD[RANGE[ONE][FIVE]][ZERO][ADD])
=> 15
>> to_integer(FOLD[RANGE[ONE][FIVE]][ONE][MULTIPLY])
=> 120
```

Имея FOLD, мы можем лаконично записать процедуру MAP:

```
MAP =
  -> k { -> f {
    FOLD[k][EMPTY][
      -> l { -> x { UNSHIFT[l][f[x]] } }
    ]
  } }
```

Работает ли MAP?

```
>> my_list = MAP[RANGE[ONE][FIVE]][INCREMENT]
=> #<Proc (lambda)>
>> to_array(my_list).map { |p| to_integer(p) }
=> [2, 3, 4, 5, 6]
```

Да. Заменяем #map в FizzBuzz:

```
MAP[RANGE[ONE][HUNDRED]][-> n {
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    'FizzBuzz'
  ][IF[IS_ZERO[MOD[n][THREE]]][
    'Fizz'
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    'Buzz'
  ][
    n.to_s
  ]]]
}]
```

Почти закончили! Осталось разобраться со строками.

Строки

Со строками все просто: их можно представить в виде списков чисел, если только договориться о кодировке – какое число какому символу соответствует.

Выбрать можно любую кодировку, поэтому вместо универсальной, например ASCII, придумаем свою собственную, более удобную для FizzBuzz. Нам требуется кодировать только цифры и строки 'FizzBuzz', 'Fizz' и 'Buzz', поэтому воспользуемся числами от 0 до 9 для представления символов от '0' до '9' и числами от 10 до 14 для кодирования символов 'B', 'F', 'i', 'u', 'z'.

Этого достаточно для представления интересующих нас строковых литералов (мы позаботились о том, чтобы не затереть Z-комбинатор, представленный константой Z):

```
TEN      = MULTIPLY[TWO][FIVE]
B        = TEN
F        = INCREMENT[B]
I        = INCREMENT[F]
U        = INCREMENT[I]
ZED      = INCREMENT[U]
FIZZ     = UNSHIFT[UNSHIFT[UNSHIFT[UNSHIFT[EMPTY][ZED]][ZED]][I]][F]
BUZZ     = UNSHIFT[UNSHIFT[UNSHIFT[UNSHIFT[EMPTY][ZED]][ZED]][U]][B]
FIZZBUZZ = UNSHIFT[UNSHIFT[UNSHIFT[UNSHIFT[BUZZ][ZED]][ZED]][I]][F]
```

Чтобы убедиться, что это работаст, напишем пару внешних методов для преобразования процедур в строки Ruby:

```
def to_char(c)
  '0123456789BFiuz'.slice(to_integer(c))
end

def to_string(s)
  to_array(s).map { |c| to_char(c) }.join
end
```

Ну так, работают строки или нет?

```
>> to_char(ZED)
=> "z"
>> to_string(FIZZBUZZ)
=> "FizzBuzz"
```

Все отлично, можно вставлять в FizzBuzz:

```
MAP[RANGE[ONE][HUNDRED]][-> n {
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    FIZZBUZZ
  ][IF[IS_ZERO[MOD[n][THREE]]][
    FIZZ
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    BUZZ
  ][
    n.to_s
  ]]]
}]
```

Остался только метод `Fixnum#to_s`. Для этого нужно разбить строку на составляющие цифры, и сделать это можно, например, так:

```
def to_digits(n)
  previous_digits =
```

```
    if n < 10
      []
    else
      to_digits(n / 10)
    end
    previous_digits.push(n % 10)
end
```

Мы не реализовывали оператор `<`, но эту проблему легко обойти, написав `n <= 9` вместо `n < 10`. К несчастью, увильнуть подобным образом от реализации методов `Fixnum#/` и `Array#push` не получится, поэтому приводим код соответствующих процедур:

```
DIV =
  Z[-> f { -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      -> x {
        INCREMENT[f[SUBTRACT[m][n]][n]][x]
      }
    ][
      ZERO
    ]
  } } }]

PUSH =
  -> l {
    -> x {
      FOLD[l][UNSHIFT[EMPTY][x]][UNSHIFT]
    }
  }
```

Теперь можно преобразовать `#to_digits` в процедуру:

```
TO_DIGITS =
  Z[-> f { -> n { PUSH[
    IF[IS_LESS_OR_EQUAL[n][DECREMENT[TEN]]][
      EMPTY
    ][
      -> x {
        f[DIV[n][TEN]][x]
      }
    ]
  ][MOD[n][TEN]] } }]
```

Работает?

```
>> to_array(TO_DIGITS[FIVE]).map { |p| to_integer(p) }
=> [5]
>> to_array(TO_DIGITS[POWER[FIVE][THREE]]).map { |p| to_integer(p) }
=> [1, 2, 5]
```

Да. А поскольку мы предусмотрительно выбрали такую кодировку строк, в которой 1 представляет '1' и т. д., то массивы, порождаемые процедурой TO_DIGITS, уже являются допустимыми строками:

```
>> to_string(TO_DIGITS[FIVE])
=> "5"
>> to_string(TO_DIGITS[POWER[FIVE][THREE]])
=> "125"
```

Так что можно заменить в FizzBuzz #to_s на TO_DIGITS:

```
MAP[RANGE[ONE][HUNDRED]][-> n {
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    FIZZBUZZ
  ][IF[IS_ZERO[MOD[n][THREE]]][
    FIZZ
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    BUZZ
  ][
    TO_DIGITS[n]
  ]]]
}]
```

Решение

Наконец-то мы закончили! (Да, пожалуй, это было бы самое длинное и несуразное собеседование при приеме на работу.) Теперь у нас есть реализация FizzBuzz, написанная исключительно на процедурах. Запустим ее и убедимся, что все работает правильно:

```
>> solution =
       MAP[RANGE[ONE][HUNDRED]][-> n {
         IF[IS_ZERO[MOD[n][FIFTEEN]]][
           FIZZBUZZ
         ][IF[IS_ZERO[MOD[n][THREE]]][
           FIZZ
         ][IF[IS_ZERO[MOD[n][FIVE]]][
           BUZZ
         ][
           TO_DIGITS[n]
         ]]]
       }]
=> #<Proc (lambda)>
>> to_array(solution).each do |p|
       puts to_string(p)
     end; nil
12 Fizz
4
Buzz
Fizz
7:
```

```
94
Buzz
Fizz
97
98
Fizz
Buzz
=> nil
```

Приложив так много усилий к тому, чтобы все константы были не более чем аббревиатурами более длинных выражений, мы просто обязаны заменить каждую константу ее определением и взглянуть на программу целиком:

```
-> k { -> f { -> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }
[-> f { -> l { -> x { -> g { -> b { b }[-> p { p[-> x { -> y { x } }] }[l]][x]
[-> y { g[f[-> l { -> p { p[-> x { -> y { y } }] }[-> p { p[-> x { -> y { y } }] }
[l]] }[l]][x][g]][-> l { -> p { p[-> x { -> y { x } }] }[-> p { p[-> x { -> y
{ y } }] }[l]] }[l]][y] }] } } } }][k][-> x { -> y { -> f { f[x][y] } } }[-> x
{ -> y { x } }][-> x { -> y { x } }]][-> l { -> x { -> l { -> x { -> x { -> y
{ -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
[x][l]] } }[l][f[x]] } }] } }[-> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y
{ x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -> n { n[-> x
{ -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p { p[-
> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x
{ -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { ->
y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-
> p { -> x { x } }]]] }][m] } }[m][n]] } }[m][n]][-> x { -> l { -> x { -> x { -
> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
[x][l]] } }[f[-> n { -> p { -> x { p[n[p][x]] } } }[m]][n]][m][x] }][-> x { ->
y { -> f { f[x][y] } } }[-> x { -> y { x } }][-> x { -> y { x } }]] } } }][-> p
{ -> x { p[x] } }][-> p { -> x
{ p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[
p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[
p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]] } }]][->
n { -> b { b }[-> n { n[-> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-
> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f { -> m
{ -> n { -> b { b }[-> m { -> n { -> n { n[-> x { -> x { -> y { y } } }][-> x
{ -> y { x } }] }[-> m { -> n { n[-> n { -> p { p[-> x { -> y { x } }] }[n[-> p
{ -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y { y } }] }[p]][-> n
{ -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y { y } }] }[p]]] }][-> x
{ -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p { -> x { x } }]]] }]
[m] } }[m][n]] } }[n][m]][-> x { f[-> m { -> n { n[-> n { -> p { p[-> x { -> y
{ x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y
{ y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }][m] } }[m][n]][n][x] }][m] } } }][n][-> p { -> x
{ p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]] } }]]][-> l { -> x { -> x { -
> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }]
[-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f
{ f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }
[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -
```

```
> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x]
[y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-
> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { ->
y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x]
[y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-
> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { ->
y { -> f { f[x][y] } } }[x][l]] } }[-> x { -> y { -> f { f[x][y] } } }[-> x { -
> y { x } }][-> x { -> y { x } }]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n
{ -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -
> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { ->
x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m
{ -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { ->
p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> m { -> n { n[-> m {
-> n { n[-> n { -> p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] }
[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]][-> n { -> p
{ -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -
> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-
> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { ->
p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]][-> n { -> p { -> x
{ p[p[p[p[p[x]]]]] } }]]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p
{ -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x
{ p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }]
[-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { ->
p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p
{ -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x
{ p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }]
[-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { ->
p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]]]]][-> b { b }[-> n { n[-> x { -> x { -> y { y } } }[-
> x { -> y { x } }] }[-> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x]
[y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -> n { n[-> x { -> x
{ -> y { y } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p { p[-> x
{ -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -
> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { f[-> m { -> n { n[-> n { -
> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p
{ p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-
> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x
{ x } }][-> p { -> x { x } }]]] }][m] } }[m][n]][n][x] }][m] } } }][n][-> p { -
> x { p[p[p[x]]] } }]]][-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x
{ -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { ->
x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x]
[y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -
> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x {
-> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x]
[y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -
> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> x { -> y { -> f {
f[x][y] } } }[x][l]] } }[-> x { -> y { -> f { f[x][y] } } }[-> x { -> y { x } }]
[-> x { -> y { x } }]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
```

```
{ p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p]
[x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p
{ -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n
{ -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -
> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { ->
x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m
{ -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -
> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]][-> b { b }[-> n { n[-> x
{ -> x { -> y { y } } }][-> x { -> y { x } }] }[-> f { -> x { f[-> y { x[x]
[y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m
{ -> n { -> n { n[-> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -
> n { n[-> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x]
[y] } } }[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }
[-> p { p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[->
p { -> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { f[-
> m { -> n { n[-> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -
> f { f[x][y] } } }[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> p { p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f
{ f[x][y] } } }[-> p { -> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]][n]
[x] }][m] } } }][n][-> p { -> x { p[p[p[p[p[x]]]]] } }]]][-> l { -> x { -> x { -
> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }]
[-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f
{ f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }
[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -
> y { -> f { f[x][y] } } }[x][l]] } }[-> x { -> y { -> f { f[x][y] } } }[-> x
{ -> y { x } }][-> x { -> y { x } }]][-> n { -> p { -> x { p[n[p][x]] } } }[->
n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n
{ -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -
> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { p[p[x]] } }][-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p
{ -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n {
n[-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { ->
p { -> x { p[n[p][x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]][-> f { -> x
{ f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f { -> n { -> l { -
> x { -> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f
{ -> l { -> x { -> g { -> b { b }[-> p { p[-> x { -> y { x } }] }[l]][x][-> y
{ g[f[-> l { -> p { p[-> x { -> y { y } }] }[-> p { p[-> x { -> y { y } }] }
[l]] }[l]][x][g]][-> l { -> p { p[-> x { -> y { y } }] }[-> p { p[-> x { -> y
{ y } }] }[l]] }[l]][y] }] } } } }][l][-> l { -> x { -> x { -> y { -> f { f[x]
[y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-
> x { -> y { -> f { f[x][y] } } }[-> x { -> y { x } }] }[-> x { -> y { x } }]][x]]
[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -
> y { -> f { f[x][y] } } }[x][l]] } }] } }[-> b { b }[-> m { -> n { -> n { n[-
> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p
```

```
{ p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-
> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x
{ -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }]
[-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][-> n { -> p { p[-> x { -> y
{ x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y
{ y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p]
[x]] } } }[m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p
{ -> x { p[p[p[p[p[x]]]]] } }]]]][-> x { -> y { -> f { f[x][y] } } }[-> x { ->
y { x } }][-> x { -> y { x } }]][-> x { f[-> f { -> x { f[-> y { x[x][y] }] }[-
> x { f[-> y { x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -
> n { n[-> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-
> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }
[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p
{ p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -
> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { -> n { -
> p { -> x { p[n[p][x]] } } }[f[-> m { -> n { n[-> n { -> p { p[-> x { -> y
{ x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y
{ y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }][m] } }[m][n]][n]][x] }][-> p { -> x { x } }] } } }][n][-
> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }[m] } }[m]]
[-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]][x] }]][-> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-
> y { x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -> n { n[-
> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p
{ p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-
> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { -> x { x } }]
[-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { f[-> m { -> n { n[-
> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }
[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p
{ p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -
> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]][n]][-> m
{ -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-
> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]] } }][n]]]] }]
```

Ну просто прелесть.

Более сложные приемы программирования

Конструирование программы из одних лишь процедур требует много усилий, но мы убедились, что это можно сделать, если вы не против приложить толику изобретательности. А теперь познакомимся еще с двумя приемами написания кода в таком минималистском окружении.

Бесконечные потоки

У использования кода для представления данных есть ряд любопытных достоинств. Наши списки на основе процедур не обязаны

быть статическими: список – это просто код, который делает то, что нужно, когда мы передаем его процедурам FIRST и REST, поэтому несложно реализовать списки, вычисляющие свое содержимое на лету. Они известны также под названием *потоков*. На самом деле, нет никаких причин, почему потоки должны быть конечными; ведь вычисление порождает содержимое списка, только когда оно потребляется программой, поэтому ничто не мешает порождать новые значения сколь угодно долго.

Вот, например, как можно реализовать бесконечный поток нулей:

```
ZEROS = Z[-> f { UNSHIFT[f][ZERO] }]
```

Это «честная» версия структуры данных ZEROS = UNSHIFT[ZEROS][ZERO], определенной через саму себя. Программисты обычно ничего не имеют против рекурсивного определения функций, но рекурсивное определение структуры данных может показаться странным и необычным; в данном случае обе версии функционально одинаковы, а благодаря Z-комбинатору обе вполне корректны.

В оболочке легко видеть, что ZEROS ведет себя как список, правда, один его конец теряется вдали:

```
>> to_integer(FIRST[ZEROS])
=> 0
>> to_integer(FIRST[REST[ZEROS]])
=> 0
>> to_integer(FIRST[REST[REST[REST[REST[REST[ZEROS]]]]]])
=> 0
```

Было бы удобно иметь вспомогательный метод для преобразования этого потока в массив Ruby, но метод to_array будет работать бесконечно, если его явно не остановить. Для этой цели подойдет необязательный параметр «максимальный размер»:

```
def to_array(l, count = nil)
  array = []
  until to_boolean(IS_EMPTY[l]) || count == 0
    array.push(FIRST[l])
    l = REST[l]
    count = count - 1 unless count.nil?
  end

  array
end
```

Это позволяет извлечь из потока нужное число элементов и сформировать из них массив:

```
>> to_array(ZEROS, 5).map { |p| to_integer(p) }
=> [0, 0, 0, 0, 0]
>> to_array(ZEROS, 10).map { |p| to_integer(p) }
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>> to_array(ZEROS, 20).map { |p| to_integer(p) }
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Процедура ZEROS не вычисляет новый элемент каждый раз, но сделать это несложно. Вот пример потока, который начинает отсчет от заданного числа:

```
>> UPWARDS_OF = Z[-> f { -> n { UNSHIFT[-> x { f[INCREMENT[n]][x] }][n] } }]
=> #<Proc (lambda)>
>> to_array(UPWARDS_OF[ZERO], 5).map { |p| to_integer(p) }
=> [0, 1, 2, 3, 4]
>> to_array(UPWARDS_OF[FIFTEEN], 20).map { |p| to_integer(p) }
=> [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]
```

Более хитроумный поток содержит все числа, кратные заданному:

```
>> MULTIPLES_OF =
     -> m {
       Z[-> f {
         -> n { UNSHIFT[-> x { f[ADD[m][n]][x] }][n] }
       }][m]
     }
=> #<Proc (lambda)>
>> to_array(MULTIPLES_OF[TWO], 10).map { |p| to_integer(p) }
=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>> to_array(MULTIPLES_OF[FIVE], 20).map { |p| to_integer(p) }
=> [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100]
```

Что удивительно, бесконечными потоками можно манипулировать как обычными списками. Например, можно создать новый поток, применив процедуру к существующему:

```
>> to_array(MULTIPLES_OF[THREE], 10).map { |p| to_integer(p) }
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
>> to_array(MAP[MULTIPLES_OF[THREE]][INCREMENT],10).map { |p| to_integer(p) }
=> [4, 7, 10, 13, 16, 19, 22, 25, 28, 31]
>> to_array(MAP[MULTIPLES_OF[THREE]][MULTIPLY[TWO]], 10).map { |p| to_integer(p) }
=> [6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
```

Можно даже написать процедуру, которая комбинирует два потока, порождая третий:

```
>> MULTIPLY_STREAMS =
       Z[-> f {
          -> k { -> l {
            UNSHIFT[-> x {f[REST[k]][REST[l]][x]}][MULTIPLY[FIRST[k]][FIRST[l]]]
          } }
       }]
=> #<Proc (lambda)>
>> to_array(MULTIPLY_STREAMS[UPWARDS_OF[ONE]][MULTIPLES_OF[THREE]], 10).
       map { |p| to_integer(p) }
=> [3, 12, 27, 48, 75, 108, 147, 192, 243, 300]
```

Поскольку содержимое потока можно порождать с помощью любого вычисления, ничто не мешает нам создать бесконечную последовательность чисел Фибоначчи или простых чисел или всех возможных строк в алфавитном порядке – вообще все, что можно вычислить. Эта мощная абстракция не требует каких-то хитрых средств сверх того, что мы уже имеем.

Встроенные в Ruby потоки

В Ruby имеется класс `Enumerator`, который можно использовать для построения бесконечных потоков, не прибегая к процедурам. Вот как с его помощью реализуется поток «кратные заданному числу»:

```
def multiples_of(n)
  Enumerator.new do |yielder|
    value = n
    loop do
      yielder.yield(value)
      value = value + n
    end
  end
end
```

Этот метод возвращает объект `Enumerator`, который выполняет одну итерацию цикла `loop` при каждом вызове метода `#next` и возвращает значение, полученное от `yield`:

```
>> multiples_of_three = multiples_of(3)
=> #<Enumerator: #<Enumerator::Generator>:each>
>> multiples_of_three.next
=> 3
>> multiples_of_three.next
=> 6
>> multiples_of_three.next
=> 9
```

Класс `Enumerator` включает модуль `Enumerable`, так что мы можем вызывать, в частности, методы `#first`, `#take`, `#detect`:

```
>> multiples_of(3).first
=> 3
>> multiples_of(3).take(10)
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
>> multiples_of(3).detect { |x| x > 100 }
=> 102
```

Другие методы из модуля Enumerable, например #map и #select, не будут правильно работать с этим объектом Enumerator, потому что они пытаются обработать каждый элемент бесконечного потока. Однако в версии Ruby 2.0 имеется класс Enumerator::Lazy, в котором некоторые методы Enumerable реализованы так, что продолжают работать, даже если Enumerator порождает бесконечный поток. Для получения экземпляра Enumerator::Lazy нужно вызвать метод #lazy объекта Enumerator, после чего можно будет манипулировать бесконечными потоками точно так же, как в версии, основанной на процедурах:

```
>> multiples_of(3).lazy.map { |x| x * 2 }.take(10).force
=> [6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
>> multiples_of(3).lazy.map { |x| x * 2 }.select { |x| x > 100 }.take(10).force
=> [102, 108, 114, 120, 126, 132, 138, 144, 150, 156]
>> multiples_of(3).lazy.zip(multiples_of(4)).map { |a, b| a * b }.take(10).force
=> [12, 48, 108, 192, 300, 432, 588, 768, 972, 1200]
```

Это не так элегантно, как в списках на базе процедур, – нам приходится писать специальный код для работы с бесконечными потоками, вместо того чтобы трактовать их как обычные объекты Enumerable, – но все же доказывает, что в Ruby имеется встроенный способ работы с такими необычными структурами данных.

Предотвращение произвольной рекурсии

Решая задачу FizzBuzz, мы использовали рекурсивные функции типа MOD и RANGE, чтобы продемонстрировать применение Z-комбинатора. Это удобно, потому что позволяет перейти от ничем не ограниченной рекурсивной реализации на Ruby к реализации с помощью одних лишь процедур, не изменяя структуру коду. Однако технически мы можем реализовать эти функции и без Z-комбинатора, воспользовавшись поведением нумералов Чёрча.

Например, наша реализация MOD[m][n] повторно вычитает n из m, пока n <= m, всякий раз проверяя это условие, чтобы решить, нужен ли следующий рекурсивный вызов. Но того же результата можно достичь, выполнив действие «вычесть n из m, если n <= m» фиксированное число раз и не используя рекурсию для динамического управления повторением. Мы не знаем точно, сколько раз повторить

это действие, но m раз заведомо достаточно (в худшем случае, когда n равно 1), а лишние попытки ничем не грозят:

```
def decrease(m, n)
  if n <= m
    m - n
  else
    m
  end
end

>> decrease(17, 5)
=> 12
>> decrease(decrease(17, 5), 5)
=> 7
>> decrease(decrease(decrease(17, 5), 5), 5)
=> 2
>> decrease(decrease(decrease(decrease(17, 5), 5), 5), 5), 5)
=> 2
>> decrease(decrease(decrease(decrease(decrease(17, 5), 5), 5), 5), 5), 5)
=> 2
```

Поэтому можно переписать MOD с использованием процедуры, которая принимает число и либо вычитает из него n (если оно больше n), либо возвращает без изменения. Чтобы получить окончательный ответ, эта процедура вызывается m раз для самого числа m:

```
MOD =
  -> m { -> n {
    m[-> x {
      IF[IS_LESS_OR_EQUAL[n][x]][
        SUBTRACT[x][n]
      ][
        x
      ]
    }][m]
  } }
```

Этот вариант MOD работает ничуть не хуже рекурсивного:

```
>> to_integer(MOD[THREE][TWO])
=> 1
>> to_integer(MOD[
     POWER[THREE][THREE]
   ][
     ADD[THREE][TWO]
   ])
=> 2
```

Хотя эта реализация, пожалуй, проще первоначальной, читать ее труднее, и в общем случае она менее эффективна, потому что выполняет столько вызовов, сколько необходимо в худшем случае,

а не останавливается, когда больше делать нечего. Кроме того, она и экстенсионально не равна оригиналу, потому что прежняя версия MOD зациклилась бы при попытке деления на ZERO (условие n <= m никогда не стало бы ложным), тогда как новая просто возвращает свой первый аргумент:

```
>> to_integer(MOD[THREE][ZERO])
=> 3
```

Процедура RANGE чуть потруднее, но можно применить тот же трюк, благодаря которому мы заставили работать DECREMENT: написать такую функцию, которая при вызове n раз с некоторым начальным аргументом возвращает список n чисел из нужного диапазона. Как и в случае DECREMENT, секрет в том, чтобы использовать пару для хранения как результирующего списка, так и информации, необходимой на следующей итерации:

```
def countdown(pair)
  [pair.first.unshift(pair.last), pair.last - 1]
end

>> countdown([[], 10])
=> [[10], 9]
>> countdown(countdown([[], 10]))
=> [[9, 10], 8]
>> countdown(countdown(countdown([[], 10])))
=> [[8, 9, 10], 7]
>> countdown(countdown(countdown(countdown([[], 10]))))
=> [[7, 8, 9, 10], 6]
```

Этот метод легко переписать с помощью процедур:

```
COUNTDOWN = -> p { PAIR[UNSHIFT[LEFT[p]][RIGHT[p]]][DECREMENT[RIGHT[p]]] }
```

Теперь нужно только реализовать RANGE, так чтобы она вызвала COUNTDOWN нужное число раз (диапазон от m до n содержит m - n + 1 элементов) и распаковала результирующий список, хранящийся в конечной паре:

```
RANGE = -> m { -> n { LEFT[INCREMENT[SUBTRACT[n][m]][COUNTDOWN][PAIR[EMPTY][n]]] } }
```

И в этом случае свободная от комбинаторов версия работает правильно:

```
>> to_array(RANGE[FIVE][TEN]).map { |p| to_integer(p) }
=> [5, 6, 7, 8, 9, 10]
```

Мы можем реализовать процедуры MOD и RANGE, выполнив заранее определенное число итераций, – а не гонять цикл, пока его условие не станет истинным, – потому что это *примитивно рекурсивные* функции. Дополнительные сведения по этому вопросу см. в разделе «Частичные рекурсивные функции» на стр. 260.

Реализация лямбда-исчисления

Наш эксперимент с программой FizzBuzz позволил почувствовать, как пишутся программы в бестиповом лямбда-исчислении. Из-за наложенных ограничений мы были вынуждены реализовать большой объем функциональности с нуля, не опираясь на средства языка, но в конце концов нам удалось построить все структуры данных и алгоритмы, необходимые для решения поставленной задачи.

Конечно, мы не писали *настоящих* программ в лямбда-исчислении, потому у нас нет для него интерпретатора; мы всего лишь написали на Ruby несколько программ в духе лямбда-исчисления, чтобы составить представление о том, как может работать такой минималистский язык. Но теперь у нас есть знания, необходимые, чтобы построить интерпретатор лямбда-исчисления и с его помощью вычислять настоящие выражения лямбда-исчисления. Попробуем.

Синтаксис

Бестиповое лямбда-исчисление – это язык программирования, содержащий всего три вида выражений: переменные, определения функций и вызовы. Вместо того чтобы вводить новый конкретный синтаксис выражений лямбда-исчисления, мы будем придерживаться нотации Ruby: переменные записываются в виде `x`, функции – в виде `-> x { x }`, вызовы – в виде `x[y]` – и постараемся не путать оба языка.

Почему «лямбда-исчисления»?

В этом контексте слово «исчисление» означает систему правил для манипулирования цепочками символов[1]. В стандартном синтаксисе лямбда-исчисления вместо символа `->` применяется греческая буква лямбда (`λ`); например, процедура `ONE` записывается в виде `λp.λx.p x`.

---

[1] У многих людей оно ассоциируется с дифференциальным и интегральным исчислением, математическими понятиями, связанными со скоростью изменения и суммированием величин.

Мы можем реализовать синтаксические классы `LCVariable`, `LCFunction` и `LCCall` обычным образом:

```
class LCVariable < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    to_s
  end
end

class LCFunction < Struct.new(:parameter, :body)
  def to_s
    "-> #{parameter} { #{body} }"
  end

  def inspect
    to_s
  end
end

class LCCall < Struct.new(:left, :right)
  def to_s
    "#{left}[#{right}]"
  end

  def inspect
    to_s
  end
end
```

Эти классы позволят строить абстрактные синтаксические деревья для выражений лямбда-исчисления – точно так же, как мы это делали для языка Simple в главе 2 и для регулярных выражений в главе 3.

```
>> one =
       LCFunction.new(:p,
         LCFunction.new(:x,
           LCCall.new(LCVariable.new(:p), LCVariable.new(:x))
         )
       )
=> -> p { -> x { p[x] } }
>> increment =
       LCFunction.new(:n,
         LCFunction.new(:p,
           LCFunction.new(:x,
             LCCall.new(
               LCVariable.new(:p),
               LCCall.new(
                 LCCall.new(LCVariable.new(:n), LCVariable.new(:p)),
```

```
                LCVariable.new(:x)
              )
            )
          )
        )
      )
=> -> n { -> p { -> x { p[n[p][x]] } } }
>> add =
      LCFunction.new(:m,
        LCFunction.new(:n,
          LCCall.new(
            LCCall.new(LCVariable.new(:n), increment), LCVariable.new(:m)
          )
        )
      )
=> -> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }
```

Поскольку синтаксис языка столь минимален, этих трех классов достаточно для представления любой программы в лямбда-исчислении.

Семантика

Теперь мы придадим лямбда-исчислению операционную семантику мелких шагов, реализовав в каждом синтаксическом классе метод `#reduce`. Семантика мелких шагов хороша тем, что позволяет видеть отдельные шаги вычисления – для выражений Ruby это сделать не так-то просто.

Замена переменных

Прежде чем приступать к методу `#reduce`, напишем еще один метод `#replace`, который будет искать все вхождения указанной переменной в выражение и заменять их другим выражением:

```
class LCVariable
  def replace(name, replacement)
    if self.name == name
      replacement
    else
      self
    end
  end
end

class LCFunction
  def replace(name, replacement)
    if parameter == name
      self
    else
      LCFunction.new(parameter, body.replace(name, replacement))
    end
```

```
  end
end

class LCCall
  def replace(name, replacement)
    LCCall.new(left.replace(name, replacement), right.replace(name, replacement))
  end
end
```

Для переменных и вызовов работа этого метода очевидна:

```
>> expression = LCVariable.new(:x)
=> x
>> expression.replace(:x, LCFunction.new(:y, LCVariable.new(:y)))
=> -> y { y }
>> expression.replace(:z, LCFunction.new(:y, LCVariable.new(:y)))
=> x
>> expression =
       LCCall.new(
         LCCall.new(
           LCCall.new(
             LCVariable.new(:a),
             LCVariable.new(:b)
           ),
           LCVariable.new(:c)
         ),
         LCVariable.new(:b)
       )
=> a[b][c][b]
>> expression.replace(:a, LCVariable.new(:x))
=> x[b][c][b]
>> expression.replace(:b, LCFunction.new(:x, LCVariable.new(:x)))
=> a[-> x { x }][c][-> x { x }]
```

Для функций ситуация сложнее. Метод `#replace` применяется только к телу функции и заменяет лишь *свободные переменные* – такие, которые не *привязаны* к функции, то есть не являются ее параметрами:

```
>> expression =
       LCFunction.new(:y,
         LCCall.new(LCVariable.new(:x), LCVariable.new(:y))
       )
=> -> y { x[y] }
>> expression.replace(:x, LCVariable.new(:z))
=> -> y { z[y] }
>> expression.replace(:y, LCVariable.new(:z))
=> -> y { x[y] }
```

Это позволяет заменять вхождения переменной в выражение, не затрагивая по ошибке не относящиеся к делу переменные, которые по чистому совпадению имеют такое же имя:

```
>> expression =
      LCCall.new(
        LCCall.new(LCVariable.new(:x), LCVariable.new(:y)),
        LCFunction.new(:y, LCCall.new(LCVariable.new(:y), LCVariable.new(:x)))
      )
=> x[y][-> y { y[x] }]
>> expression.replace(:x, LCVariable.new(:z))
=> z[y][-> y { y[z] }] ❶
>> expression.replace(:y, LCVariable.new(:z))
=> x[z][-> y { y[x] }] ❷
```

❶ Оба вхождения x свободны в исходном выражении, поэтому оба заменяются.

❷ Только первое вхождение y – свободная переменная, поэтому только оно и заменяется. Второе вхождение y – параметр функции, а не переменная, а третье – переменная, принадлежащая функции, поэтому трогать ее не следует.

Наша простая реализация #replace на некоторых исходных данных не работает. Она неправильно обрабатывает случай, когда подставляемая строка содержит свободные переменные:

```
>> expression =
      LCFunction.new(:x,
        LCCall.new(LCVariable.new(:x), LCVariable.new(:y))
      )
=> -> x { x[y] }
>> replacement = LCCall.new(
      LCVariable.new(:z),
      LCVariable.new(:x)
   )
=> z[x]
>> expression.replace(:y, replacement)
=> -> x { x[z[x]] }
```

Нельзя просто вставить z[x] в тело функции -> x { … }, потому что x в выражении z[x] – свободная переменная и должна остаться таковой после замены, но сейчас она оказалась *захваченной* параметром функции с таким же именем[1].

Мы можем не обращать внимания на этот дефект, потому что будем вычислять только выражения, не содержащие свободных переменных, так что никакой проблемы не возникнет, но помните, что в общем случае необходима более тщательная реализация.

[1] Правильно было бы автоматически переименовывать параметр функции, так чтобы избежать конфликтов со свободными переменными: переписать -> x { x[y] } в виде эквивалентного выражения, к примеру -> w { w[y] }, а затем безопасно произвести замену, получив -> w { w[z[x]] }, оставив x свободной.

Вызов функций

Единственное назначение метода `#replace` – дать нам возможность реализовать семантику вызовов функции. В Ruby, если процедура вызывается с одним или более аргументами, то ее тело вычисляется в окружении, где каждый аргумент присвоен локальной переменной, так что вхождения этих переменных ведут себя как сами аргументы: в метафорическом смысле вызов процедуры `-> x, y { x + y }` с аргументами `1` и `2` порождает промежуточное выражение `1 + 2`, и именно оно вычисляется для получения конечного результата.

Ту же идею, но более буквально, мы можем применить в лямбда-исчислении, реально заменив переменные в теле функции при вычислении вызова. Для этого определим метод `LCFunction#call`, который производит замену и возвращает результат:

```
class LCFunction
  def call(argument)
    body.replace(parameter, argument)
  end
end
```

Теперь мы можем смоделировать вызов функции:

```
>> function =
      LCFunction.new(:x,
        LCFunction.new(:y,
          LCCall.new(LCVariable.new(:x), LCVariable.new(:y))
        )
      )
=> -> x { -> y { x[y] } }
>> argument = LCFunction.new(:z, LCVariable.new(:z))
=> -> z { z }
>> function.call(argument)
=> -> y { -> z { z }[y] }
```

Свертка выражений

Вызовы функций – единственное, что *происходит* при выполнении программы в лямбда-исчислении, поэтому мы готовы к тому, чтобы реализовать метод `#reduce`. Этот метод находит в выражении место, где может произойти вызов функции, а затем с помощью `#call` производит его. Нам нужно только научиться понимать, какие выражения действительно допускают вызов...

```
class LCVariable
  def callable?
```

```
    false
  end
end

class LCFunction
  def callable?
    true
  end
end

class LCCall
  def callable?
    false
  end
end
```

...после чего можно написать `#reduce`:

```
class LCVariable
  def reducible?
    false
  end
end

class LCFunction
  def reducible?
    false
  end
end

class LCCall
  def reducible?
    left.reducible? || right.reducible? || left.callable?
  end

  def reduce
    if left.reducible?
      LCCall.new(left.reduce, right)
    elsif right.reducible?
      LCCall.new(left, right.reduce)
    else
      left.call(right)
    end
  end
end
```

В этой реализации вызовы функций – единственная синтаксическая конструкция, допускающая свертывание. Свертывание `LCCall` немного напоминает свертывание `Add` и `Multiply` в языке SIMPLE: если хотя бы одно подвыражение допускает свертку, то мы его сворачиваем; если нет, то выполняется вызов левого подвыражения (которое должно иметь тип `LCFunction`), передавая правое в качестве

аргумента. Эта стратегия называется *вызовом по значению* – сначала аргумент сворачивается до конца, а затем выполняется вызов.

Протестируем нашу реализацию, применив лямбда-исчисление для вычисления «один плюс один»:

```
>> expression = LCCall.new(LCCall.new(add, one), one)
=> -> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }↵
[-> p { -> x { p[x] }}][-> p { -> x { p[x] } }]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
     end; puts expression
-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[-> p↵
{ -> x { p[x] } }][-> p { -> x { p[x] } }]
-> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][-> p { -> x { p[x] } }] }↵
[-> p { -> x { p[x] } }]
-> p { -> x { p[x] } }[-> n { -> p { -> x { p[n[p][x]] } } }][-> p↵
{ -> x { p[x] } }]
-> x { -> n { -> p { -> x { p[n[p][x]] } } }[x] }[-> p { -> x { p[x] } }]
-> n { -> p { -> x { p[n[p][x]] } } }[-> p { -> x { p[x] } }]
-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }
=> nil
```

Что-то, безусловно, произошло, но результат не совпадает с ожидаемым: получилось выражение `> p { -> x { p[-> p { -> x { p[x] } } [p][x]] } }`, тогда как в лямбда-исчислении число 2 представляется выражением `-> p { -> x{ p[p[x]] } })]`. Что не так?

Расхождение вызвано используемой нами стратегией вычисления. В результате остались вызовы функций, допускающие свертывание, – например, вызов `-> p { -> x { p[x] } }[p]` можно было бы свернуть в `-> x { p[x] }` – но метод `#reduce` не обратил на них внимания, потому что они находятся внутри тела функции, а согласно нашей семантике функции не сворачиваются[1].

Однако, как было сказано в разделе «Равенство» на стр. 206, два выражения с разным синтаксисом, можно считать равными, если они обладают одинаковым поведением. Мы знаем, как должно вести себя представление числа 2 в лямбда-исчислении: если передать ему два аргумента, то оно дважды вызовет первый аргумент, передав ему второй. Поэтому попробуем вызвать наше выражение с двумя

[1] Мы могли бы исправить эту ошибку, применив в методе `#reduce` более агрессивную стратегию вычисления (например, *аппликативный* или *нормальный* порядок вычисления), при которой в телах функций свертка производится, однако тело функции, взятое само по себе, обычно содержит свободные переменные, так что для этого понадобилась бы более качественная реализация метода `#replace`.

специально созданными переменными inc и zero[1] и посмотрим, что оно делает на самом деле:

```
>> inc, zero = LCVariable.new(:inc), LCVariable.new(:zero)
=> [inc, zero]
>> expression = LCCall.new(LCCall.new(expression, inc), zero)
=> -> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }[inc][zero]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
   end; puts expression
-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }[inc][zero]
-> x { inc[-> p { -> x { p[x] } }[inc][x]] }[zero]
inc[-> p { -> x { p[x] } }[inc][zero]]
inc[-> x { inc[x] }[zero]]
inc[inc[zero]]
=> nil
```

Это именно то поведение, которого мы ожидаем от числа два, поэтому -> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } } – все-таки правильный результат, несмотря на то, что выглядит он не так, как мы ожидали.

Синтаксический разбор

Теперь, имея работающую семантику, доведем дело до конца, построив синтаксический анализатор выражений лямбда-исчисления. Как обычно, для описания грамматики воспользуемся библиотекой Treetop:

```
grammar LambdaCalculus
  rule expression
    calls / variable / function
  end

  rule calls
    first:(variable / function) rest:('[' expression ']')+ {
      def to_ast
        arguments.map(&:to_ast).inject(first.to_ast) { |l, r| LCCall.new(l,r) }
      end

      def arguments
        rest.elements.map(&:expression)
      end
    }
```

[1] Мы рискуем, вычисляя выражение, содержащее свободные переменные inc и zero, но, к счастью, ни у одной из функций, встречающихся в выражении, нет аргументов с такими именами, поэтому в данном частном случае нет опасности случайно захватить какую-то переменную.

```
  end

  rule variable
    [a-z]+ {
      def to_ast
        LCVariable.new(text_value.to_sym)
      end
    }
  end

  rule function
    '-> ' parameter:[a-z]+ ' { ' body:expression ' }' {
      def to_ast
        LCFunction.new(parameter.text_value.to_sym, body.to_ast)
      end
    }
  end
end
```

В разделе «Реализация синтаксических анализаторов» на стр. 82 отмечалось, что Treetop-грамматики обычно порождают правоассоциативные деревья, из-за чего в этой грамматике приходится прилагать дополнительные усилия, чтобы правильно отразить принятый в лямбда-исчислении левоассоциативный синтаксис вызова функций. Правило `calls` соответствует одному или нескольким последовательным вызовам (например, `a[b][c][d]`), а метод `#to_ast` получающегося конкретного узла синтаксического дерева использует метод `Enumerable#inject`, чтобы преобразовать эти вызовы в левоассоциативное абстрактное синтаксическое дерево.

Синтаксический анализатор вкупе с операционной семантикой дают полную реализацию лямбда-исчисления, позволяя читать и вычислять выражения:

```
>> require 'treetop'
=> true
>> Treetop.load('lambda_calculus')
=> LambdaCalculusParser
>> parse_tree = LambdaCalculusParser.new.parse('-> x { x[x] }[-> y { y }]')
=> SyntaxNode+Calls2+Calls1 offset=0, "…}[-> y { y }]" (to_ast,arguments,first,rest):
     SyntaxNode+Function1+Function0 offset=0, "… x { x[x] }" (to_ast,parameter,body):
       SyntaxNode offset=0, "-> "
       SyntaxNode offset=3, "x":
         SyntaxNode offset=3, "x"
       SyntaxNode offset=4, " { "
       SyntaxNode+Calls2+Calls1 offset=7, "x[x]" (to_ast,arguments,first,rest):
         SyntaxNode+Variable0 offset=7, "x" (to_ast):
           SyntaxNode offset=7, "x"
         SyntaxNode offset=8, "[x]":
           SyntaxNode+Calls0 offset=8, "[x]" (expression):
```

```
              SyntaxNode offset=8, "["
              SyntaxNode+Variable0 offset=9, "x" (to_ast):
                SyntaxNode offset=9, "x"
              SyntaxNode offset=10, "]"
          SyntaxNode offset=11, " }"
        SyntaxNode offset=13, "[-> y { y }]":
          SyntaxNode+Calls0 offset=13, "[-> y { y }]" (expression):
            SyntaxNode offset=13, "["
            SyntaxNode+Function1+Function0 offset=14, "… { y }" (to_ast,parameter,body):
              SyntaxNode offset=14, "-> "
              SyntaxNode offset=17, "y":
                SyntaxNode offset=17, "y"
              SyntaxNode offset=18, " { "
              SyntaxNode+Variable0 offset=21, "y" (to_ast):
                SyntaxNode offset=21, "y"
              SyntaxNode offset=22, " }"
            SyntaxNode offset=24, "]"
>> expression = parse_tree.to_ast
=> -> x { x[x] }[-> y { y }]
>> expression.reduce
=> -> y { y }[-> y { y }]
```

Глава 7. Универсальность повсюду

Источником наблюдаемой нами в жизни сложности по большей части являются сложные системы – млекопитающие, микропроцессоры, экономика, погода, – поэтому естественно предположить, что простая система умеет делать только простые вещи. Однако в этой книге мы видели, что простые системы могут обладать впечатляющими возможностями: в главе 6 было показано, что даже совсем минимальный язык программирования способен выполнять полезную работу, а в главе 5 мы намекнули, как построить универсальную машину Тьюринга, которая умеет читать кодированное описание другой машины и моделировать ее выполнение.

Существование универсальной машины Тьюринга – факт исключительной важности. Пусть даже каждая отдельно взятая машина Тьюринга имеет фиксированный свод правил, наличие универсальной машины Тьюринга доказывает, что можно сконструировать устройство, которое сможет адаптироваться к любой задаче, прочитав с ленты инструкции. Инструкции по сути дела представляют собой программу, которая управляет работой оборудования машины, – точно так же, как обстоит дело в программируемых компьютерах общего назначения, которыми мы ежедневно пользуемся[1]. Автоматы – конечные и с магазинной памятью – *слишком* просты для поддержки такой полноценной программируемости, но машина Тьюринга обладает для этого достаточной сложностью.

---

[1] Под «оборудованием» понимается головка считывания/записи, лента и свод правил. Это не есть оборудование в буквальном понимании, потому машина Тьюринга — обычно умозрительный эксперимент, а не физический объект, однако оно «твердое» (hard) в том смысле, что является неизменной частью системы, в противоположность постоянно изменяющейся «мягкой» (soft) информации, которая существует в виде записанных на ленте символов.

В этой главе мы рассмотрим несколько простых систем и убедимся, что все они универсальны – каждая способна смоделировать машину Тьюринга, а, значит, и выполнить произвольную заданную программу, а не только зашитые в систему правила. Таким образом, универсальность встречается куда чаще, чем можно было бы ожидать.

Лямбда-исчисление

Мы видели, что лямбда-исчисление – пригодный к использованию язык программирования, но пока не знаем, является ли он столь же мощным, как машина Тьюринга. На самом деле, лямбда-исчисление должно быть никак не менее мощным, потому что, как выясняется, с его помощью можно смоделировать любую машину Тьюринга, в том числе (а как же иначе?) *универсальную*.

Давайте посмотрим, как это работает, для чего по-быстрому реализуем часть машины Тьюринга – ленту – на лямбда-исчислении.

Как и в главе 6, мы будем представлять выражения лямбда-исчисления в виде Ruby-кода, это возможно при условии, что код не делает ничего, кроме создания процедур, вызова процедур и использования констант в качестве аббревиатур.

Немного рискованно принимать в игру Ruby, коль скоро не он является языком, который мы собираемся исследовать, но в обмен мы получаем знакомый синтаксис выражений и простой способ их вычисления, а все наши открытия останутся в силе, если мы не станем выходить за пределы ограничений.

У ленты машины Тьюринга есть четыре атрибута: список символов в левой части ленты, символ в середине ленты (там, где находится головка считывания/записи), список символов в правой части и символ, обозначающий незаполненную позицию. Мы можем представить эти четыре значения в виде пары пар:

```
TAPE         = -> l { -> m { -> r { -> b {PAIR[PAIR[l][m]][ PAIR[r][b]]} } } }
TAPE_LEFT    = -> t { LEFT[LEFT[t]] }
TAPE_MIDDLE  = -> t { RIGHT[LEFT[t]] }
TAPE_RIGHT   = -> t { LEFT[RIGHT[t]] }
TAPE_BLANK   = -> t { RIGHT[RIGHT[t]] }
```

`TAPE` играет роль конструктора, который принимает четыре атрибута ленты в виде аргументов и возвращает представляющую ленту процедуру, а `TAPE_LEFT`, `TAPE_MIDDLE`, `TAPE_RIGHT` и `TAPE_BLANK` – ак-

цессоры, которые принимают представление ленты и возвращают соответствующий атрибут.

Имея такую структуру данных, мы можем реализовать процедуру TAPE_WRITE, которая принимает ленту и символ и возвращает новую ленту, на которой этот символ записан в средней позиции:

```
TAPE_WRITE = -> t { -> c { TAPE[TAPE_LEFT[t]][c][TAPE_RIGHT[t]][TAPE_BLANK[t]] } }
```

Мы также можем определить операции для перемещения головки. Ниже приведена процедура TAPE_MOVE_HEAD_RIGHT, которая сдвигает головку на один квадратик вправо, – она получена путем преобразования не имеющего ограничений Ruby-метода Tape#move_head_right из раздела «Моделирование» на стр. 180[1].

```
TAPE_MOVE_HEAD_RIGHT =
  -> t {
    TAPE[
      PUSH[TAPE_LEFT[t]][TAPE_MIDDLE[t]]
    ][
      IF[IS_EMPTY[TAPE_RIGHT[t]]][
        TAPE_BLANK[t]
      ][
        FIRST[TAPE_RIGHT[t]]
      ]
    ][
      IF[IS_EMPTY[TAPE_RIGHT[t]]][
        EMPTY
      ][
        REST[TAPE_RIGHT[t]]
      ]
    ][
      TAPE_BLANK[t]
    ]
  }
```

Вместе взятые, эти операции дают все необходимое для создания ленты, чтения, записи и перемещения головки. Например, можно начать с чистой ленты и записать последовательность чисел в последовательные квадратики:

```
>> current_tape = TAPE[EMPTY][ZERO][EMPTY][ZERO]
=> #<Proc (lambda)>
>> current_tape = TAPE_WRITE[current_tape][ONE]
=> #<Proc (lambda)>
```

[1] Реализация метода TAPE_MOVE_HEAD_LEFT аналогична, хотя требует дополнительных функций работы со списками, которые не были определены в разделе «Списки» на стр. 228.

```
>> current_tape = TAPE_MOVE_HEAD_RIGHT[current_tape]
=> #<Proc (lambda)>
>> current_tape = TAPE_WRITE[current_tape][TWO]
=> #<Proc (lambda)>
>> current_tape = TAPE_MOVE_HEAD_RIGHT[current_tape]
=> #<Proc (lambda)>
>> current_tape = TAPE_WRITE[current_tape][THREE]
=> #<Proc (lambda)>
>> current_tape = TAPE_MOVE_HEAD_RIGHT[current_tape]
=> #<Proc (lambda)>
>> to_array(TAPE_LEFT[current_tape]).map { |p| to_integer(p) }
=> [1, 2, 3]
>> to_integer(TAPE_MIDDLE[current_tape])
=> 0
>> to_array(TAPE_RIGHT[current_tape]).map { |p| to_integer(p) }
=> []
```

Мы опускаем остальные детали, но следуя по этому пути, нетрудно построить основанные на процедурах представления состояния, конфигураций, правил и сводов правил. Собрав воедино все эти кусочки, мы сможем написать с помощью одних процедур реализации методов `DTM#step` и `DTM#run`: процедура `STEP` моделирует один шаг машины Тьюринга путем применения свода правил к одной конфигурации для порождения другой, а процедура `RUN` моделирует весь процесс выполнения машины, применяя Z-комбинатор для повторяющихся вызовов `STEP`, до тех пор пока не окажется применимых правил или машина не попадет в состояние останова.

Иными словами, `RUN` – это программа на лямбда-исчислении, которая может смоделировать любую машину Тьюринга[1]. Оказывается, что верно и обратное: машина Тьюринга может работать как интерпретатор лямбда-исчисления: сохранить представление выражения лямбда-исчисления на ленте и повторно обновлять его в соответствии с правилами свертки – в точности, как операционная семантика из раздела «Семантика» на стр. 247.

Поскольку любая машина Тьюринга может быть смоделирована программой на лямбда-исчислении, а любая программа на лямбда-исчислении может быть смоделирована машиной Тьюринга, то обе системы равномощны. Это удивительный результат, так как машины Тьюринга и программы на лямбда-исчислении работают совершенно по-разному, и априори нет никаких причин ожидать, что их возможности одинаковы.

[1] Термином «Тьюринг-полный» часто называют язык программирования или систему, способные смоделировать произвольную машину Тьюринга.

Это означает, что существует по крайней мере один способ смоделировать лямбда-исчисление собственными средствами: сначала реализовать на лямбда-исчислении машину Тьюринга, а затем использовать ее для прогона интерпретатора лямбда-исчисления. Такая модель внутри модели крайне неэффективна, и такой же результат можно получить более элегантно: спроектировать структуры данных для представления выражений лямбда-исчисления, а затем непосредственно реализовать операционную семантику. Тем не менее, это доказывает, что лямбда-исчисление универсально само по себе и не нуждается в чем-то дополнительном. Самоинтерпретатор – это версия универсальной машины Тьюринга, написанная на лямбда-исчислении: хотя сама интерпретирующая программа фиксирована, мы можем поручить ей любую работу, подав на вход подходящее выражение лямбда-исчисления.

Как мы видели, истинное достоинство универсальной системы заключается в том, что ее можно запрограммировать для решения разных задач, а не зашивать единственную жесткую программу. В частности, универсальную систему можно запрограммировать для моделирования другой универсальной системы; универсальная машина Тьюринга способна вычислять выражения лямбда-исчисления, а интерпретатор лямбда-исчисления может моделировать выполнение машины Тьюринга.

Частично рекурсивные функции

Если выражения лямбда-исчисления состоят исключительно из создания и вызовов процедур, то *частично рекурсивные функции* – это программы, собираемые из четырех фундаментальных структурных элементов в различных комбинациях. Первые два называются `zero` и `increment`, мы можем реализовать их в виде методов Ruby следующим образом:

```
def zero
  0
end

def increment(n)
  n + 1
end
```

Никаких хитростей: один метод возвращает числу нуль, второй прибавляет к числу единицу:

```
>> zero
=> 0
>> increment(zero)
=> 1
>> increment(increment(zero))
=> 2
```

Мы можем использовать `#zero` и `#increment` для определения новых методов, хотя и не очень интересных:

```
>> def two
     increment(increment(zero))
   end
=> nil
>> two
=> 2
>> def three
     increment(two)
   end
=> nil
>> three
=> 3
>> def add_three(x)
     increment(increment(increment(x)))
   end
=> nil
>> add_three(two)
=> 5
```

Третий структурный элемент, метод `#recurse`, более сложен:

```
def recurse(f, g, *values)
  *other_values, last_value = values
  if last_value.zero?
    send(f, *other_values)
  else
    easier_last_value = last_value - 1
    easier_values = other_values + [easier_last_value]
    easier_result = recurse(f, g, *easier_values)
    send(g, *easier_values, easier_result)
  end
end
```

`#recurse` принимает в качестве аргументов два имени методов, `f` и `g`, и использует их для выполнения рекурсивного вычисления с некоторыми входными значениями. Результат вызова `#recurse` вычисляется путем делегирования работы одному из методов `f` или `g` в зависимости от последнего входного значения.

- Если последнее входное значение равно нулю, то `#recurse` вызывает метод `f`, передавая ему в качестве аргументов все значения, кроме последнего.

- Если последнее входное значение не равно нулю, то `#recurse` сначала декрементирует его, затем вызывает себя же с новыми входными аргументами, а затем – метод `g`, которому передает те же самые аргументы и результат рекурсивного вызова.

Звучит это сложнее, чем есть на самом деле; `#recurse` – это просто шаблон для определения рекурсивных функций определенного вида. Например, с его помощью можно определить метод `#add`, который принимает два аргумента, `x` и `y`, и складывает их. Для этого нам потребуются еще два метода, отвечающие на следующие вопросы.

- Если дано значение `x`, то чему равно значение `add(x, 0)`?
- Если даны значения `x`, `y - 1` и `add(x, y - 1)`, то чему равно значение `add(x, y)`?

Первый вопрос простой: сложение числа с нулем не изменяет его, поэтому, если мы знаем значение `x`, то значение `add(x, 0)` точно такое же. Мы можем реализовать это в виде метода `#add_zero_to_x`, который просто возвращает свой аргумент:

```
def add_zero_to_x(x)
  x
end
```

Второй вопрос посложнее, но лишь чуть-чуть: если мы уже знаем значение `add(x, y - 1)`, то для получения `add(x,y)`должны просто увеличить его на единицу[1]. Это означает, что нам нужен метод, который инкрементирует свой третий аргумент (`#recurse` вызывает его с аргументами `x`, `y - 1` и `add(x, y - 1)`). Назовем его `#increment_easier_result`:

```
def increment_easier_result(x, easier_y, easier_result)
  increment(easier_result)
end
```

Собирая все вместе, получаем определение метода `#add`, построенное на основе `#recurse` и `#increment`:

```
def add(x, y)
  recurse(:add_zero_to_x, :increment_easier_result, x, y)
end
```

[1] Поскольку вычитание – операция, обратная сложению, то `(x + (y – 1)) + 1 = (x + (y + –1)) + 1`. Поскольку сложение ассоциативно, то `(x + (y + –1)) + 1 = (x + y) + (–1 + 1)`. А поскольку `–1 + 1 = 0`, нейтральному элементу относительно сложения, то `(x + y) + (–1 + 1) = x + y`.

Здесь применен тот же подход, что и в главе 6: мы можем использовать определения методов только для придания удобных имен выражениям, но включать в них рекурсию запрещено[1]. Если требуется написать рекурсивный метод, следует использовать `#recurse`.

Проверим, что `#add` делает то, что нужно:

```
>> add(two, three)
=> 5
```

Нормально. Ту же стратегию можно использовать для реализации других знакомых примеров, например `#multiply`...

```
def multiply_x_by_zero(x)
  zero
end

def add_x_to_easier_result(x, easier_y, easier_result)
  add(x, easier_result)
end

def multiply(x, y)
  recurse(:multiply_x_by_zero, :add_x_to_easier_result, x, y)
end
```

...и `#decrement`...

```
def easier_x(easier_x, easier_result)
  easier_x
end

def decrement(x)
  recurse(:zero, :easier_x, x)
end
```

...и `#subtract`:

```
def subtract_zero_from_x(x)
  x
end

def decrement_easier_result(x, easier_y, easier_result)
  decrement(easier_result)
end
```

[1] Конечно, в реализации самого метода `#recurse` рекурсия используется, но это допустимо, потому что мы считаем `#recurse` одним из четырех встроенных примитивов системы, а не методом, определенным пользователем.

```
def subtract(x, y)
  recurse(:subtract_zero_from_x, :decrement_easier_result, x, y)
end
```

Все они работают, как и ожидается:

```
>> multiply(two, three)
=> 6
>> def six
       multiply(two, three)
   end
=> nil
>> decrement(six)
=> 5
>> subtract(six, two)
=> 4
>> subtract(two, six)
=> 0
```

Программы, которые можно составить из методов `#zero`, `#increment` и `#recurse` называются *примитивно* рекурсивными функциями.

Все примитивно рекурсивные функции являются *тотальными*: вне зависимости от количества входных аргументов они обязательно останавливаются и возвращают результат. Объясняется это тем, что `#recurse` – единственный допустимый способ определить рекурсивный метод, а сам `#recurse` останавливается всегда: после каждого рекурсивного вызова последний аргумент становится ближе к нулю, и, как только он станет равен нулю, а это неизбежно произойдет, рекурсия остановится.

Методов `#zero`, `#increment` и `#recurse` достаточно для построения многих полезных функций, в том числе всех операций, необходимых для выполнения одного шага машины Тьюринга: содержимое ленты машины Тьюринга можно представить в виде одного большого числа, а затем использовать примитивно рекурсивные функции, чтобы считать символ под головкой, записать на ленту новый символ и сдвинуть головку влево или вправо. Однако смоделировать полное выполнение произвольной машины Тьюринга с помощью примитивно рекурсивных функций невозможно, потому что некоторые машины Тьюринга циклятся бесконечно. Следовательно, примитивно рекурсивные функции не универсальны.

Чтобы получить действительно универсальную систему, нужно добавить четвертую фундаментальную операцию, `#minimize`:

```
def minimize
  n = 0
  n = n + 1 until yield(n).zero?
  n
end
```

Метод `#minimize` принимает блок и повторно вычисляет его, передавая один числовой аргумент. При первом вызове блоку передается `0`, затем `1`, `2` и так далее до тех пор, пока блок не вернет нуль.

Добавив `#minimize` в набор, состоящий из `#zero`, `#increment` и `#recurse`, мы сможем построить гораздо больше функций – все *частично* рекурсивные функции – в том числе такие, которые не останавливаются. Например, с помощью `#minimize` нетрудно реализовать метод `#divide`:

```
def divide(x, y)
  minimize { |n| subtract(increment(x), multiply(y, increment(n))) }
end
```

Выражение `subtract(increment(x), multiply(y, increment(n)))` построено так, чтобы для значений `n`, при которых `y * (n + 1)` больше `x`, возвращался нуль. Если мы попытаемся разделить 13 на 4 (`x = 13, y = 4`), то при увеличении `n` выражение `y * (n + 1)` будет принимать следующие значения:

n	x	y * (n + 1)	y * (n + 1) больше x?
0	13	4	нет
1	13	8	нет
2	13	12	нет
3	13	16	да
4	13	20	да
5	13	24	да

Первое значение `n`, для которого условие выполняется, равно 3, поэтому блок, переданный методу `#minimize`, в первый раз вернет нуль при `n = 3`, так что результат деления `divide(13, 4)` равен 3.

При вызове метода `#divide` с разумными аргументами он всегда возвращает результат, как примитивно рекурсивная функция:

```
>> divide(six, two)
=> 3
>> def ten
       increment(multiply(three, three))
   end
```

```
=> nil
>> ten
=> 10
>> divide(ten, three)
=> 3
```

Но `#divide` может и не вернуть ответ, поэтому `#minimize` может циклиться бесконечно. Деление на нуль не определено:

```
>> divide(six, zero)
SystemStackError: stack level too deep
```

Немного странно видеть здесь ошибку переполнения стека, потому что реализация `#minimize` итеративна и сама по себе не приводит к росту стека вызовов, однако переполнение происходит из-за того, что `#divide` обращается к рекурсивному методу `#multiply`. Глубина рекурсии при вызове `#multiply` определяется его вторым аргументом, `increment(n)`, а, поскольку `#minimize` исполняет бесконечный цикл, то значение `n` становится очень большим, что в конце концов и приводит к переполнению стека.

Имея в своем арсенале `#minimize`, мы можем полностью смоделировать машину Тьюринга, повторно вызывая примитивно рекурсивную функцию, которая выполняет один шаг модели. Моделирование продолжается до тех пор, пока машина не остановится, – если этого не произойдет, то и модель будет работать вечно.

SKI-исчисление

SKI-исчислением называется система правил для манипулирования синтаксисом выражений – как и лямбда-исчисление. И хотя лямбда-исчисление очень простое, в нем все же есть выражения трех видов – переменные, функции и вызовы – и, как мы видели в разделе «Семантика» на стр. 247, наличие переменных несколько усложняет правила свертки. SKI-исчисление еще проще, в нем всего два вида выражений – вызовы и буквенные *символы* – и гораздо более простые правила. Вся его мощь проистекает из трех специальных символов `S`, `K` и `I` (называемых *комбинаторами*), для каждого из которых есть свое правило свертки:

- свернуть `S[a][b][c]` в `a[c][b[c]]`, где `a`, `b`, `c` – произвольные выражения SKI-исчисления;
- свернуть `K[a][b]` в `a`;
- свернуть `I[a]` в `a`.

Например, вот один из способов свернуть выражение `I[S][K][S][I[K]]`:

```
I[S][K][S][I[K]]  → S[K][S][I[K]] (свернуть I[S] в S)
                  → S[K][S][K] (свернуть I[K] to K)
                  → K[K][S[K]] (свернуть S[K][S][K] в K[K][S[K]])
                  → K (свернуть K[K][S[K]] в K)
```

Отметим, что никаких замен переменных в духе лямбда-исчисления здесь нет и в помине, есть лишь символы, которые переупорядочиваются, дублируются и отбрасываются в соответствии с правилами свертки.

Легко реализовать абстрактный синтаксис SKI-выражений:

```
class SKISymbol < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    to_s
  end
end

class SKICall < Struct.new(:left, :right)
  def to_s
    "#{left}[#{right}]"
  end

  def inspect
    to_s
  end
end

class SKICombinator < SKISymbol
end

S, K, I = [:S, :K, :I].map { |name| SKICombinator.new(name) }
```

Здесь мы определяем классы `SKICall` и `SKISymbol` для представления вызовов и символов соответственно, а затем создаем объекты `S`, `K`, `I` (каждый в единственном экземпляре), представляющие конкретные символы, которые будут выступать в роли комбинаторов.

Мы могли бы сделать `S`, `K` и `I` экземплярами самого класса `SKISymbol`, но решили сделать их экземплярами подкласса `SKICombinator`. Прямо сейчас это не дает никаких преимуществ, но впоследствии упростит добавление в объекты-комбинаторы новых методов.

Эти классы и объекты можно использовать для построения абстрактных синтаксических деревьев SKI-выражений:

```
>> x = SKISymbol.new(:x)
=> x
>> expression = SKICall.new(SKICall.new(S, K), SKICall.new(I, x))
=> S[K][I[x]]
```

Мы можем придать SKI-исчислению операционную семантику мелких шагов, реализовав правила свертки и применив их к выражениям. Сначала определим метод `#call` для экземпляров класса `SKICombinator`; каждый из объектов `S`, `K`, `I` получит собственное определение этого метода, в котором реализовано соответствующее правило свертки:

```
# свернуть S[a][b][c] в a[c][b[c]]
def S.call(a, b, c)
  SKICall.new(SKICall.new(a, c), SKICall.new(b, c))
end

# свернуть K[a][b] в a
def K.call(a, b)
  a
end

# свернуть I[a] в a
def I.call(a)
  a
end
```

Итак, мы теперь имеем возможность применять правила исчисления, если только знаем, с какими аргументами комбинатор вызывается...

```
>> y, z = SKISymbol.new(:y), SKISymbol.new(:z)
=> [y, z]
>> S.call(x, y, z)
=> x[z][y[z]]
```

...но чтобы использовать `#call` с реальным SKI-выражением, нам нужно извлечь из него комбинатор и аргументы. Это довольно мутное дело, потому что выражение представлено в виде двоичного дерева объектов типа `SKICall`:

```
>> expression = SKICall.new(SKICall.new(SKICall.new(S, x), y), z)
=> S[x][y][z]
>> combinator = expression.left.left.left
=> S
>> first_argument = expression.left.left.right
```

```
=> x
>> second_argument = expression.left.right
=> y
>> third_argument = expression.right
=> z
>> combinator.call(first_argument, second_argument, third_argument)
=> x[z][y[z]]
```

Чтобы упростить работу с этой структурой, мы можем определить методы `#combinator` и `#arguments` для абстрактных синтаксических деревьев:

```
class SKISymbol
  def combinator
    self
  end

  def arguments
    []
  end
end

class SKICall
  def combinator
    left.combinator
  end

  def arguments
    left.arguments + [right]
  end
end
```

Это дает простой способ определить, какой комбинатор вызывать и с какими аргументами:

```
>> expression
=> S[x][y][z]
>> combinator = expression.combinator
=> S
>> arguments = expression.arguments
=> [x, y, z]
>> combinator.call(*arguments)
=> x[z][y[z]]
```

Для выражения `S[x][y][z]` все работает отлично, но в общем случае есть две проблемы. Во-первых, метод `#combinator` просто возвращает самый левый *символ* в выражении, а этот символ вовсе необязательно является комбинатором:

```
>> expression = SKICall.new(SKICall.new(x, y), z)
=> x[y][z]
```

```
>> combinator = expression.combinator
=> x
>> arguments = expression.arguments
=> [y, z]
>> combinator.call(*arguments)
NoMethodError: undefined method `call' for x:SKISymbol
```

Во-вторых, даже если самый левый символ *является* комбинатором, он необязательно вызывается с правильным числом аргументов:

```
>> expression = SKICall.new(SKICall.new(S, x), y)
=> S[x][y]
>> combinator = expression.combinator
=> S
>> arguments = expression.arguments
=> [x, y]
>> combinator.call(*arguments)
ArgumentError: wrong number of arguments (2 for 3)
```

Чтобы предотвратить обе проблемы, определим предикат `#callable?`, который проверяет, пригодны ли для вызова метода `#call` результаты, возвращенные методами `#combinator` и `#arguments`. Символ сам по себе не допускает вызова, а комбинатор допускает вызов, только если количество аргументов правильное:

```
class SKISymbol
  def callable?(*arguments)
    false
  end
end

def S.callable?(*arguments)
  arguments.length == 3
end

def K.callable?(*arguments)
  arguments.length == 2
end

def I.callable?(*arguments)
  arguments.length == 1
end
```

По счастью, в Ruby уже есть способ узнать у метода, сколько аргументов он ожидает (его *арность*):

```
>> def add(x, y)
     x + y
   end
=> nil
>> add_method = method(:add)
```

```
=> #<Method: Object#add>
>> add_method.arity
=> 2
```

Поэтому можно было бы заменить отдельные реализации метода `#callable?` в объектах `S`, `K`, `I` одной общей:

```
class SKICombinator
  def callable?(*arguments)
    arguments.length == method(:call).arity
  end
end
```

Теперь мы научились распознавать выражения, к которым правила свертки применимы непосредственно:

```
>> expression = SKICall.new(SKICall.new(x, y), z)
=> x[y][z]
>> expression.combinator.callable?(*expression.arguments)
=> false
>> expression = SKICall.new(SKICall.new(S, x), y)
=> S[x][y]
>> expression.combinator.callable?(*expression.arguments)
=> false
>> expression = SKICall.new(SKICall.new(SKICall.new(S, x), y), z)
=> S[x][y][z]
>> expression.combinator.callable?(*expression.arguments)
=> true
```

Наконец, мы готовы реализовать методы `#reducible?` и `#reduce` для SKI-выражений:

```
class SKISymbol
  def reducible?
    false
  end
end

class SKICall
  def reducible?
    left.reducible? || right.reducible? || combinator.callable?(*arguments)
  end

  def reduce
    if left.reducible?
      SKICall.new(left.reduce, right)
    elsif right.reducible?
      SKICall.new(left, right.reduce)
    else
      combinator.call(*arguments)
    end
  end
end
```

Метод `SKICall#reduce` рекурсивно ищет подвыражение, которое мы умеем сворачивать – например, комбинатор S, вызываемый с тремя аргументами, – а затем применяет соответствующее правило с помощью `#call`.

Ну вот! Теперь мы можем вычислять SKI-выражения, сворачивая их до тех пор, пока не окажется, что дальнейшая свертка невозможна. Например, выражение `S[K[S[I]]][K]`, будучи вызвано с символами `x` и `y`, меняет два аргумента местами:

```
>> swap = SKICall.new(SKICall.new(S, SKICall.new(K, SKICall.new(S, I))), K)
=> S[K[S[I]]][K]
>> expression = SKICall.new(SKICall.new(swap, x), y)
=> S[K[S[I]]][K][x][y]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
   end; puts expression
S[K[S[I]]][K][x][y]
K[S[I]][x][K[x]][y]
S[I][K[x]][y]
I[y][K[x][y]]
y[K[x][y]]
y[x]
=> nil
```

SKI-исчисление, обладающее всего тремя простыми правилами, может порождать на удивление сложное поведение – настолько сложное, что оказывается универсальным. Доказать его универсальность можно, показав, как транслировать произвольное выражение лямбда-исчисления в SKI-выражение, которое делает то же самое, то есть воспользоваться SKI-исчислением, чтобы придать денотационную семантику лямбда-исчислению. Мы уже знаем, что лямбда-исчисление универсально, поэтому если SKI-исчисление может его полностью смоделировать, значит, и SKI-исчисление универсально.

В основе трансляции лежит метод `#as_a_function_of`:

```
class SKISymbol
  def as_a_function_of(name)
    if self.name == name
      I
    else
      SKICall.new(K, self)
    end
  end
end

class SKICombinator
  def as_a_function_of(name)
```

```
    SKICall.new(K, self)
  end
end

class SKICall
  def as_a_function_of(name)
    left_function = left.as_a_function_of(name)
    right_function = right.as_a_function_of(name)
    SKICall.new(SKICall.new(S, left_function), right_function)
  end
end
```

Понимать все детали работы метода `#as_a_function_of` необязательно; грубо говоря, он преобразует SKI-выражение в другое, которое превращается в исходное при вызове с одним аргументом. Например, выражение `S[K][I]` преобразуется в `S[S[K[S]][K[K]]][K[I]]`:

```
>> original = SKICall.new(SKICall.new(S, K), I)
=> S[K][I]
>> function = original.as_a_function_of(:x)
=> S[S[K[S]][K[K]]][K[I]]
>> function.reducible?
=> false
```

Когда `S[S[K[S]][K[K]]][K[I]]` вызывается с одним аргументом, скажем символом `y`, оно сворачивается обратно в `S[K][I]`:

```
>> expression = SKICall.new(function, y)
=> S[S[K[S]][K[K]]][K[I]][y]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
   end; puts expression
S[S[K[S]][K[K]]][K[I]][y]
S[K[S]][K[K]][y][K[I][y]]
K[S][y][K[K][y]][K[I][y]]
S[K[K][y]][K[I][y]]
S[K][K[I][y]]
S[K][I]
=> nil
>> expression == original
=> true
```

Параметр `name` используется, только если исходное выражение содержит символ с таким именем. В данном случае метод `#as_a_function_of` порождает нечто более интересное: выражение, которое при вызове с одним аргументом сворачивается в исходное выражение с этим аргументом вместо символа:

```
>> original = SKICall.new(SKICall.new(S, x), I)
=> S[x][I]
>> function = original.as_a_function_of(:x)
=> S[S[K[S]][I]][K[I]]
>> expression = SKICall.new(function, y)
=> S[S[K[S]][I]][K[I]][y]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
   end; puts expression
S[S[K[S]][I]][K[I]][y]
S[K[S]][I][y][K[I][y]]
K[S][y][I[y]][K[I][y]]
S[I[y]][K[I][y]]
S[y][K[I][y]]
S[y][I]
=> nil
>> expression == original
=> false
```

Это явная реализация способа замены переменных в теле функции лямбда-исчисления при ее вызове. По существу, метод `#as_a_function_of` позволяет использовать SKI-выражение как тело функции: он создаст новое выражение, которое ведет себя в точности как функция с конкретным телом и именем параметра, хотя в SKI-исчислении и нет явного синтаксиса для функций.

Способность SKI-исчисления имитировать поведение функций позволяет без труда транслировать выражения лямбда-исчисления в SKI-выражения. Переменные и вызовы лямбда-исчисления становятся символами и вызовами SKI-исчисления, а тело каждой функции лямбда-исчисления преобразуется в «функцию» SKI-исчисления с помощью метода `#as_a_function_of`:

```
class LCVariable
  def to_ski
    SKISymbol.new(name)
  end
end

class LCCall
  def to_ski
    SKICall.new(left.to_ski, right.to_ski)
  end
end

class LCFunction
  def to_ski
    body.to_ski.as_a_function_of(parameter)
  end
end
```

Проверим, как работает эта трансляция, преобразовав представление числа два в лямбда-исчислении (см. раздел «Числа» на стр. 209) в SKI-исчисление:

```
>> two = LambdaCalculusParser.new.parse('-> p { -> x { p[p[x]] } }').to_ast
=> -> p { -> x { p[p[x]] } }
>> two.to_ski
=> S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]]
```

Делает ли выражение SKI-исчисления `S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]]` то же самое, что выражение лямбда-исчисления `-> p { -> x { p[p[x]] } }`? Что касается последнего, то оно должно дважды вызывать свой первый аргумент, передавая ему второй аргумент, так что мы можем передать выражению SKI-исчисления какие-нибудь аргументы и посмотреть, действительно ли оно именно это и делает, – так же, как в разделе «Семантика» на стр. 247:

```
>> inc, zero = SKISymbol.new(:inc), SKISymbol.new(:zero)
=> [inc, zero]
>> expression = SKICall.new(SKICall.new(two.to_ski, inc), zero)
=> S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]][inc][zero]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]][inc][zero]
S[K[S]][S[K[K]][I]][inc][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
K[S][inc][S[K[K]][I][inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[S[K[K]][I][inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[K][inc][I[inc]]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[I[inc]]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[inc]][S[K[S]][S[K[K]][I]][inc][K[I][inc]]][zero]
S[K[inc]][K[S][inc][S[K[K]][I][inc]][K[I][inc]]][zero]
S[K[inc]][S[S[K[K]][I][inc]][K[I][inc]]][zero]
S[K[inc]][S[K[K][inc][I[inc]]][K[I][inc]]][zero]
S[K[inc]][S[K[I[inc]]][K[I][inc]]][zero]
S[K[inc]][S[K[inc]][K[I][inc]]][zero]
S[K[inc]][S[K[inc]][I]][zero]
K[inc][zero][S[K[inc]][I][zero]]
inc[S[K[inc]][I][zero]]
inc[K[inc][zero][I[zero]]]
inc[inc[I[zero]]]
inc[inc[zero]]
=> nil
```

Как и следовало ожидать, в результате вычисления вызова преобразованного выражения с символами `inc` и `zero` получилось

inc[inc[zero]], то есть в точности то, что мы хотели. Такая трансляция работает и для любого другого выражения лямбда-исчисления, поэтому SKI-исчисление может полностью смоделировать лямбда-исчисление и потому обязано быть универсальным.

Хотя в SKI-исчислении есть три комбинатора, на самом деле комбинатор I избыточен. Существует много выражений, содержащих только S и K, которые делают то же самое, что I; например, рассмотрим поведение выражения S[K][K]:

```
>> identity = SKICall.new(SKICall.new(S, K), K)
=> S[K][K]
>> expression = SKICall.new(identity, x)
=> S[K][K][x]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
   end; puts expression
S[K][K][x]
K[x][K[x]]
x
=> nil
```

Таким образом, поведение S[K][K] такое же, как у I, и в действительности это верно для любого SKI-выражения вида S[K][что угодно]. Комбинатор I – не более чем синтаксическая глазурь, без которой вполне можно жить; для универсальности достаточно всего двух комбинаторов S и K.

Iota

Греческой буквой йота (ι) обозначается дополнительный комбинатор, который можно добавить в SKI-исчисление. Правило свертки для него выглядит так: свернуть ι[a] в a[S][K].

Наша реализация SKI-исчисления позволяет без труда добавить еще один комбинатор:

```
IOTA = SKICombinator.new('ι')

# свернуть ι[a] в a[S][K]
def IOTA.call(a)
  SKICall.new(SKICall.new(a, S), K)
end

def IOTA.callable?(*arguments)
  arguments.length == 1
end
```

Крис Баркер (Chris Barker) предложил язык Iota (*http://semarch.linguistics.fas.nyu.edu/barker/Iota/*), в программах на котором используется *только* комбинатор ι. И хотя в нем есть всего один комбинатор, Iota – универсальный язык, потому что на него можно транслировать любое выражение SKI-исчисления, а, как мы видели, SKI-исчисление универсально.

Для преобразования SKI-выражения в Iota следует применить такие правила подстановки:

- заменить S на ι[ι[ι[ι[ι]]]];
- заменить на ι[ι[ι[ι]]];
- заменить I на ι[ι].

Нетрудно реализовать это преобразование:

```
class SKISymbol
  def to_iota
    self
  end
end

class SKICall
  def to_iota
    SKICall.new(left.to_iota, right.to_iota)
  end
end

  def S.to_iota
    SKICall.new(IOTA, SKICall.new(IOTA, SKICall.new(IOTA, SKICall.new(IOTA, IOTA))))
  end

  def K.to_iota
    SKICall.new(IOTA, SKICall.new(IOTA, SKICall.new(IOTA, IOTA)))
  end

  def I.to_iota
    SKICall.new(IOTA, IOTA)
end
```

Вовсе не очевидно, что Iota-версии комбинаторов S, K, I эквивалентны исходным, поэтому изучим этот вопрос, свернув каждый из них в SKI-исчислении и понаблюдав за поведением. Вот что происходит, когда мы транслируем S в Iota, а затем выполняем свертку:

```
>> expression = S.to_iota
=> ι[ι[ι[ι[ι]]]]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
```

```
   end; puts expression
ι[ι[ι[ι[ι]]]]
ι[ι[ι[ι[S][K]]]]
ι[ι[ι[S[S][K][K]]]]
ι[ι[ι[S[K][K[K]]]]]
ι[ι[S[K][K[K]][S][K]]]
ι[ι[K[S][K[K][S]][K]]]
ι[ι[K[S][K][K]]]
ι[ι[S[K]]]
ι[S[K][S][K]]
ι[K[K][S[K]]]
ι[K]
K[S][K]
S
=> nil
```

Как видим, `ι[ι[ι[ι[ι]]]]` действительно эквивалентно `S`. То же самое верно и для `K`:

```
>> expression = K.to_iota
=> ι[ι[ι[ι]]]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
   end; puts expression
ι[ι[ι[ι]]]
ι[ι[ι[S][K]]]
ι[ι[S[S][K][K]]]
ι[ι[S[K][K[K]]]]
ι[S[K][K[K]][S][K]]
ι[K[S][K[K][S]][K]]
ι[K[S][K][K]]
ι[S[K]]
S[K][S][K]
K[K][S[K]]
K
=> nil
```

Для `I` все не так гладко. Правило свертки `ι` порождает выражения, содержащие только комбинаторы `S` и `K`, поэтому нет никакой надежды получить в итоге литерал `I`:

```
>> expression = I.to_iota
=> ι[ι]
>> while expression.reducible?
       puts expression
       expression = expression.reduce
   end; puts expression
ι[ι]
ι[S][K]
S[S][K][K]
S[K][K[K]]
=> nil
```

Очевидно, что `S[K][K[K]]` синтаксически не *равно* `I`, но это еще один пример выражения, которое содержит только комбинаторы `S` и `K` и *делает то же самое, что* `I`:

```
>> identity = SKICall.new(SKICall.new(S, K), SKICall.new(K, K))
=> S[K][K[K]]
>> expression = SKICall.new(identity, x)
=> S[K][K[K]][x]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[K][K[K]][x]
K[x][K[K][x]]
K[x][K]
x
=> nil
```

Таким образом, трансляция на язык Iota действительно сохраняет *поведение* всех трех комбинаторов SKI, хотя и не в точности сохраняет их синтаксис. Чтобы проверить результат, преобразуем уже знакомое выражения лямбда-исчисления в Iota через его представление на SKI-исчислении, а затем вычислим его и понаблюдаем за поведением:

```
>> two
=> -> p { -> x { p[p[x]] } }
>> two.to_ski
=> S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]]
>> two.to_ski.to_iota
=> ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[↵
ι]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι[ι]↵
]]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι]]][ι[ι]]]]
>> expression = SKICall.new(SKICall.new(two.to_ski.to_iota, inc), zero)
=> ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]][ι[↵
ι[ι[ι]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι↵
[ι[ι]]]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι]]][ι[ι]]↵
]][inc][zero]
>> expression = expression.reduce while expression.reducible?
=> nil
>> expression
=> inc[inc[zero]]
```

И снова получился ожидаемый результат `inc[inc[zero]]`, так что Iota-выражение `ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]][ι[ι[ι[ι]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι]]][ι[ι]]]]` –действительно результат трансля-

ции выражения `-> p { -> x { p[p[x]] } }` на язык, в котором нет ни переменных, ни функций, а только один комбинатор. И поскольку такая трансляция возможна для любого выражения лямбда-исчисления, Iota также является универсальным языком.

Таг-системы

Таг-система – это модель вычислений, напоминающая упрощенную машину Тьюринга, только вместо перемещения головки вдоль ленты таг-система работает с цепочкой, добавляя новые символы в ее конец и удаляя символы из начала. В некоторых отношениях цепочка таг-системы аналогична ленте машины Тьюринга, но таг-система может работать только с краями цепочки и «перемещается» только в одном направлении – к ее концу.

Описание таг-системы состоит из двух частей: во-первых, набора правил, в котором каждое правило определяет, какие символы добавлять в конец цепочки при появлении определенного символа в начале – например, «если в начале цепочки находится символ `a`, дописать в конец символы `bcd`» – и, во-вторых, число, называемое *количеством удалений*, которое определяет, сколько символов удалить из начала цепочки после выполнения правила.

Приведем пример таг-системы:

- если в начале цепочки находится символ `a`, дописать в конец символы `bc`;
- если в начале цепочки находится символ `b`, дописать в конец символы `caad`;
- если в начале цепочки находится символ `c`, дописать в конец символы `ccd`;
- после выполнения любого из перечисленных выше правил удалить из начала цепочки три символа – иными словами, количество удалений равно 3.

Мы можем выполнить вычисление в таг-системе, раз за разом применяя правила и удаляя символы, пока не получится цепочка, к первому символу которой неприменимо ни одно правило или длина которой меньше количества удалений[1]. Попробуем прогнать эту таг-систему с начальной цепочкой `'aaaaaa'`:

---

[1] Второе условие позволяет избежать ситуации, когда требуется удалить больше символов, чем имеется в цепочке.

aaaaaa	Если в начале цепочки находится символ a, дописать в конец символы bc
aaabc	Если в начале цепочки находится символ a, дописать в конец символы bc
bcbc	Если в начале цепочки находится символ b, дописать в конец символы caad
ccaad	Если в начале цепочки находится символ c, дописать в конец символы ccd
adccd	Если в начале цепочки находится символ a, дописать в конец символы bc
cdbc	Если в начале цепочки находится символ c, дописать в конец символы ccd
cccd	Если в начале цепочки находится символ c, дописать в конец символы ccd
dccd	–

Таг-системы умеют работать только с цепочками символов, но можно заставить их производить нетривиальные операции и со значениями других типов, например числами, при условии, что их можно подходящим образом представить в виде цепочек символов. Вот один из возможных способов кодирования чисел: число n представляется в виде цепочки aa, за которой n раз следует цепочка bb; например, числу 3 соответствует цепочка aabbbbbb.

Некоторые аспекты этого представления могут показаться избыточными – число 3 можно было бы представить просто цепочкой aaa – но использование пар символов и явного маркера в начале цепочки, как мы вскоре увидим, оказывается весьма полезным.

Выбрав схему кодирования чисел, мы можем построить таг-систему, которая будет выполнять операции с числами, манипулируя их представлениями в виде цепочек. Так, следующая таг-система удваивает исходное число:

- если в начале цепочки находится символ a, дописать в конец символы aa;
- если в начале цепочки находится символ b, дописать в конец символы bbbb;
- после выполнения любого из перечисленных выше правил удалить из начала цепочки три символа (количество удалений равно 2).

Посмотрим, как поведет себя эта таг-система, если подать ей на вход цепочку aabbbb, представляющую число 2:

```
aabbbb → bbbbaa
       → bbaabbbb
       → aabbbbbbbb (представляет число 4)
       → bbbbbbbbaa
       → bbbbbbaabbbb
       → bbbbaabbbbbbbb
       → bbaabbbbbbbbbbbb
       → aabbbbbbbbbbbbbbbb (число 8)
       → bbbbbbbbbbbbbbbbaa
       → bbbbbbbbbbbbbbbaabbbb
       ⋮
```

Удвоение, очевидно, имеет место, но эта таг-система работает бесконечно – сначала удваивает число, представленное входной цепочкой, потом удваивает результат, снова удваивает и так далее – это совсем не то, что мы хотели. Чтобы построить систему, которая удваивает число один раз и затем останавливается, нам нужно использовать для кодирования результата другие символы, которые бы не инициировали следующий круг удвоения. Это можно сделать, разрешив использовать в нашей схеме удвоения символы `c` и `d` вместо `a` и `b` и изменив правила так, что при создании удвоенного числа вместо `aa` и `bbbb` в конец цепочки дописывается `cc` и `dddd` соответственно.

После этих изменений вычисление выглядит так:

```
aabbbb → bbbbcc
       → bbccdddd
       → ccdddddddd (число 4, закодированное с помощью c и d вместо a и b)
```

Модифицированная система останавливается по достижении `ccdddddddd`, потому что не существует правила для цепочек, начинающихся символом `c`.

В данном случае для остановки вычисления в нужный момент нам необходим только символ `c`, а символ `b` вполне можно было бы оставить и не заменять символом `d`, но в использовании большего числа символов, чем строго необходимо, нет никакого вреда.

Вообще говоря, система, в которой для кодирования входных и выходных значений используются разные наборы символов, проще для понимания, чем та, в которой они пересекаются; как мы вскоре увидим, это также позволяет объединять несколько малых систем в одну более крупную, организовав выходную кодировку одной системы, так чтобы она совпадала с входной кодировкой другой.

Для моделирования таг-системы на Ruby нам необходима реализация одного правила (`TagRule`), свода правил (`TagRulebook`) и самой таг-системы (`TagSystem`):

```
class TagRule < Struct.new(:first_character, :append_characters)
  def applies_to?(string)
    string.chars.first == first_character
  end

  def follow(string)
    string + append_characters
  end
end

class TagRulebook < Struct.new(:deletion_number, :rules)
  def next_string(string)
    rule_for(string).follow(string).slice(deletion_number..-1)
  end

  def rule_for(string)
    rules.detect { |r| r.applies_to?(string) }
  end
end

class TagSystem < Struct.new(:current_string, :rulebook)
  def step
    self.current_string = rulebook.next_string(current_string)
  end
end
```

Эта реализация позволяет выполнять вычисление пошагово – по одному правилу за раз. Воспользуемся ей для удвоения числа, только на этот раз будем удваивать число 3 (`aabbbbbb`):

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'aa'), TagRule.new('b', 'bbbb')])
=> #<struct TagRulebook …>
>> system = TagSystem.new('aabbbbbb', rulebook)
=> #<struct TagSystem …>
>> 4.times do
       puts system.current_string
       system.step
   end; puts system.current_string
aabbbbbb
bbbbbbaa
bbbbaabbbb
bbaabbbbbbbb
aabbbbbbbbbbbb
=> nil
```

Поскольку эта таг-система работает бесконечно, нам нужно заранее знать, сколько шагов выполнять до появления результата – в данном случае четыре шага – но если бы мы воспользовались мо-

дифицированной версией, которая кодирует результат с помощью c и d, то могли бы оставить ее работать, пока она автоматически не завершится. Добавим необходимый для этого код.

```
class TagRulebook
  def applies_to?(string)
      !rule_for(string).nil? && string.length >= deletion_number
  end
end

class TagSystem
  def run
    while rulebook.applies_to?(current_string)
      puts current_string
      step
    end
    puts current_string
  end
end
```

Теперь можно вызвать метод `TagSystem#run` модифицированной системы и дать ей возможность остановиться естественным образом:

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'cc'), TagRule.new('b', 'dddd')])
=> #<struct TagRulebook …>
>> system = TagSystem.new('aabbbbbb', rulebook)
=> #<struct TagSystem …>
>> system.run
aabbbbbb
bbbbbbcc
bbbbccdddd
bbccdddddddd
ccddddddddddd
=> nil
```

Эта реализация таг-систем позволяет исследовать, на что еще они способны. Наша схема кодирования дает возможность легко строить системы, выполняющие и другие операции над числами, например, деление пополам:

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'cc'), TagRule.new('b', 'd')])
=> #<struct TagRulebook …>
>> system = TagSystem.new('aabbbbbbbbbbbb', rulebook)
=> #<struct TagSystem …>
>> system.run
aabbbbbbbbbbbb
bbbbbbbbbbbbcc
bbbbbbbbbbccd
bbbbbbbbccdd
bbbbbbccddd
bbbbccdddd
bbccddddd
```

```
ccdddddd
=> nil
```

Или увеличивать число на единицу:

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'ccdd'), TagRule.new('b', 'dd')])
=> #<struct TagRulebook …>
>> system = TagSystem.new('aabbbb', rulebook)
=> #<struct TagSystem …>
>> system.run
aabbbb
bbbbccdd
bbccdddd
ccdddddd
=> nil
```

Можно также соединить две таг-системы при условии, что выходная кодировка первой совпадает с входной кодировкой второй. Ниже приведен пример одной системы, которая комбинирует правила удвоения и инкремента, применяя символы `c` и `d` для кодирования чисел, подаваемых на вход правил инкремента, и `e` и `f` – для кодирования их выхода:

```
>> rulebook = TagRulebook.new(2, [
       TagRule.new('a', 'cc'), TagRule.new('b', 'dddd'),  # double
       TagRule.new('c', 'eeff'), TagRule.new('d', 'ff')   # increment
    ])
=> #<struct TagRulebook …>
>> system = TagSystem.new('aabbbb', rulebook)
=> #<struct TagSystem …>
>> system.run
aabbbb (число 2)
bbbbcc
bbccdddd
ccddddddddd (число 4) ❶
ddddddddeeff
dddddddeffff
ddddeffffff
ddeffffffff
eeffffffffff (число 5) ❷
=> nil
```

❶ Правила удвоения преобразуют 2 в число 4, кодированное символами `c` и `d`.

❷ Правила инкремента преобразуют 4 в число 5, кодированное символами `e` и `f`.

Таг-системы могут не только преобразовывать одни числа в другие, но и проверять их математические свойства. Следующая таксистема проверяет, является число четным или нечетным:

```
>> rulebook = TagRulebook.new(2, [
       TagRule.new('a', 'cc'), TagRule.new('b', 'd'),
       TagRule.new('c', 'eo'), TagRule.new('d', ''),
       TagRule.new('e', 'e')
    ])
=> #<struct TagRulebook …>
```

Если на вход подано четное число, то система останавливается на односимвольной цепочке `e` (от слова «even» – четный):

```
>> system = TagSystem.new('aabbbbbbbb', rulebook)
=> #<struct TagSystem …>
>> system.run
aabbbbbbbb (число 4)
bbbbbbbcc
bbbbbbccd
bbbbccdd
bbccddd
ccdddd ❶
ddddeo ❷
ddeo
eo ❸
e ❹
=> nil
```

❶ Правила для `a` и `b` делят число пополам; `ccdddd` представляет число 2.

❷ Правило для `c` удаляет начальную пару `cc` и дописывает символы `eo`, один из которых станет конечным результатом.

❸ Пустое правило для `d` удаляет из начала цепочки все пары `dd`, оставляя только `eo`.

❹ Правило для `e` заменяет `eo` на `e`, после чего система останавливается.

Если на вход было подано нечетное число, то результатом будет цепочка `o` (от слова «odd» – нечетный):

```
>> system = TagSystem.new('aabbbbbbbbbb', rulebook)
=> #<struct TagSystem …>
>> system.run
aabbbbbbbbbb (число 5)
bbbbbbbbbcc
bbbbbbbccd
bbbbbbccdd
bbbbccddd
bbccddddd
ccdddddd ❶
ddddddeo
dddeo
```

```
deo ❷
o ❸
=> nil
```

❶ Число, как и раньше, делится пополам, но поскольку теперь оно нечетно, в результате получается цепочка с нечетным числом символов `d`. В нашей схеме кодирования чисел применяются только пары символов, поэтому цепочка `ccdddddd`, строго говоря, не представляет ничего, но коль скоро она содержит «две с половиной» пары, разумно считать ее неформальным представлением числа 2,5.

❷ Все начальные пары `dd` удаляются, оставляя единственный символ `d` перед `eo`.

❸ Оставшийся символ `d` удаляется, забирая с собой `e`, в итоге остается только `o`, и система останавливается.

Эта таг-система работает только потому, что количество удалений больше 1. Поскольку каждому *второму* символу соответствует правило, то на поведение системы можно повлиять, выбирая, какие символы появляются (или не появляются) в таких позициях. Эта техника синхронизации (или рассинхронизации) символов с поведением удаления – ключ к построению мощных таг-систем.

Рассмотренные приемы манипулирования числами можно использовать для моделирования машины Тьюринга. Для построения машины Тьюринга поверх чего-то столь простого, как таг-система, требуется рассмотреть множество деталей, но один из способов работает следующим образом (очень грубо):

1. В качестве простейшего примера возьмем машину Тьюринга, лента которой содержит всего два символа – назовем их `0` и `1`, причем `0` будет играть роль пустого символа.
2. Разобьем ленту машины Тьюринга на две части: левая состоит из символа под головкой и всех символов слева от него, а правая – из всех символов справа от головки.
3. Будем рассматривать левую часть ленты как двоичное число: если исходная лента содержит цепочку `0001101(0)0011000`, то левая часть – это двоичное число 11010, то есть 26 в десятичной записи.
4. Будем рассматривать правую часть ленты как двоичное число, *записанное задом наперед*: в нашем примере правая часть ленты – число `1100`, или 12 в десятичной записи.

5. Закодируем оба числа в виде цепочек, воспринимаемых таг-системой. В нашем примере можно было бы написать `aa`, затем 26 раз повторить пару `bb`, затем `cc` и 12 раз повторить пару `dd`.
6. Используем простые операции над числами – удвоение, деление пополам, инкремент, декремент и проверку на четность-нечетность – чтобы смоделировать чтение ленты, запись на ленту и перемещение головки ленты. Например, для перемещения головки ленты вправо можно умножить число, представленное левой частью и разделить пополам число, представленное правой частью[1]: удвоение 26 дает 52, то есть 110100 в двоичной записи; половина от 12 равна 6, то есть 110 в двоичной записи; таким образом, новая лента содержит цепочку `011010(0)011000`. Чтение ленты сводится к проверке четности или нечетности числа, представленного левой частью, а запись `1` или `0` на ленту – к инкременту или декременту этого числа.
7. Представим текущее состояние смоделированной машины Тьюринга путем выбора символов, используемых для кодирования чисел в левой и правой части ленты: например, если машина находится в состоянии 1, то будем кодировать ленту символами `a`, `b`, `c` и `d`, когда она переходит в состояние 2 – будем использовать символы `e`, `f`, `g` и `h` и так далее.
8. Преобразуем каждое правило машины Тьюринга в таг-систему, которая подходящим образом переписывает текущую цепочку. Правилу, которое читает `0`, записывает `1`, перемещает головку вправо и переходит в состояние 2, можно сопоставить таг-систему, которая проверяет, что число в левой части ленты четно, инкрементирует его, удваивает число в левой части, делит пополам число в правой части и порождает окончательную цепочку, закодированную символами, выбранными для состояния 2.
9. Объединим построенные таг-системы в одну более крупную, которая моделирует все правила машины Тьюринга.

Полное объяснение принципа моделирования машины Тьюринга с помощью таг-системы см. в элегантной конструкции Мэттью Кука (Matthew Cook) в разделе 2.1 статьи *http://www.complex-systems.com/pdf/15-1-1.pdf*.

---

[1] При удвоении числа все цифры в его двоичном представлении сдвигаются на один разряд влево, а при делении пополам — на один разряд вправо.

Модель Кука сложнее описанной выше. В ней изобретательно используется «выравнивание» текущей цепочки для представления символа под головкой моделируемой ленты вместо того, чтобы включать его в одну из частей ленты, и идея легко обобщается на моделирование машины Тьюринга с произвольным числом символов путем увеличения количества удалений системы.

Раз таг-системы позволяют смоделировать произвольную машину Тьюринга, значит, они тоже универсальны.

Циклические таг-системы

Циклической называет таг-система, еще упрощенная за счет наложения следующих дополнительных ограничений:

- циклическая таг-система может содержать только два символа, `0` и `1`;
- правило циклической таг-системы применимо, только если цепочка начинается символом 1[1];
- количество удалений в циклической таг-системе всегда равно 1.

Сами по себе эти ограничения слишком сильны, чтобы обеспечить хоть какое-то полезное вычисление, поэтому у циклических таг-систем есть дополнительная компенсирующая особенность: первое правило в своде правил циклической таг-системы является *текущим правилом*, с которого начинается вычисление, а после каждого шага вычисления текущим становится следующее правило из свода, причем по достижении конца свода правил происходит возврат к первому правилу.

Система такого вида называется циклической из-за цикличности назначения текущего правила. Использование понятия текущего правила в сочетании с ограничением, гласящим, что любое правило применяется только к строкам, начинающимся символом `1`, позволяет избежать издержек на просмотр всего свода правил в поисках правила, применимого на каждом шаге выполнения; если первый символ равен `1`, то применяется текущее правило, иначе – никакое.

---

[1] В правиле циклической таг-системы не нужно говорить «если цепочка начинается символом `1`, дописать в конец символы `011`», потому что первая часть подразумевается — достаточно формулировки «дописать в конец символы `011`».

В качестве примера рассмотрим циклическую таг-систему с тремя правилами, которые дописывают в конце символы `1`, `0010` и `10` соответственно. Вот что происходит, если начать выполнение с цепочки `11`:

Текущая цепочка	Текущее правило	Правило применимо?
11	дописать символ 1	да
11	дописать символы 0010	да
10010	дописать символы 10	да
001010	дописать символ 1	нет
01010	дописать символы 0010	нет
1010	дописать символы 10	да
01010	дописать символ 1	нет
1010	дописать символы 0010	да
0100010	дописать символы 10	нет
100010	дописать символ 1	да
000101	дописать символы 0010	нет
00101	дописать символы 10	нет
0101	дописать символ 1	нет
101	дописать символы 0010	да
010010	дописать символы 10	нет
10010	дописать символы 1	да
00101	дописать символы 0010	нет
⋮	⋮	⋮

Несмотря на исключительную простоту этой системы, видны признаки сложного поведения: неочевидно, что произойдет дальше. Немного подумав, мы можем убедиться в том, что система будет работать бесконечно, не достигая пустой строки, потому что каждое правило дописывает одну `1`, то есть коль скоро начальная цепочка содержит `1`, она никогда не выродится в пустую цепочку[1]. Но будет ли текущая цепочка, пульсируя, становиться все длиннее, или установится стационарный режим расширения и сжатия? Простой взгляд на правила не даст ответа на этот вопрос; нужно дать системе поработать подольше и посмотреть, что происходит.

---

[1] В отличие от обычной, циклическая таг-система продолжает работать, когда нет применимых правил, иначе она вообще не сдвинулась бы с места. Циклическая таг-система останавливается только в одном случае: когда текущая цепочка оказывается пустой, а это происходит, например, когда входная цепочка содержит только символы `0`.

У нас уже имеется реализация обычной таг-системы на Ruby, так что для моделирования циклической много дополнительной работы не потребуется. Класс `CyclicTagRule` мы реализуем в виде подкласса `TagRule`, фиксировав в качестве первого символа `'1'`:

```
class CyclicTagRule < TagRule
  FIRST_CHARACTER = '1'

  def initialize(append_characters)
    super(FIRST_CHARACTER, append_characters)
  end

  def inspect
    "#<CyclicTagRule #{append_characters.inspect}>"
  end
end
```

`#initialize` – это метод-*конструктор*, который автоматически вызывается при создании экземпляра класса. Метод `CyclicTagRule#initialize` вызывает конструктор своего суперкласса, `TagRule`, чтобы установить атрибуты `first_character` и `append_characters`.

Свод правил для циклической таг-системы работает немного по-другому, поэтому класс `CyclicTagRulebook` мы напишем с нуля, предоставив реализации методов `#applies_to?` и `#next_string`:

```
class CyclicTagRulebook < Struct.new(:rules)
  DELETION_NUMBER = 1

  def initialize(rules)
    super(rules.cycle)
  end

  def applies_to?(string)
    string.length >= DELETION_NUMBER
  end

  def next_string(string)
    follow_next_rule(string).slice(DELETION_NUMBER..-1)
  end

  def follow_next_rule(string)
    rule = rules.next
    if rule.applies_to?(string)
      rule.follow(string)
    else
      string
    end
  end
end
```

В отличие от TagRulebook, класс CyclicTagRulebook всегда применяется к непустой цепочке, даже если текущее правило к ней неприменимо.

Метод Array#cycle создает объект Enumerator (См. раздел «Встроенные в Ruby потоки» на стр. 241), который бесконечно перебирает элементы массива по кругу:

```
>> numbers = [1, 2, 3].cycle
=> #<Enumerator: [1, 2, 3]:cycle>
>> numbers.next
=> 1
>> numbers.next
=> 2
>> numbers.next
=> 3
>> numbers.next
=> 1
>> [:a, :b, :c, :d].cycle.take(10)
=> [:a, :b, :c, :d, :a, :b, :c, :d, :a, :b]
```

Именно такое поведение нам требуется для текущего правила циклической таг-системы, поэтому в методе CyclicTagRulebook#initialize объект Enumerator присваивается атрибуту rules, а при каждом вызове метода #follow_next_rule происходит обращение к методу rules.next для получения следующего правила.

Теперь мы можем создать объект CyclicTagRulebook, заполнить его правилами CyclicTagRule и, подключив к TagSystem, посмотреть, как он будет работать:

```
>> rulebook = CyclicTagRulebook.new([
       CyclicTagRule.new('1'), CyclicTagRule.new('0010'),
       CyclicTagRule.new('10')
    ])
=> #<struct CyclicTagRulebook …>
>> system = TagSystem.new('11', rulebook)
=> #<struct TagSystem …>
>> 16.times do
       puts system.current_string
       system.step
   end; puts system.current_string
11
11
10010
001010
01010
1010
01010
```

```
1010
0100010
100010
000101
00101
0101
101
010010
10010
00101
=> nil
```

Это мы уже видели при ручном выполнении. Продолжим:

```
>> 20.times do
       puts system.current_string
       system.step
   end; puts system.current_string
00101
0101
101
011
11
110
101
010010
10010
00101
0101
101
011
11
110
101
010010
10010
00101
0101
101
=> nil
```

Как выясняется, действительно устанавливается стационарный режим, когда система начинает работать с цепочки `11`: после начального периода нестабильности начинает повторяться последовательность из девяти цепочек: (`101`, `010010`, `10010`, `00101`, ...). Конечно, если мы изменяем начальную цепочку или какое-нибудь правило, поведение в долгосрочном плане будет другим.

Циклические таг-системы крайне ограничены – у них негибкие правила, всего два символа и минимально возможное количество удалений – но, как это ни удивительно, с их помощью тем не менее можно смоделировать *любую* таг-систему.

Моделирование обычной таг-системы с помощью циклической производится следующим образом.

1. Определяем алфавит таг-системы – множество используемых в ней символов.
2. Строим схему кодирования, которая ассоциирует с каждым символом уникальную цепочку, пригодную для использования в циклической таг-системе (то есть состоящую только из нулей и единиц).
3. Преобразуем каждое правило исходной системы в правило циклической таг-системы путем кодирования дописываемых в конец символов.
4. Дополняем свод правил циклической таг-системы пустыми правилами для моделирования количества удаления исходной таг-системы.
5. Кодируем начальную цепочку исходной таг-системы и подаем результат на вход циклической таг-системы.

Конкретизируем эти идеи, воплотив их в код. Сначала нужно научиться спрашивать у таг-системы, какие символы в ней используются:

```
class TagRule
  def alphabet
    ([first_character] + append_characters.chars.entries).uniq
  end
end

class TagRulebook
  def alphabet
    rules.flat_map(&:alphabet).uniq
  end
end

class TagSystem
  def alphabet
    (rulebook.alphabet + current_string.chars.entries).uniq.sort
  end
end
```

Мы можем протестировать эти классы на таг-системе для инкрементирования чисел, описанной в разделе «Таг-системы» на стр. 280. Метод `TagSystem#alphabet` сообщает, что в этой системе используются символы a, b, c и d:

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'ccdd'), TagRule.new('b', 'dd')])
=> #<struct TagRulebook …>
>> system = TagSystem.new('aabbbb', rulebook)
```

```
=> #<struct TagSystem …>
>> system.alphabet
=> ["a", "b", "c", "d"]
```

Далее нам необходим способ кодирования каждого символа в виде строки, пригодной для циклической таг-системы. Это конкретная схема кодирования, при которой модель будет работать: каждый символ представляется цепочкой, длина которой совпадает с длиной алфавита, с нулями во всех позициях, кроме одной, номер которой равен позиции данного символа в алфавите; в этой позиции находится 1[1].

У нашей таг-системы алфавит содержит четыре символа, поэтому буквы кодируются четырехсимвольными цепочками с единицей в разных позициях:

Символ таг-системы	Позиция в алфавите	Кодированное представление
a	0	1000
b	1	0100
c	2	0010
d	3	0001

Для реализации этой схемы кодирования напишем класс `CyclicTagEncoder`; передав конструктору его экземпляра алфавит, мы затем можем попросить его закодировать символы этого алфавита.

```
class CyclicTagEncoder < Struct.new(:alphabet)
  def encode_string(string)
    string.chars.map { |character| encode_character(character) }.join
  end

  def encode_character(character)
    character_position = alphabet.index(character)
    (0...alphabet.length).map { |n| n == character_position ? '1' : '0' }.join
  end
end

class TagSystem
  def encoder
    CyclicTagEncoder.new(alphabet)
  end
end
```

[1] Получающаяся последовательность нулей и единиц *не* представляет двоичное число. Это просто строка нулей, в которой единственная единица отмечает конкретную позицию.

Теперь класс `CyclicTagEncoder` можно использовать для кодирования любых цепочек, состоящих из символов `a`, `b`, `c`, `d`:

```
>> encoder = system.encoder
=> #<struct CyclicTagEncoder alphabet=["a", "b", "c", "d"]>
>> encoder.encode_character('c')
=> "0010"
>> encoder.encode_string('cab')
=> "001010000100"
```

Кодировщик позволяет преобразовать любое правило таг-системы в правило циклической таг-системы. Нужно лишь закодировать цепочку `append_characters` из объекта `TagRule` и использовать получившуюся цепочку для построения объекта `CyclicTagRule`:

```
class TagRule
  def to_cyclic(encoder)
    CyclicTagRule.new(encoder.encode_string(append_characters))
  end
end
```

Испытаем этот метод на одном правиле `TagRule`:

```
>> rule = system.rulebook.rules.first
=> #<struct TagRule first_character="a", append_characters="ccdd">
>> rule.to_cyclic(encoder)
=> #<CyclicTagRule "0010001000010001">
```

Отлично, цепочку `append_characters` мы преобразовали, но при этом потеряли информацию о том, какой первый символ `first_character` должен инициировать правило – каждое правило `CyclicTagRule` инициируется символом `1` независимо от того, из какого правила `TagRule` оно получено.

Утраченная же информация передается с помощью *порядка* следования правил в циклической таг-системе: первое правило соответствует первому символу алфавита, второе – второму и т. д. Символам, для которых не было правил в исходной таг-системе, соответствует пустое правило в своде правил циклической таг-системы.

Мы можем реализовать метод `TagRulebook#cyclic_rules`, который будет возвращать преобразованные правила в нужном порядке:

```
class TagRulebook
  def cyclic_rules(encoder)
    encoder.alphabet.map { |character| cyclic_rule_for(character, encoder) }
  end
```

```
  def cyclic_rule_for(character, encoder)
    rule = rule_for(character)
    if rule.nil?
      CyclicTagRule.new('')
    else
      rule.to_cyclic(encoder)
    end
  end
end
```

Вот что метод `#cyclic_rules` дает для нашей таг-системы:

```
>> system.rulebook.cyclic_rules(encoder)
=> [
       #<CyclicTagRule "0010001000010001">,
       #<CyclicTagRule "00010001">,
       #<CyclicTagRule "">,
       #<CyclicTagRule "">
]
```

Как и следовало ожидать, сначала идут преобразованные правила для `a` и `b`, а затем два пустых правила в позициях, соответствующих `c` и `d`.

Такое решение согласуется со схемой кодирования символов, благодаря которой модель и работает. Например, если начальная цепочка моделируемой таг-системы состоит из одного символа `b`, то на входе циклической таг-системы она будет выглядеть как `0100`. Посмотрим, что происходит при запуске системы с таким входом.

Текущая цепочка	Текущее правило	Правило применимо?
0100	дописать символы 0010001000010001 (правило для a)	нет
100	дописать символы 00010001 (правило для b)	да
00**000010001**	не дописывать ничего (правило для c)	нет
000010001	не дописывать ничего (правило для d)	нет
⋮	⋮	⋮

На первом шаге вычисления текущим является преобразованное правило для `a`, и оно не используется, потому что текущая цепочка начинается с `0`. На втором шаге текущим становится правило для `b`, а начальный `0` из текущей цепочки удаляется, оставляя в начале `1`, что инициирует выполнение правила. Следующие два символа равны `0`, поэтому правила для `c` и `d` также не используются.

Итак, согласовав позиции символа `1` во входной цепочке с цикличностью появления правил в циклической таг-системе, мы можем инициировать нужное правило в нужный момент и тем самым точно смоделировать поведение правил обычной таг-системы в части сопоставления с символами.

Наконец, мы должны смоделировать количество удалений в исходной таг-системе, но это легко сделать, вставив дополнительные пустые правила в свод правил циклической таг-системы, так чтобы после успешной обработки одного закодированного символа удалялось требуемое число символов. Если в алфавите исходной таг-системы было `n` символов, то каждый символ входной цепочки для нее представляется `n` символами в цепочке для циклической таг-системы, поэтому нам нужно `n` пустых правил для каждого дополнительного моделируемого символа, подлежащего удалению:

```
class TagRulebook
  def cyclic_padding_rules(encoder)
    Array.new(encoder.alphabet.length, CyclicTagRule.new('')) * (deletion_number - 1)
  end
end
```

В нашей таг-системе алфавит состоит из четырех символов, а количество удалений равно 2, поэтому нам нужны четыре лишних пустых правила, чтобы удалить еще один моделируемый символ в дополнение к тому, что уже удаляется преобразованными правилами:

```
>> system.rulebook.cyclic_padding_rules(encoder)
=> [
       #<CyclicTagRule "">,
       #<CyclicTagRule "">,
       #<CyclicTagRule "">,
       #<CyclicTagRule "">
     ]
```

Теперь можно собрать все вместе, чтобы полностью реализовать метод `#to_cyclic` в классе `TagRulebook`, а затем использовать его в методе `TagSystem#to_cyclic`, который преобразует свод правил и текущую цепочку, порождая законченную циклическую таг-систему:

```
class TagRulebook
  def to_cyclic(encoder)
    CyclicTagRulebook.new(cyclic_rules(encoder) + cyclic_padding_rules(encoder))
  end
```

```
end

class TagSystem
  def to_cyclic
    TagSystem.new(encoder.encode_string(current_string), rulebook.to_cyclic(encoder))
  end
end
```

Вот что произойдет, после того как мы преобразуем таг-систему для инкрементирования чисел и запустим ее:

```
>> cyclic_system = system.to_cyclic
=> #<struct TagSystem …>
>> cyclic_system.run
100010000100010001000100 (aabbbb) ❶
000100001000100010001000010001000010001
00100001000100010001000010001000010001
0100001000100010001000010001000010001
100001000100010001000010001000010001 (abbbbccdd) ❷
00001000100010001000010001000010001
0001000100010001000010001000010001
001000100010001000010001000010001
01000100010001000010001000010001 (bbbbccdd) ❸
1000100010001000010001000010001 ❹
00010001000100001000100001000100010001
0010001000100001000100001000100010001
010001000100001000100001000100010001 (bbbccdddd)
10001000100001000100001000100010001
0001000100001000100001000100010001
001000100001000100001000100010001
01000100001000100001000100010001 (bbccdddd)
1000100001000100001000100010001 ❺
00010000100010000100010001000100010001
0010000100010000100010001000100010001
010000100010000100010001000100010001 (bccdddddd)
10000100010000100010001000100010001
0000100010000100010001000100010001
000100010000100010001000100010001
00100010000100010001000100010001 (ccdddddd) ❻
0100010000100010001000100010001
100010000100010001000100010001
00010000100010001000100010001 ❼
⋮
001
01
1
  ❽
=> nil
```

❶ Здесь вступает в дело кодированная версия правила таг-системы.

❷ Первый полный символ смоделированной цепочки обработан, поэтому на последующих четырех шагах используются четыре

пустых правила для удаления следующего смоделированного символа.

❸ После восьми шагов циклической таг-системы завершен один полный шаг моделируемой таг-системы.

❹ Здесь инициируется закодированное правило для b...

❺ ...и здесь еще раз.

❻ Через 24 шага вычисления циклической таг-системы мы достигли представления конечной цепочки моделируемой таг-системы – ccdddddd.

❼ В моделируемой таг-системе нет правил для цепочек, начинающихся символом c или d, поэтому текущая цепочка циклической таг-системы становится все короче и короче...

❽ ...пока наконец не станет пустой, и в этот момент система останавливается.

Эту технику можно использовать для моделирования любой таг-системы, в том числе той, что сама моделирует машину Тьюринга, а это означает, что циклические таг-системы тоже универсальны.

Игра «Жизнь» Конвея

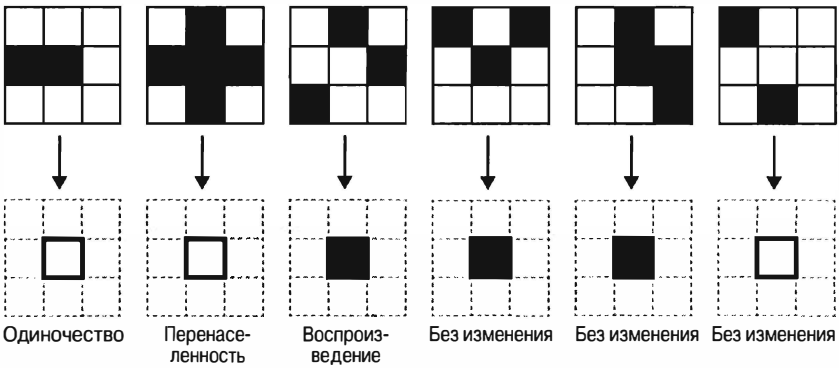
В 1970 году Джон Конвей изобрел универсальную систему, которую назвал *игрой «Жизнь»*. Игра разыгрывается на бесконечной двумерной квадратной сетке, в которой каждая клетка может быть *живой* или *мертвой*. Любая клетка окружена восьмью *соседками*: три клетки сверху, по одной слева и справа и три снизу.

Игра «Жизнь» состоит из последовательности шагов, как в конечном автомате. На каждом шаге любая клетка может изменить состояние с живой на мертвую или наоборот согласно правилам, в которых фигурируют текущие состояния самой клетки и ее соседок. Правила просты: живая клетка умирает, если у нее меньше двух живых соседок (от одиночества) или больше трех (от перенаселенности), а в мертвой клетке зарождается жизнь, если у нее ровно три живых соседки (воспроизведение).

Ниже показано шесть примеров[1] того, как может меняться состояние клетки в результате выполнения одного шага (живые клетки изображены черным цветом, мертвые – белым):

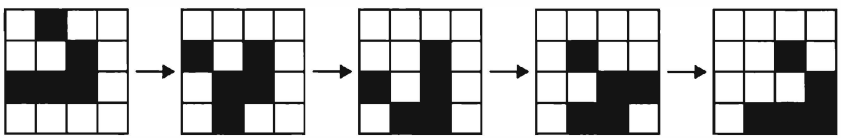
---

[1] Из 512 возможных: участвуют девять клеток, каждая может находиться в двух состояниях, всего, следовательно, существует $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 512$ возможностей.



Подобные системы, состоящие из массива клеток и набора правил для обновления состояния каждой клетки на каждом шаге, называются *клеточными автоматами*.

Как и прочие рассмотренные в этой главе системы, игра «Жизнь» демонстрирует удивительную сложность, несмотря на простоту правил. Интересное поведение возникает при определенных конфигурациях живых клеток, самой известной является *планер* – расстановка пяти живых клеток, которая смещается вдоль сетки по диагонали на один квадрат за каждые четыре шага:



Открыто немало других интересных конфигураций (*http://conway.life.com*), в том числе фигуры, которые разными способами перемещаются по сетке (*космические корабли*), генерируют потоки других фигур (*ружья*) и даже порождают полные копии самих себя (*репликаторы*).

В 1982 году Конвей показал, как с помощью потока планеров представить потоки двоичных данных, а также как построить логические вентили И, ИЛИ и НЕ для выполнения цифровых вычислений путем хитроумного сталкивания планеров. Этим было доказано, что имеется теоретическая возможность смоделировать цифровой компьютер с помощью игры «Жизнь», но Конвей не стал проектировать действующую машину:

> Начиная с этого момента, конструирование сколь угодно большого конечного (и очень медленного!) компьютера становится чисто инженерной задачей. Инженеру даны инструменты – так не мешайте ему закончить работу! [...] Тот вид компьютера, который мы смоделировали, технически называется *универсальной машиной*, потому что его можно запрограммировать для выполнения любого вычисления.
>
> *–Джон Конвей*
> *«Как выигрывать в математических играх»*

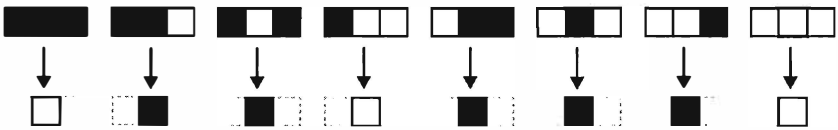
В 2002 году Пол Чепмэн (Paul Chapman) построил конкретный универсальный компьютер на принципах «Жизни», а в 2010 году Пол Ренделл (Paul Rendell) сконструировал универсальную машину Тьюринга.

Вот крупный план одного маленького фрагмента конструкции Ренделла.

Правило 110

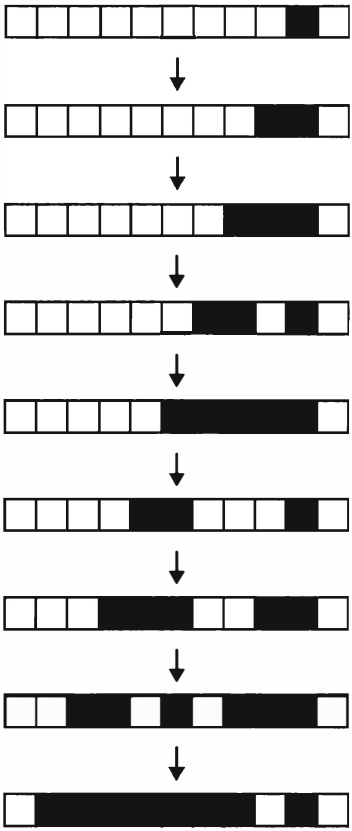
Правило 110 – еще один клеточный автомат, изобретенный Стивеном Вольфрамом в 1983 году. Каждая клетка также может быть живой или мертвой, как и в игре «Жизнь» Конвея, но правило 110 воздействует на клетки, расположенные в одномерной строке, а не на двумерной сетке. Это означает, что у клетки всего две соседки – слева и справа – а не восемь, как в игре «Жизнь».

На каждом шаге автомата «Правило 110» следующее состояние клетки определяется ее собственным состоянием и состоянием обеих соседок. В отличие от игры «Жизнь», где правила носят общий характер и применимы к различным конфигурациям живых и мертвых клеток, в «Правиле 110» имеет отдельное правило для каждой возможной конфигурации:

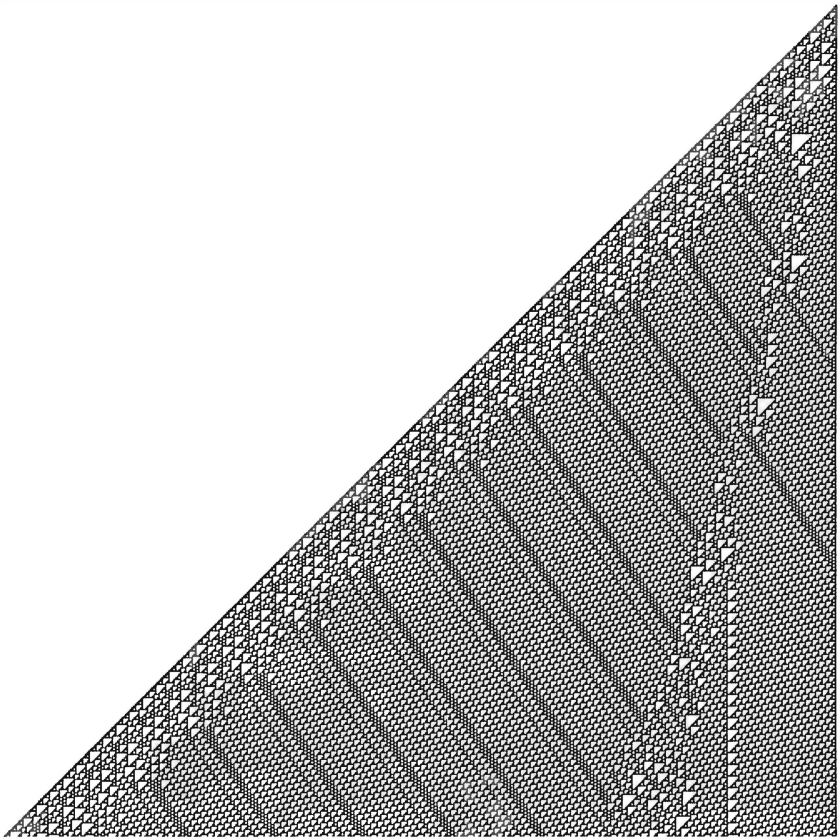


Если выписать подряд значения клеток «после» во всех восьми правилах, интерпретируя мертвую клетку как 0, а живую – как 1, то получится двоичное число 01101110, которое в десятичной записи равно 110, что и дало название этому клеточному автомату.

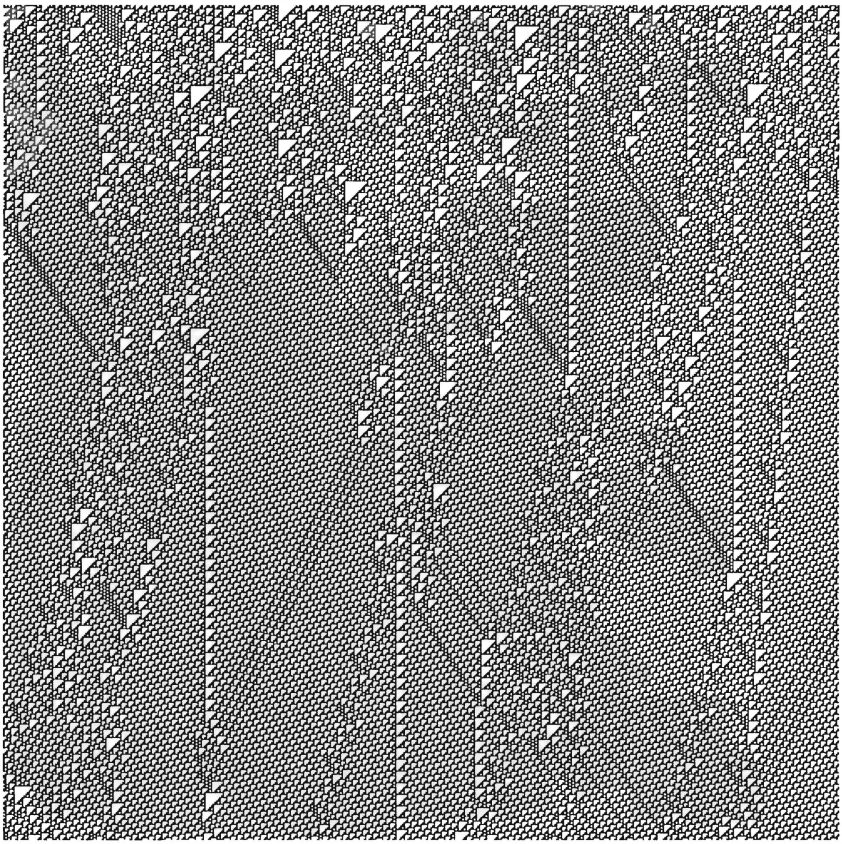
Правило 110 гораздо проще игры «Жизнь», но и этот автомат демонстрирует весьма сложное поведение. Вот несколько первых шагов автомата, начавшего работу с одной живой клетки:



Уже это поведение далеко от очевидного – так, оно не порождает сплошной ряд живых клеток – а если проследить поведение этого автомата на протяжении 500 шагов, то начнут проявляться интересные закономерности:



Если же запустить правило 110 в начальном состоянии, содержащем случайную комбинацию живых и мертвых клеток, то можно наблюдать всевозможные фигуры, перемещающиеся в разных направлениях и взаимодействующие между собой:



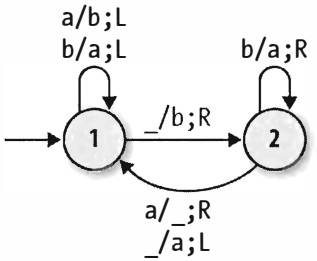
Сложность, проистекающая из этих восьми простых правил, оказывается поразительно мощной: в 2004 году Мэтью Кук (Matthew Cook) опубликовал доказательство того, что правило 110 фактически универсально. Доказательство изобилует деталями (см. разделы 3 и 4 статьи по адресу *http://www.complex-systems.com/pdf/15-1-1.pdf*), но, грубо говоря, Кук рассматривает несколько конфигураций правила 110, которые можно уподобить планерам, а затем показывает, как можно смоделировать любую циклическую таг-систему, организовав эти планеры специальным образом.

Это означает, что правило 110 может смоделировать циклическую таг-систему, которая моделирует обычную таг-систему, которая моделирует универсальную машину Тьюринга, – не самый эффектив-

ный способ реализовать универсальное вычисление, но тем не менее впечатляющий технический результат для столь простого клеточного автомата.

Вольфрамова 2,3 машина Тьюринга

Чтобы завершить наш краткий экскурс по простым универсальным системам, расскажем еще об одной, которая даже проще правила 110: Вольфрамовой 2,3 машине Тьюринга. Свое название она получила от двух состояний и трех символов (a, b и пустышка), которые в сочетании дают шесть правил:



Эта машина Тьюринга необычна тем, что не имеет заключительного состояния, то есть никогда не останавливается, но это в основном техническая деталь. Мы можем получить результат даже от не прекращающей работу машины, наблюдая за некоторым поведением – например, появлением на ленте определенной комбинации символов – и считая это указанием на то, что текущая лента содержит полезную выходную информацию.

На первый взгляд, непохоже, чтобы Вольфрамова 2,3 машина Тьюринга обладала мощностью, достаточной для универсальных вычислений, но в 2007 году компания Wolfram Research объявила награду в 25 000 долларов тому, кто докажет ее универсальность, а годом позже Алекс Смит (Alex Smith) заявил права на награду (*http://www.theguardian.com/technology/2007/nov/29/research*), предъявив корректное доказательство. Как и в случае правила 110, идея состоит в том, чтобы показать, что эта машина может смоделировать любую циклическую таг-систему; доказательство также содержит множество деталей, но желающие могут ознакомиться с ним в статье по адресу *http://www.wolframscience.com/prizes/tm23/*.

Глава 8. Невозможные программы

> Мне думается, что высшее милосердие, явленное нашему миру, заключается в неспособности человеческого ума согласовывать между собой свои собственные составляющие.
>
> – *Г. Ф. Лавкрафт*

В этой книге мы исследовали различные модели компьютеров и языков программирования, в том числе несколько разновидностей абстрактных машин. Одни из них мощнее, другие слабее, а у двух есть очевидные ограничения: конечные автоматы не способны решать задачи, в которых требуется подсчитывать потенциально неограниченное количество объектов, например, установить факт сбалансированности скобок, а автоматы с магазинной памятью не могут справиться с задачами, где информацию требуется повторно использовать в нескольких местах, например, решить, содержит ли цепочка одинаковое количество символов `a`, `b` и `c`.

Но самое развитое из рассмотренных нами устройств, машина Тьюринга, похоже, имеет все, что нужно: неограниченную память, к которой можно обращаться в любом порядке, произвольные циклы, условные выражения и подпрограммы. Предельно минималистский язык программирования, с которым мы познакомились в главе 6, – лямбда-исчисление – также оказался неожиданно мощным: проявив смекалку, мы смогли представить в виде чистого кода простые значения и сложные структуры данных, а также реализовать операции для манипулирования этими представлениями. А в главе 7 мы видели много других простых систем, которые, подобно лямбда-исчислению, обладают такой же вычислительной мощностью, как машины Тьюринга.

Как далеко можно продолжить этот ряд все более мощных систем? Пожалуй, не до бесконечности: наши попытки сделать машину Тьюринга более мощной путем добавления дополнительных возможностей никуда не привели, и это наводит на мысль, что у вычислительной мощности есть предел. Так на что же принципиально способны компьютеры и языки программирования и есть ли что-то такое, чего они сделать не могут? Существуют ли невозможные программы?

Факты как они есть

Это весьма глубокие вопросы, поэтому прежде чем подступаться к ним, сделаем обзор фундаментальных фактов из мира вычислений. Одни из них очевидны, другие – не очень, но все это важные предпосылки для размышлений о возможностях и ограничениях вычислительных машин.

Универсальные системы могут выполнять алгоритмы

Чего, вообще говоря, мы можем ожидать от таких универсальных систем, как машины Тьюринга, лямбда-исчисление и частично рекурсивные функции? Если мы сможем правильно понять их возможности, то сможем и исследовать их ограничения.

Практическая цель вычислительной машины состоит в том, чтобы выполнять *алгоритмы*. Алгоритм – это перечень инструкций, описывающих некий процесс преобразования входного значения в выходное, которые должны удовлетворять определенным критериям.

- *Конечность* – число инструкций конечно.
- *Простота* – каждая отдельная инструкция настолько проста, что может быть выполнена человеком, имеющим только ручку и бумагу, и не требует никакой изобретательности.
- *Завершение* – для любых входных данных человек, выполняющий инструкции, должен завершить работу за конечное число шагов.
- *Правильность* – для любых входных данных человек, выполняющий инструкции, должен получить правильный ответ.

Например, один из старейших известных алгоритмов – алгоритм Евклида – датируется примерно 300 годом до нашей эры. Получая на входе два положительных целых числа, он возвращает наиболь-

шее число, на которое они оба делятся без остатка, – *наибольший общий делитель*. Вот как выглядят соответствующие инструкции.

1. Назвать два числа x и y.
2. Определить, какое из чисел x и y больше.
3. Вычесть меньшее число из большего (если x больше, вычесть y из x и сделать результат новым значением x; если y больше, поступить наоборот).
4. Повторять шаги 2 и 3, пока x и y не станут равны.
5. Получившееся в этот момент значение x (и равное ему значение y) и является наибольшим общим делителем исходных чисел.

Этот набор инструкций действительно можно признать алгоритмом, поскольку он, похоже, отвечает сформулированным критериям. Он содержит всего несколько инструкций, и все они достаточно просты, чтобы любой человек, не знакомый с существом задачи, мог выполнить их с помощью бумаги и ручки. Немного подумав, мы приходим к выводу, что алгоритм должен заканчиваться за конечное число шагов при любых входных данных: при каждом повторении шага 3 одно из двух чисел становится меньше, поэтому в конечном итоге они должны сравняться[1], и в этот момент работа алгоритма закончится. Не вполне очевидно, дает ли он правильный ответ, но достаточно нескольких тождеств из элементарной алгебры, чтобы убедиться, что в результате обязательно получается наименьший общий делитель исходных чисел.

Таким образом, алгоритм Евклида заслуживает такого названия, но, как и любой другой алгоритм, это не более чем совокупность идей, изложенных в виде понятных человеку слов и символов. Если мы хотим сделать с ним что-то полезное – например, исследовать его математические свойства или построить машину, которая будет выполнять его автоматически, – то должны представить алгоритм в более строгой, не допускающей двояких толкований форме, которая была бы пригодна для математического анализа или механического выполнения.

У нас уже есть несколько моделей вычислений, которые можно было бы использовать для этой цели: мы могли бы попытаться записать алгоритм Евклида в виде свода правил машины Тьюринга или выражения лямбда-исчисления или определения частично ре-

---

[1] Ни x, ни y не могут стать меньше 1, поэтому в худшем случае они станут равны, когда оба обратятся в 1.

курсивной функции, но все эти способы требуют многочисленных организационных действий и других неинтересных деталей. Поэтому пока просто запишем его в виде программы на языке Ruby, не налагая на него никаких ограничений[1]:

```
def euclid(x, y)
  until x == y
    if x > y
      x = x - y
    else
      y = y - x
    end
  end

  x
end
```

Метод `#euclid` содержит по существу те же инструкции, что в описании алгоритма Евклида на естественном языке, только записаны они в виде, имеющем строго определенный смысл (определяемый операционной семантикой Ruby), и потому могут быть интерпретированы машиной:

```
>> euclid(18, 12)
=> 6
>> euclid(867, 5309)
=> 1
```

В данном случае оказалось нетрудно превратить неформальное описание алгоритма на понятном человеку языке в набор недвусмысленных инструкций для машины. Иметь машиночитаемое представление алгоритма Евклида очень удобно; теперь мы можем выполнять его быстро, многократно и надежно, не полагаясь на ручной труд.

Надеемся, не вызывает сомнений, что этот алгоритм, можно было бы с тем же успехом реализовать на лямбда-исчислении, следуя методике, описанной в разделе «Операции над числами» на стр. 206, или в виде частично рекурсивной функции, построенной из операций, представленных в разделе «Частично рекурсивные функции» на стр. 260, или в виде набора правил машины Тьюринга, как для простых арифметических действий в разделе «Правила» на стр. 176.

---

[1] На самом деле, в Ruby уже встроен алгоритм Евклида в виде метода `Integer#gcd`, но это к делу не относится.

В связи с этим возникает важный вопрос: *любой* ли алгоритм можно представить в виде инструкций, пригодных для выполнения машиной? При поверхностном взгляде ответ кажется тривиальным – ведь очевидно же, как превратить в программу алгоритм Евклида, а мы, программисты, склонны считать объекты взаимозаменяемыми, – но между абстрактной, интуитивной идеей алгоритма и конкретной, логической реализацией алгоритма в вычислительной системе существует огромная разница. Может ли существовать алгоритм настолько большой, сложный и необычный, что его суть невозможно представить в виде бездумного механического процесса?

Окончательного и строгого ответа на этот вопрос быть не может, потому что он носит скорее философский, чем научный характер. Инструкции алгоритма должны быть «простыми» и «не требующими изобретательности», так чтобы «их мог выполнить любой человек», но это лишь неточные высказывания об интуиции и способностях человека, а не математические утверждения, которые можно было бы доказать или опровергнуть.

Можно было бы собрать какие-то свидетельства, придумывая различные алгоритмы и изучая, может ли их реализовать та или иная вычислительная система – машины Тьюринга, лямбда-исчисление, частично рекурсивные функции или Ruby. Математики и специалисты по информатике занимаются этим с 1930-х годов, и до сих пор никому не удалось изобрести разумный алгоритм, который невозможно было бы выполнить в таких системах, поэтому можно со значительной долей уверенности утверждать, что наша эмпирическая догадка истинна: *похоже* на то, что машина может выполнить любой алгоритм.

Еще одним убедительным аргументом, подкрепляющим эту догадку, служит тот факт, что большинство вышеупомянутых систем было разработано независимо в попытках формализовать и проанализировать неформальную идею алгоритма, и лишь позже было обнаружено, что все они эквивалентны. Все известные попытки построить модель идеи алгоритма приводили к системам, возможности которых таковы же, как у машины Тьюринга, и это весомый довод в пользу того, что машина Тьюринга адекватно представляет возможности алгоритма.

Идея о том, что любой алгоритм может быть выполнен машиной – точнее, детерминированной машиной Тьюринга – называется *тезисом Чёрча-Тьюринга*, и, хотя это лишь гипотеза, а не доказанный факт, подтверждающих свидетельств достаточно, чтобы можно было принять ее за истину.

Высказывание «машина Тьюринга может выполнить любой алгоритм» – это философское предположение о связи между интуитивной идеей алгоритма и формальными системами, которыми мы пользуемся для их реализации. Что оно означает на самом деле – вопрос интерпретации: можно рассматривать его как утверждение о том, что можно и чего нельзя вычислить, или как более строгое определение слова «алгоритм».

Как бы то ни было, называется оно «тезисом Чёрча-Тьюринга», а не «теоремой Чёрча-Тьюринга», потому что это неформальное предположение, а не доказуемое математическое утверждение, – его нельзя выразить на чисто математическом языке, а потому и невозможно построить математическое доказательство. Многие полагают, что оно верно, потому что согласуется с интуитивным представлением о природе вычислений и фактическими данными о том, что могут делать алгоритмы, но мы все равно называем его «тезисом», напоминая себе, что оно отличается от доказуемых утверждений типа теоремы Пифагора.

Из тезиса Чёрча-Тьюринга вытекает, что машины Тьюринга, несмотря на свою простоту, обладают мощностью, необходимой для выполнения любого вычисления, которое в принципе может быть проделано человеком, который следует простым инструкциям. Многие идут дальше и утверждают, что коль скоро все попытки формализовать идею алгоритма приводили к универсальным системам, которые с точки зрения вычислительной мощности эквивалентны машинам Тьюринга, то ничего лучшего получать вообще невозможно: любой реальный компьютер или язык программирования может сделать только то, на что способна машина Тьюринга, и не больше. Возможно ли когда-то в будущем построить более мощную машину – в которой использовались бы какие-то экзотические физические законы для решения задач, выходящих за рамки того, что мы называем «алгоритмами», – точно не известно, но сейчас мы определенно не знаем, как это сделать.

Программы могут замещать машины Тьюринга

В главе 5 мы видели, что из-за простоты машины Тьюринга свод правил для нее получается громоздким и лишенным наглядности. Чтобы не затемнять наше исследование вычислимости кропотливыми деталями программирования машины Тьюринга, мы будем ис-

пользовать вместо нее программы на Ruby, как поступили в случае алгоритма Евклида.

Эта уловка оправдывается универсальностью: в принципе, любую Ruby-программу можно преобразовать в эквивалентную машину Тьюринга и наоборот, поэтому Ruby-программы не более и не менее мощны, чем машины Тьюринга, и все, что мы выясним об ограничениях Ruby, равным образом применимо и к машинам Тьюринга.

Можно возразить, что в Ruby встроена практическая функциональность, отсутствующая у машин Тьюринга, – и это правда. Ruby-программа может обращаться к файловой системе, отправлять и получать сообщения по сети, принимать данные от пользователя, рисовать графику на растровом дисплее и т. д., тогда как даже самая навороченная машина Тьюринга умеет лишь читать и записывать символы на ленту. Но это проблема не принципиальная, потому что всю эту дополнительную функциональность можно *смоделировать* машиной Тьюринга: если необходимо, мы можем считать некоторые участки ленты «файловой системой», «сетью», «дисплеем» и вообще чем угодно и рассматривать чтение и запись в эти области, как взаимодействие с внешним миром. Ни одно из таких дополнений не изменяет вычислительную мощность машины Тьюринга, все они являются не более чем высокоуровневыми интерпретациями операций с лентой.

На практике мы можем вообще обойти это возражение, ограничившись простыми Ruby-программами, в которых не используются вызывающие сомнения средства языка. Далее в этой главе мы будем писать лишь программы, которые читают строку из стандартного ввода, производят какое-то вычисление, а по завершении записывают строку на стандартный вывод; входная строка будет аналогом начального содержимого ленты машины Тьюринга, а выходная – аналогом ее конечного содержимого.

Код – это данные

Программы живут двойной жизнью. Программу можно рассматривать не только как набор инструкций для управления конкретной системой, но и как чистые данные: дерево выражений, неструктурированную строку символов или даже как одно большое число. Программисты обычно принимают этот дуализм, не задумываясь, но для компьютеров общего назначения весьма важно, что программу можно представить как данные и использовать в качестве входа для

другой программы; именно благодаря такой унификации кода и данных программное обеспечение вообще стало возможным.

Мы уже встречались с «программами-как-данными» при рассмотрении универсальной машины Тьюринга, которая ожидает, что на ленту будет записан в виде последовательности символов свод правил другой машины Тьюринга. В *гомоиконных* языках программирования типа Lisp[1] и XSLT программы явно записываются в виде структур данных, которыми сам язык может манипулировать: любая Lisp-программа представляет собой вложенный список, который называется *s-выражением*, а любая таблица стилей XSLT – это XML-документ.

В Ruby обычно только интерпретатору (который, по крайней мере в случае MRI, написан не на Ruby) доводится видеть структурированное представление программы, но принцип «код-как-данные» все равно применим. Рассмотрим следующую простую Ruby-программу:

```
puts 'hello world'
```

Для наблюдателя, который понимает синтаксис и семантику Ruby, это программа, которая отправляет объекту `main` сообщение `puts` со строковым параметром `'hello world'`, в результате чего метод `Kernel#puts` печатает сообщение `hello world` на стандартный вывод. Но на более низком уровне это просто последовательность символов, а поскольку символы представлены в виде байтов, то эту последовательность можно рассматривать как одно большое число:

```
>> program = "puts 'hello world'"
=> "puts 'hello world'"
>> bytes_in_binary = program.bytes.map { |byte| byte.to_s(2).rjust(8, '0') }
=> ["01110000", "01110101", "01110100", "01110011", "00100000", "00100111",
    "01101000", "01100101", "01101100", "01101100", "01101111", "00100000",
    "01110111", "01101111", "01110010", "01101100", "01100100", "00100111"]
>> number = bytes_in_binary.join.to_i(2)
=> 979654384950070652110298049571774002183479l
```

В каком-то смысле `puts 'hello world'` – это Ruby-программа за номером 979654384950070652110298049571774002183479l[2]. Обрат-

[1] В действительности Lisp — это целое семейство языков программирования, включающее Common Lisp, Scheme и Clojure, с очень похожим синтаксисом.

[2] Было бы полезнее присваивать номера только *синтаксически корректным* Ruby-программам, но это более сложно.

но, если кто-то сообщит нам номер Ruby-программы, то мы легко сможем восстановить по нему саму программу и выполнить ее:

```
>> number = 979654384950070652110298049571774002183479l
=> 979654384950070652110298049571774002183479l
>> bytes_in_binary = number.to_s(2).scan(/.{7}(?=.{6}*\z)/)
=> ["1110000", "01110101", "01110100", "01110011", "00100000", "00100111",
    "01101000", "01100101", "01101100", "01101100", "01101111", "00100000",
    "01110111", "01101111", "01110010", "01101100", "01100100", "00100111"]
>> program = bytes_in_binary.map { |string| string.to_i(2).chr }.join
=> "puts 'hello world'"
>> eval program
hello world
=> nil
```

Разумеется, именно такая схема представления программы в виде большого числа и позволяет хранить ее на диске, посылать через Интернет и подавать на вход интерпретатора Ruby (который и сам является гигантским числом на диске!), чтобы осуществить конкретное вычисление.

Поскольку у каждой Ruby-программы есть уникальный номер, мы можем автоматически сгенерировать все возможные программы: начать с генерации программы под номером 1, затем сгенерировать программу под номером 2 и т. д.[1] Если делать это достаточно долго, то в конце концов мы сгенерируем очередной крутой каркас для асинхронной веб-разработки и сможем почивать на лаврах.

Универсальные системы могут зацикливаться

Мы видели, что компьютеры общего назначения универсальные: мы можем построить машину Тьюринга, способную моделировать любую другую машину Тьюринга, или написать программу, которая может вычислить любую другую программу. Универсальность – действенная идея, которая позволяет использовать для решения разнообразных задач единственную адаптируемую машину, а не множество специализированных устройств, но у нее есть и неприятное след-

[1] Большая часть номеров будет соответствовать синтаксически некорректным Ruby-программам, но мы можем подать каждую потенциальную программу синтаксическому анализатору Ruby и отбросить те, что содержат синтаксические ошибки.

ствие: любая система, достаточно мощная, чтобы быть универсальной, неизбежно позволяет сконструировать вычисления, которые бесконечно выполняют некоторый цикл, никогда не останавливаясь.

Очень длительные вычисления

> – Я лишь хотел сказать, – оглушительно взревел компьютер, – что все мои цепи заняты сейчас поисками ответа на Основной Вопрос Жизни, Вселенной и Всего Остального. – Он замолчал и, удостоверившись, что владеет вниманием безраздельно, тихо продолжил: – Но на такую программу понадобится некоторое время.
>
> Фук нетерпеливо взглянул на часы:
>
> – Сколько же?
>
> – Семь с половиной миллионов лет, – произнес Пронзительный Интеллектомат.
>
> *– Дуглас Адамс*
> *«Автостопом по Галактике»*

Если мы стремимся выполнить алгоритм – список инструкций, задача которых состоит в том, чтобы преобразовать вход в выход, – то бесконечный цикл – это плохо. Мы хотим, чтобы машина (или программа) поработала ограниченное время, а затем остановилась и выдала какой-то результат, а не просто молча стояла и грелась. При прочих равных условиях было бы лучше иметь такие компьютеры и языки, которые гарантированно заканчивают любую задачу за конечное число шагов, чтобы нам не приходилось думать, получим мы в конце концов ответ или нет.

Однако на практике встречаются приложения, в которых наличие бесконечного цикла желательно. Например, от веб-сервера типа Apache или Nginx было бы мало толку, если бы он принял один HTTP-запрос, отправил ответ и завершился; мы хотим, чтобы он работал неопределенно долго, продолжая обслуживать поступающие запросы, пока не будет принудительно остановлен. Однако концептуально однопоточный веб-сервер можно разделить на две части: код обработки одного запроса, который обязательно должен останавливаться, чтобы можно было отправить ответ, и обертывающий его бесконечный цикл, в котором обработчик запросов повторно вызывается при поступлении каждого нового запроса. В данном случае внутри сложного кода обработки запроса бесконечный цикл по-прежнему нежелателен, пусть даже простая обертка вокруг него и должна работать бесконечно.

В реальном мире можно встретить много примеров программ, которые повторно выполняют конечные вычисления внутри бесконечного цикла: веб-серверы, приложения с графическим интерфейсом, операционные системы и т. д. И хотя в общем случае мы хотим, чтобы алгоритмические программы, преобразующие вход в выход, всегда останавливались, аналог этой цели для подобных длительно работающих

систем – оставаться *продуктивными*, то есть продолжать работать, не застревая в состоянии, когда они ни на что не реагируют.

Так почему к любой универсальной системе в довесок прилагается опасность не завершиться? Нет ли какого-нибудь хитроумного способа, не принося в жертву полезность, наложить на машины Тьюринга такое ограничение, чтобы они непременно останавливались? Откуда мы знаем, что в один прекрасный день не будет изобретен язык программирования такой же мощный, как Ruby, но не имеющий бесконечных циклов? Можно привести множество конкретных примеров, показывающих, почему это нельзя сделать, но существует и более общая аргументация, которую мы сейчас и рассмотрим.

Ruby – универсальный язык программирования, поэтому должна быть возможность написать на нем код, который вычислял бы Ruby-код. В принципе, мы можем определить метод `#evaluate`, который принимает исходный код Ruby-программы и строку, подаваемую ей на вход, и возвращает результат (то есть строку, посылаемую на стандартный вывод) вычисления данной программы с данным входом.

Такая реализация `#evaluate` слишком сложна для включения в эту главу, но тем не менее приведем самый общий набросок его работы:

```
def evaluate(program, input)
  # разобрать программу
  # вычислить программу для данного входа и запомнить выход
  # вернуть выход
end
```

По существу, `#evaluate` – это интерпретатор Ruby, написанный на Ruby. И хотя мы опустили его реализацию, нет сомнений, что она возможна: сначала нужно преобразовать `program` в последовательность лексем и подвергнуть их синтаксическому анализу для построения дерева разбора (см. раздел «Разбор с помощью автоматов с магазинной памятью» на стр. 160), а затем вычислить это дерево согласно операционной семантике Ruby (см. раздел «Операционная семантика» на стр. 36). Это большая и сложная работа, но ее безусловно можно проделать, иначе Ruby не мог бы считаться универсальным языком.

Для простоты предположим, что наша воображаемая реализация `#evaluate` не содержит ошибок и не «грохнется» при выполнении

программы – если уж мы воображаем какой-то код, то почему бы заодно не вообразить, что он идеален? Конечно, он может вернуть результат, означающий, что программа возбудила исключение при выполнении, но это совсем не то же самое, что фатальная ошибка в самом методе `#evaluate`.

Вообще-то, в Ruby имеется встроенный метод `Kernel#eval`, который умеет вычислять Ruby-код, представленный в виде строки, но воспользоваться им значило бы смошенничать – не в последнюю очередь потому, что написан этот метод (в MRI) на C, а не на Ruby. К тому же, для настоящего обсуждения это и не нужно; мы используем Ruby как репрезентативный пример универсального языка программирования, а во многих универсальных языках встроенного `eval` нет.

Однако же, раз уж этот метод имеется, то было бы стыдно не использовать его, чтобы сделать метод `#evaluate` не таким воображаемым. Вот грубая попытка – с извинениями за мелкий обман:

```
require 'stringio'

def evaluate(program, input)
  old_stdin, old_stdout = $stdin, $stdout
  $stdin, $stdout = StringIO.new(input), (output = StringIO.new)

  begin
    eval program
  rescue Exception => e
    output.puts(e)
  ensure
    $stdin, $stdout = old_stdin, old_stdout
  end

  output.string
end
```

В этой реализации много практических и философских проблем, которых можно было бы избежать, написав `#evaluate` на чистом Ruby. С другой стороны, она коротенькая и достаточно хорошо работает, так что для демонстрации ее можно и включить:

```
>> evaluate('print $stdin.read.reverse', 'hello world')
=> "dlrow olleh"
```

Наличие метода `#evaluate` позволяет определить еще один метод, `#evaluate_on_itself`, который возвращает результат вычисления `program`, когда ей на вход подается *ее собственный исходный код*:

```
def evaluate_on_itself(program)
  evaluate(program, program)
end
```

Вещь, на первый взгляд, странная, но совершенно законная; `program` – обычная строка, поэтому мы вправе рассматривать ее и как Ruby-программу, и как входные данные для этой программы. Ведь код – это данные, не так ли?

```
>> evaluate_on_itself('print $stdin.read.reverse')
=> "esrever.daer.nidts$ tnirp"
```

Поскольку мы знаем, что можем реализовать методы `#evaluate` и `#evaluate_on_itself` на Ruby, то должна быть и возможность написать на Ruby законченную программу *does_it_say_no.rb*:

```
def evaluate(program, input)
  # разобрать программу
  # вычислить программу для данного входа и запомнить выход
  # вернуть выход
end

def evaluate_on_itself(program)
  evaluate(program, program)
end

program = $stdin.read

if evaluate_on_itself(program) == 'no'
  print 'yes'
else
  print 'no'
end
```

Эта программа – прямое применение уже имеющегося кода: мы определяем методы `#evaluate` и `#evaluate_on_itself`, затем читаем другую Ruby-программу из стандартного ввода и передаем ее методу `#evaluate_on_itself`, чтобы посмотреть, что будет делать эта программа, если подать ей на вход ее же саму. Если результатом является строка `'no'`, то *does_it_say_no.rb* выводит `'yes'`, иначе `'no'`. Например[1]:

```
$ echo 'print $stdin.read.reverse' | ruby does_it_say_no.rb
no
```

[1] Мы используем здесь синтаксис оболочки Unix. В Windows следует опустить одиночные кавычки вокруг аргумента `echo` или поместить его в текстовый файл и подать на вход `ruby` с помощью оператора перенаправления ввода `<`.

Результат вполне ожидаем; как мы видели выше, подав на вход программы `print $stdin.read.reverse` ее саму, мы получаем на выходе `esrever.daer.nidts$ tnirp`, то есть совсем не `no`. А как насчет программы, которая таки печатает `no`?

```
$ echo 'if $stdin.read.include?(«no») then print «no» end' |
  ruby does_it_say_no.rb
yes
```

Снова ожидаемо.

А теперь интересный вопрос: что произойдет, если выполнить команду *ruby does_it_say_no.rb < does_it_say_no.rb*[1]? Помните, что *does_it_say_no.rb* – настоящая программа, которую мы могли бы написать, будь у нас достаточно времени и желания, поэтому она должна делать *что-то*, вот только сразу неясно, что именно. Попробуем разобраться, рассмотрев все возможности и исключив заведомо бессмысленные.

Во-первых, прогон этой конкретной программы с ее собственным исходным кодом в качестве входа никак не может привести к выводу `yes`. В соответствии с логикой программы, `yes` выводится, только когда выполнение *does_it_say_no.rb* со своим исходным кодом приводит к выводу `no`, что противоречит исходному предположению. Так что этот ответ неправилен.

Ладно, тогда, быть может, `no`? Но опять же программа устроена так, что может выводить `no`, только если точно такое вычисление *не* выводит `no` – снова противоречие.

Быть может, она могла бы вывести еще какую-то строку, например `maybe` или даже пустую? Но и тут мы сталкиваемся с противоречием: если `evaluate_on_itself(program, program)` не возвращает `no`, то программа печатает `no`, а не что-либо другое.

Итак, она не может вывести ни `yes`, ни `no`, ни что-то еще. И даже «грохнуться» не может, если метод `#evaluate` не содержит ошибок, а мы предположили, что их нет. Единственная оставшаяся возможность – что она вообще ничего не выводит, а такое может случиться, только если программа никогда не завершается: метод `#evaluate` должен крутиться в бесконечном цикле, не возвращая результат.

На практике команда *ruby does_it_say_no.rb < does_it_say_no.rb* почти наверняка исчерпает конечную память машины, что вызовет крах *ruby*, а не будет циклиться бесконечно. Однако это

[1] Это команда оболочки, которая запускает программу *does_it_say_no.rb*, подавая ей на вход ее собственный исходный код.

ограничение на ресурсы, наложенное извне, а не свойство самой программы; в принципе мы могли бы добавить компьютеру столько памяти, сколько необходимо, и вычисление продолжалось бы неопределенно долго.

Возможно, сказанное выше показалось вам неоправданно сложным способом доказать, что на Ruby можно писать программы, которые не останавливаются. В конце концов, `while true do end` – куда более простой пример с тем же результатом.

Но рассуждая о поведении программы *does_it_say_no.rb*, мы показали, что незавершающиеся программы – неизбежное следствие универсальности, не зависящее от конкретных особенностей системы. Наше рассуждение не опиралось ни на какие свойства Ruby, кроме его универсальности, поэтому те же идеи применимы к машинам Тьюринга, к лямбда-исчислению и к любой другой универсальной системе. Всякий раз работая с языком, достаточно мощным, чтобы выполнить написанную на нем же программу, мы можем быть уверены, что сумеем воспользоваться написанным на нем эквивалентом метода `#evaluate` для построения незавершающейся программы, пусть даже больше ничего о возможностях языка не знаем.

В частности, невозможно удалить из языка программирования какие-либо средства (например, циклы `while`), так чтобы предотвратить создание незавершающихся программ, оставив язык универсальным. Если удаление некоторого средства делает невозможным написание программы с бесконечным циклом, то станет также невозможно реализовать метод `#evaluate`.

Языки, тщательно спроектированные так, что написанные на них программы обязательно останавливаются, называются *тотальными* – в противоположность более распространенным *частичным языкам программирования*, программы на которых иногда останавливаются и дают ответ, а иногда нет. Тотальные языки могут быть очень мощными, позволяющими выразить много полезных вычислений, но интерпретировать себя им не дано.

Это удивительно, так как эквивалент метода `#evaluate` для тотального языка программирования по определению должен останавливаться, и тем не менее реализовать его на этом языке невозможно – если бы это было не так, то мы могли бы использовать ту же технику, что в программе *does_it_say_no.rb*, чтобы зациклить его.

И вот вам первое дразнящее видение невозможной программы: мы не можем написать интерпретатор тотального языка про-

граммирования на нем самом, пусть даже существует вполне респектабельный, гарантированно останавливающийся алгоритм его интерпретации. На самом деле, настолько респектабельный, что его *можно было бы* реализовать на другом, более развитом тотальном языке, но на том в свою очередь нельзя было бы написать интерпретатор его самого.

Интересная диковинка, но тотальные языки специально создаются с искусственными ограничениями; а мы ведь искали нечто такое, что *никакой* компьютер и *никакой* язык программирования не в состоянии сделать. Будем искать дальше.

Программы могут ссылаться сами на себя

Трюк со ссылкой на себя же, который мы использовали в программе *does_it_say_no.rb*, опирается на возможность построить программу, которая умеет читать собственный исходный код, но, пожалуй, было бы не совсем честно предполагать, что такое всегда возможно. В нашем примере программа получала свой исходный код в виде явных входных данных – спасибо функциональности, предоставляемой окружающей средой (то есть оболочкой); не будь такой любезности, она могла бы прочитать данные прямо с диска с помощью метода `File.read(__FILE__)`, воспользовавшись встроенным в Ruby API для работы с файловой системой и специальной константой `__FILE__`, которая всегда содержит имя текущего файла.

Но предполагалось, что мы сможем предъявить аргумент общего характера, зависящий только от универсальности Ruby, а не от возможностей операционной системы или класса `File`. Как быть с компилируемыми языками, например Java или C, которые могут и не иметь доступа к своему исходному коду во время выполнения? А с программами на JavaScript, которые загружаются в память по сети и могут вообще не храниться в локальной файловой системе? А как насчет автономных универсальных систем типа машины Тьюринга или лямбда-исчисления, где понятия «файловая система» и «стандартный ввод» вовсе отсутствуют?

К счастью, аргументация на основе скрипта *does_it_say_no.rb* успешно отражает это возражение, потому что умение программы читать собственный исходный код из стандартного ввода – просто удобный эвфемизм для действия, на которое способны все универсальные системы независимо от окружения или других аспектов. Это следствие фундаментальной *второй теоремы Клини о рекурсии*, которая утверждает, что любую программу можно преобразо-

вать в эквивалентную форму, которая может вычислить собственный исходный код. Теорема о рекурсии возрождает уверенность в оправданности нашего эвфемизма: мы могли бы заменить строку `program = $stdin.read` кодом, который генерирует исходный код *does_it_say_no.rb* и присваивает его переменной `program`, не выполняя вообще никакого ввода-вывода.

Посмотрим, как выглядит такое преобразование для простой Ruby-программы, например, такой:

```
x = 1
y = 2
puts x + y
```

Мы хотим преобразовать в программу примерно такого вида...

```
program = '…'
x = 1
y = 2
puts x + y
```

... где переменной `program` присваивается строка, содержащая исходный код полной программы. Но каким должно быть значение `program`?

Наивная попытка состряпать простенький строковый литерал, который можно присвоить переменной `program` быстро заводит в тупик, потому что такой литерал должен был бы являться частью исходного кода программы, а, значит, встречаться где-то внутри себя самого. Таким образом, `program` должна была бы начинаться строкой `'program ='`, за которой следует значение `program`, за которым следует строка `'program ='`, за которой снова следует значение `program`, и так далее до бесконечности:

```
program = %q{program = %q{program = %q{program = %q{program = %q{program = %q{…}}}}}}
x = 1
y = 2
puts x + y
```

Конструкция Ruby `%q` позволяет закавычить неинтерполируемые строки парой символов-ограничителей, в данном случае фигурных скобок, а не одиночными кавычками. Преимущество в том, что строковый литерал может содержать неэкранированные экземпляры ограничителей при условии, что они правильно сбалансированы:

```
>> puts %q{Curly brackets look like { and }.}
Curly brackets look like { and }.
=> nil
>> puts %q{An unbalanced curly bracket like } is a problem.}
SyntaxError: syntax error, unexpected tIDENTIFIER, expecting end-of-input
```

Использование `%q` позволяет избежать сложностей с экранированием в строках, которые содержат символы, совпадающие с ограничителями:

```
program = 'program = \'program = \\\'program = \\\\\\\'…\\\\\\\'\\\'\'
```

Выбраться из этой бесконечно глубокой ямы можно, воспользовавшись тем фактом, что используемое в программе значение необязательно должно присутствовать в коде буквально; оно может динамически вычисляться по другим данным. Это означает, что преобразованная программа может состоять из трех частей:

A. Присвоить строковый литерал переменной (назовем ее `data`).
B. Использовать эту строку для вычисления исходного кода текущей программы и присвоить ее `program`.
C. Выполнить прочие действия, для которых и был написан код, с которого мы начинали.

Таким образом, структура программы будет, скорее, такой:

```
data = '…'
program = …
x = 1
y = 2
puts x + y
```

В качестве общей стратегии звучит обнадеживающе, но не хватает проработанности деталей. Откуда нам знать, какую строку присвоить переменной `data` в части A, и как использовать ее в части B для вычисления `program`? Вот одно из возможных решений:

- В части A создать строковый литерал, который содержит исходный код частей B и C, и присвоить эту строку переменной `data`. Эта строка не обязана «содержать себя», потому что это код не всей программы, а только ее участка, следующего за частью A.
- В части B сначала вычислить строку, которая содержит исходный код части A. Это можно сделать, потому что часть A состоит в основном из большого строкового литерала, значение которого доступно через `data`, так что для воссоздания ис-

ходного кода части A нам нужно лишь добавить к значению `data` префикс `'data ='`. Затем просто конкатенировать результат с `data` для получения исходного кода всей программы (поскольку `data` содержит исходный код частей B и C) и присвоить то, что получится, переменной `program`.

План все еще страдает цикличностью – часть A порождает исходный код части B, а часть B порождает исходный код части A – но нам удалось избежать бесконечного возврата, поскольку часть B только *вычисляет* исходный код части A, а не содержит его буквально.

Теперь можно потихоньку двигаться вперед, подставляя недостающие кусочки, о которых мы уже знаем. Почти весь исходный код частей B и C у нас уже есть, так что можно частично определиться со значением `data`:

```
data = %q{
program = …
x = 1
y = 2
puts x + y
}
program = …
x = 1
y = 2
puts x + y
```

`data` должна содержать символы новой строки. Представляя их фактическими символами новой строки в неинтерполируемом строковом литерале, а не интерполируемыми управляющими последовательностями `\n`, мы можем включить исходный код частей B и C буквально, не прибегая к специальному кодированию или экранированию[1]. Такое прямое копирование упрощает вычисление исходного кода части A.

Мы также знаем, что исходный код части A – это просто строка `'data = %q{…}'`, в которой место внутри фигурных скобок занимает `data`, поэтому можем частично заполнить и значение `program`:

```
data = %q{
program = …
```

[1] Это сошло нам с рук только потому, что в частях B и C по счастливой случайности нет «трудных» символов, например, знаков обратной черты или несбалансированных фигурных скобок. А если бы они были, то пришлось бы их как-то экранировать, а потом снимать экранирование по ходу сборки значения `program`.

```
x = 1
y = 2
puts x + y
}
program = "data = %q{#{data}}" + …
x = 1
y = 2
puts x + y
```

Теперь в `program` не хватает только кода частей B и C, то есть именно того, что содержит `data`, поэтому мы можем дописать значение `data` в конец `program` – и дело сделано:

```
data = %q{
program = …
x = 1
y = 2
puts x + y
}
program = "data = %q{#{data}}" + data
x = 1
y = 2
puts x + y
```

Наконец, мы можем вернуться и подправить значение `data` с учетом того, как на самом деле выглядит часть B:

```
data = %q{
program = "data = %q{#{data}}" + data
x = 1
y = 2
puts x + y
}
program = "data = %q{#{data}}" + data
x = 1
y = 2
puts x + y
```

Вот теперь все! Эта программа делает то же самое, что первоначальная, но теперь в ней есть дополнительная локальная переменная, содержащая ее же исходный код, – пиррова победа, так как ничего с этой переменной программа *не делает*. А что, если преобразовать программу, которая ожидает локальную переменную с именем `program` и что-то делает с ней? Возьмем классический пример:

```
puts program
```

Эта программа пытается напечатать собственный исходный код[1], но, очевидно, в таком виде ничего не выйдет, потому что переменная `program` не определена. Если применить к ней описанную выше трансформацию в самоссылающийся код, то получится следующий результат

```
data = %q{
program = "data = %q{#{data}}" + data
puts program
}
program = "data = %q{#{data}}" + data
puts program
```

Уже интереснее. Посмотрим, что эта программа выводит на консоль:

```
>> data = %q{
     program = "data = %q{#{data}}" + data
     puts program
     }
=> "\nprogram = \"data = %q{\#{data}}\" + data\nputs program\n"
>> program = "data = %q{#{data}}" + data
=> "data = %q{\nprogram = \"data = %q{\#{data}}\" + data\nputs program\n}\n↵
program = \"data = %q{\#{data}}\" + data\nputs program\n"
>> puts program
data = %q{
program = "data = %q{#{data}}" + data
puts program
}
program = "data = %q{#{data}}" + data
puts program
=> nil
```

Как и следовало ожидать, строка `puts program` действительно печатает исходный код всей программы.

Надеемся, всем понятно, что эта трансформация не зависит ни от каких специальных свойств самой программы, то есть она будет работать для любой Ruby-программы, и, следовательно, от использования `$stdin.read` или `File.read(__FILE__)` для чтения программой собственного исходного кода всегда можно отказаться[2]. Трансфор-

---

[1] Дуглас Хофштадтер придумал для программ, распечатывающих свой исходный текст, термин *квайн*.

[2] Можете ли вы противиться настоятельной потребности написать Ruby-программу, которая сможет применить такую трансформацию к любой Ruby-программе? Если использовать `%q{}` для закавычивания значения, то как быть со знаками обратной косой черты и несбалансированными фигурными скобками?

мация не зависит также от свойств самого Ruby – нужна лишь возможность вычислять новые значения по старым, которой обладает любая универсальная система, – а отсюда следует, что любую машину Тьюринга можно адаптировать для ссылки на свою собственную кодировку, любое выражение лямбда-исчисления можно расширить так, что оно будет содержать представление собственного синтаксиса в лямбда-исчислении, и т. д.

Разрешимость

Мы уже видели, что машины Тьюринга обладают значительной мощностью и гибкостью: они могут исполнять произвольные программы, закодированные как данные, выполнять любые мыслимые алгоритмы, работать сколь угодно долго и вычислять собственные описания. И несмотря на свою простоту, эти крохотные воображаемые машины представляют универсальные системы в целом.

Но если они такие мощные и гибкие, то есть ли что-то такое, чего машины Тьюринга – и, следовательно, реальные компьютеры и языки программирования – сделать не в состоянии?

Прежде чем искать ответ на этот вопрос, необходимо уточнить постановку. Что именно мы можем попросить машину Тьюринга сделать и как определить, что это не может быть сделано? Нужно ли исследовать все возможные типы задач или достаточно ограничиться лишь некоторыми? Ищем ли мы задачи, решение которых просто выходит за пределы нашего нынешнего понимания, или задачи, которые заведомо *никогда* не могут быть решены?

Чтобы сузить рамки рассмотрения, сосредоточимся на *проблеме разрешимости*. Проблема разрешимости возникает всякий раз, как нужно дать ответ «да» или «нет» на какой-нибудь вопрос, например: «верно ли, что 2 меньше 3?» или «соответствует ли регулярное выражение `(a(|b))*` строке `'abaab'`?». Проблемы разрешимости проще *функциональных проблем*, в которых ответом является число или еще какое-нибудь не-булево значение, например: «чему равен наибольший общий делитель 18 и 12?», но все равно достаточно интересны для исследования.

Проблема разрешимости называется *разрешимой* (или *вычислимой*), если существует алгоритм, который гарантированно решает ее за конечное время при любых входных данных. Тезис Чёрча-Тьюринга утверждает, что любой алгоритм может быть выполнен машиной Тьюринга, поэтому доказать разрешимость проблемы можно,

построив машину Тьюринга, которая всегда дает правильный ответ и останавливается, если дать ей поработать достаточно долго. Нетрудно придумать интерпретацию конечной конфигурации машины Тьюринга как ответа «да» или «нет»: например, можно посмотреть, записала ли она в текущую позицию символ `Y` или `N`, или вообще не обращать внимания на содержимое ленты, а анализировать лишь, является ли конечное состояние заключительным («да») или не заключительным («нет»).

Все проблемы разрешимости, встречавшиеся нам в предыдущих главах, разрешимы. Некоторые, например «допускает ли данный конечный автомат эту цепочку?» или «соответствует ли данное регулярное выражение этой строке?», очевидно, разрешимы, потому что мы написали на Ruby программы для их решения, смоделировав конечные автоматы напрямую. При наличии достаточного времени и мотивации эти программы можно преобразовать в машины Тьюринга, а поскольку для их выполнения требуется конечное число шагов – каждый шаг ДКА потребляет один входной символ и число символов во входном потоке конечно – то они гарантированно останавливаются и дают ответ «да» или «нет». Таким образом, первоначальная проблема действительно разрешима.

С другими проблемами не все так ясно. Проблема «допускает ли данный автомат с магазинной памятью эту цепочку?» может показаться неразрешимой, потому что, как мы видели, прямое моделирование автомата с магазинной памятью на Ruby потенциально может зациклиться и никогда не вернуть ответ. Однако, как выясняется, существует способ точно вычислить, сколько шагов должна будет проделать модель конкретного автомата с магазинной памятью, чтобы допустить или отвергнуть входную строку заданной длины[1], поэтому проблема все-таки разрешима: мы должны лишь вычислить число потребных шагов, выполнить столько шагов модели, а затем проверить, допустила она входную цепочку или нет.

---

[1] В двух словах для любопытных: для каждого автомата с магазинной памятью существует эквивалентная контекстно-свободная грамматика (КСГ) и наоборот. Любую КСГ можно переписать в *нормальной форме Хомского*, а любой КСГ, записанной в этой форме, требуется ровно $2n - 1$ шагов для порождения цепочки длиной n. Таким образом, мы можем преобразовать исходный АМП в КСГ, переписать КСГ в нормальной форме Хомского, а затем преобразовать эту КСГ обратно в АМП. Получившийся автомат с магазинной памятью распознает тот же язык, что исходный, но теперь мы точно знаем, сколько для этого нужно шагов.

Так что, это всегда можно сделать? Существует ли какой-нибудь изобретательный подход к проблеме, что позволил бы реализовать машину или программу, которая гарантированно решит ее за конечное время?

Увы, нет. Существует много проблем разрешимости – *бесконечно* много – и, как выясняется, среди них множество неразрешимых: не существует алгоритма их решения, который гарантированно останавливается. Любая такая проблема неразрешима не потому, мы пока не нашли для нее подходящего алгоритма, а потому что ее в принципе невозможно решить при некоторых входных данных, и можно даже доказать, что алгоритм никогда не будет найден.

Проблема остановки

Многие неразрешимые проблемы касаются поведения машин и программ во время выполнения. Самая известная из них, *проблема остановки*, заключается в том, чтобы решить, остановится ли выполнение данной машины Тьюринга с данной начальной лентой. С учетом универсальности эту задачу можно переформулировать в виде, более удобном для практических целей: если дана некоторая строка, содержащая исходный код Ruby-программы, и другая строка – с данными, которые эта программа должна прочитать из стандартного ввода, – то будет ли в результате прогона этой программы получен ответ (в конечном итоге) или она попадет в бесконечный цикл?

Построение анализатора остановки

Сразу не очевидно, почему проблема остановки неразрешима. Ведь нетрудно придумать конкретные программы, для которых вопрос имеет ответ. Например, следующая программа заведомо останавливается независимо от входной строки:

```
input = $stdin.read
puts input.upcase
```

Мы будем предполагать, что метод `$stdin.read` всегда возвращает значение немедленно – то есть стандартный ввод любой программы конечный и неблокирующий – потому что нас интересует внутреннее поведение программы, а не ее взаимодействие с операционной системой.

Напротив, после незначительного изменения исходного кода получается программа, которая, очевидно, никогда не остановится:

```
input = $stdin.read

while true
  # ничего не делать
end

puts input.upcase
```

Мы, конечно, можем написать анализатор остановки, различающий только эти два случая. Достаточно проверить, есть ли в исходном коде программы строка `while true`:

```
def halts?(program, input)
  if program.include?('while true')
    false
  else
    true
  end
end
```

Эта реализация метода `#halts?` дает правильные ответы для обеих приведенных выше программ:

```
>> always = "input = $stdin.read\nputs input.upcase"
=> "input = $stdin.read\nputs input.upcase"
>> halts?(always, 'hello world')
=> true
>> never = "input = $stdin.read\nwhile true\n# do nothing\nend\nputs input.upcase"
=> "input = $stdin.read\nwhile true\n# do nothing\nend\nputs input.upcase"
>> halts?(never, 'hello world')
=> false
```

Однако для других программ предикат `#halts?`, скорее всего, даст неверный ответ. Например, существуют программы, которые останавливаются или не останавливаются в зависимости от входных данных:

```
input = $stdin.read

if input.include?('goodbye')
  while true
    # ничего не делать
  end
else
  puts input.upcase
end
```

Всегда можно дополнить анализатор для учета таких частных случаев, поскольку мы знаем, что искать:

```
def halts?(program, input)
  if program.include?('while true')
    if program.include?('input.include?(\'goodbye\')')
      if input.include?('goodbye')
        false
      else
        true
      end
    else
      false
    end
  else
    true
  end
end
```

Теперь у нас имеется анализатор, который дает правильный ответ для всех трех программ и любых входных строк:

```
>> halts?(always, 'hello world')
=> true
>> halts?(never, 'hello world')
=> false
>> sometimes = "input = $stdin.read\nif input.include?('goodbye')\nwhile true\n↵
# do nothing\nend\nelse\nputs input.upcase\nend"
=> "input = $stdin.read\nif input.include?('goodbye')\nwhile true\n# do nothing\n↵
end\nelse\nputs input.upcase\nend"
>> halts?(sometimes, 'hello world')
=> true
>> halts?(sometimes, 'goodbye world')
=> false
```

Можно было бы продолжать в том же духе до бесконечности: добавлять все новые проверки и частные случаи для поддержки все расширяющегося набора программ, но так не решить общую проблему: как узнать, остановится ли *произвольная* программа? Реализацию методом грубой силы можно сделать все более и более точной, но у нее всегда будут существовать «слепые зоны»; бесхитростный поиск конкретных синтаксических конструкций, очевидно, не может покрыть все возможные программы.

Написать метод `#halts?` в общем виде, так чтобы она работал для любой программы и любой входной строки, похоже, не так-то просто. Если программа вообще содержит циклы – явные, типа `while`, или неявные, например рекурсивные вызовы, – то она потенциально может работать бесконечно, и, если мы хотим что-то сказать о ее

поведении для конкретных входных данных, то необходим сложный анализ *семантики* программы. Человек, конечно, сразу увидит, что следующая программа всегда останавливается:

```
input = $stdin.read
output = ''

n = input.length
until n.zero?
  output = output + '!'
  n = n - 1
end

puts output
```

Но *почему* она всегда останавливается? Понятно, что никакой лежащей на поверхности синтаксической причины нет. Объяснение заключается в том, что метод `IO#read` всегда возвращает объект `String`, а метод `String#length` может вернуть только неотрицательное целое число (объект `Integer`), а повторный вызов метода `-(1)` неотрицательного целого числа рано или поздно породит объект, метод `#zero?` которого возвращает `true`. Это цепочка тонких рассуждений легко уязвима для небольших модификаций; если предложение `n = n - 1` внутри цикла заменить на `n = n - 2`, то программа будет останавливаться только для исходных строк четной длины. Анализатор остановки, который знает все эти особенности Ruby и чисел и умеет связать факты между собой для принятия точных решений о программах такого вида, окажется большим и сложным.

Фундаментальная трудность заключается в том, что сложно предсказать, что будет делать программа, не запуская ее. Возникает соблазн прогнать программу с помощью `#evaluate` и посмотреть, останавливается она или нет, но ни к чему хорошему это не приведет: если программа не останавливается, то `#evaluate` будет работать вечно и, сколько бы мы ни ждали, ответа от `#halts?` не получим. Надежный алгоритм обнаружения остановки должен каким-то образом давать определенный ответ за конечное время, основываясь лишь на анализе текста программы и не прибегая к запуску и ожиданию завершения.

Это никогда работать не будет

Ну ладно, интуиция подсказывает, что правильно реализовать метод `#halts?` будет сложно, но это еще не значит, что проблема остановки неразрешима. Существует немало трудных задач (например,

написание метода `#evaluate`), которые тем не менее можно решить, приложив достаточно усилий и изобретательности; неразрешимость проблемы остановки означает, что метод `#halts?` не просто очень трудно, а *невозможно* написать.

Откуда мы знаем, что корректной реализации `#halts?` нет и быть не может? Если это лишь инженерная проблема, то что мешает бросить на нее легион программистов, которые в конце концов найдут решение?

Слишком хорошо, чтобы быть правдой

Допустим на минутку, что проблема остановки разрешима. В этом воображаемом мире можно написать полную реализацию `#halts?`, так что вызов `halts?(program, input)` всегда возвращает `true` или `false` для любых значений `program` и `input`, и полученный ответ правильно предсказывает, остановится ли программа `program`, если прочитает из стандартного ввода строку `input`. Общая структура метода `#halts?` могла бы выглядеть как-то так:

```
def halts?(program, input)
  # разобрать программу
  # проанализировать программу
  # вернуть true, если программа останавливается на входной строке, и false
  #    в противном случае
end
```

Если мы можем написать `#halts?`, то можем построить программу *does_it_halt.rb*, которая читает другую программу в качестве входных данных и печатает `yes` или `no` в зависимости от того, останавливается эта программа, прочитав пустую строку, или нет[1]:

```
def halts?(program, input)
  # разобрать программу
  # проанализировать программу
  # вернуть true, если программа останавливается на входной строке, и false
  #    в противном случае
end

def halts_on_empty?(program)
  halts?(program, '')
end

program = $stdin.read
```

[1] Выбор именно пустой строки несущественен, подойдет любая фиксированная строка. Наш план состоит в том, чтобы прогнать *does_it_halt.rb* для автономной программы, которая ничего не читает из стандартного ввода, так что значение `input` может быть любым.

```
if halts_on_empty?(program)
  print 'yes'
else
  print 'no'
end
```

Имся программу *does_it_halt.rb*, мы можем использовать ее для решения очень трудных задач. Рассмотрим знаменитую гипотезу, высказанную Кристианом Гольдбахом в 1742 году:

> Любое четное целое число, большее 2, можно представить в виде суммы двух простых чисел.

Знаменита эта гипотеза тем, что никому не удалось доказать ее истинность или ложность. Эмпирические факты свидетельствуют в пользу истинности, потому что взятое наугад четное число неизменно удается разложить в сумму двух простых – 12 = 5 + 7, 34 = 3 + 31, 567890 = 7 + 567883 и так далее – и с помощью компьютеров было проверено, что это верно для всех четных чисел от 4 до 4 квинтильонов (4 000 000 000 000 000 000). Но четных чисел бесконечно много, так что никакой компьютер не сможет проверить все, а доказательство возможности такого разложения неизвестно. Существует вероятность, пусть даже очень маленькая, что найдется какое-то очень большое четное число, *не* представимое в виде суммы двух простых.

Доказательство гипотезы Гольдбаха – одна из заветных целей теории чисел; в 2000 году издательство «Фабер и Фабер» объявило награду в миллион долларов любому, кто сможет его представить. Однако секундочку: у нас ведь уже есть инструмент, который позволит узнать, верна ли гипотеза! Нам всего-то и нужно написать программу, которая ищет контрпример:

```
require 'prime'

def primes_less_than(n)
  Prime.each(n - 1).entries
end

def sum_of_two_primes?(n)
  primes = primes_less_than(n)
  primes.any? { |a| primes.any? { |b| a + b == n } }
end

n = 4

while sum_of_two_primes?(n)
  n = n + 2
end

print n
```

Таким образом, мы установили связь между истинностью гипотезы Гольдбаха и проблемой остановки конкретной программы. Если гипотеза истинна, то эта программа никогда не найдет контрпример и будет крутиться в цикле вечно; если ложна, то переменной `n` в конечном итоге будет присвоено четное число, не являющееся суммой двух простых, и программа остановится. Поэтому осталось сохранить эту программу в файле *goldbach.rb*, запустить ее командой *ruby does_it_halt.rb < goldbach.rb* и посмотреть, останавливается ли она. В результате мы узнаем, истинна ли гипотеза Гольдбаха. Миллион долларов наш[1]!

Очевидно, это слишком хорошо, чтобы быть правдой. Чтобы написать программу, которая сможет точно предсказать поведение *goldbach.rb*, нужно знать теорию чисел куда лучше, чем мы. Математики сотни лет пытаются доказать или опровергнуть гипотезу Гольдбаха, так что маловероятно, что кучка алчных программистов сможет каким-то чудом решить не только эту, но и *любую* нерешенную математическую задачу, которую можно представить в виде циклической программы.

Принципиально невозможно

До сих пор мы видели лишь сильные аргументы в пользу неразрешимости проблемы остановки, но не неопровержимое доказательство. Интуиция, может, и подсказывает, что доказать или опровергнуть гипотезу Гольдбаха путем преобразования ее в программу, скорее всего, не получится, но вычисления иногда идут вразрез с интуицией, так что не следует довольствоваться лишь рассуждениями о том, как это невероятно. Если проблема остановки действительно неразрешима, а не просто очень трудна, то это надо доказать.

Объясним, почему метод `#halts?` никогда не будет работать. *Если бы* он работал, то мы смогли бы построить новый метод `#halts_on_itself?`, который вызывает `#halts?`, чтобы решить, что будет делать программа, когда на вход подается ее собственный исходный код[2]:

```
def halts_on_itself?(program)
  halts?(program, program)
end
```

[1] Предложение миллиона долларов Фабером утратило силу в 2002 году, но и сегодня любой человек, представивший доказательство, обретет славу и почет рок-звезды в математических кругах.

[2] Это напоминает метод `#evaluate_on_itself` (см. раздел «Универсальные системы могут зацикливаться» на стр. 316), только вместо `#evaluate` мы подставили `#halt`.

Как и `#halts?`, метод `#halts_on_itself?` всегда завершается и возвращает булево значение: `true`, если `program` останавливается на своем же исходном коде, и `false`, если циклится бесконечно.

Имея работающие реализации `#halts?` и `#halts_on_itself?`, мы можем написать программу *do_the_opposite.rb*:

```
def halts?(program, input)
  # разобрать программу
  # проанализировать программу
  # вернуть true, если программа останавливается на входной строке, и false
  #    в противном случае
end

def halts_on_itself?(program)
  halts?(program, program)
end

program = $stdin.read

if halts_on_itself?(program)
  while true
    # ничего не делать
  end
end
```

Этот код читает программу `program` из стандартного ввода, выясняет, останавливается ли она на собственном исходном коде, и делает прямо противоположное: если `program` останавливается, то *do_the_opposite* циклится бесконечно, а если `program` бесконечно циклится, то *do_the_opposite.rb* останавливается.

А теперь позвольте вопрос: что произойдет при запуске команды *ruby do_the_opposite.rb < do_the_opposite.rb*[1]? Как и при рассмотрении программы *does_it_say_no.rb* выше, этот вопрос приводит к неустранимому противоречию.

`#halts_on_itself?` должен вернуть `true` или `false`, если передать ему в качестве аргумента исходный код *do_the_opposite.rb*. Если он возвращает `true`, то есть программа останавливается, то команда `ruby do_the_opposite.rb < do_the_opposite.rb` будет циклиться бесконечно, а, значит, `#halts_on_itself?` ошибся. С другой стороны, если `#halts_on_itself?` возвращает `false`, то `do_the_opposite.rb` немедленно останавливается, что снова противоречит предсказанию `#halts_on_itself?`.

---

[1] Или эквивалентно: что вернет метод `#halts_on_itself?`, если вызвать его с исходным кодом do_the_opposite.rb в качестве аргумента?

И винить `#halts_on_itself?` тут не за что – это всего лишь невинная однострочная обертка вокруг метода `#halts?`, доверяющая его вердикту. Таким образом, мы доказали, что метод `#halts?` не может вернуть правильный ответ, будучи вызван с исходным кодом *do_the_opposite.rb* в качестве аргументов `program` и `input`; как бы усердно он ни работал, любой вывод, к которому он придет, автоматически окажется неверным. Это означает, что любая реализация `#halts?` обречена на один из двух уделов:

- иногда она дает неверный ответ, например, предсказывает, что `do_the_opposite.rb` будет бесконечно циклиться, хотя в действительности он останавливается (или наоборот);
- иногда она бесконечно циклится и вообще не возвращает никакого ответа, как метод `#evaluate` при запуске команды *ruby does_it_say_no.rb < does_it_say_no.rb*.

Таким образом, правильной реализации `#halts?` существовать не может: обязательно найдутся такие исходные данные, на которых она дает неверное предсказание или не дает никакого.

Напомним определение разрешимости:

> Проблема разрешимости называется *разрешимой* (или *вычислимой*), если существует алгоритм, который гарантированно решает ее за конечное время при любых входных данных.

Мы доказали, что невозможно написать Ruby-программу, которая полностью решает проблему остановки, а поскольку Ruby-программы по мощности эквивалентны машинам Тьюринга, то и с помощью машины Тьюринга эту проблему не решить. Тезис Чёрча-Тьюринга утверждает, что *любой* алгоритм может быть выполнен машиной Тьюринга, поэтому если не существует машины Тьюринга, решающей проблему остановки, то не существует и алгоритма ее решения; иными словами, проблем остановки неразрешима.

Другие неразрешимые проблемы

Тот факт, что столь просто формулируемая задача («останавливается ли данная программа?») не может быть надежно решена компьютером, несколько обескураживает. Однако эта конкретная проблема относительно абстрактна, а программа *do_the_opposite.rb*, которую мы использовали для ее иллюстрации, практически бесполезна и придумана искусственно; маловероятно, что нам когда-ни-

будь понадобится в действительности реализовывать метод `#halts?` или писать программу типа *do_the_opposite.rb* как часть реального приложения. Быть может, просто отнести неразрешимость к разряду академических курьезов и продолжать жить, как жили?

К сожалению, все не так просто, потому что проблема остановки – отнюдь не единственная неразрешимая проблема. Существует много других, с которыми мы вполне можем столкнуться в своей повседневной работе по созданию программного обеспечения, и их неразрешимость имеет вполне реальные последствия в плане практических ограничений на автоматизированные инструменты и процессы.

Рассмотрим выдуманный пример. Допустим, вам поручили разработать на Ruby программу, которая печатает строку `'hello world'`. Вроде ничего сложного, но, по глубоко укоренившейся привычке тянуть резину[1] вы заодно собираетесь разработать автоматизированный инструмент, который может надежно определить, печатает ли данная программа строку `hello world`, если подать ей определенные входные данные[2]. Вооружившись этим инструментом, мы сможем проанализировать конечную программу и убедиться, что она делает то, что нужно.

Теперь представим, что нам удалось написать метод `#prints_hello_world?`, который умеет корректно принимать вышеупомянутое решение относительно любой программы. Опуская детали реализации, приведем лишь общий вид этого метода:

```
def prints_hello_world?(program, input)
  # разобрать программу
  # проанализировать программу
  # вернуть true, если программа печатает "hello world", и false
  #    в противном случае
end
```

Закончив писать изначально порученную программу, мы сможем воспользоваться методом `#prints_hello_world?`, чтобы проверить, делает ли она то, что нужно; если делает, сохраняем ее в системе управления версиями и посылаем электронной почтой шефу – все довольны и счастливы. Но ситуация складывается даже лучше, чем

[1] Пардон, надо было сказать «ответственно и профессионально подходить к разработке ПО».

[2] И на этот раз входные данные, возможно, и несущественны, если программа ничего не читает из стандартного ввода, но мы решили включить их ради полноты и согласованности.

ожидалось, так как мы можем использовать #prints_hello_world? для реализации еще одного интересного метода:

```
def halts?(program, input)
  hello_world_program = %Q{
    program = #{program.inspect}
    input = $stdin.read
    evaluate(program, input) # вычислить программу, игнорируя выход
    print 'hello world'
  }

  prints_hello_world?(hello_world_program, input)
end
```

Синтаксическая конструкция %Q применяется так же, как %q, но в закавыченных таким образом строках производится интерполяция, то есть #{program.inspect} заменяется строковым литералом Ruby, содержащим значение program.

Принцип работы новой версии #halts? основан на построении специальной программы hello_world_program, которая делает две вещи:

1. Вычисляет program, подав на стандартный ввод input.
2. Печатает hello world.

Программа hello_world_program построена так, что ее исполнение может завершиться только двумя способами: либо evaluate(program, input) завершается успешно и тогда печатается hello world, либо evaluate(program, input) бесконечно циклится и тогда ничего не печатается.

Эта специальная программа передается методу #prints_hello_world?, чтобы выяснить, какой из двух исходов будет иметь место. Если #prints_hello_world? возвращает true, значит evaluate(program, input) в конечном итоге завершится и напечатает hello world, поэтому #halts? возвращает true, показывая, что program останавливается на входных данных input. Если же #prints_hello_world? возвращает false, значит hello_world_program никогда не дойдет до последней строки, поэтому #halts? возвращает false, сообщая, что evaluate(program, input) будет бесконечно циклиться.

Наша реализация #halts? показывает, что проблему остановки можно *свести* к проблеме проверки того, печатает ли некоторая программа строку hello world. Иными словами, любой алгоритм, который вычисляет #prints_hello_world?, можно преобразовать в алгоритм, вычисляющий #halts?.

Мы уже знаем, что корректно реализовать метод `#halts?` невозможно, а, следовательно, невозможно реализовать и метод `#prints_hello_world?`. А раз его невозможно реализовать, то согласно тезису Чёрча-Тьюринга для него не существует и алгоритма, то есть вопрос «печатает ли данная программа строку `hello world`?» также относится к классу неразрешимых проблем.

На практике никого не интересует автоматическое распознавание того, печатает ли некая программа определенную строку, но схема доказательства неразрешимости указывает на куда более масштабную и общую проблему. Нам всего-то и нужно было построить программу, которая обнаруживает свойство «печатает `hello world`» всякий раз, как некоторая другая программа останавливается, и этого оказалось достаточно для утверждения о неразрешимости. Что мешает нам повторно использовать то же самое рассуждение для *любого* свойства поведения программы, включая и те, которые нам действительно интересны?

Да ничего. Это и есть *теорема Райса*: любое нетривиальное свойство поведения программы порождает неразрешимую проблему, поскольку проблему остановки всегда можно свести к задаче об определении того, истинно ли данное свойство; если бы можно было придумать алгоритм, устанавливающий истинность данного свойства, то мы смогли бы построить другой алгоритм, который решает проблему остановки, а это невозможно.

Грубо говоря, под «нетривиальным свойством» понимается утверждение о том, *что* программа делает, а не *как* она это делает. Например, теорема Райса неприменима к чисто синтаксическому свойству вида «содержит ли исходный код программы строку `'reverse'`?», потому что это несущественная деталь реализации, которую можно убрать, не меняя видимого поведения программы. С другой стороны, семантическое свойство вида «верно ли, что эта программа обращает входные данные?» отвечает условиям теоремы Райса и потому порождает неразрешимую проблему.

Из теоремы Райса вытекает, что имеется огромное число неразрешимых проблем, в которых задается вопрос, что будет делать программа во время выполнения.

Печальные следствия

Неразрешимость – неприглядный факт. Проблема остановки вызывает досаду, поскольку показывает, что нельзя иметь все сразу: мы

хотели бы располагать неограниченной мощностью универсального языка программирования, но при этом писать программы, не опасаясь, что они увязнут в бесконечном цикле, или, по крайней мере, такие, *подпрограммы* которых гарантированно останавливаются при выполнении в составе более крупной длительной операции (см. раздел «Очень длительные вычисления» на стр. 317).

Это разочарование мрачно резюмировано в классической статье, написанной в 2004 году:

> Из-за проблемы остановки в дизайне языков программирования наблюдается раздвоение. Мы вынуждены выбирать одно из двух:
> A. Безопасность – язык, все программы на котором заведомо завершаются.
> B. Универсальность – язык, на котором можно написать
> i. все завершающиеся программы;
> ii. дурацкие программы, которые не завершаются.
> причем в общем случае для произвольно выбранной программы нельзя сказать, принадлежит она классу (i) или (ii).
>
> Пятьдесят лет назад, в начале эпохи электронных вычислений, мы выбрали вариант B.
>
> – Дэвид Тёрнер, *Total Functional Programming (http://www.jucs.org/jucs_10_7/total_functional_programming)*

Да, мы хотели бы избежать дурацких программ, но что поделаешь – не повезло. О произвольно взятой программе невозможно сказать, дурацкая она или нет, поэтому полностью избежать их, не жертвуя универсальностью, не получится[1].

Следствия из теоремы Райса также удручают: неразрешим не только вопрос «останавливается ли данная программа?», но и вопрос «делает ли данная программа то, я от нее хочу?». Мы живем в мире, где не существует способа построить машину, которая может точно предсказать, напечатает ли некая программа строку `hello world`, вычислит ли она определенную математическую функцию или обратится ли с определенным вызовом к операционной системе. Так уж устроена жизнь.

Это печально, потому что было бы очень полезно иметь возможность механически проверять свойства программы; надежность современного ПО была бы выше, если бы существовал инструмент, который определял бы, соответствует ли программа своей специфи-

---

[1] Потенциальным решением этой проблемы могут служить тотальные языки программирования, но до сих пор ни одного такого не создано, быть может, потому что они труднее для понимания, чем традиционные.

кации или содержит ли она ошибки. Для отдельных программ свойства можно проверить механически, но коль скоро в общем случае они не могут быть проверены, мы никогда не сможем до конца доверять машинам, которые работают на нас.

Допустим, к примеру, что мы заняты изобретением новой программной платформы и хотим зарабатывать на продаже совместимых с ней программ через сетевой магазин – «application superstore», если хотите, – от имени сторонних разработчиков для нашей платформы. Мы хотим, чтобы наши клиенты были уверены в своей покупке, поэтому решаем продавать только программы, отвечающие определенным критериям: они не должны аварийно завершаться, не должны обращаться к закрытым API и не должны выполнять произвольный код, загруженный из Интернета.

Но когда тысячи разработчиков начнут отправлять нам свои программы, как мы будем их анализировать на предмет соответствия нашим требованиям? Мы сэкономили бы кучу времени и денег, если бы могли воспользоваться автоматизированной системой, проверяющей каждую отправленную программу, но из-за неразрешимости построить систему, которая делает это надежно, невозможно. У нас нет иного выбора, кроме как нанять небольшую армию людей, которые вручную будут проверять программы: запускать, дизассемблировать и вставлять в операционную систему специальные средства для наблюдения за динамическим поведением.

Ручной анализ медленный, дорогой и подверженный ошибкам, к тому же на прогон каждой программы можно отвести лишь очень немного времени, так что срез ее динамического поведения будет ограничен. Даже если никто не допускает ошибок, время от времени через эту сеть что-то да проскользнет, и нам придется иметь дело с толпами разгневанных клиентов. Спасибо тебе, неразрешимость.

За всеми этими неудобствами стоят две фундаментальные проблемы. Во-первых, мы не умеем прозревать будущее и видеть, что произойдет при выполнении программы; в общем случае единственный способ узнать, что программа делает, – взять и запустить ее. Хотя некоторые программы достаточно просты и их поведение легко предсказать, на универсальном языке всегда можно написать программу, поведение которой невозможно предсказать только путем анализа исходного кода[1].

---

[1] Стивен Вольфрам придумал термин вычислительная неприводимость (*http://mathworld.wolfram.com/ComputationalIrreducibility.html*), означающий, что поведение программы невозможно предсказать, не выполнив ее.

Вторая проблема заключается в том, что решив запустить программу, мы не можем заранее сказать, сколько времени она будет работать. В общем случае единственное решение – запустить и подождать, но поскольку, как мы знаем, программы на универсальном языке могут циклиться бесконечно и никогда не останавливаться, то всегда найдутся программы, для которых ждать придется бесконечно долго.

Почему так происходит?

В этой главе мы видели, что все универсальные системы обладают достаточной мощностью для ссылки на себя же. Программы оперируют числами, числа могут представлять строки, а строками записываются инструкции программы, поэтому программа вполне способна оперировать собственным исходным кодом.

Из-за способности ссылаться на себя невозможно написать программу, которая могла бы надежно предсказывать поведение программ. Написав какую-нибудь программу анализа поведения, мы всегда сможем построить большую программу, которая ее обманет: новая программа включает анализатор в качестве подпрограммы, анализирует свой исходный код и сразу же делает противоположное тому, что, по мнению анализатора, она должна делать. Такие внутренне противоречивые программы – всего лишь курьезы, которые никто не станет писать на практике, но это симптом, а не первопричина настоящей проблемы: в общем случае, поведение программы слишком сложно, чтобы его можно было точно предсказать.

У естественных языков сравнимая мощность и аналогичные проблемы. Утверждение «данное высказывание – ложь» (*парадокс лжеца*) не может быть ни истинным, ни ложным. Но парадокс лжеца зависит от специального самоссылающегося слова «данное», а, как мы видели в разделе «Программы могут ссылаться сами на себя» на стр. 323, любую компьютерную программу можно сделать самоссылающейся *по построению*, не прибегая к специальным свойствам языка.

И раз уж зашла речь об этом, существуют две основные причины, по которым так трудно предсказать поведение программы.

1. Любая система, обладающая достаточной мощностью для ссылки на себя, не может правильно ответить на любой во-

прос о себе[1]. Мы всегда можем построить программу типа *do_the_opposite.rb*, поведение которой система предсказать не сможет. Чтобы избежать этой проблемы, мы должны выйти за пределы самоссылающейся системы и использовать другую, более мощную систему для ответа на вопросы об исходной.

2. Но в случае универсальных языков программирования более мощной системы не существует. Тезис Чёрча-Тьюринга утверждает, что любой алгоритм, который мы могли бы придумать для предсказания поведения программ, сам может быть выполнен некоторой программой, поэтому мы остаемся в рамках возможностей универсальных систем.

Жизнь в условиях невычислимости

Весь смысл написания программ состоит в том, чтобы заставить компьютер сделать нечто полезное. И как программистам жить в мире, где проверка правильности работы программы, – неразрешимая проблема?

Соблазнительно было бы просто отказаться от попыток. *Не обращаем внимания. Хорошо было бы автоматически верифицировать поведение программы, но не получается, поэтому будем надеяться на лучшее и никогда не станем утруждать себя проверкой того, что программа работает правильно.*

Но это, пожалуй, перебор, потому что ситуация не так плоха, как кажется. Теорема Райса не означает, что анализировать программы невозможно, а лишь говорит, что нельзя написать нетривиальный анализатор, который *всегда* останавливается и дает правильный ответ. Как мы видели в разделе «Построение анализатора остановки» на стр. 331, ничто не мешает написать инструмент, который дает правильный ответ для *некоторых* программ, если мы готовы смириться с тем, что всегда найдется *другая* программа, для которой этот инструмент дает неверный ответ или вообще зацикливается и ничего не возвращает.

Вот несколько практических способов анализа и предсказания поведения программ вопреки принципиальной неразрешимости.

- Задавать неразрешимые вопросы, но отступаться, если ответ не удается получить. Например, чтобы проверить, печатает ли программа определенную строку, мы можем запустить ее и по-

[1] В этом по существу заключается смысл первой теоремы Гёделя о неполноте.

дождать, а если она не напечатает строку в течение заданного времени, скажем 10 секунд, то завершаем программу и считаем, что она некорректна. Можно случайно отклонить программу, которая напечатает ожидаемую строку через 11 секунд, но во многих случаях это приемлемый риск, особенно с учетом того, что медленные программы сами по себе нежелательны.

- Задавать несколько мелких вопросов, ответы на которые, вместе взятые, дают эмпирические свидетельства в пользу того или иного ответа на более крупный вопрос. В ходе автоматизированного приемочного тестирования мы обычно не можем проверить, что программа работает правильно для всех возможных входных данных, но стремимся выполнить ее для ограниченного числа *репрезентативных* данных и посмотреть, что получается. Каждый тест дает информацию о поведении программы в каком-то частном случае, но на основе собранных сведений мы получаем большую уверенность относительно вероятного поведения программы в целом. Конечно, остается возможность, что на входных данных, которые не тестировались, поведение будет разительно отличаться, но с этим можно смириться при условии, что контрольные примеры представляют наиболее реалистичные случаи.

  Другой пример такого подхода – использование автономных тестов для проверки поведения небольших фрагментов программы по отдельности, а не верификация всей программы в целом. Изолированный автономный тест сосредоточен на проверке свойств простого блока кода и делает предположения о других частях программы, представляя их тестовыми двойниками (заглушками и подставными объектами). Отдельные автономные тесты, проверяющие небольшие участки хорошо понятного кода, можно сделать простыми и быстрыми, сведя к минимуму риск зацикливания или ложного ответа.

  Подвергнув автономному тестированию все блоки программы, мы можем выстроить цепочку допущений и следствий, напоминающую математическое доказательство: «если блок A работает, то блок B тоже работает, а если работает блок B, то работает блок C». Решение о том, насколько обоснованы все эти допущения, принимает человек, а не автоматизированная процедура верификации, хотя комплексное и приемочное тестирование могут повысить степень уверенности в правильности работы всей системы.

- Задавать разрешимые вопросы, проявляя необходимую осторожность. Предложенные выше рекомендации подразумевают фактическое выполнение программы и наблюдение за результатом, при этом неизбежно существует риск зацикливания. Однако существуют полезные вопросы, на которые можно ответить путем одного лишь статического анализа исходного кода. Самый очевидный из них – «содержит ли исходный код синтаксические ошибки?», но есть и более интересные, на которые можно ответить, если мы готовы принять безопасную аппроксимацию в тех случаях, когда получение настоящего ответа – неразрешимая проблема.

  Один из распространенных видов анализа – поиск *нерабочего кода*, где вычисляются значения, которые никогда не используются, или *недостижимого кода*, который никогда не выполняется. Не всегда можно точно сказать, является ли код действительно нерабочим или недостижимым, и тогда следует проявить осторожность и предположить, что это не так. Но бывают случаи, когда это очевидно: в некоторых языках присваивание значения локальной переменной, которая более нигде не упоминается, – заведомо нерабочий код, а предложение, следующее сразу за `return`, заведомо недостижимо[1]. Оптимизирующий компилятор, например GCC, применяет эти приемы для обнаружения и исключения ненужного кода, что делает программы короче и быстрее без изменения поведения.
- Аппроксимировать программу, преобразовав ее в более простую, а затем задавать разрешимые вопросы об аппроксимации. Эта важная идея является предметом следующей главы.

---

[1] Спецификация Java требует, чтобы компилятор отвергал программу, содержащую недостижимый код. В документе *http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.21* имеется длинное описание того, как компилятор Java должен определять, какие участки программы недостижимы, не исполняя программу.

Глава 9. Программирование в игрушечной стране

Задача программирования – сообщить машине идеи с помощью синтаксических конструкций. Разрабатывая программу, мы представляем, чего хотим от машины, которая будет ее исполнять, а знание семантики языка программирования дает уверенность в том, что машина поймет смысл каждого крохотного кусочка программы.

Но сложная компьютерная программа больше, чем сумма ее отдельных предложений и выражений. Соединив много мелких деталей для получения крупного целого, было бы полезно проверить, что полная программа действительно делает то, что мы задумывали. Возможно, мы захотим убедиться, что она всегда возвращает определенные результаты или что результатом прогона являются некоторые побочные эффекты в файловой системе либо сети или просто что программа не содержит очевидных ошибок, приводящих к краху на неожиданных входных данных.

На самом деле, у программ может быть много самых разных желательных свойств и было бы здорово, если бы наличие или отсутствие этих свойств можно было вывести из одного лишь анализа ее текста, но, как мы уже знаем, теорема Райса утверждает, что предсказание поведения программы путем изучения исходного кода не всегда дает верный ответ. Конечно, самый прямой путь выяснить, что делает программа, – запустить ее, и иногда в этом нет ничего плохого – тестирование программ сплошь и рядом выполняется посредством их прогона на известных входных данных и сравнения полученных результатов с ожидаемыми – однако существует несколько причин, по которым выполнение кода может оказаться неприемлемым решением.

Прежде всего, любая полезная программа, скорее всего, должна обрабатывать информацию, которая неизвестна до момента выпол-

нения: действия пользователя; файлы, переданные через аргументы; данные, прочитанные из сети, и т. д. Мы, конечно, можем прогнать программу с фиктивными входными данными, чтобы составить представление о ее работе, но это расскажет лишь о поведении программы на таких данных, а что произойдет на реальных? Прогонять программу на всех возможных комбинациях входных данных часто непрактично или попросту невозможно, а попытка прогнать один раз на конкретном наборе, сколь угодно реалистичном, может и не дать подробной информации о поведении программы в общем случае.

В разделе «Универсальные системы могут зацикливаться» на стр. 316 мы исследовали еще одну проблему: программы, написанные на достаточно мощных[1] языках, вполне могут работать бесконечно, не порождая никакого результата. Поэтому невозможно надежно изучить произвольную программу, запуская ее, поскольку иногда нельзя заранее сказать, будет ли программа работать неопределенно долго (см. раздел «Проблема остановки» на стр. 331), так что любой автоматический анализатор, который попытается прогнать такую программу, рискует никогда не получить ответ.

И наконец, даже если заранее известны все возможные входные данные для программы и по какой-то причине она гарантированно завершается, никогда не попадая в бесконечный цикл, все равно может оказаться, что прогонять ее и смотреть, что получается, слишком накладно или неудобно. Возможно, она работает слишком долго или имеет необратимые побочные эффекты – отправляет почту, переводит деньги, запускает ракету – нежелательные при тестировании.

По указанным причинам было бы полезно умение получать информацию о программе, не выполняя ее. Один из таких способов заключается в использовании *абстрактной интерпретации* – технике анализа, при которой мы выполняем упрощенную версию программы и используем результаты, чтобы сделать выводы о свойствах исходной программы.

Абстрактная интерпретация

Абстрактная интерпретация дает способ подойти к задаче, которая по каким-то причинам слишком трудна – очень велика, очень сложна или имеет так много неизвестных, что напрямую к ней не

[1] Здесь «достаточно мощный» означает «универсальный».

подступиться. Основная идея абстрактной интерпретации – использовать *абстракцию*, модель реальной задачи, в которой какие-то детали опущены ради того, чтобы сделать ее поддающейся решению – меньше, проще, с меньшим числом неизвестных – но при этом все же оставлено столько деталей, чтобы решение упрощенной задачи что-то говорило об исходной.

Чтобы конкретизировать эту туманную идею, рассмотрим простое применение абстрактной интерпретации.

Планирование маршрута

Представьте, что вы – турист в незнакомой стране и хотите спланировать поездку на машине в другой город. Как решить, по какому маршруту ехать? Прямолинейное решение – сесть в арендованную машину и двигаться в направлении, которое кажется самым многообещающим. В зависимости от вашей удачливости и информативности чужих дорожных знаков, такое исследование незнакомой дорожной сети методом грубой силы в конце концов все же может привести вас в пункт назначения. Но эта дорогостоящая стратегия, и с большой вероятностью вы так и будете блуждать, пока окончательно не сдадитесь.

Куда разумнее воспользоваться картой. Напечатанный дорожный атлас – это абстракция, которая приносит в жертву многочисленные детали физической дорожной сети. Он ничего не говорит об интенсивности движения, о том, какие дороги сейчас закрыты, где находятся отдельные строения да и вообще ничего о третьем измерении. Атлас – и это очень важно – двумерен и гораздо меньше реальной местности. Но на карте сохранена самая важная информация, необходимая для планирования поездки, – относительное расположение всех населенных пунктов, дороги, ведущие в каждый населенный пункт, и связи между дорогами.

Несмотря на опущенные детали, точная карта полезна, потому что проложенный по ней маршрут, вероятно, окажется пригодным и в реальности, а не только в абстрактном мире карты. Картограф проделал огромную работу по созданию модели реальности, и теперь у вас есть возможность проводить вычисления на этой модели: планировать свой маршрут, глядя на упрощенное представление дорожной сети. Затем результат вычисления можно будет перенести в реальный мир, когда вы сядете в машину и отправитесь в путь, – дешево обошедшиеся решения, принятые в абстрактном мире карты, позволят избежать дорогостоящих ошибок на физической дороге.

Аппроксимации, примененные в карте, существенно упрощают навигационные вычисления, не делая результаты фатально не соответствующими действительности. Существует немало ситуаций, когда решения, принятые с помощью карты, оказываются неверными – не гарантируется, что карта говорит абсолютно *все*, что нужно знать о поездке, – но заблаговременное планирование маршрута позволяет избежать определенных ошибок и существенно упростить задачу перемещения из одного места в другое.

Абстракция: умножение знаков

Планирование маршрута с помощью бумажной карты – это реальное приложение абстрактной интерпретации, но уж очень неформальное. В качестве более формального примера рассмотрим умножение чисел; и хотя это всего лишь игрушечный пример, он даст нам возможность приступить к написанию кода для исследования рассматриваемых идей.

Представьте на минутку, что умножение двух чисел – трудная или дорогая операция, а мы хотим получить некую информацию о ее результате, не выполняя умножение в действительности. Точнее, нас интересует *знак* результата: является ли результат умножения положительным числом, отрицательным или нулем?

Гипотетически накладный способ узнать это – вычислить произведение в *конкретном* мире, используя *стандартную интерпретацию* умножения: перемножить числа, посмотреть на получившееся число и решить, какое оно: положительное, отрицательное или нуль. Например, в Ruby это выглядит так:

```
>> 6 * -9
=> -54
```

`-54` – отрицательное число, следовательно, мы узнали, что произведение `6` и `-9` отрицательно. Дело сделано.

Однако ту же информацию можно получить путем вычисления в *абстрактном* мире, используя *абстрактную интерпретацию* умножения. Подобно линиям на плоской карте, изображающим реальные дороги, мы можем использовать абстрактные значения для представления чисел; мы можем спланировать маршрут по карте, а не искать его методом проб и ошибок на местности – и точно так же можно определить операцию абстрактного умножения абстрактных значений, вместо того чтобы конкретно перемножать конкретные числа.

Для этого следует выбрать абстрактные значения, так чтобы, с одной стороны, упростить вычисление, а, с другой, сохранить достаточно информации, чтобы полученный ответ был полезен. Можно воспользоваться тем фактом, что знак произведения не зависит от абсолютных величин сомножителей[1]:

```
>> (6 * -9) < 0
=> true
>> (1000 * -5) < 0
=> true
>> (1 * -1) < 0
=> true
```

Еще в начальной школе нас учили, что важны только знаки сомножителей: произведение двух положительных или двух отрицательных чисел положительно, произведение положительного и отрицательного числа отрицательно, произведение любого числа и нуля равно нулю.

Поэтому в качестве абстрактных значений выберем три вида чисел: «negative», «zero» и «positive». В Ruby для этого нужно определить класс `Sign` и создать три его экземпляра:

```
class Sign < Struct.new(:name)
  NEGATIVE, ZERO, POSITIVE = [:negative, :zero, :positive].map { |name| new(name) }

  def inspect
    "#<Sign #{name}>"
  end
end
```

Так мы получаем объекты Ruby, которые можно использовать как абстрактные значения: `Sign::NEGATIVE` представляет «любое отрицательное число», `Sign::ZERO` – «число нуль», а `Sign::POSITIVE` – «любое положительное число». Эти три объекта класса `Sign` и составляют крохотный абстрактный мирок, в котором мы будем производить вычисления, памятуя о том, что конкретный мир состоит из практически неограниченного множества целых чисел, представимых в Ruby[2].

В определении операции абстрактного умножения значений типа `Sign` используется только информация о знаках:

---

[1] Абсолютная величина числа — это то, что остается после отбрасывания знака. К примеру, абсолютная величина –10 равна 10.

[2] Объект `Bignum` в Ruby позволяет представить целое число произвольного размера, ограниченное лишь объемом доступной памяти.

```
class Sign
  def *(other_sign)
    if [self, other_sign].include?(ZERO)
      ZERO
    elsif self == other_sign
      POSITIVE
    else
      NEGATIVE
    end
  end
end
```

Теперь экземпляры `Sign` можно «перемножать», как числа, а наша реализация метода `Sign#*` дает ответы, не противоречащие определению умножения настоящих чисел:

```
>> Sign::POSITIVE * Sign::POSITIVE
=> #<Sign positive>
>> Sign::NEGATIVE * Sign::ZERO
=> #<Sign zero>
>> Sign::POSITIVE * Sign::NEGATIVE
=> #<Sign negative>
```

Например, в последней строке мы спрашиваем, что получится при умножении произвольного положительного числа на произвольное отрицательное. И получаем ответ: отрицательное число. Это по-прежнему умножение, но оно гораздо проще привычного и работает лишь для «чисел», утративших почти всю определяющую их информацию. Если мы все еще настаиваем на том, что настоящее умножение – дорогая операция, то эту урезанную версию можно, наверное, считать дешевой.

Подготовив абстрактный мир чисел и абстрактную интерпретацию их умножения, мы можем взглянуть на исходную задачу под другим углом. Вместо того чтобы непосредственно перемножать два числа, чтобы узнать их знак, мы преобразуем числа в их абстрактные представления и перемножим последние. Но сначала нужно научиться преобразовывать конкретные числа в абстрактные:

```
class Numeric
  def sign
    if self < 0
      Sign::NEGATIVE
    elsif zero?
      Sign::ZERO
    else
      Sign::POSITIVE
    end
  end
end
```

Вот теперь можно преобразовать два числа и выполнить умножение в абстрактном мире:

```
>> 6.sign
=> #<Sign positive>
>> -9.sign
=> #<Sign negative>
>> 6.sign * -9.sign
=> #<Sign negative>
```

Мы снова получили, что `6 * -9` – отрицательное число, но на этот раз обошлись без перемножения реальных чисел. Переход в абстрактный мир дает новый способ выполнения вычислений, и, что особенно важно, абстрактный результат можно разумно интерпретировать в реальном мире, хотя ответ получается лишь приблизительным, потому что, абстрагируясь от реальности, мы принесли в жертву кое-какие детали. В данном случае абстрактный результат `Sign::NEGATIVE` говорит, что произведение `6 * -9` может быть равно любому из конкретных чисел `-1`, `-2`, `-3` и т. д., но заведомо не `0` и не положительному числу, например, `1` или `500`.

Отметим, что поскольку в Ruby значениями являются объекты – структуры данных, несущие в себе определения операций, – одно и то же Ruby-выражение можно использовать как для конкретного, так и для абстрактного вычисления в зависимости от того, являются ли его аргументами конкретные (`Fixnum`) или абстрактные (`Sign`) объекты. Возьмем, к примеру, метод `#calculate`, который определенным образом перемножает три числа:

```
def calculate(x, y, z)
  (x * y) * (x * z)
end
```

Если вызывать `#calculate` с объектами типа `Fixnum`, то в вычислениях будет участвовать метод `Fixnum#*`, и мы получим конкретный результат типа `Fixnum`. Если же вызвать его с экземплярами класса `Sign`, то будет использована операция `Sign#*` и получится результат типа `Sign`.

```
>> calculate(3, -5, 0)
=> 0
>> calculate(Sign::POSITIVE, Sign::NEGATIVE, Sign::ZERO)
=> #<Sign zero>
```

Это дает нам ограниченную возможность выполнять абстрактную интерпретацию, не меняя основной код Ruby-программы, а лишь заменяя конкретные аргументы их абстрактными аналогами.

Эта техника напоминает использование *тестовых двойников* (например, *заглушек* и *подставных объектов*) в автоматизированном автономном тестировании. Тестовые двойники – это специальные замещающие объекты, которые внедряются в код для управления и проверки его поведения. Они полезны в любой ситуации, когда применять в качестве тестовых данных реальные объекты неудобно или слишком накладно.

Аппроксимация и безопасность: сложение знаков

До сих пор мы видели, что вычисления в абстрактном мире дают менее точные результаты, чем в конкретном, поскольку при абстрагировании отбрасываются некоторые детали: маршрут, спланированный на карте, подскажет, где повернуть, но ничего не скажет о том, по какой полосе ехать, а умножение объектов `Sign` дает информацию о том, по какую сторону от нуля находится результат, но не о его реальном значении.

Часто с неточностью результата вполне можно смириться, но если мы хотим, чтобы абстракция была полезной, такая неточность должна быть *безопасной*. Это означает, что абстракция должна быть правдивой: результат абстрактного вычисления не должен противоречить вычислению конкретному. Если это не так, то сообщаемая абстракцией информация ненадежна, а это, пожалуй, даже хуже чем бесполезна.

Наша абстракция `Sign` безопасна, потому что перемножение объектов `Sign`, полученных преобразованием двух чисел, дает тот же результат, что перемножение исходных чисел с последующим преобразованием в `Sign`:

```
>> (6 * -9).sign == (6.sign * -9.sign)
=> true
>> (100 * 0).sign == (100.sign * 0.sign)
=> true
>> calculate(1, -2, -3).sign == calculate(1.sign, -2.sign, -3.sign)
=> true
```

В этом смысле абстракция `Sign` абсолютно точна. Она оставляет ровно столько информации, сколько требуется, и не теряет ее при вычислениях. Но если абстракция не так хорошо согласуется с вычислением, то вопрос о безопасности возникает во всей остроте, что можно наблюдать на примере абстрактного сложения.

Существуют *некоторые* правила, позволяющие определить знак суммы по знакам слагаемых, но они работают не для всех комбинаций знаков. Мы знаем, что сумма положительных чисел положительна, а сумма отрицательных – отрицательна, но как быть, когда одно слагаемое положительно, а другое отрицательно? В этом случае знак суммы зависит от абсолютных величин слагаемых: если положительное число по абсолютной величине больше отрицательного, то сумма будет положительна (–20 + 30 = 10); если же по абсолютной величине больше отрицательное слагаемое, то сумма будет отрицательна (–30 + 20 = –10); наконец, если оба слагаемых равны по абсолютной величине, то сумма будет равна нулю. Но ведь абсолютная величина – это именно то, что в нашей абстракции отброшено, поэтому принять такое решение в абстрактном мире невозможно.

Проблема возникала из-за того, что наша абстракция *слишком* абстрактна для вычисления суммы во всех возможных случаях. Как ее решить? Можно было бы состряпать определение абстрактного сложения, так чтобы оно возвращало хоть какой-нибудь результат – скажем, `Sign::ZERO` – если не знает правильного ответа, но это будет небезопасно, потому что тогда абстрактное вычисление может дать ответ, грубо противоречащий тому, что был бы получен при конкретном вычислении.

Правильное решение – расширить абстракцию с учетом этой неопределенности. В дополнение к значениям `Sign`, означающим «любое положительное число» и «любое отрицательное число», мы можем ввести еще одно, интерпретируемое как «любое число». По сути дела, это единственный честный ответ, который можно дать, когда для правильного ответа не хватает информации: результат может быть отрицательным, нулевым или положительным, никаких гарантий мы не даем. Назовем это новое значение `Sign::UNKNOWN`:

```
class Sign
  UNKNOWN = new(:unknown)
end
```

Вот теперь у нас есть все необходимое для безопасной реализации абстрактного сложения. Правила вычисления знака суммы двух чисел `x` и `y` формулируются следующим образом:

- если знаки `x` и `y` одинаковы (оба положительны, оба отрицательны или оба равны нулю), то знак суммы будет таким же;

- если x равно нулю, то суммы имеет такой же знак, как y, и наоборот;
- в противном случае знак суммы неизвестен.

Эти правила нетрудно реализовать в виде метода `Sign#+`:

```
class Sign
  def +(other_sign)
    if self == other_sign || other_sign == ZERO
      self
    elsif self == ZERO
      other_sign
    else
      UNKNOWN
    end
  end
end
```

В результате получаем требуемое поведение:

```
>> Sign::POSITIVE + Sign::POSITIVE
=> #<Sign positive>
>> Sign::NEGATIVE + Sign::ZERO
=> #<Sign negative>
>> Sign::NEGATIVE + Sign::POSITIVE
=> #<Sign unknown>
```

Оказывается даже, что эта реализация делает то, что нужно, и тогда, когда знак одного из операндов неизвестен:

```
>> Sign::POSITIVE + Sign::UNKNOWN
=> #<Sign unknown>
>> Sign::UNKNOWN + Sign::ZERO
=> #<Sign unknown>
>> Sign::POSITIVE + Sign::NEGATIVE + Sign::NEGATIVE
=> #<Sign unknown>
```

Однако мы должны поправить реализацию `Sign#*`, чтобы она корректно обрабатывала случай `Sign::UNKNOWN`:

```
class Sign
  def *(other_sign)
    if [self, other_sign].include?(ZERO)
      ZERO
    elsif [self, other_sign].include?(UNKNOWN)
      UNKNOWN
    elsif self == other_sign
      POSITIVE
    else
      NEGATIVE
    end
  end
end
```

Теперь у нас есть две абстрактные операции. Отметим, что значение `Sign::UNKNOWN` не стопроцентно «заразно»: если неизвестное число умножить на нуль, то получится нуль, так что прокравшаяся в вычисление неопределенность, может исчезнуть к моменту его завершения:

```
>> (Sign::POSITIVE + Sign::NEGATIVE) * Sign::ZERO + Sign::POSITIVE
=> #<Sign positive>
```

Еще нам следует скорректировать свое представление о правильности с учетом неточности, которую вносит `Sign::UNKNOWN`. Поскольку у абстракции иногда не хватает информации для точного ответа, может случиться так, что результаты абстрактного и конкретного вычисления расходятся:

```
>> (10 + 3).sign == (10.sign + 3.sign)
=> true
>> (-5 + 0).sign == (-5.sign + 0.sign)
=> true
>> (6 + -9).sign == (6.sign + -9.sign)
=> false
>> (6 + -9).sign
=> #<Sign negative>
>> 6.sign + -9.sign
=> #<Sign unknown>
```

Что здесь происходит? Можно ли считать нашу абстракцию по-прежнему безопасной? Да, можно, потому что в тех случаях, когда абстракция теряет точность и возвращает `Sign::UNKNOWN`, она все-таки не врет: действительно, «результат может быть отрицательным, нулевым или положительным». Этот ответ не так полезен, как полученный в результате конкретного вычисления, но он все же не *ошибочен* и настолько хорош, насколько возможно без включения дополнительной информации в абстрактные значения и соответственно усложнения вычислений.

Мы можем отразить это в коде, изменив способ сравнения объектов `Sign`, поскольку метод `#==` является слишком категоричным. А хотим мы знать следующее: *согласуется* ли результат конкретного вычисления с прогнозом, который дало абстрактное? Если абстрактное вычисление говорит, что возможно несколько разных результатов, то верно ли, что результат конкретного вычисления совпадает с одним из них, или он не имеет с ними ничего общего?

Определим для объектов `Sign` операцию, которая говорит, как два абстрактных значения соотносятся между собой. Назовем этот ме-

тод #<=, так как он примерно соответственно семантике «совпадает с одним из», или «попадает в диапазон возможных значений»:

```
class Sign
  def <=(other_sign)
    self == other_sign || other_sign == UNKNOWN
  end
end
```

Это позволяет выполнить интересующую нас проверку:

```
>> Sign::POSITIVE <= Sign::POSITIVE
=> true
>> Sign::POSITIVE <= Sign::UNKNOWN
=> true
>> Sign::POSITIVE <= Sign::NEGATIVE
=> false
```

Теперь можно контролировать безопасность, проверяя, согласуется ли результат конкретного вычисления с прогнозом абстрактного:

```
>> (6 * -9).sign <= (6.sign * -9.sign)
=> true
>> (-5 + 0).sign <= (-5.sign + 0.sign)
=> true
>> (6 + -9).sign <= (6.sign + -9.sign)
=> true
```

Это свойство безопасности имеет силу для вычислений, включающих сложение и умножение, потому что абстракция спроектирована так, что при невозможности дать точный ответ переходит к безопасной аппроксимации.

Кстати говоря, наличие такой абстракции позволяет проводить простой анализ Ruby-кода, в котором складываются и умножаются числа. Рассмотрим, например, метод, который вычисляет сумму квадратов своих аргументов:

```
def sum_of_squares(x, y)
  (x * x) + (y * y)
end
```

Если мы хотим подвергнуть этот метод автоматическому анализу, чтобы получить информацию о его поведении, то можем поступить двумя способами: рассматривать его как черный ящик и прогонять со всеми возможными аргументами, что потребует бесконечного времени, или изучить исходный код и с помощью математических

рассуждений попытаться вывести какие-то его свойства – а это довольно сложно (и в общем случае обречено на неудачу в силу теоремы Райса). Абстрактная интерпретация дает третий вариант: вызвать метод с абстрактными значениями и посмотреть, какой результат дает абстрактная версия вычислений. При этом можно проверить все возможные входные значения, поскольку количество их комбинаций невелико.

Каждый аргумент x и y может быть отрицательным числом, нулем или положительным числом. Посмотрим, какие при этом возможны результаты:

```
>> inputs = Sign::NEGATIVE, Sign::ZERO, Sign::POSITIVE
=> [#<Sign negative>, #<Sign zero>, #<Sign positive>]
>> outputs = inputs.product(inputs).map { |x, y| sum_of_squares(x, y) }
=> [
       #<Sign positive>, #<Sign positive>, #<Sign positive>,
       #<Sign positive>, #<Sign zero>, #<Sign positive>,
       #<Sign positive>, #<Sign positive>, #<Sign positive>
     ]
>> outputs.uniq
=> [#<Sign positive>, #<Sign zero>]
```

Безо всякого искусного анализа мы узнали, что метод `#sum_of_squares` может возвращать только положительные числа или нуль, но никак не отрицательные, – тривиальное свойство, очевидное любому образованному человеку, ознакомившемуся с исходным кодом, но не очевидное машине. Конечно, такие приемы работают только для очень простого кода, но даже оставаясь игрушкой, этот пример показывает, как абстрагирование помогает подступиться к трудной задаче.

Статическая семантика

До сих пор мы рассматривали искусственные примеры того, как получать приблизительную информацию о вычислениях, не выполняя их. Реальное выполнение помогло бы узнать больше, но и приблизительная информация лучше, чем ничего, а в некоторых приложениях (например, при планировании маршрута) ее и вовсе достаточно.

В примерах с умножением и сложением мы смогли преобразовать небольшую программу в более простую и абстрактную версию, просто подав ей на вход абстрактные значения вместо конкретных чисел, но к исследованию более сложных и запутанных программ

эта техника применима с серьезными ограничениями. Легко создать объекты, которые предоставляют собственные реализации умножения и сложения, но Ruby не позволит им контролировать свое поведение более точно – например, когда они используются в предложении `if`, – потому что существуют зашитые в язык правила[1] работы синтаксических конструкций. К тому же, все равно остается проблема, связанная с тем, что в общем случае невозможно получить информацию о программе путем ее запуска и ожидания результата, так как некоторые программы циклятся бесконечно и ничего не возвращают.

У примеров с умножением и сложением есть еще один недостаток – они не очень интересны: никого не интересует, возвращает программа положительное или отрицательное число. На практике задаются другие вопросы: «завершится ли моя программа аварийно?» или «можно ли преобразовать мою программу, сделав ее более эффективной?».

На эти, более интересные, вопросы о программе можно ответить, рассмотрев ее *статическую семантику*. В главе 2 мы говорили о *динамической семантике* языков программирования, то есть о том, как определить смысл кода во время выполнения; статическая семантика сообщает о свойствах программы, которые можно исследовать, не выполняя ее. Классический пример статической семантики – *система типов*: набор правил, с помощью которых можно проанализировать программу на предмет наличия ошибок определенного вида. В разделе «Корректность» на стр. 60 мы рассматривали программы на языке SIMPLE вида «`x = true; x = x + 1`», которые, будучи синтаксически корректными, при выполнении приводят к проблемам, связанным с динамической семантикой. Система типов могла бы обнаружить такие ошибки заранее и автоматически отвергать некорректные программы, даже не позволяя их запустить.

Абстрактная интерпретация дает способ рассуждать о статической семантике программы. Программы предназначены для выполнения, поэтому наша стандартная интерпретация смысла программы – та, что диктуется ее динамической семантикой: программа «`x = 1 + 2; y = x * 3`» манипулирует числами, выполняя над ними арифметические операции и сохраняя где-то в памяти. Но если у нас имеется альтернативная, более абстрактная семантика языка, то мы можем «выполнить» ту же программу по другим правилам и получить бо-

---

[1] В отличие, скажем, от Smalltalk.

лее абстрактные результаты, сообщающие частичную информацию о том, что произойдет при нормальной интерпретации.

Реализация

Конкретизируем эти идеи, построив систему типов для языка SIMPLE из главы 2. На первый взгляд, это будет выглядеть как операционная семантика крупных шагов из раздела «Семантика крупных шагов» на стр. 62: мы реализуем некий метод в каждом из классов, представляющих синтаксис SIMPLE-программ (`Number`, `Add` и т. д.), и вызов этого метода будет возвращать конечный результат. В динамической семантике соответствующий метод называется `#evaluate`, а его результатом является либо полностью вычисленное значение SIMPLE, либо окружение, ассоциирующее имена со значениями SIMPLE, – в зависимости от того, что вычисляется: выражение или предложение.

```
>> expression = Add.new(Variable.new(:x), Number.new(1))
=> «x + 1»
>> expression.evaluate({ x: Number.new(2) })
=> «3»
>> statement = Assign.new(:y, Number.new(3))
=> «y = 3»
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«1», :y=>«3»}
```

Для нашей статической семантики мы реализуем другой метод, который делает меньше работы и возвращает более абстрактный результат. Вместо конкретных значений и окружений мы будем оперировать абстрактными значениями – *типами*. Тип представляет множество возможных значений: результатом вычисления выражения языка SIMPLE может быть число или булево значение, поэтому для выражений есть два типа: «любое число» и «любое булево значение». Эти типы аналогичны значениям `Sign`, с которыми мы встречались выше, в особенности значению `Sign::UNKNOWN`, которое по существу означает «любое число». Как и в случае `Sign`, мы можем реализовать типы, определив класс `Type` и создав несколько его экземпляров:

```
class Type < Struct.new(:name)
  NUMBER, BOOLEAN = [:number, :boolean].map { |name| new(name) }

  def inspect
    "#<Type #{name}>"
  end
end
```

Новый метод будет возвращать тип, поэтому назовем его `#type`. Предполагается, что он должен отвечать на вопрос: «Каков тип значения, возвращаемого в результате вычисления синтаксической конструкции SIMPLE?». В синтаксических классах `Number` и `Boolean` реализовать его очень просто: числа и булевы значения возвращают сами себя, так что мы точно, значение какого типа получим.

```
class Number
  def type
    Type::NUMBER
  end
end

class Boolean
  def type
    Type::BOOLEAN
  end
end
```

Для операций `Add`, `Multiply` и `LessThan` дело обстоит чуть сложнее. Например, мы знаем, что вычисление `Add` возвращает число, но знаем и то, что вычисление завершается успешно, только если оба аргумента `Add` являются числами, в противном случае интерпретатор SIMPLE завершится с ошибкой:

```
>> Add.new(Number.new(1), Number.new(2)).evaluate({})
=> «3»
>> Add.new(Number.new(1), Boolean.new(true)).evaluate({})
TypeError: true can't be coerced into Fixnum
```

Как узнать, является ли значением аргумента число? А об этом как раз и говорит его тип. Поэтому в случае `Add` правило звучит примерно так: если оба аргумента имеют тип `Type::NUMBER`, то и результат имеет тип `Type::NUMBER`, в противном случае у результата вообще нет типа, потому вычисление выражения, в котором делается попытка сложить нечисловые значения, аварийно завершится еще до возврата результата. Простоты ради разрешим методу `#type` возвращать `nil` для индикации такой ошибки, хотя в других случаях, возможно, стоило бы возбуждать исключение или возвращать какое-то специальное значение, обозначающее ошибку (например, `Type::ERROR`), если это окажется удобнее.

Код в классе `Add` принимает вид:

```
class Add
  def type
    if left.type == Type::NUMBER && right.type == Type::NUMBER
```

```
      Type::NUMBER
    end
  end
end
```

Реализация метода `Multiply#type` ничем не отличается, а метод `LessThan#type` очень похож, только возвращает не `Type::NUMBER`, а `Type::BOOLEAN`:

```
class LessThan
  def type
    if left.type == Type::NUMBER && right.type == Type::NUMBER
      Type::BOOLEAN
    end
  end
end
```

В оболочке можно убедиться, что этого достаточно, чтобы отличить выражения, вычисляемые успешно, от тех, что не могут быть вычислены, пусть даже синтаксис SIMPLE допускает те и другие.

```
>> Add.new(Number.new(1), Number.new(2)).type
=> #<Type number>
>> Add.new(Number.new(1), Boolean.new(true)).type
=> nil
>> LessThan.new(Number.new(1), Number.new(2)).type
=> #<Type boolean>
>> LessThan.new(Number.new(1), Boolean.new(true)).type
=> nil
```

Мы предполагаем, что абстрактное синтаксическое дерево как минимум *синтаксически* правильно. Фактические значения, хранящиеся в листьях дерева, статической семантикой игнорируются, поэтому метод `#type` может некорректно предсказывать поведение при вычислении неправильно сформированных выражений:

```
>> bad_expression = Add.new(Number.new(true), Number.new(1)) ❶
=> «true + 1»
>> bad_expression.type
=> #<Type number> ❷
>> bad_expression.evaluate({})
NoMethodError: undefined method `+' for true:TrueClass ❸
```

❶ Структура верхнего уровня этого АСД правильна (`Add` содержит два объекта `Number`), но первый объект `Number` сформирован некорректно, так как его атрибут `value` равен `true`, а должен быть значением типа `Fixnum`.

❷ В статической семантике предполагается, что сложение двух объектов `Number` всегда дает объект `Number`, поэтому метод `#type` говорит, что вычисление завершится успешно...

❸ ...однако на самом деле при попытке вычислить это выражение возникает исключение, потому что Ruby пытался прибавить `1` к `true`.

Синтаксический анализатор SIMPLE никогда не должен порождать неправильно сформированные АСД, поэтому на практике такая проблема маловероятна.

Это более общий вариант приема, использованного ранее в примере со сложением, умножением и классом `Sign`. Хотя мы ничего не складываем и не сравниваем, статическая семантика дает альтернативный способ «выполнить» программу и получить при этом полезный результат.

Вместо того чтобы рассматривать выражение «`1 + 2`» как программу, имеющую дело со *значениями*, мы отбрасываем некоторые детали и считаем, что эта программа имеет дело с *типами*, а статическая семантика предоставляет альтернативные интерпретации «`1`», «`2`» и «`+`», которые позволяют выполнить такую программу и посмотреть на результат. Результат при этом получается менее конкретный – более абстрактный – чем в случае нормального выполнения программы в соответствии с динамической семантикой, но он тем не менее полезен, потому что его можно сформулировать в виде, имеющем смысл в конкретном мире: `Type::NUMBER` означает «вызов `#evaluate` для этого выражения вернет `Number`», и `nil` – «вызов `#evaluate` может привести к ошибке».

У нас уже есть почти полная статическая семантики выражений SIMPLE, но мы еще не рассмотрели переменные. Что должен возвращать метод `Variable#type`? Зависит от значения, хранящегося в переменной: в программе «`x = 5; y = x + 1`» переменная `y` имеет тип `Type::NUMBER`, а в программе «`x = 5; y = x < 1`» – тип `Type::BOOLEAN`. Как это обработать?

В разделе «Семантика мелких шагов» на стр. 37 мы видели, что динамическая семантика класса `Variable` подразумевает использование хеша окружения для сопоставления именам переменных их значений, и в статической семантике нужно сделать нечто подобное: сопоставить именам переменных *типы*. Можно было бы назвать такой хеш «окружением типов», но, чтобы не путать два окружения, будем использовать названия *контекст типов*. Если передать кон-

текст типов методу `Variable#type`, тому останется лишь поискать в этом контексте переменную:

```
class Variable
  def type(context)
    context[name]
  end
end
```

А откуда берется контекст типов? Пока предположим, что предоставляется каким-то внешним механизмом по мере надобности. Например, каждая программа на Simple могла бы сопровождаться заголовочным файлом, в котором объявлены типы всех используемых переменных. На выполнение программы этот файл никак не влияет, но с его помощью можно автоматически проверять статическую семантику на этапе разработки.

Раз метод `#type` теперь ожидает аргумент `context`, то нужно вернуться назад и соответственно подправить другие реализации `#type`:

```
class Number
  def type(context)
    Type::NUMBER
  end
end

class Boolean
  def type(context)
    Type::BOOLEAN
  end
end

class Add
  def type(context)
    if left.type(context) == Type::NUMBER && right.type(context) == Type::NUMBER
      Type::NUMBER
    end
  end
end

class LessThan
  def type(context)
    if left.type(context) == Type::NUMBER && right.type(context) == Type::NUMBER
      Type::BOOLEAN
    end
  end
end
```

Это позволяет задавать вопросы о типе выражений, включающих переменные, коль скоро мы предоставляем контекст, в котором им сопоставлены правильные типы:

```
>> expression = Add.new(Variable.new(:x), Variable.new(:y))
=> «x + y»
>> expression.type({})
=> nil
>> expression.type({ x: Type::NUMBER, y: Type::NUMBER })
=> #<Type number>
>> expression.type({ x: Type::NUMBER, y: Type::BOOLEAN })
=> nil
```

Таким образом, мы имеем реализации метода `#type` для всех видов синтаксиса выражений, но как быть с предложениями? Вычисление предложения языка SIMPLE возвращает окружение, а не значение, и как это выразить в статической семантике?

Проще всего рассматривать предложения как своего рода пассивные выражения: предполагать, что они не возвращают никакого значения (так оно и есть) и игнорировать воздействие на окружение. Мы можем завести новый тип, означающий «не возвращает значение» и ассоциировать его с любым предложением при условии, что все его составные части имеют правильные типы. Назовем этот тип `Type::VOID`:

```
class Type
  VOID = new(:void)
end
```

Реализации метода `#type` в классах `DoNothing` и `Sequence` простые. Вычисление `DoNothing` всегда завершается успешно, а вычисление `Sequence` завершается успешно, если все в порядке с входящими в него предложениями:

```
class DoNothing
  def type(context)
    Type::VOID
  end
end

class Sequence
  def type(context)
    if first.type(context) == Type::VOID && second.type(context) == Type::VOID
      Type::VOID
    end
  end
end
```

`If` и `While` несколько требовательнее. Оба содержат выражение, играющее роль условия, и, чтобы программа работала правильно, результатом вычисления этого выражения должно быть булево значение:

```
class If
  def type(context)
    if condition.type(context) == Type::BOOLEAN &&
        consequence.type(context) == Type::VOID &&
        alternative.type(context) == Type::VOID
      Type::VOID
    end
  end
end

class While
  def type(context)
    if condition.type(context) == Type::BOOLEAN && body.type(context) == Type::VOID
      Type::VOID
    end
  end
end
```

Это позволяет отличить предложения, которые приведут к ошибке во время вычисления, от тех, что будут вычислены нормально:

```
>> If.new(
       LessThan.new(Number.new(1), Number.new(2)), DoNothing.new, DoNothing.new
     ).type({})
=> #<Type void>
>> If.new(
       Add.new(Number.new(1), Number.new(2)), DoNothing.new, DoNothing.new
     ).type({})
=> nil
>> While.new(Variable.new(:x), DoNothing.new).type({ x: Type::BOOLEAN })
=> #<Type void>
>> While.new(Variable.new(:x), DoNothing.new).type({ x: Type::NUMBER })
=> nil
```

Здесь `Type::VOID` и `nil` имеют разный смысл. Если `#type` возвращает `Type::VOID`, значит «этот код правилен, но намеренно не возвращает никакого значения», а если `nil`, то «этот код содержит ошибку».

Осталось только реализовать метод `Assign#type`. Мы знаем, что он должен возвращать `Type::VOID`, но при каких обстоятельствах? Как решить, допустимо присваивание или нет? Мы хотим убедиться, что выражение в правой части присваивания имеет смысл в рамках статической семантики, но важен ли нам его тип?

Эти вопросы заставляют задуматься над тем, что считать допустимой SIMPLE-программой. Например, допустима ли программа «`x = 1; y = 2; x = x < y`»? С точки зрения динамической семантики с ней все в порядке – при ее выполнении ничего страшного не про-

изойдет – но, возможно, нам не хочется (или, наоборот, хочется!), чтобы тип переменной мог изменяться во время выполнения. Одним программистам такая гибкость представляется достоинством, а другим – источником случайных ошибок.

С точки зрения человека, проектирующего статическую семантику, язык, в котором переменные могут менять тип, менее удобен. Напомним, что пока мы предполагаем, что контекст типов поступает из какого-то внешнего источника и остается неизменным в течение всего времени работы программы, но можно было бы выбрать более изощренный подход, при котором в начальный момент контекст пуст и заполняется по мере объявления переменных или присваивания им значений – точно так же, как в динамической семантике постепенно строится окружение. Однако это существенно усложняет дело: если предложения могут модифицировать контекст типов, то метод `#type` должен возвращать не только тип, но и контекст, – точно так же, как в динамической семантике метод `#reduce` возвращает свернутую программу и окружение, чтобы предыдущее предложение могло передать измененный контекст последующему. Кроме того, надо как-то обрабатывать ситуации вида «`if (b) { x = 1 } else { y = 2 }`», когда на разных путях выполнения порождаются разные контексты, а также вида «`if (b) { x = 1 } else { x = true }`», когда разные контексты явно противоречат друг другу[1].

Существует фундаментальное противоречие между ограничительностью системы типов и выразительностью программ. Ограничительная система типов – вещь хорошая, поскольку дает строгие гарантии, исключающие целые классы возможных ошибок, но одновременно и плохая, потому что не позволяет писать программы так, как нам хотелось бы. Хорошая система типов ищет приемлемый компромисс между ограничительностью и выразительностью, стремясь исключить как можно больше ошибок, но не вставать на пути программиста, и при этом остается достаточно простой для понимания.

Мы разрешим это противоречие, оставшись верны упрощающей идее о том, что контекст предоставляется каким-то внешним по отношению к программе механизмом и не обновляется отдельными предложениями. Этим мы исключаем некоторые программы и опре-

---

[1] Проще всего было бы постулировать, что система типов отвергает предложение, если на разных путях его выполнения порождаются разные контексты.

деленно уходим от ответа на вопрос, как и где берет начало контекст, зато сохраняем статическую семантику простой и получаем правило, с которым легко работать.

Таким образом, для предложений присваивания постулируем, что тип выражения должен совпадать с типом переменной, которой оно присваивается:

```
class Assign
  def type(context)
    if context[name] == expression.type(context)
      Type::VOID
    end
  end
end
```

Это правило вполне приемлемо для всех программ, в которых мы можем выбрать типы все переменных заранее и впоследствии не изменять, – с таким ограничением можно смириться. Например, можно проверить цикл `While`, динамическая семантика которого была реализована в главе 2:

```
>> statement =
       While.new(
         LessThan.new(Variable.new(:x), Number.new(5)),
         Assign.new(:x, Add.new(Variable.new(:x), Number.new(3)))
       )
=> «while (x < 5) { x = x + 3 }»
>> statement.type({})
=> nil
>> statement.type({ x: Type::NUMBER })
=> #<Type void>
>> statement.type({ x: Type::BOOLEAN })
=> nil
```

Достоинства и ограничения

Построенная нами система типов может предотвратить простые ошибки. Запустив игрушечную версию программы с такой статической семантикой, мы сможем узнать, значения каких типов могут встречаться в каждой точке исходной программы и убедиться, что эти типы соответствуют тем, которых ожидает динамическая семантика во время реального выполнения. Из-за простоты этой игрушечной интерпретации мы получаем только ограниченную информацию о том, что может произойти во время выполнения, но зато и проверки просты и незамысловаты. Например, можно проверить программу, которая работает вечно:

```
>> statement =
       Sequence.new(
         Assign.new(:x, Number.new(0)),
         While.new(
           Boolean.new(true),
           Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
         )
       )
=> «x = 0; while (true) { x = x + 1 }»
>> statement.type({ x: Type::NUMBER })
=> #<Type void>
>> statement.evaluate({})
SystemStackError: stack level too deep
```

Программа дурацкая, но ошибок типизации в ней нет: условие цикла – булево значение, в переменной `x` всегда находится число. Конечно, система типов не настолько умна, чтобы сказать, что программа делает именно то, что мы имели в виду, или что она вообще делает нечто полезное. Она проверяет лишь, что все части программы согласованы друг с другом. А поскольку система типов обязана быть безопасной, как и абстракция `Sign`, иногда она дает излишне пессимистичные ответы о наличии в программе ошибок. Чтобы убедиться в этом, достаточно добавить в программу выше еще одно предложение:

```
>> statement = Sequence.new(statement, Assign.new(:x, Boolean.new(true)))
=> «x = 0; while (true) { x = x + 1 }; x = true»
>> statement.type({ x: Type::NUMBER })
=> nil
```

Метод `#type` возвращает `nil`, сообщая тем самым об ошибке, потому что существует предложение, где переменной `x` присваивается булево значение, однако во время выполнения это никак не может привести к проблеме, потому что данное предложение никогда не выполняется. Но наша система типов недостаточно умная, чтобы это понять, поэтому дает безопасный ответ ««эта программа *может* вести себя неправильно» – излишне осторожный, но правильный. Где-то в программе производится попытка присвоить булево значение числовой переменной, так что эта часть потенциально может оказаться неверной, хотя *по совершенно другим причинам* в реальности такое не произойдет.

К проблемам приводят не только бесконечные циклы. С точки зрения динамической семантики, показанная ниже программа вполне корректна:

```
>> statement =
       Sequence.new(
```

```
        If.new(
          Variable.new(:b),
          Assign.new(:x, Number.new(6)),
          Assign.new(:x, Boolean.new(true))
        ),
        Sequence.new(
          If.new(
            Variable.new(:b),
            Assign.new(:y, Variable.new(:x)),
            Assign.new(:y, Number.new(1))
          ),
          Assign.new(:z, Add.new(Variable.new(:y), Number.new(1)))
        )
      )
=> «if (b) { x = 6 } else { x = true }; if (b) { y = x } else { y = 1 }; z = y + 1»
>> statement.evaluate({ b: Boolean.new(true) })
=> {:b=>«true», :x=>«6», :y=>«6», :z=>«7»}
>> statement.evaluate({ b: Boolean.new(false) })
=> {:b=>«false», :x=>«true», :y=>«1», :z=>«2»}
```

Переменная x используется для хранения числа или булева значения в зависимости от того, чему равно b — true или false; но во время вычисления здесь никакой проблемы не возникнет, потому что не существует пути выполнения, на котором x могла бы рассматриваться то как число, то как булево значение. Однако в абстрактных значениях, применяемых в статической семантике, недостаточно деталей, чтобы удостовериться в отсутствии ошибок[1], поэтому безопасная аппроксимация обязана сказать «эта программа может вести себя неправильно».

```
>> statement.type({})
=> nil
>> context = { b: Type::BOOLEAN, y: Type::NUMBER, z: Type::NUMBER }
=> {:b=>#<Type boolean>, :y=>#<Type number>, :z=>#<Type number>}
>> statement.type(context)
=> nil
>> statement.type(context.merge({ x: Type::NUMBER }))
=> nil
>> statement.type(context.merge({ x: Type::BOOLEAN }))
=> nil
```

Это *статическая система типов*, предназначенная для проверки программы до ее выполнения; в статически типизированном языке с каждой *переменной* ассоциирован тип. *Динамическая система типов*, применяемая в Ruby, работает иначе: у переменных нет типов, а проверяются только типы *значений*, когда

[1] В данном случае деталь состоит в том, что тип x зависит от значения b. В наших типах нет информации о конкретных значениях переменных, и они неспособны выразить зависимости между типами и значениями.

они используются в ходе выполнения программы. Благодаря этому Ruby разрешает присваивать одной переменной значения разных типов, но расплачивается за это невозможностью обнаружить ошибки типизации до начала выполнения программы.

В этой системе исследуется один конкретный вид ошибок в программе: у динамической семантики каждой синтаксической конструкции имеются определенные ожидания в части типов обрабатываемых значений, а система типов проверяет эти ожидания, чтобы в том месте, где должно быть булево значение, не оказалось число и наоборот. Но возможны и другие программные ошибки, которая данная статическая семантика не обнаруживает. Например, система типов не смотрит, было ли переменной присвоено значение перед использованием, поэтому если в программе имеются неинициализированные переменные, то она пройдет контроль типов, но «грохнется» при выполнении:

```
>> statement = Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
=> «x = x + 1»
>> statement.type({ x: Type::NUMBER })
=> #<Type void>
>> statement.evaluate({})
NoMethodError: undefined method `value' for nil:NilClass
```

Любую информацию, получаемую от системы типов, следует принимать с долей здорового скептицизма и обращать внимание на ограничения, решая, в какой мере ей доверять. Успешное прохождение сквозь сито статической семантики означает не что «эта программа точно будет работать», а что «эта программа точно не упадет из-за ошибки определенного вида». Было бы здорово иметь автоматизированную систему, которая могла бы сказать, что программа не содержит никаких мыслимых ошибок, но, как мы видели в главе 8, этот мир не настолько совершенен.

Приложения

В этой главе мы вкратце рассмотрели основную идею абстрактной интерпретации – применение дешевых аппроксимаций для изучения поведения дорогих вычислений – и продемонстрировали простую систему типов как пример полезности аппроксимации для анализа программ.

Наше обсуждение абстрактной интерпретации было очень неформальным. Если же говорить формально, то абстрактная интерпре-

тация – это математическая техника, позволяющая связать между собой различные семантики одного языка функциями, которые преобразуют наборы конкретных значений в абстрактные и наоборот, открывая тем самым возможность осмысливать результаты и свойства абстрактных программ в терминах конкретных.

Известным промышленным приложением этой техники является статический анализатор Astrée (*http://www.astree.ens.fr/*), в котором абстрактная интерпретация применяется для автоматического доказательства отсутствия в программах, написанных на языке C, таких ошибок времени выполнения, как деление на нуль, выход за границы массива и целочисленное переполнение. Astrée использовался для верификации программ управления полетов для самолетов Airbus A340 и A380, а также программ автоматической стыковки для космического аппарата *Jules Verne* ATV-001, который доставлял грузы на Международную космическую станцию. Абстрактная интерпретация не противоречит теореме Райса, потому что дает безопасные аппроксимации, а не гарантированные ответы, поэтому Astrée в принципе может сообщить о возможном наличии ошибки, когда ее на самом деле не существует (*ложное срабатывание*); впрочем, на практике его абстракции оказались достаточно точными, и в процессе верификации программного обеспечения A340 не было ни одного ложного срабатывания.

Программы, написанные на языке SIMPLE, могут манипулировать только рудиментарными значениями – числами и булевыми величинами – поэтому и типы, рассмотренные в этой главе, были очень простыми. В настоящих языках программирования разнообразие значений гораздо шире, поэтому реальные статические системы типов куда более развиты. Например, в статически типизированных функциональных языках программирования, подобных ML или Haskell, значениями могут быть функции (как процедуры в Ruby), поэтому их системы типов поддерживают *типы функций*, например «функция, которая принимает два числовых аргумента и возвращает булево значение», что позволяет верификатору типов проверять соответствие вызова функции ее определению.

Системы типов могут нести и другую информацию: в Java имеется *система типов и последствий*, которая отслеживает не только типы аргументов и возвращаемых значений методов, но и то, какие *контролируемые исключения* может возбуждать тело метода (возбуждение исключения – это последствие). Тем самым гарантируется, что любое возможное исключение либо обрабатывается, либо явно распространяется дальше.

Послесловие

Вот мы и добрались до конца путешествия по теории вычислений. Мы проектировали языки и машины с разными возможностями, проводили вычисления в необычных системах и с боем прокладывали дорогу к теоретическим пределам компьютерного программирования.

Исследуя конкретные машины и технические приемы, мы попутно познакомились со многими общими идеями.

- Любой человек может спроектировать и реализовать язык программирования. Базовые идеи синтаксиса и семантики просты, а инструменты типа Treetop могут позаботиться о малоинтересных деталях.
- Любая компьютерная программа – это математический объект. Синтаксически программа – всего лишь большое число, а семантически она может представлять математическую функцию или иерархическую структуру, которой можно манипулировать с помощью формальных правил свертки. Это означает, что многие методы и результаты, полученные математиками, например теорема Клини о рекурсии или теорема Гёделя о неполноте, применимы также и к программам.
- Вычисление, которое мы поначалу определили как «то, что делает компьютер», оказалось чем-то сродни силам природы. Соблазнительно думать о вычислениях как об изобретении изощренного человеческого ума и полагать, что их можно выполнить только с помощью специально сконструированных систем, состоящих из сложных частей, однако вычисления присутствуют и в системах, которые не кажутся достаточно сложными для их поддержки. Таким образом, вычисление – это не стерильный, искусственный процесс, который происходит исключительно внутри микропроцессора, а всепроникающее явление, неожиданно возникающее в самых разных местах и под совсем непохожими личинами.

- Вычисление не следует рассматривать как «все или ничего». У разных машин разная вычислительная мощность и соответственно полезность: у ДКА и НКА возможности ограничены, ДАМП помощнее, НАМП еще мощнее, а машина Тьюринга – самая мощная из всех, что мы знаем.
- Для обуздания мощи вычислений очень важны кодировки и уровни абстракции. Компьютеры – это машины для подпирания башни абстракций, которая начинается с самого низкого уровня физики полупроводников и возносится все выше и выше, доходя до мультисенсорных графических интерфейсов пользователя. Чтобы вычисление было полезным, мы должны научиться кодировать сложные идеи, берущие начало в реальном мире, то есть представлять их в более простой форме, которой могут манипулировать машины, а затем декодировать результаты, переводя их в осмысленное высокоуровневое представление.
- У вычислений есть предел доступного. Мы не знаем, как построить компьютер, который был бы принципиально мощнее машины Тьюринга, но существуют корректно поставленные задачи, которые машина Тьюринга решить не может, и многие из них касаются получения информации о тех программах, которые мы же и пишем. Мы можем приспособиться к этим ограничениям, научившись использовать неоднозначные или неполные ответы на вопросы о поведении программ.

Возможно, эти идеи сразу и не изменят ваш подход к работе, но я надеюсь, что они отчасти удовлетворили ваше любопытство и помогут получать удовольствие от времени, потраченного на выполнение вычислений в этом мире.

Предметный указатель

Том Стюарт

Теория вычислений для программистов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 30.01.2014. Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 16. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

Теория вычислений для программистов

Наконец-то появился увлекательный и практичный способ изучать теорию вычислений и проектирование языков программирования! В этой книге теоретическая информатика излагается в хорошо знакомом вам контексте, что поможет оценить, почему ее идеи важны и как они отражаются на том, чем программист изо дня в день занимается на работе. Вместо математической нотации или незнакомого академичного языка программирования типа Haskell или Lisp в этой книге для объяснения формальной семантики, теории автоматов и функционального программирования вкупе с лямбда-исчислением применяется язык Ruby, сведенный к минимуму. Это идеальное решение для программистов, знакомых хотя бы с одним из современных языков, но не имеющих формальной подготовки в информатике.

Рассматриваются следующие темы:

- Фундаментальные концепции вычислений, в том числе полнота языков программирования по Тьюрингу
- Использование в программах динамической семантики для сообщения смысла машине
- Исследование возможностей компьютера, сведенного к элементам самого низкого уровня
- Путь от универсальной машины Тьюринга к современным компьютерам общего назначения
- Выполнение сложных вычислений с помощью простых языков и клеточных автоматов
- Какие особенности языка программирования по-настоящему важны для вычислений
- Что такое проблема остановки и самореферентность и почему некоторые вычислительные задачи неразрешимы
- Анализ программ с использованием абстрактной интерпретации и системы типов

***Том Стюарт** — специалист по информатике и программист, основатель компании Codon, которая находится в Лондоне и занимается консультированием в области цифровых продуктов. Работает консультантом, преподавателем, инструктором, помогая различным компаниям выработать наиболее правильный подход к производству программных продуктов и повысить их качество.*

Интернет-магазин:
www.dmkpress.com

Книга – почтой:
orders@alians-kniga.ru

Оптовая продажа:
"Альянс-книга"
тел.(499)725-54-09
books@alians-kniga.ru

O'REILLY®

ДМК ПРЕСС
издательство
www.дмк.рф

ISBN 978-5-94074-979-0

9 785940 749790 >