ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНДУСТРИАЛЬНЫЙ УНИВЕРСИТЕТ

Е. А. Роганов, Н. А. Роганова

Программирование на языке Ruby

Учебное пособие

МГИУ
Москва 2008

ББК 32.97
УДК 681.3
Р59

Рецензент:

И. В. Абрамов, кандидат физико-математических наук,
доцент Московского государственного индустриального университета

Роганов Е. А., Роганова Н. А.

Р59 Программирование на языке Ruby: Учебное пособие. — М.: МГИУ, 2008. — 56 с.
ISBN 978-5-2760-1495-1

Настоящее пособие содержит описание языка программирования Ruby и предназначено прежде всего для студентов первого курса направления «Прикладная математика и информатика» и специальности «Математическое обеспечение и администрирование информационных систем». Оно может быть также полезно студентам инженерных и экономических специальностей, изучающих программирование, а также учащимся старших классов.

ББК 32.97
УДК 681.3

Редактор К. В. Шмат

Подписано в печать 05.02.08
Формат бумаги 60x84/16 Бум. офсетная
Усл.печ.л. 3,5 Уч.-изд.л. 3,75 Изд. № 1-08/08
Тираж 100 Заказ № 75

РИЦ МГИУ, 115280, Москва, Автозаводская, 16
www.izdat.msiu.ru
izdat@msiu.ru
тел. 677-23-15

ISBN 978-5-2760-1495-1

Оглавление

Предисловие

Ruby, названный так в честь драгоценного камня рубина, — один из самых молодых языков современного промышленного программирования.

В МГИУ с 2003 года именно Ruby является первым из языков, которые изучают студенты-программисты. Его используют для написания простейших программ на занятиях по информатике старшеклассники подшефных школ нашего университета. Ruby применяется сотрудниками информационно-вычислительного центра университета для генерации индивидуальных заданий по математике и информатике для студентов и слушателей факультета довузовского образования. Он же позволяет с минимальными затратами сил и времени решать многие другие задачи организации эффективного учебного процесса. Наконец, именно на Ruby реализована основная часть информационной системы, позволившей автоматизировать работу университета в целом.

Е. А. Роганов, Н. А. Роганова.

Москва, 2007

1. Основы языка Ruby

Ruby, названный так в честь драгоценного камня рубина, — один из самых молодых языков современного промышленного программирования. Первая версия интерпретатора была обнародована создателем языка, японским программистом Юкихиро Мацумото (Yukihiro Matsumoto) в 1995 году. Официальный сайт, посвящённый языку Ruby, размещён по адресу `http://www.ruby-lang.org`, а много дополнительной полезной и интересной информации можно найти в Википедии — свободной Интернет-энциклопедии (`http://en.wikipedia.org/wiki/Ruby_programming_language`).

Ruby — это чрезвычайно мощный, динамический, чисто объектно-ориентированный язык, при разработке которого основное внимание было уделено удобству программирования на нём. Многие удачные идеи, использованные ранее в таких языках, как Perl, Python, Smalltalk, LISP и некоторых других, в Ruby удалось гармонично объединить. Благодаря этому язык легко изучать, на нём очень легко и приятно писать программы, а в уже написанные программы легко вносить необходимые изменения.

В МГИУ с 2003 года Ruby является первым из языков, которые изучают студенты-программисты. Его используют для написания простейших программ на занятиях по информатике старшеклассники подшефных школ нашего университета. Ruby применяется сотрудниками информационно-вычислительного центра университета для генерации индивидуальных заданий по математике и информатике для студентов и слушателей факультета довузовского образования. Он же позволяет с минимальными затратами сил и времени решать многие другие задачи организации эффективного учебного процесса. Наконец, именно на Ruby реализована основная часть информационной системы, позволившей автоматизировать работу университета в целом (см. [6]) .

В этой небольшой книге мы не сможем рассказать обо всех особенностях языка, однако уже после знакомства с текущей главой читатель сможет писать небольшие программы на Ruby. Следующая глава позволит научиться применять этот язык для решения типичных задач обработки текстов и простейшего системного администрирования, а далее будет рассказано, как использовать язык Ruby для веб-программирования.

1.1 Установка Ruby. Если Ваша операционная система — Linux или Mac OS X, то, скорее всего, интерпретатор языка Ruby вместе со всеми необходимыми библиотеками уже установлен. Команда **ruby -v** в этом случае выведет информацию о версии интерпретатора, подобную следующей: `ruby 1.8.4 (2005-12-24) [i686-linux]`.

Для Microsoft Windows существует так называемый One-Click Installer, который можно взять с сайта `http://rubyinstaller.rubyforge.org`.

Подходящий RPM-пакет для операционной системы Linux легко найти на сайте `http://www.rpmfind.net`, набрав в поле поиска слово **ruby**.

Так как Ruby — свободный программный продукт, то его исходные тексты доступны и могут быть получены с сайта `http://www.ruby-lang.org`. Установка из исходных текстов требует определённых знаний, но, как правило, сводится к выполнению лишь нескольких команд, подобных следующим: **tar xvfz ruby-1.8.4.tar.gz; cd ruby-1.8.4; ./cofigure; make install**.

1.2 Первые программы.

Программа на языке Ruby представляет собой последовательность выражений и инструкций (expressions and statements), которые размещаются обычно по одному (одной) на строке. Точка с запятой используется для отделения друг от друга инструкций на одной и той же строке, обратный слэш \ в конце строки позволяет продолжить запись выражения на следующей строчке, а комментарии начинаются с символа # и продолжаются до конца текущей строки.

Пример 1. Напишите программу, печатающую строку-приветствие.

```
puts "Здравствуй, мир!"
```

Эта программа содержит единственную инструкцию — вызов метода `puts`, который печатает на стандартный вывод данный ему список объектов, состоящий в нашем случае из одной строки — объекта класса `String`. Если указанный в рамке текст разместить в файле `hello.rb`, то команда **ruby hello.rb** запустит программу и мы увидим приветствие на экране, который является стандартным выводом по умолчанию. Перенаправить вывод в файл с именем `res.txt` можно командой **ruby hello.rb > res.txt**.

Ruby является объектно-ориентированным языком, но допускает написание программ в директивном стиле. Можно считать, что наша программа содержит команду (директиву) «напечатать строку», хотя на самом деле всё устроено немного сложнее. Те сущности (называемые объектами), с которыми имеет дело программа на языке Ruby, в процессе выполнения программы взаимодействуют между собой, посылая друг другу сообщения (называемые также вызовами методов), содержащие иногда дополнительную информацию (объекты-параметры). Вызов метода `m` объекта `obj` с параметром `arg` записывают в виде `obj.m(arg)` или просто `obj.m arg`[1], а единственная инструкция нашей программы является сокращением от `STDOUT.puts "Здравствуй, мир!"`. Теперь уже ясно, что сообщение «напечатать строку» посылается объекту `STDOUT` — стандартному выводу.

[1] Подробнее о вызовах методов рассказано на стр. 45.

Пример 2. Напишите программу, печатающую сумму первых `n` членов начинающейся с единицы геометрической прогрессии со знаменателем `q`.

Рассмотрим сначала решение, основанное на использовании известной формулы для суммы геометрической прогрессии $S = (q^n - 1)/(q - 1)$. Запуск этой программы даёт результат 7, а заменив 3 на 64 и запустив программу ещё раз, мы получим знаменитое «шахматное»[2] число 18446744073709551615.

```
q = 2; n = 3
puts (q**n-1)/(q-1)
```

★ **В Ruby (в отличие от многих других языков) легко работать с большими числами.**

Программа содержит три инструкции, первые две из которых присваивают переменным `q` и `n` значения 2 и 3 соответственно. Отметим, что сами переменные объектами не являются, а лишь указывают (ссылаются) на различные объекты (в данном случае на числа — объекты класса `Fixnum`). Далее вычисляется выражение `(q**n-1)/(q-1)`, которое неявно преобразуется в строку и печатается методом `puts` на стандартный вывод. Вот развёрнутая форма второй инструкции программы: `STDOUT.puts ((q**n-1)/(q-1)).to_s`, где метод `to_s`, применяемый к числу, возвращает представление этого числа в виде строки.

Кроме чисел и уже встречавшихся нам строк в языке Ruby существует ещё ряд базовых типов (см. стр. 42), для работы с которыми можно использовать много различных операторов (см. стр. 48), включая стандартные арифметические операторы и оператор присваивания. Имена локальных (другие нам встретятся чуть позже) переменных должны начинаться с маленькой латинской буквы и могут содержать также большие буквы, цифры и символ подчёркивания. Имена констант (например, `STDOUT`) должны начинаться с большой буквы. Более полная информация об использовании имён приведена на стр. 45.

Другое решение этой задачи, использующее цикл `while`, требует чуть больших комментариев. Переменные `s` и `i` используются в этой программе как аккумулятор для накопления суммы прогрессии и счётчик цикла соответственно. Вначале они обнуляются, а на каждой итерации цикла, выполнение которого продолжается, пока величина `i` остаётся меньше `n`, значение `s` увеличивается на очередной член прогрессии, а значение `i` — на единицу. После завершения цикла остаётся только напечатать результат. Заметим, что выражение `i += a` эквивалентно `i = i + a`.

```
q = 2; n = 3
i = s = 0
while i < n
  s += q**i
  i += 1
end
puts s
```

[2] Согласно легенде, изобретатель шахмат Сета потребовал от индусского царя Шерама, решившего наградить его, выдать за первую клетку доски одно пшеничное зерно, за вторую — два, за третью — четыре и так далее, вплоть до последней, шестьдесят четвёртой.

Обратите внимание, насколько первый вариант программы проще второго. Ещё важнее, что в последнем случае используется *низкоуровневый* стиль программирования, который чреват ошибками и которого стараются избегать знатоки Ruby. Кроме того, вторая программа чрезвычайно *неэффективна* по сравнению с первой (попробуйте найти сумму миллиарда членов прогрессии со знаменателем 0.5 с помощью обеих программ).

★ **Даже хорошее знание лучших языков программирования не заменяет знания математики.**

1.3 Использование базовых типов.

Часто программы получают данные из файлов, запрашивают их с клавиатуры или берут непосредственно из командной строки. Проиллюстрируем последнюю из этих возможностей на примере решения следующей несложной задачи.

Пример 3. Напишите программу, печатающую сумму цифр десятичного представления натурального числа, задаваемого в командной строке.

После запуска программы `test.rb` командой **ruby test.rb 123 Маша** в массиве `ARGV` окажутся все аргументы командной строки, указанные при её запуске, то есть строки `123` и `Маша`. Выражение `ARGV[0]` даёт первый элемент массива (строку `123`). Для решения задачи этого достаточно, а более подробно с массивами (`Arrays`) можно познакомиться, заглянув на стр. 43. Таблица А.10 на стр. 47 содержит перечень наиболее часто используемых предопределённых объектов Ruby-программы (`ARGV` — один из них).

Так как последняя цифра числа `n` равна остатку от деления на 10 (`n%10`), а целочисленное деление на 10 (`n/10`) равносильно удалению последней цифры, то построить цикл `while`, вычисляющий требуемую сумму, не слишком сложно. Обратите внимание на метод `to_i`, преобразующий строку в целое число. Его необходимо делать явно в отличие от вызываемого автоматически при работе `puts` метода `to_s`, выполняющего обратное преобразование числа в строку. Запустить эту программу, содержащуюся в файле `sum1.rb`, можно так: **ruby sum1.rb 12345**.

```
n = ARGV[0].to_i
s = 0
while n > 0
  s += n % 10
  n /= 10
end
puts s
```

Для опытного человека написание подобной программы не является проблемой, но новичок может сделать в ней несколько самых разных ошибок, поэтому будущих программистов знакомят с математическими методами проектирования циклов. Подобные знания зачастую абсолютно необходимы, но рассматриваемую задачу даже новичок сможет решить безошибочно, если воспользуется так называемым *подходом Ruby* (*Ruby way*). Он предполагает активное использование стандартных библиотек, что поз-

воляет создавать короткие и ясные программы, не содержащие циклов, при проектировании которых так легко совершить ошибку.

Новая версия программы содержит последовательный вызов всего трёх методов и состоит из одной строки.

```
puts ARGV[0].scan(/./).inject(0){|s,x| s+x.to_i}
```

Метод `scan`[3] с параметром `/./` преобразует строковое представление числа `ARGV[0]` в массив цифр, который обрабатывается методом `inject` — одним из так называемых *итераторов*, последовательно выполняющим действия, указанные в *блоке* (код в фигурных скобках), над каждым из элементов массива. При этом переменная `s` сначала инициализируется нулём, а на каждой итерации увеличивается на величину `x.to_i`, равную числовому значению очередной цифры `x`. Полученный результат затем печатается методом `puts`.

Отметим, что `inject` является методом модуля `Enumerable`, который включается (mix in) в различные классы, расширяя их возможности. Кроме массивов таким классом являются диапазоны (`Ranges`), чаще всего используемые в качестве последовательностей элементов: натуральные числа от 3 до 9 (`3..9` или `3...10`), латинские буквы от `D` до `H` (`'D'..'H'`) и т. п. При решении следующей задачи использование итератора `each` для диапазона чисел является вполне естественной идеей.

Пример 4. Является ли простым число 15 485 863?

Напомним, что натуральное число является простым, если оно имеет ровно два различных делителя. Интерес к простым числам связан прежде всего с их использованием в криптографии. Подробнее об этом можно узнать на сайте `http://www.utm.edu/research/primes` и на страницах Википедии (`http://en.wikipedia.org/wiki/Prime_numbers`).

Наивный алгоритм проверки простоты числа `n` очевиден: будем последовательно находить остатки от деления `n` на числа 2, 3, 4 и так далее, вплоть до `n-1`. Если хотя бы один из остатков окажется нулевым, то число простым не является. Итератор `each` для каждого элемента диапазона выполняет блок (начинающийся `do` и заканчивающийся `end`[4]), проверяющий это условие и прекращающий выполнение программы (метод `exit`) в случае его выполнения. В Ruby операторы `if`, `while` и некоторые другие являются про-

```
n = 15_485_863
(2 ... n).each do |i|
  if n%i == 0
    puts false; exit
  end
end
puts true
```

[3] Список всех методов класса `String` приведён в разделе Library Reference книги [3]. Команда **ri String** печатает его целиком, а команда **ri String#scan** — описание метода `scan`.

[4] Блок принято ограничивать `do-end` вместо `{}`, если его тело занимает несколько строк.

сто модификаторами выражений, что не так во многих других языках (обязательно ознакомьтесь с примерами на стр. 49).

Программа станет намного изящнее и понятнее, если воспользоваться методом `any?` (в Ruby имя метода может оканчиваться вопросительным знаком[5]) модуля `Enumerable`. Этот метод проверяет для элементов `i` диапазона `(2...n)` заданное в блоке условие `n%i==0` и возвращает «истину» (`true`), как только встретит элемент, ему удовлетворяющий. Если же таких элементов не найдётся, то метод возвратит «ложь» (`false`). Отрицание (`!`) полученного результата является ответом на вопрос о простоте числа `n`.

```
n=15_485_863; puts !(2...n).any?{|i| n%i==0}
```

Аналогичный методу `any?` метод `all?` модуля `Enumerable` позволяет проверить истинность заданного в блоке условия *для всех* элементов коллекции. Этот модуль определяет ещё целый ряд методов, полный перечень можно получить с помощью команды **ri Enumerable**.

Пример 5. Найдите минимальное четырёхзначное число, уменьшающееся на 27 после перестановки его последней цифры на первую позицию.

Последняя цифра числа `x` — это `x%10`, а число, получающееся после её перестановки на первую позицию, равно значению выражения `x/10+1000*(x%10)`. Таким образом, нам нужно найти (find) первое число из диапазона `1000...10000`, для которого выполняется условие `x/10+1000*(x%10) == x-27`. Это делает метод `find` (или `detect`).

```
puts (1000...10000).find{|x|x/10+1000*(x%10) == x-27}
```

Заметим, что методы `find_all` и `select`, будучи применёнными к коллекции, возвращают массив всех тех её элементов, которые удовлетворяют заданному в блоке условию, а метод `reject` возвращает элементы, для которых условие оказывается ложным.

Пример 6. Напечатайте квадраты всех натуральных чисел, не больших тысячи, десятичная запись которых заканчивается на 396.

Для решения задачи достаточно применить метод `map` (или `collect`), возвращающий массив, содержащий результаты применения заданного блока к

```
p (1..1000).map{|t|t*t}.select{|x|x%1000==396}
```

[5] Подробнее об именах методов рассказано на стр. 45.

элементам коллекции, а затем выбрать из него нужные числа. Метод `p` печатает массив (как и иные объекты) в более удобном виде, чем `puts`.

Пример 7. Выясните, является ли заданная в командной строке последовательность символов палиндромом. Напомним, что палиндром — это последовательность, которая не изменяется после её инвертирования (переворачивания). Ограничимся в этой задаче только малыми русскими буквами и пробелами, которые должны игнорироваться. Вот примеры палиндромов: «поп», «шалаш», «аргентина манит негра», «а роза упала на лапу азора». Если программа, решающая эту задачу, содержится в файле `palindrome.rb`, то команда **`ruby palindrome.rb аргентина манит негра`** должна напечатать `true`.

Как мы уже знаем, объект `ARGV` содержит массив аргументов командной строки. Метод `join` класса `Array` «склеивает» в одну строку все его элементы, а метод `reverse` класса `String` инвертирует строку.

```
p ARGV.join==ARGV.join.reverse
```

★ **Время, потраченное на знакомство с библиотеками Ruby, многократно окупится в дальнейшем.**

Пока мы использовали только методы, определённые для классов и модулей стандартной библиотеки, но часто при написании программ полезно создавать свои методы. Например, при решении задачи о простоте числа естественно определить метод `prime?` с одним параметром (числом), возвращающий `true` или `false`. Затем его можно использовать для того, чтобы найти и напечатать все простые числа из диапазона `15_485_800..15_485_863`. Подробнее про определение методов рассказано на стр. **??**.

```
def prime?(n)
  not (2 ... n).any?{|i| n%i == 0}
end
(15_485_800..15_485_863).each do |x|
  puts x if prime?(x)
end
```

Пример 8. Реализованный выше метод `prime?` работает достаточно медленно. Говорят, что его сложность линейна[6], ибо для простых чисел выполняется почти n итераций. Реализуйте метод `prime?`, работающий быстрее.

Заметим, что если $n = pq$, то одно из чисел p и q заведомо не превосходит $\sqrt{n}$. До этой величины и достаточно выполнять проверку[7]. Для вычисления квадратного корня используем метод `sqrt` модуля `Math`.

[6] Более точно это формулируют так: асимптотическая сложность метода есть $\Theta(n)$.

[7] Поэтому сложность получающейся программы будет равна $\Theta(\sqrt{n})$

```
def prime?(n) not (2 .. Math.sqrt(n)).any?{|i| n%i == 0} end
(15_485_800 .. 15_485_863).each{ |x| puts x if prime?(x) }
```

★ **Без знания математики хорошей программы не напишешь.**

Пример 9. Многочлен можно задать массивом его коэффициентов. Например, многочлену x^3+2x-3 соответствует массив `[1,0,2,-3]`, а массив `[1,2]` задаёт многочлен $x+2$. Реализуйте метод, позволяющий перемножать два многочлена, заданные их коэффициентами.

Если $P(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n = \sum_{i=0}^{n} a_i x^{n-i}$, а $Q(x) = b_0x^m + b_1x^{m-1} + \ldots + b_{m-1}x + b_m = \sum_{j=0}^{m} b_j x^{m-j}$, то, как легко проверить, $P(x) \cdot Q(x) = \sum_{j=0}^{m+n} \left(\sum_{i=0}^{j} a_i b_{j-i} \right) x^j$. На самом деле для написания программы эта формула совершенно не нужна. Достаточно заметить, что степень итогового многочлена на единицу меньше суммы степеней исходных, перемножать необходимо каждый из коэффициентов первого многочлена a_i на каждый из коэффициентов второго b_j, а получающееся произведение a_ib_j следует добавлять к коэффициенту итогового многочлена при степени $i+j$. Реализовать данную идею проще всего с помощью метода `each_with_index` модуля `Enumerable`, который последовательно выполняет заданный блок для всех элементов коллекции, передавая в него сам элемент и его индекс. Метод `size` класса `Array` используется для определения количества элементов в нём, а метод `new` с двумя параметрами этого же класса создаёт новый массив указанного размера, заполненный нулями. Массивы в Ruby применяют очень часто, так как их можно использовать в качестве *стеков*, *очередей*, *деков*, *списков и других структур данных*. Некоторые из методов класса `Array` описаны на стр. **??**.

```
def mul(p,q)
  r = Array.new(p.size+q.size-1,0)
  p.each_with_index do |u, i|
    q.each_with_index do |v, j|
      r[i + j] += u * v
    end
  end
  r
end
p mul([1,0,2,-3],[1,2])
```

Пример 10. Реализованный в примере 8 метод `prime?` является слишком медленным для получения списка всех простых чисел от 2 до миллиона.

Напишите программу, решающую эту задачу за «разумное время».

Более двух тысяч лет назад греческий математик Эратосфен придумал алгоритм, называемый сейчас «решетом Эратосфена» [8]. В простейшем варианте он выглядит так. Выпишем сначала все числа от 2 до n. Затем отметим первое простое число 2 и вычеркнем все числа, кратные ему. Далее отметим первое из ещё не отмеченных и не вычеркнутых чисел (это будет число 3), и вычеркнем все числа, кратные ему (включая и те, которые уже вычеркнуты). Будем продолжать данный процесс, пока это возможно. В итоге останутся только простые числа. Слегка модифицированный алгоритм приводит к программе, которая через несколько секунд после её запуска командой **ruby sieve.rb 1000000** печатает список всех простых чисел, меньших миллиона. Первая инструкция программы содержит вызов метода `Integer` модуля `Kernel`, который делает почти то же самое[9], что используемый нами ранее метод `to_i` класса `String`, — преобразует строку в число. В созданный далее пустой массив `sieve` заносятся числа от `2` до `n` с помощью цикла `for`. Массив при этом растёт: после первого присваивания он содержит три элемента (`[nil,nil,2]`), после второго — уже четыре и т. д. Цикл `for` является лишь удобным сокращением для известного нам итератора `each`: запись `for i in 2 .. m` неявно преобразуется в `(2 .. m).each do |i|`.

```
n = Integer(ARGV[0])
sieve = []
for i in 2 .. n
  sieve[i] = i
end
for i in 2 .. Math.sqrt(n)
  next unless sieve[i]
  (i*i).step(n,i){|j|sieve[j]=nil}
end
p sieve.compact
```

В следующем цикле число не обрабатывается, если оно равно `nil`, то есть уже вычеркнуто (см. таблицу А.12 на стр. 49). Вычёркивание всех ему кратных осуществляет итератор `min.step(limit,step){|i| block}` класса `Numeric`, который выполняет `block`, начиная со значения `i=min`, увеличивая его после каждой итерации на `step` до тех пор, пока оно не станет больше, чем `limit`. Воспользуйтесь командой **ri compact** для выяснения того, что делает метод `compact`.

★ **Учитесь находить нужную Вам информацию в книгах, справочных системах и сети Интернет.**

Пример 11. Числами Фибоначчи называют последовательность, задаваемую следующими формулами: $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$ для $n > 1$. Напишите рекурсивный метод вычисления величины f_n.

[8] Смотри, например, `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.

[9] `Integer` учитывает также индикаторы системы счисления — префиксы `0`, `0b` и `0x`.

Числа Фибоначчи встречаются и в науке, и в природе чрезвычайно часто[10], а последовательность этих чисел занимает весьма почётное место в Интернет-энциклопедии целочисленных последовательностей[11], которая является очень интересной сама по себе.

Рекурсивным называют метод, который вызывает сам себя, и написать его в данном случае очень легко, ибо он дословно повторяет определение последовательности чисел Фибоначчи. Отметим только, что тернарный оператор `a ? b : c` эквивалентен условному оператору `if a then b; else c end` (см. таблицу А.12 на стр. 49).

```
def fib(n)
  n < 2 ? n : fib(n-2) + fib(n-1)
end
p fib(40)
```

Пример 12. С помощью написанного рекурсивного метода практически невозможно находить числа Фибоначчи f_n для больших значений n, так как даже на вычисление сорокового числа на компьютере с тактовой частотой процессора 2.4 Ghz требуется почти 6 минут. Реализуйте метод, позволяющий найти миллионное число Фибоначчи за «разумное время».

Требуемое решение можно получить, рассматривая преобразование F, определённое на парах чисел формулой $F(a,b) = (b, a+b)$. Если начать с пары чисел $(0,1)$, то многократное применение F порождает последовательность чисел Фибоначчи. Такая программа (обратите внимание на множественное (параллельное) присваивание, см. стр. 49) имеет линейную сложность, ибо количество необходимых для нахождения f_n итераций цикла не превосходит n. Миллионное число Фибоначчи на компьютере с заданными в постановке задачи характеристиками эта программа находит примерно за две минуты, а ведь в его десятичной записи содержится 86784 цифры!

```
def fib(n)
  return n if n < 2
  a, b = 0, 1
  for i in 2 .. n
    a, b = b, a+b
  end
  b
end
p fib(1_000_000)
```

```
require 'matrix'
def fib(n)
  return n if n < 2
  (Matrix[ [1,1], [1,0] ]**n)[0,1]
end
p fib(1_000_000)
```

Интересно, что если вместо пар чисел рассматривать четвёрки, записанные в виде таблицы 2×2 — квадратной матрицы, то можно получить ещё более быструю программу. Заметим, что матрица $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, возведённая в

[10] Загляните на сайт `http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci`.

[11] `http://www.research.att.com/~njas/sequences`.

квадрат, равна $\begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}$, в куб — $\begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}$, в четвёртую степень — $\begin{pmatrix} 5 & 3 \\ 1 & 0 \end{pmatrix}$. Первая строка этих матриц содержит числа Фибоначчи, а так как возведение в степень (в том числе и матриц) выполняется быстро, то вычисление чисел Фибоначчи таким способом оказывается весьма эффективным.

Для работы с матрицами, которые не входят в число базовых типов языка Ruby, необходимо с помощью директивы `require` подключить библиотеку, в которой определён класс `Matrix` и методы для манипулирования с объектами этого класса. Миллионное число Фибоначчи последняя программа находит почти в два с половиной раза быстрее, чем предыдущая, и разница в скорости их работы увеличивается с ростом номера n числа f_n.

1.4 Упражнения

Задача 1. Оператор `**` в Ruby реализован весьма эффективно по сравнению с *наивным* способом умножения основания на себя нужное число раз. Напишите программу с применением цикла `while`, которая будет возводить число `a` в натуральную степень `n` за время, сопоставимое со временем работы оператора `a**n` даже для больших значений показателя.

Задача 2. Измените написанную при рассмотрении примера 7 программу так, чтобы она работала и в ситуации, когда исследуемая строка заключается в апострофы или кавычки. Например, команда **ruby palindrome.rb 'аргентина манит негра'** должна печатать `true`.

Задача 3. Какие изменения следует внести в программу умножения многочленов (пример 9), если коэффициенты многочленов задавать в обратном порядке — от младших степеней к старшим?

Задача 4. Напишите программу, находящую для всех чисел от 1 до задаваемого в командной строке натурального n, массив списков (массивов) всех делителей этих чисел. Для $n = 6$, например, программа должна напечатать следующую строку: `[[1], [1, 2], [1, 3], [1, 2, 4], [1, 5], [1, 2, 3, 6]]`.

Задача 5. Назовём билет с натуральным номером, десятичная запись которого состоит из чётного количества $(2n)$ цифр, счастливым, если сумма первых его n цифр равна сумме n последних. Напишите программу, вычисляющую количество счастливых билетов для заданного натурального n.

Задача 6*. Напишите программу, способную вычислить количество счастливых билетов (см. задачу 5) для $n = 1000$ за «разумное время» (например, около 30 секунд на компьютере с тактовой частотой процессора 2.4 Ghz).

Задача 7*. Напишите такую программу, решающую задачу 4, чтобы время её работы для $n = 10^6$ удовлетворяло требованиям, сформулированным в задаче 6 (это ограничение касается собственно нахождения массива; вывод такого огромного количества данных требует заметного дополнительного времени).

Задача 8*. Программа умножения многочленов, приведённая на стр. 12, имеет квадратичную сложность (количество операций при перемножении двух многочленов степени n пропорционально n^2). Так называемое быстрое преобразование Фурье (см. книгу [5]) позволяет сделать это быстрее. Реализуйте алгоритм быстрого умножения многочленов таким образом.

2. Модификация и создание пользовательских классов

Ruby является объектно-ориентированным языком программирования. Давайте рассмотрим особенности представления классов.

Объекты и классы

В реальной жизни все объекты обладают индивидуальными свойствами. Но, введя некоторые обобщения, можно поделить объекты реальной жизни на группы. Каждая из таких групп будет обладать одинаковым набором свойств и, как следствие, к ней будут применимы одинаковые наборы действий.

В языке программирования Ruby для описания набора свойств групп объектов и применимых к этим объектам действий используется такое понятие, как классы.

Класс — это формальное описание основных свойств объекта (его атрибутов) и методов, применимых к нему. Задав описание класса, в дальнейшем можно создавать столько его экземпляров, сколько потребуется.

Классы в Ruby — открыты; если нам не хватает функциональности класса, то мы можем дополнить класс новым методом или переопределить старый.

```
class String
  def reverse
    "?????"
  end
  def msiu
    "МГИУ"
  end
end
a = "12345"
puts a.reverse
puts a.msiu
```

Иерархия классов и полиморфизм

Зачастую из класса можно выделить группу объектов, обладающую каким-либо специальным свойством. Такую группу называют подклассом, или дочерним классом. Дочерний класс наследует все свойства родительского класса, но обладает отдельной функциональностью. С помощью символа < указывается родительский класс.

Пример 1.

Птицы всех видов несут яйца, но не все птицы умеют летать. Хотя в классе `Penguin` и не описан метод `lay_egg`, при необходимости он ищется (и находится!) в родительском классе (принцип наследования, иерархическая организация классов). Метод `fly` переопределен для пингвинов, и реакция на вызов этого метода определяется принадлежностью к тому или

иному классу (полиморфизм, различные реалиции на одинаковую команду).

```
class Bird
  def lay_egg
    puts "Я из класса #{self.class}. Яйцо снесено!"
  end
  def fly
    puts "Все представители класса #{self.class} умеют летать!"
  end
end
class Penguin < Bird
  def fly
    puts "Представители класса #{self.class} не летают..."
  end
end
b = Bird.new; b.lay_egg; b.fly
p = Penguin.new; p.lay_egg; p.fly
```

Создание класса

Дальнейшее описание проиллюстрируем обыкновенными дробями. В процессе вычислений нередко возникает необходимость работы с дробными числами. Учитывая ограничения по точности представления чисел с плавающей точкой в компьютере, иногда бывает необходимо производить вычисления в терминах обыкновенных дробей, представляющих собой отношение целого числителя и целого знаменателя. В Ruby имеется класс `Rational`, описывающий такие дроби. Представим на время, что его нет и создадим нечто подобное.

```
class Frac
  def initialize(a, b)
    raise 'Division by zero' if b.to_i == 0
    @numerator, @denominator = a.to_i, b.to_i
  end
end
fr = Frac.new(1, 2)
```

Метод `initialize` называется *конструктором* класса. Он вызывается каждый раз, когда мы создаем экземпляр класса при помощи метода `new`.

Конструкция `raise` вызывает *исключительную ситуацию* (об этом будет рассказано позже) и выдает соответствующее сообщение об ошибке.

Переменные экземпляра

Переменные `@numerator`, `@denominator` — это *переменные* атрибуты *экземпляра* класса. В именах переменных экземпляра необходимо использовать префикс `@`. Каждый объект, принадлежащий данному классу, имеет свои собственные значения этих атрибутов (свойства), но пока их можно использовать только внутри методов самого класса. При попытке обратиться к таким переменным извне класса будет выдано сообщение об ошибке. Что же делать, если хочется иметь доступ к переменным экземпляра вне класса?

Один способ состоит в создании методов, возвращающих значение соответствующего атрибута. Но в случае большого числа атрибутов такой подход не удобен. Язык Ruby предлагает более удобную возможность.

```
class Frac
  def numerator
    @numerator
  end
end
```

Для контроля доступа к переменным экземпляра можно использовать макросы `attr_reader` (для чтения), `attr_writer` (для изменения значения) и `attr_accessor` (для выдачи разрешения на оба действия).

Метод `to_s`, присутствующий в классе, определяет, как объект должен отображаться при печати.

```
class Frac
  attr_reader :numerator, :denominator
  attr_writer :numerator, :denominator
  def to_s
    "#{@numerator} / #{@denominator}"
  end
end
x = Frac.new(1, 2); puts x
x.denominator = 4
puts x.numerator, x.denominator
```

Переопределение операторов

Оперирование дробями предполагает возможность производить математические действия, например, сложение. Символ `+` является оператором языка Ruby. Но некоторые операторы для удобства пользователя могут быть переопределены. Но не все! Например, `+` может быть переопределен, `=` нет.

Реализуем операцию сложения дробей самостоятельно. Чтобы дроби не получались избыточными, нам надо будет уметь их сокращать. Это позволяет сделать алгоритм Евклида.

```
class Frac
  def initialize(a, b)
    raise 'Division by zero' if b.to_i == 0
    @numerator,@denominator  = a.to_i, b.to_i
    simplify()
  end

  def +(b)
    if b.class != Frac
      raise "Undefined method + for class Frac and #{b.class}!"
    end
    return Frac.new(self.numerator * b.denominator +
                    b.numerator * self.denominator,
                    self.denominator * b.denominator)
  end
  private
  def simplify()
    x, y = @numerator.abs, @denominator.abs
    x, y = y, x % y while x * y > 0
    m = x + y
    @numerator, @denominator = @numerator / m, @denominator / m
  end
end
```

В операторе сложения мы проверяем, что второй аргумент операции + тоже является объектом типа `Frac`.

```
if b.kind_of?(Frac)
```

Для каждого объекта в Ruby можно применить метод `class`, чтобы выяснить, к какому классу он относится. Более правильно проверять, что объект относится к тому или иному классу, при помощи метода `kind_of?`.

Ключевое слово `self` указывает на объект, вызвавший метод. Запись `self.numerator` эквивалентна `@numerator`.

Метод `simplify` реализует алгоритм Евклида, упрощающий дроби. Чтобы у нас всегда были упрощенные дроби, он вызывается в конструкторе класса. Ключевое слово `private` определяет право доступа к следующим за ним методам только внутри класса. Таким образом, метод `simplify` не может быть использован вне класса `Frac`.

Аналогично + мы можем также переопределять и логические операторы. Например, оператор ==, позволяющий сравнивать два объекта на равество.

```
class Frac
  def ==(b)
    if !b.kind_of?(Frac)
      raise "Undefined method == for class Frac and #{b.class}!"
    end
    return (self.numerator == b.numerator and
            self.denominator == b.denominator)
  end
end
```

Если нам хочется, чтобы объекты нашего класса можно было бы сравнивать между собой (и, как следствие, сортироовать массив таких объектов методом sort), то необходимо переопределить метод <=>. Метод <=> должен возвращать 0 если объекты равны между собой; отрицательное число, если первый аргумент меньше второго; и положительное число в остальных случаях.

```
class Frac
  def <=>(b)
    if !b.kind_of?(Frac)
      raise "Undefined method <=> for class Frac and #{b.class}!"
    end
    return self.numerator * b.denominator -
                           b.numerator * self.denominator <=> 0
  end
end
```

Переменные класса

В случае необходимости иметь какую-то одинаковую для всех объектов класса характеристику используют *переменные класса*, которые синтаксически выделяются указанием префикса `@@` в их имени.

```
@@quantity += 1
```

Предположим, что нам потребовалось знать общее количество дробей, созданных при работе с классом `Frac`. Добавив в конструктор класса команду инкремента такой переменной, мы сможем узнать количество созданных дробей.

Методы класса

С понятием применения метода связано понятие ответственности за его выполнение. Иногда за то или иное действие отвечает не объект, а сам класс. В этом случае применяется *метод класса*. При его задании перед именем метода указывается имя класса.

```
class R2Point
  def initialize(x, y)
    @x, @y = x, y
  end
  def dist(a)
    sqrt((a.x-@x)**2 + (a.y-@y)**2)
  end
  def R2Point.dist(a, b)
    sqrt((a.x-b.x)**2 + (a.y-b.y)**2)
  end
end
p1 = R2Point.new(1,2)
p2 = R2Point.new(-1,-2)
puts p1.dist(p2)
puts R2Point.dist(p1, p2)
```

Пример 2. Рассмотрим класс `R@Point`, описывающий точки на плоскости. Пусть нам требуется найти расстояние между двумя точками. В первом случае мы перекладываем ответственность за выполнение метода на одну из точек. Требование выглядит так: «точка, вычисли расстояние до другой точки». В этом случае непонятно, почему одна из точек берет на себя ответственность за выполнение этой обязанности, ведь они обе участвуют в данной операции на равных правах. Второй подход состоит в команде, отдаваемой самому классу: «класс, определи расстояние между двумя объектами».

3. Тестирование и обработка исклительных ситуаций

3.1 Unit-тесты. Проанализировав, на что уходит время у большинства программистов, — можно обнаружить, что на написание кода в действительности тратится совсем небольшая часть. Какая-то часть уходит на понимание задачи, еще кусок на проектирование, а большую часть времени занимает отладка. Отладка — это процесс проверки программы на соответствие поставленной задаче, хотя чаще под этим определением понимают работу по нахождению ошибок в программе. Отладка — это «страшная вещь», и любой программист может рассказать о том, как на поиски какой-то ошибки ушел день или даже больше. Исправить ошибку можно довольно быстро, но самое сложное — найти её. А при исправлении ошибки всегда существует возможность добавить новую, которая проявится гораздо позднее.

Из опыта предыдущих курсов было замечено, что студент написав программу, как правило, поверхностно проверяет её на работоспособность, а некоторые даже не запускают программу, почему-то считая, что раз уж они её написали, то она просто обязана работать правильно. Доходит до того, что студенты пытаются сдать программы, не обрабатываемые интерпретатором или компилятором.

Поэтому одной из основных целей этого курса можно считать задачу обучения студента проверке (тестированию) программ. У новичков процесс тестирования выглядел обычно следующим образом. Написав программу, студент запускал ее один раз (или несколько, в зависимости от количества частных случаев, рассмотренных в программе) и, получив похожий на правду ответ, считал, что задача решена. Если же программа выдавала неверный ответ, то он пытался найти ошибку и затем заново запускал программу.

Очевидно такая схема написания неудобна. Например, пусть отлаживаемая программа ищет пересечение двух прямоугольников. В процессе проверки нам придется неоднократно (для проверки различных частных случаев) вводить координаты различных прямоугольников (по восемь чисел на тест). Понятно, что необходимо как-то автоматизировать этот процесс. Один из способов — поместить тестовые данные в файл и считывать их оттуда при старте программы. Такой способ позволяет нам экономить на вводе тестовых данных, но этапа сравнения полученных решений с требуемыми при этом способе не избежать. Для решения этой задачи программисты на Ruby обычно используют специальную библиотеку TestUnit

(подобного рода библиотеки есть в большинстве современных языков программирования, в языке Java, например, для этих целей используется пакет JUnit).

Приведем простой пример, в котором протестируем довольно простую программу о разложении чётного числа на сумму простых.

```
def number_decomposition(number)
  num1, num2 = number/2, number/2

  while num1 > 1 and num2 < number
    if prime?(num1) and prime?(num2)
      return num1, num2
    end
    num1 -= 1
    num2 += 1
  end
  return nil
end
if __FILE__ == $0
  p number_decomposition(142)
end
```

Метод number_decomposition использует рассмотренный (стр. 10) нами ранее метод prime? (проверка числа на простоту).

Исходная программа была подготовлена к процессу тестирования. Метод number_decomposition возвращает полученный результат (вместо печати, который мы будем сравнивать с набором подготовленных шаблонов). Тело условного оператора в конце программы будет выполнено только в случае, если интерпретатору на выполнение будет передан данный файл с программой, при подключении этого файла командой require условие `__FILE__ == $0` будет ложно.

Для тестирования создадим подкласс базовой библиотеки тестирования. Обратите внимание на стили наименований: имя такого класса должно начинаться с префикса Test, а имена тестирующих методов — с лексемы test. Каждый тест содержит набор сравнений — специальных функций, имя которых начинается со слова assert.

Метод assert получает два аргумента (второй — необязательный): логическое выражение и текстовое сообщение. Если логическое выражение окажется ложно, то программа добавит 1 к числу неуспешных проверок и выведет сообщение о неудаче. Кроме этого, будет напечатано текстовое сообщение, переданное методу в качестве второго аргумента. Метод assert_equal получает три аргумента (третий — необязательный): ожидае-

мое значение, сравниваемое значение (как правило результат работы тестируемой функции) и сообщение, которое будет выведено при несовпадении первого и второго аргументов. Метод assert_nil проверяет, является ли аргумент объектом nil.

```
# подключение библиотеки тестирования
require "test/unit"
# подключение файла с программой
require "number_decomposition4test.rb"
class TestNumberDecomposition < Test::Unit::TestCase
  def setup
    # набор начальных действий выполняемых
    # перед запуском каждого тестового случая.
  end
  # проверка работоспособности функции prime?
  def test_prime?
    assert(false == prime?(1))
    assert false == prime?(10)
    assert_equal(true, prime?(2), "Ошибка при проверке числа 2")
    assert_equal true, prime?(7), "Ошибка при проверке числа 7"
    assert_equal(true, prime?(103))
  end
  def test_number_decomposition
    assert_nil(number_decomposition(2))
    assert_equal([5, 5], number_decomposition(10))
    assert_equal [7, 11], number_decomposition(18)
    assert_equal [97, 103], number_decomposition(200)
  end
end
```

Вы можете добавлять в класс вспомогательные методы и использовать их в тестовых случаях, но только методы, начинающиеся с test, будут запущены во время выполнения теста! Тесты вызываются в том порядке, в котором они представлены в программе. Если в тестирующем классе присутствуют методы с именами setup и (или) teardown, то они будут выполнены соответственно перед запуском всех тестов и после их завершения.

★ **Очень полезно писать тесты до начала программирования.**

Посмотрим на результат работы данной программы. Его можно интерпретировать таким образом. Выполнено 2 теста (tests) и 9 сравнений (assertions). Обнаружено 0 сбоев (failures) и 0 ошибок (errors). Это означает, что программа теоретически работает и, наверное, все в порядке.

```
$ ruby test_number_decomposition.rb
Loaded suite test_number_decomposition
Started
..
Finished in 0.003189 seconds.

2 tests, 9 assertions, 0 failures, 0 errors
$
```

Ради эксперимента искусственно внесем ошибку в метод prime?, сделав так, чтобы он считал единицу простым числом.

Запуск теста моментально обнаруживает ошибку, показав место, где произошел сбой — седьмую строку с выражением assert false == prime?(1) (отметим, что количество сбоев стало равным одному). Изучив тестовый случай, можно понять, где произошла ошибка и в чем причина. Теперь выполнять тесты стало очень легко.

```
$ ruby test_number_decomposition.rb
Loaded suite test_number_decomposition
Started
.F
Finished in 0.061807 seconds.

  1) Failure:
test_prime?(TestNumDecomposition) [test_number_decomposition.rb:7]:
<false> is not true.

2 tests, 5 assertions, 1 failures, 0 errors
$
```

Когда требуется ввести в программу новую функцию, начните с создания теста. Это не так странно, как может показаться. Когда вы пишите тест, то спрашиваете себя, что нужно сделать для добавления этой функции и как она должна работать.

3.2 Обработка исключительных ситуаций. В большинстве современных языков программирования предусмотрена возможность работы с исключительными (особыми) ситуациями. В языке Ruby программисты могут работать как со встроенными ситуациями, так и с ситуациями, создаваемыми по указанию программиста при наличии того или иного события. Типичные встроенные ситуации — это «деление на ноль» (ZeroDivisionError) или «достижение конца файла» (EOFError).

Перейдем к простому примеру, в котором мы будем обрабатывать исключительную ситуацию «деление на ноль». При запуске данной программы возникает исключительная ситуация (ZeroDivisionError).

```
a = 0
puts 10/a
```

Для реализации механизма исключений в языке Ruby присутствуют следующие ключевые слова: `rescue`, `raise`, `ensure`.

- `rescue` — поймать и обработать исключение, находящееся в блоке.
- `raise` — генерировать исключительную ситуацию.
- `ensure` — всегда выполнить код, заключенный в этот блок.

Обработаем ситуацию, описанную выше, внеся опасный участок кода в блок обработки исключения `rescue`.

Если кроме блока `rescue` присутствует блок `ensure`, то он будет выполнен в любом случае. Поэтому зачастую блок `ensure` используют для освобождения зарезервированных ресурсов. В следующей программе блок `ensure` следит за освобождением дескриптора файла.

```
a = 0
begin
  puts 10/a
rescue ZeroDivisionError
  puts "Произошло деление на ноль"
ensure
  puts "Блок ensure всегда выполняется!"
end
```

```
f = File.open("testfile")
begin   # .. выполнение
rescue  # .. обработка ошибки
ensure  # закрыть соединение с файлом
  f.close unless f.nil?
end
```

★ **Будьте внимательны при использовании исключений. Использование конструкции `rescue` без указания конкретного исключения приводит к обработке всех ошибок.**

Любая библиотека Ruby возбуждает исключения при возникновении любой ошибки, и вы также можете явно возбуждать ошибки в своем коде. Создать исключение программист может с помощью метода `raise` модуля `Kernel` (или его синонима `fail`). Рассмотрим его применение на следующих примерах:

```
raise

raise "Missing name" if name.nil?

if i >= names.size
  raise IndexError, "#{i} >= size (#{names.size})"
end
```

Первая форма просто перехватывает текущее исключение и позволяет тем или иным способом затем обработать его. Старайтесь не использовать форму возбуждения исключения без аргумента, так как она не информативна. Второй пример демонстрирует передачу сообщения при возбуждении исключения. Вид возбужденного исключения в этом случае можно узнать, определив тип переменной **$!**

```
sum = 0
begin
  a.each { |x| sum += 1 }
rescue
  puts sum
end
```

Обратите внимание на следующую программу. Она не должна работать, т.к. объект `a` не проинициализирован, но ...

Бесконечный ввод данных, последовательности

В некоторых ситуациях программист не может сказать, когда закончится процесс ввода данных. Например, на вход поступает некая последовательность чисел. При завершении ввода требуется выдать какую-нибудь характеристику последовательности. В такой систуации уместно использовать обработку исключений (конкретно, ситуацию EOFError). Начнем с ряда простых примеров.

```
sum = 0
begin
  while true
    sum += readline.to_f
  end
rescue EOFError
  puts "Сумма последовательности #{sum}"
end
```

Сумма элементов последовательности

Пусть в задаче требуется вычислить сумму всех элементов последовательности чисел, получаемых из входного потока. Мы вводим данные в бесконечном цикле и при получении нового числа прибавляем его значение к переменной `sum`. При завершении ввода печа-

тается значение переменной `sum`. (При вводе даннных с консоли нажатие Ctrl-D (CTRL-Z в Windows) завершает ввод.)

Обратим внимание на распространенную ошибку. При работе с последовательностью не следует использовать контейнеры (объекты классов `Array` или `Hash`) для ее хранения, так как мы считаем, что элементов может быть бесконечно много и памяти для хранения не достаточно. Следующая программа может аварийно завершиться, если входная последовательность содержит большое число членов.

```
arr = [] # не последовательность
begin
  arr << readline.to_i while true
rescue EOFError
  sum = 0
  arr.each{ |v| sum += v }
  puts sum
end
```

★ **Похожая на `readline` функция `gets` не возбуждает исключительной ситуации при прерывании ввода и, по этой причине, не может использоваться в задачах, требующих обработки исключений.**

Среднее арифметическое элементов последовательности

Для решения этой задачи нам недостаточно одной переменной `sum`, так как функция, вычисляющая среднее арифметическое значение последовательности, не является индуктивной. Потребуется построить её *индуктивное расширение*. Для одной и той же функции f можно придумать разные расширения. Однако, наибольший практический интерес представляет такое индуктивное расширение, которое содержит минимум дополнительной информации.

★ **Разложение функции в композицию индуктивных — творческая задача и применяется только в простейших случаях, где оно более или менее очевидно.**

В данной задаче минимальным индуктивным расширением является объединение двух индуктивных функций — количества элементов последовательности и их суммы. Как правило, решение таких задач сводится к тому, что программист должен понять, какой информации (переменных) ему не хватает, и ввести ее в программу. Более подробно об этом можно причитать в пособии Е.А. Роганова «Основы информатики и программирования» (стр. 132–138).

```
sum, count = 0.0, 0
begin
  while true
    sum += readline.to_f
    count += 1
  end
rescue EOFError
  puts count == 0 ? 0 : sum/count
end
```

3.3 Тестирование последовательностей. Для тестирования программ, работающих с последовательностью, возможно применение следующего приёма. Продемонстрируем его на тестировании программы, находящей среднее арифметическое элементов последовательности.

```
require 'test/unit'
class TestAverage < Test::Unit::TestCase
  def setup    # запуск программы с последовательностью
    @seq = IO.popen("ruby average.rb", "r+")
  end
  def teardown # выключение программы (закрытие потока)
    @seq.close
  end
  # метод направляющий поток данных в тестируемую
  # программу и возвращающий результат в виде строки
  def sequence(*items)
    items.each { |i| @seq.puts(i) }
    @seq.close_write
    @seq.read.chomp
  end
  def test_average
    assert_equal("2.5",sequence(1,2,3,4))
  end
  def test_empty
    assert_equal("0", sequence())
  end
end
```

Отметим что функция `sequence()` возвращает результат выполнения программы `average.rb` в виде строки (выводимой коммандой `puts`). Таким образом, если бы оператор печати выглядет так:

```
    puts "average = #{count == 0 ? 0 : sum/count}"
```

то тестовый случай должен принять вид

```
    assert_equal(sequence(1,2,3,4),"average = 2.5")
```

Задача 1. Найдите максимальный элемент последовательности чисел.

Задача 2. Найдите второй по величине элемент последовательности.

Задача 3. Напишите программу, определяющую, является ли последовательность чисел возрастающей?

Задача 4. Найдите наибольший общий делитель последовательности целых чисел.

Задача 5. Найдите сумму всех попарных произведений элементов последовательности чисел.

4. Стековый калькулятор и языки

Прежде, чем начать рассматривать сам компилятор формул, необходимо ознакомиться с такими понятиями как стек, стековый калькулятор, языки, грамматики, а также компилятор.

Итак, введем определение стека. *Стеком* называется любая структура, в которой могут накапливаться какие-то элементы и для которой выполнено следующее основное условие: *элементы из стека можно доставать только в порядке, обратном порядку добавления их в стек*. Это условие часто называют принципом «последним пришел — первым ушел» или *LIFO* (Last In — First Out).

Элемент стека, который в данный момент можно взять или посмотреть называется *вершиной стека*, максимальное число элементов в стеке — *глубина стека*. Стек, в котором нет ни одного элемента называется *пустым*.

Уже из названия видно, что *стековый калькулятор* использует стек. При выполнении любых арифметический операций (сложить, умножить, вычесть, разделить) стековый калькулятор достает из стека последовательно сначала правый, а затем левый аргументы операции, выполняет операцию и полученный результат помещает в стек.

При выполнении любой арифметической операции число элементов в стеке уменьшается на 1. Попытки выполнить операцию, когда в стеке меньше двух элементов, или показать результат, когда стек пуст, приводят к отказу.

Рассмотрим пример для стекового калькулятора. Пусть требуется получить значение формулы $5{\cdot}(7{+}8){+}25$. Тогда последовательность для стекового калькулятора будет такой: $5, 7, 8, +, \cdot, 25, +$.

Из предыдущего примера видно, что обычная формула преобразуется в некую последовательность для стекового калькулятора. Причем эта последовательность преобразуем мы сами. Наша же задача написать программу, которая делает это для любой формулы автоматически. Такая программа называется *компилятором* с языка арифметических формул на язык понятный стековому калькулятору.

4.1 Рекурсивная реализация компилятора правильных формул. Будем для определенности считать, что язык правильных формул задается грамматикой:

формула→терм | терм+формула | терм−формула

терм→множитель | множитель · терм | множитель / терм

множитель→(формула) | переменная

переменная→ $a \mid b \mid \ldots \mid z$

и рассмотрим реализацию компилятора в соответствии с этой грамматикой.

Правильная формула задается грамматикой с четырьмя метасимволами. Для каждого метасимвола пишется отдельный метод обработки. Например, метод «обработать терм» находит в начале непрочитанной части формулы терм максимальной длины, читает и печатает соответствующий фрагмент последовательности для стекового калькулятора.

В используемой нами грамматике понятие *терм* определяется через понятия *терм* и *формула*. Естественно поэтому попытаться реализовать обработку формулы *рекурсивно*. (Напомним, что *рекурсия* — это ситуация или программистский прием, состоящий в том, что программа непосредственно или через другие программы обращается к себе как к подпрограмме.)

В соответствии с определением формулы при ее компиляции можно сначала найти и откомпилировать терм: либо этот терм совпадает со всей формулой, либо за ним должен следовать знак + или −. Таким образом, после компиляции терма возможны три ситуации:

- в формуле больше символов нет;
- далее идет знак +, а за ним формула;
- далее идет знак −, а за ним формула.

Соответственно в целом обработку формулы можно записать так: обрабатывается терм, затем проводится выбор: если нет непрочитанных элементов, то ничего не делать; если идет знак + или −, то пропустить его, обработать формулу и напечатать соответственно + или −.

Обратите внимание, что в методе «обработать формулу» мы обрабатываем непрочитанную часть формулы не до ее конца, а лишь до конца *правильной* формулы. Это связанно с тем, что далее при обработке множителя в соответствии с правилом:

множитель→(формула) | переменная

мы будем вызывать метод «обработать формулу» для компиляции выражения внутри скобок, а такая обработка должна заканчиваться при достижении закрывающейся скобки.

Наконец, коль скоро мы воспользовались рекурсией, мы обязаны гарантировать, что метод «обработать формулу» не будет вызывать себя бесконечно. В данном случае это действительно так, потому что перед каждым новым вызовом метода «обработать формулу» непрочитанная часть формулы уменьшается, а т.к. она конечна, то и вызовов может быть лишь конечное число. Следовательно, рано или поздно обработка формулы закончится.

Рекурсивный компилятор формул (RecursCompf.rb)

```
class RecursComp
  def compile(str)
    @str,@index = str,0
    compileF
  end

  private

  def compileF
    compileT
    return if @index >= @str.length
    cur = @str[@index].chr
    if cur == '+' or cur == '-'
      @index += 1
      compileF
      print "#{cur} "
    end
  end
  def compileT
    compileM
    return if @index >= @str.length
    cur = @str[@index].chr
    if cur == '*' or cur == '/'
      @index += 1
      compileT
      print "#{cur} "
    end
  end
  def compileM
    if @str[@index].chr == '('
      @index += 1
      compileF
      @index += 1
    else
      compileV
    end
  end
  def compileV
    print "#{@str[@index].chr} "
    @index += 1
  end
end
```

Аналогичным способом в соответствии с грамматикой реализуются методы «обработать терм», «обработать множитель». А метод «обработать переменную» заключается лишь в выводе на экран переменной и пропуске очередного элемента.

Однако, если попытаться откомпилировать нашим компилятором формулу $a - b - c$, то мы увидим, что последовательность символов для стекового калькулятора соответствует формуле $a - (b - c)$, т.е. в этой ситуации компилятор работает неверно (традиционная операция вычитания ассоциативна слева, а у нас скобки «накапливаются» справа)[1].

Обработка ввода формул (RunRecursComp.rb)

```
require "RecursCompf"
c = RecursComp.new
while true
  print "\nВведите формулу -> "
  c.compile(readline.chomp)
end
```

4.2 Реализация компилятора с помощью стека. Грамматику, описанную в предыдущем разделе, легко преобразовать к виду, соответствующему правильному порядку выполнения арифметических действий:

формула→терм | формула + терм | формула−терм
терм→множитель | терм∘множитель | терм/множитель
множитель→(формула) | переменная
переменная→a | b | c | ... | z

Однако рекурсивно реализовать соответствующий этой грамматике компилятор так, как это было сделано раньше, нельзя. Невозможно, например, при обработке формулы по очередному элементу определить, надо ли обрабатывать терм или формулу. Но даже если бы это оказалось возможным, мы бы не имели права в программе обработки формулы рекурсивно обратиться к себе, так как в этот момент непрочитанная часть еще не изменилась и, следовательно, в этом месте программа бы обращалась к себе до бесконечности.

Приведенную выше реализацию компилятора правильных формул можно подправить так, чтобы операции выполнялись в нужном порядке. Заме-

[1] На самом деле ошибка возникла не из-за работы компилятора, а из-за некорректно написанной грамматики, в чем легко убедиться, рассмотрев дерево вывода формул для данной грамматики.

тим, однако, что при этом основное достоинство рекурсивной реализации — простая связь грамматики и программы — будет утеряно. Поэтому мы сначала изменим форму описания языка и лишь потом переделаем компилятор.

Введем новые понятия: аддитивная операция $\oplus = +, -$; мультипликативная операция $\otimes = \cdot, /$; и зададим понятия терма и формулы в новой форме:

формула→терм{$\oplus$ терм }

терм→множитель{$\otimes$ множитель},

где фигурные скобки означают повторение содержимого нуль или более раз. Остальные понятия грамматики, описанной в предыдущем разделе, оставим без изменения. При реализации компилятора в соответствии с новым описанием языка паре фигурных скобок будет соответствовать цикл.

Метод «обработать формулу», например, будет заключаться в следующем: сначала обрабатывается терм, затем начинается цикл, внутри которого происходит выбор: непрочитанных элементов нет → выход из цикла; очередной элемент + → пропустить очередной элемент, обработать терм, напечатать "+"; очередной элемент − → пропустить очередной элемент, обработать терм, напечатать "−"; иначе → выход из цикла.

Аналогичным образом модифицируется метод «обработать терм». При компиляции по новой программе формула $a - b - c$ будет обработана правильно.

Такая реализация компилятора корректно обрабатывает правильные арифметические формулы, однако, чтобы внести какие-нибудь изменения в процесс обработки формулы, надо переписать изрядную часть программы. Поэтому давайте рассмотрим реализацию компилятора правильных формул с помощью стека.

Прежде всего заметим, что любую правильную формулу можно скомпилировать так, что: 1) переменные в последовательности для стекового калькулятора будут идти в том же порядке, что и переменные в формуле; 2) все операции в последовательности будут расположены позже соответствующих знаков операций в формуле. (Этот факт легко доказывается индукцией по числу знаков операций в формуле.)

Таким образом, формулу можно компилировать так: встретив имя переменной, немедленно его напечатать, а встретив знак операции или скобку, печатать те из предыдущих, но еще невыполненных операций (будем их называть *отложенными*), которые выполнимы в данный момент, после чего «откладывать» и новый знак. Поскольку среди оставшихся отложенных операций нет таких, которые выполнимы до пришедшего знака, то для хранения можно воспользоваться стеком (назовем его стеком операций). Этот стек и есть та информация, которая необходима для индуктивной компиляции формулы.

Рассмотрим реализацию стека. Для этого мы создадим класс Stack. Вообще стек реализовать достаточно просто, так как фактически для его правильной работы необходимы всего лишь четыре метода: конструктор; метод, помещающий элемент в стек; метод, берущий элемент из стека, и метод, показывающий вершину стека.

```
class Stack
  def initialize
    @array = Array.new
  end
  def push(c)
    @array.push(c)
  end
  def pop
    @array.pop
  end
  def top
    @array.last
  end
end
```

Класс Compf

Давайте теперь разберем класс Compf — компилятор формул, использующий стек операций. Класс Compf является подклассом класса Stack и имеет методы всех трёх типов доступа: public, protected и private. Компилятор допускает только однобуквенные имена переменных.

```
require 'Stack'
class Compf < Stack
  def compile(str)
    "(#{str})".each_byte { |c| processSymbol(c.chr) }
  end

  private
  def symType(c)
    case c
    when '('
      SYM_LEFT
    when ')'
      SYM_RIGHT
    when '+', '-', '*', '/'
      SYM_OPER
    else
      symOther(c)
    end
  end

  def processSymbol(c)
    case symType(c)
    when SYM_LEFT
      push(c)
    when SYM_RIGHT
      processSuspendedSymbol(c)
      pop
    when SYM_OPER
      processSuspendedSymbol(c)
      push(c)
    when SYM_OTHER
      nextOther(c)
    end
  end

  def processSuspendedSymbol(c)
    while precedes(top, c)
      nextOper(pop)
    end
  end
```

Работа начинается с вызова метода compile, в котором все символы строки str последовательно передаются методу processSymbol.

```
  def priority(c)
    (c == '+' or c == '-') ? 1 : 2
  end

  def precedes(a, b)
    return false if symType(a) == SYM_LEFT
    return true  if symType(b) == SYM_RIGHT
    priority(a) >= priority(b)
  end
  protected

  SYM_LEFT  = 0; SYM_RIGHT = 1; SYM_OPER  = 2; SYM_OTHER = 3

  def symOther(c)
    # Сравнение символа с образцом (регулярным выражением).
    raise "Недопустимый символ #{c}" if c !~ /[a-z]/
    SYM_OTHER
  end

  def nextOper(c)
    print "#{c} "
  end

  def nextOther(c)
    nextOper(c)
  end
end
```

Квалификатор доступа protected и метод nextOther нужны для создания на базе класса Compf нового класса Calc, реализующего калькулятор формул[2].

Класс Calc (калькулятор числовых формул) выведен из класса Compf, переопределяет некоторые методы последнего, и имеет дополнительный стек для размещения в нём чисел. Калькулятор работает только с цифрами (числами от 0 до 9).

Несколько комментариев к методу nextOper(c) класса Calc. Множественное присваивание в первой строке метода корректно, т.к. в языке Ruby при выполнении множественного (параллельного) присваивания *сна-*

[2]Хотя в языке Ruby в данном случае можно убрать "protected", тем самым размещая все нижеописываемые константы и методы в зоне действия квалификатора private, в языках Java и C++ здесь нужен именно квалификатор protected.

чала последовательно вычисляются все выражения в правой части оператора присваивания.

```
require 'Compf'

class Calc < Compf
  def initialize
    # Вызов метода initialize базового класса Compf.
    super
    # Создание стека результатов операций.
    @s = Stack.new
  end

  def compile(str)
    super
    return @s.top
  end

  protected

  def symOther(c)
    raise "Недопустимый символ #{c}" if c !~ /[0-9]/
    SYM_OTHER
  end

  def nextOper(c)
    second, first = @s.pop, @s.pop
    @s.push(first.method(c).call(second))
  end

  def nextOther(c)
    @s.push(c.to_i)
  end
end
```

Конструкция first.method(c).call(second) во второй строке метода может быть объяснена таким примером: выражение 3.metod('-').call(2), эквивалентно выражению 3.-(2) или просто 3-2.

Задача 1. Добавьте операции *sin*, *cos* и унарный минус.

Задача 2. Добавьте правоассоциативную операцию ^ возведения в степень.

Задача 3. Добавьте квадратные и фигурные скобки.

Задача 4. Измените программу так, чтобы допускались в качестве имен переменных произвольные идентификаторы языка Ruby.

Задача 5. Добавьте левоассоциативную операцию % с приоритетом, равным приоритету операции /.

Задача 6. Добавьте возможность записи формулы с пробелами и комментариями двух типов (/* */ и //).

Задача 7. Измените программу так, чтобы ввод, содержащий в качестве аргументов только восьмеричные числа (начинающиеся с нуля, например 056), компилировался в программу, содержащую десятичные числа.

Задача 8. Измените программу так, чтобы ввод, содержащий в качестве аргументов только шестнадцатеричные числа (начинающиеся с $0x$, например $0x56$), компилировался в программу, содержащую восьмеричные числа.

Задача 9. Измените программу так, чтобы ввод, содержащий в качестве аргументов только римские числа, не превосходящие 5000, компилировался в программу, содержащую десятичные числа.

Задача 10. Измените программу так, чтобы для коммутативной операции аргументы выдавались в алфавитном порядке.

Задача 11. Добавьте фигурные скобки, означающие возведение в квадрат. Используйте операцию DUP стекового калькулятора.

Задача 12. Считая, что $a = 0$, оптимизируйте формулу (уберите лишние сложения).

Задача 13. Считая, что $b = 1$, оптимизируйте формулу (уберите лишние умножения).

Задача 14. Добавьте возможность ввода формулы на нескольких строках.

A. Язык программирования Ruby

A.1 Базовые типы. Базовыми типами языка Ruby являются числа, строки (объекты класса `String`), массивы (класс `Array`), диапазоны (`Range`), хэши или ассоциативные массивы (`Hash`), символы (`Symbol`) и регулярные выражения (объекты класса `Regexp`). Любое целое число $x \in \mathbb{Z}$ может быть представлено объектом класса `Fixnum` (если величина $|x|$ не слишком велика) или `Bignum` (иначе), но лишь *конечное* подмножество из несчётного множества действительных чисел $\mathbb{R}$ представимо в виде объектов класса `Float`, часто называемых *числами с плавающей точкой*.

Таблица A.1. Примеры чисел

Выражение	Значение	Комментарий
`123`	`123`	целое число — объект класса `Fixnum`
`-1234567890`	`-1234567890345`	целое число — объект класса `Bignum`
`1_234_567_890`	`1234567890345`	подчёркивания в записи чисел игнорируются
`-123.45`	`-123.45`	«действительное» число (класс `Float`)
`1.2345e+2`	`123.45`	экспоненциальная формы записи
`0xff`	`255`	шестнадцатеричное (hexadeciamal) число
`037`	`31`	восьмеричное (octal) число
`0b1011`	`11`	двоичное (binary) число

Для задания строк можно использовать кавычки (`"`) или апострофы (`'`). В первом случае распознаются и интерпретируются так называемые эскейп-последовательности (например, `\n`, `\"`, `\t`, `\r`) и выполняется подстановка результатов вычисления выражения `expr` вместо подстроки `#{expr}`. В обоих случаях последовательности `\\` и `\'` преобразуются в символы `\` и `'` соответственно. Существуют и другие способы задания строк, некоторые из которых показаны в таблице A.2.

Таблица A.2. Примеры строк

Выражение	Значение	Комментарий
`"2 + 3 = #{2+3}"`	`"2 + 3 = 5"`	подстановка вычисленного выражения
`'2 + 3 = #{2+3}'`	`"2 + 3 = #{2+3}"`	подстановка не выполняется
`%q(Язык Ruby)`	`"Язык Ruby"`	аналог строки в апострофах
`%Q(#{2**32})`	`"4294967296"`	аналог строки в кавычках
`'a\nb'`	здесь четыре символа: буква `a`, символ `\` и буквы `n` и `b`	
`"a\nb"`	всего три символа: буквы `a` и `b` разделены символом `\n`	

Массив (`Array`) в Ruby — это набор (коллекция, множество) произвольных объектов (см. таблицу А.3).

Таблица А.3. Примеры массивов

Выражение	Значение	Комментарий
`[]`	`[]`	пустой массив
`[0]`	`[0]`	массив из одного элемента — числа 0
`[1, 2.3, "Ruby"]`	`[1, 2.3, "Ruby"]`	массив из трёх элементов
`[[1,2],[3]]`	`[[1, 2], [3]]`	массив из двух массивов
`%w(Где ёж?)`	`["Где", "ёж?"]`	способ создания массива строк
`%w(Где\ же он?)`	`["Где же", "он?"]`	экранирование пробела
`%W(2 3 #{2*3})`	`["2", "3", "6"]`	подстановка значения выражения

Диапазон (`Range`) — последовательность объектов, которая включает (для `e1..e2`) или не включает (для `e1...e2`) в себя элемент `e2`. Используемый в качестве итератора диапазон передаёт в блок все свои элементы (как при вызове метода `to_a`, преобразующего диапазон в массив).

Таблица А.4. Примеры диапазонов

Диапазон	Соответствующий ему массив
`1..9`	`[1, 2, 3, 4, 5, 6, 7, 8, 9]`
`1...10`	`[1, 2, 3, 4, 5, 6, 7, 8, 9]`
`3..1`	`[]`
`'d'..'n'`	`["d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n"]`

Хэш (`Hash`)— это набор пар ключ–значение. Хэш схож с массивом, за исключением одной особенности — индексация производится с помощью объектов любых типов, кроме `integer`. Причем порядок обхода элементов не зависит от порядка вставки.

Примеры хэшей приведены ниже:

Таблица А.5. Примеры хэшей

Выражение	Значение	Комментарий
`Hash["a",1,"b",2,"c",3 ]`	`{"a"=>1, "b"=>2,"c"=>3}`	хэш из трех элементов
`Hash["a" => 1, "b" => 2]`	`{"a" => 1, "b" => 2}`	хэш из двух элементов
`{ "a" => 1}`	`{"a" => 100}`	хэш из одного элемента

Как видно из примера, для создания хэша часто используются литералы `key => value`. Ключ и значения находятся в паре, поэтому число

аргументов должно быть четным.

Хэши имеют значение по умолчанию. Это значение возвращается каким-либо итератором при попытке обращения к ключу, не существующему в хэше. И этим значением является `nil`.

Регулярные выражения (объекты класса `Regexp`) используются для подбора шаблона строки. Для создания регулярных выражений нужно использовать литералы /.../ или %r..., а также конструктор `Regexp.new`. Отметим, что разные версии Руби используют разные средства для работы с регулярными выражениями.

При создании регулярных выражений могут идти следующие параметры:

Таблица А.6. Параметры

Параметр	Значение
`/.../i`	`не различать регистр`
`/.../x`	`игнорировать пробелы и переводы строк`
`/.../s`	`считать регулярное выражение`

С помощью регулярных выражений можно:

- Проверять, соответствует ли вся строка целиком заданному шаблону.
- Находить в строке подстроки, удовлетворяющие заданному шаблону.
- Извлекать из строки подстроки, соответствующие заданному шаблону.
- Изменять в строке подстроки, соответствующие шаблону.

Примеры использования регулярных выражений приведены в таблице А7.

Таблица А.7.

Параметр	Значение
`/Abc/`	`совпадет только со словом 'Abc'`
`/Abc/i`	`совпадет со словами 'ABC', 'abc', 'Abc' и т. д.`
`/abc/`	`совпадет с 'abc', 'abc cba'`
`/abc.*def/s`	`совпадет с 'abckghfdkdef'`

Каждый символ регулярного выражения последовательно сравнивается с проверяемой строкой. Все, что не является указанными ниже спецсим-

волами или операторами, воспринимается, как обычный символ, рассматриваемый на простое совпадение.

A.2 Термы и выражения. Термами в языке Ruby являются литералы (объекты базовых типов), результаты выполнения команд операционной системы, генерации символов и вызова методов, а также значения констант и переменных.

Вызов метода `m` объекта `obj`[1] со списком параметров `arg` и блоком `blk` (иначе называемый посылкой сообщения `m` получателю `obj`) записывают в виде `obj.m(arg){blk}` или `obj.m(arg) do blk end`. Для вызовов, выполняемых вне классов («на верхнем уровне»), получателем является `main` — экземпляр класса `Object`, создаваемый при старте Ruby-программы. Примеры вызовов методов приведены в таблице A.8.

В языке Ruby имена используются для ссылок на константы, переменные, методы, классы и модули. В таблице A.9 перечислены зарезервированные слова, которые не могут быть использованы в качестве имён.

Имена констант должны начинаться с большой латинской буквы (от `A` до `Z`), за которыми может следовать любая последовательность больших и малых латинских букв, цифр и символов подчёркивания (`_`).

Переменные в языке Ruby бывают четырёх различных видов: локальные, экземпляра, класса и глобальные. Имена локальных переменных должны начинаться с малой латинской буквы (от `a` до `z`) или символа подчёркивания, за которыми может следовать любая последовательность больших и малых латинских букв, цифр и символов подчёркивания. В именах локальных переменных, состоящих из нескольких слов, рекомендуется использовать подчёркивание, например, `day_week`.

К именам переменных экземпляра вначале добавляется символ `@` (например, `@x`), переменных класса — два таких символа (например, `@@name`), а глобальных переменных — символ `$` (например, `$_`). Некоторые предопределённые объекты имеют имена, отступающие от этого правила.

Методы, не являющиеся переопределяемыми операторами (см. таблицу A.11), должны иметь имя, образованное по тем же правилам, что и имена локальных переменных. К имени метода может быть добавлен восклицательный (`!`) или вопросительный знак (`?`), либо символ `=`. Рекомендуется использовать такие имена для методов, изменяющих объект-получатель (`self`), возвращающих логическое значение и допускающих использование в левой части оператора присваивания соответственно.

Имена классов и модулей являются константами и следуют описанным выше правилам. Рекомендуется для констант, определяемых в классах, ис-

[1] В случае отсутствия явного получателя им является объект `self` — тот экземпляр некоторого класса, в контексте которого происходит данный вызов.

Таблица А.8. Примеры вызовов методов

Вызов	Комментарий
`puts "Здравствуй, мир!"`	Получатель — предопределённый объект `main` класса `Object`. Этот класс включает в себя модуль `Kernel`, имеющий метод `puts`, вызов которого эквивалентен вызову `STDOUT.puts`
`puts`	В отличие от предыдущего случая параметров нет. Результат — вывод символа перевода строки `\n`
`2.+(3)`	Получатель — число 2 (объект класса `Fixnum`). Параметр — число 3. Выражение `2+3` (см. таблицу А.11) эквивалентно данному вызову
`[1,2,3][0]=4`	Получатель — массив `[1,2,3]` (см. таблицу А.11). В результате вызова массив станет равным `[4,2,3]`
`"123".to_i`	Получатель — строка `"123"`. Метод `to_i` класса `String` без параметров возвращает целое число 123
`"123".to_i(8)`	Параметр 8 указывает, что строку надо рассматривать, как число, записанное в восьмеричной системе счисления. Метод возвращает целое число 83
`3.times do |i| p i end`	Получатель — число 3. Параметров нет, но имеется блок. Метода `times` класса `Integer` выполняет этот блок, передавая в него последовательно значения 0, 1 и 2. В результате будут напечатаны три строки
`a=[1,2,3,4,5]` `a.inject(0){|s,x|s+x}`	Получатель — экземпляр `a` класса `Array`, включающего в себя модуль `Enumerable`. Метод `inject` присваивает переменной `s` параметр (0) и вычисляет затем выражение `s+x` последовательно для всех элементов массива `x`, запоминая результат в `s`. Метод возвращает сумму элементов массива (число 15)
`[1,2].to_i`	Получатель — массив `[1,2]`. Так как класс `Array`, его родительский класс `Object` и включённые в них модули не содержат метода с именем `to_i`, то возникает исключительная ситуация `NoMethodError`

пользовать только большие буквы и символ подчёркивания, а при построении имён классов и модулей применять так называемый MixedCase, когда каждое из слов, образующих сложное имя, пишется с большой буквы.

Выражение представляет терм или несколько термов, объединённых с помощью перечисленных в таблице А.11 операторов. Приоритеты операто-

Таблица А.9. Зарезервированные слова языка Ruby

`__FILE__`	`and`	`def`	`end`	`in`	`or`	`self`	`unless`
`__LINE__`	`begin`	`defined?`	`ensure`	`module`	`redo`	`super`	`until`
`BEGIN`	`break`	`do`	`false`	`next`	`rescue`	`then`	`when`
`END`	`case`	`else`	`for`	`nil`	`retry`	`true`	`while`
`alias`	`class`	`elsif`	`if`	`not`	`return`	`undef`	`yield`

ров, разделённых горизонтальными линиями, различны и убывают сверху вниз. Многие из операторов являются методами и могут быть переопределены. Примеры использования операторов приведены в таблице А.12.

Таблица А.10. Некоторые предопределённые стандартные объекты

Имя	Класс	Назначение
`ARGF` или `$<`	`Object`	Объект, предоставляющий доступ к конкатенации всех файлов, заданных в командной строке, или к содержимому стандартного ввода (когда в командной строке нет аргументов)
`ARGV` или `$*`	`Array`	Массив строк, содержащий аргументы командной строки запуска Ruby-программы
`ENV`	`Object`	Подобный хэшу объект, содержащий значения переменных среды (environment)
`DATA`	`IO`	Если программа содержит директиву `__END__`, то `DATA` содержит все строки файла программы, следующие за строкой с директивой `__END__`
`RUBY_PLATFORM`	`String`	Идентификатор платформы (операционной системы с дополнительными характеристиками), на которой выполняется программа
`RUBY_VERSION`	`String`	Версия интерпретатора Ruby
`STDOUT`	`IO`	Стандартный вывод, начальное значение `$stdout`
`__FILE__`	`String`	Имя файла, содержащего выполняемую программу
`__LINE__`	`String`	Номер текущей строки в программе

Таблица А.11. Операторы и их приоритеты

Операторы	Описание	Метод?
`[ ]`	Ссылка на элемент массива или хэша	Да
`[ ]=`	Присваивание элементу массива или хэша	Да
`**`	Возведение в степень	Да
`! ~ + -`	Отрицание, дополнение, унарные + и -	Да
`* / %`	Умножение, деление, нахождение остатка	Да
`+ -`	Сложение, вычитание	Да
`>> <<`	Сдвиги вправо, влево	Да
`&`	Побитовое «И»	Да
`^ |`	«Исключительное Или», «Или»	Да
`<= < >= >`	Операторы сравнения	Да
`<=> == ===`	Проверки на равенство	Да
`!=`	Проверка на неравенство	Нет
`=~`	Сравнение с образцом	Да
`!~`	Сравнение с образцом	Нет
`&&`	Условное «И»	Нет
`||`	Условное «Или»	Нет
`.. ...`	Операторы создания диапазонов	Нет
`? :`	Тернарный оператор `if-then-else`	Нет
`= %= ~= /= -= +=`	Присваивание и присваивания с операцией	Нет
`|= &= >>= <<=`	Присваивания с операцией	Нет
`*= &&= ||= **=`	Присваивания с операцией	Нет
`defined?`	Проверка: определён ли символ?	Нет
`not`	Логическое отрицание	Нет
`or and`	Логические «Или » и «И»	Нет
`if unless`	Условные выражения и модификаторы	Нет
`while until`	Условные выражения и модификаторы	Нет
`begin end`	Оператор создания блока	Нет

Таблица А.12. Примеры использования операторов

Выражение	Комментарий
`a, b = b, a`	Множественное (параллельное) присваивание позволяет легко обменять значения переменных
`b += c`	Присваивание с операцией допустимо не только для сложения (см. таблицу А.11) и эквивалентно `b = b + c`
`a<=>b`	`-1`, `0` или `1`, если `a` меньше, равно или больше `b`

`c = if a < 0`
`  b = 0`
`elsif a ==0`
`  b = 1`
`else`
`  b = 2`
`end`
	Оператор **`if-then-elsif-else-end`** может иметь много **`elsif`** частей, истинность условий в каждой из которых проверяется последовательно (если условие **`if`** части не выполнено). Выражения, вычисляемые при выполнении условий, можно размещать на той же строке после **`then`** или двоеточия. Условный оператор возвращает значение последнего вычисленного выражения

`b=if 2<3:4 else 5 end`	Переменная `b` станет равна 4
`b=2<3 ? 4 : 5`	Тернарный оператор `?:` делает то же самое
`b=-1; b=0 if b<0`	Модификатор **`if`** часто удобнее
`b=-1; b=0 unless b>=0`	Оператор **`unless`** проверяет *ложность* условия

`b = i = 0`
`c = while i < 5`
`  b += i`
`  i += 1`
`end`
	Оператор **`while`**, называемый циклом, в отличие от условного оператора *не возвращает значения*. В результате выполнения данной программы переменная `i` станет равна 5, `b` примет значение 10 (сумма всех чисел от 0 до 4), а переменная `c` — значение **`nil`**

`i=1;i+=i*i while(i<9)`	Модификатор **`while`** сделает переменную `i` равной 42
`i=1;i+=i until(i>10)`	После завершения этой программы `i` станет равно 16, ибо **`until`** выполняет тело цикла, пока условие *ложно*

`s = 0`
`for i in 1..3`
`  s += i`
`end`
	Неявно преобразуется в
	`s = 0`
	`(1..3).each do |i|`
	`  s += i`
	`end`

В дополнение к этому **`break`** немедленно прекращает выполнение цикла, передавая управление на следующую за ним инструкцию; **`redo`** начинает выполнять цикл или итератор сначала, но не перевычисляет условие продолжения (для цикла) и не переходит к следующему элементу коллекции (для итератора); **`next`** прерывает выполнение текущей итерации и начинает выполнение следующей; **`retry`** начинает выполнять цикл или итератор с самого начала.

Литература и гиперссылки

[1] `http://www.ruby-lang.org/en` — Ruby language home page.

[2] `http://www.rubycentral.com/book` — Гипертекстовая версия первого издания книги [3].

[3] Thomas D., Fowler C., Hunt A. *Programming Ruby. The Pragmatic Programmers' Guide. Second Edition.* — Dalals: The Pragmatic Bookshelf, 2004.

[4] Black D. A. *Ruby for Rails. Ruby Techniques for Rails Developers.* — Greenwich: Manning Publications Co., 2006.

[5] Кормен Т., Лейзерсон Ч., Ривест Р. *Алгоритмы. Построение и анализ.* — М.: МЦНМО, 2000, 2004.

[6] `http://www.main.msiu.ru` — Информационный портал МГИУ.

[7] Роганов Е. А. *Основы информатики и программирования*: *Учебник.* — М.: МГИУ, ????.

[8] Роганов Е. А., Тихомиров Н. Б., Шелехов А. М. *Математика и информатика для юристов*: *Учебник.* — М.: МГИУ, 2005.

Предметный указатель

Три шага к свободному ПО

Не все знают, что стоимость проприетарного (коммерческого) программного обеспечения, способного превратить «компьютерное железо» в современный компьютер, значительно больше стоимости самого «железа». Не знают зачастую потому, что используют контрафактное (то есть приобретённое без лицензии) ПО, что является нарушением действующего законодательства.

Последние несколько лет отмечены небывалым ростом интереса к свободным программным продуктам, которые могут быть приобретены бесплатно или за символическую цену, и причин у этого явления достаточно много. Кроме очевидных экономической (существенная разница в цене) и «политической» (независимость от конкретной фирмы-производителя), очень важны также наличие общедоступных исходных текстов программ (что является гарантией отсутствия «закладок») и возможность создания дистрибутивов, содержащих множество свободных программ, которые могут быть установлены на компьютер все сразу (в случае проприетарных программных продуктов каждый из них необходимо устанавливать отдельно, затрачивая на это немало времени). Свободное ПО сейчас лучше превращает компьютерное «железо» в удобный инструмент для работы, учёбы и отдыха, чем широко распространённые в России программные продукты от фирмы Microsoft. Оно является более надёжным и защищённым, почти не подвержено атакам компьютерных вирусов, а работать с ним ничуть не сложнее, чем с каким-либо иным.

К настоящему времени в целом ряде стран решения об отказе от массового использования проприетарных продуктов приняты на правительственном уровне. Среди них Германия, Франция, Индия, Китай, Япония и другие. Мы гордимся тем, что Учёный Совет МГИУ ещё несколько лет назад принял решение о всемерной поддержке перехода от проприетарного ПО к свободному. Реализуя это решение, информационно-вычислительный центр МГИУ разработал программу «Три шага к свободному ПО», основное назначение которой — познакомить пользователей компьютеров с миром свободного программного обеспечения и дать возможность переходить к его применению постепенно и безболезненно, шаг за шагом.

Шаг 1 — FSF-Windows. Знакомство с лучшими свободными программами, устанавливаемыми на компьютер с операционной системой Windows.

Шаг 2 — VMware ASPLinux. Установка на компьютер с ОС Windows эмулятора, позволяющего запускать «почти настоящий Linux», не нуждающийся в дополнительной настройке.

Шаг 3 — MSIU ASPLinux. Переход к использованию доработанного в МГИУ дистрибутива ОС Linux, который может быть установлен на компьютер без удаления имеющейся версии Windows.