Программирование на Ruby

(краткий курс для начинающих)

Язык программирования Ruby превосходит все другие современные языки по своей мощи, простоте и красоте, а потому познакомиться с ним должен каждый программист, опытный или начинающий. Единственный недостаток - программы на Ruby работают на интерпретаторе. Тем, кто пока не знаком с этим термином, на этот факт можно не обращать никакого внимания, впоследствии вы всё это узнаете. (Справедливости ради надо отметить, что Ruby очень многое позаимствовал из более раннего и тоже замечательного языка Perl. Сейчас появилась новая версия этого языка - Perl6, которая по некоторым своим возможностям способна потягаться с Ruby). Существует огромное количество книг по Ruby, но как обычно для литературы подобного рода, все они ужасно многословны и перегружены подробностями, которые начинающий программист не в состоянии сразу освоить, а тем более применить на практике. Изучение языка должно начинаться с базовых понятий с постепенным переходом к более сложным конструкциям в процессе реального программирования. В этом руководстве я буду знакомить читателя с основами языка без нудных рассуждений и деталей, которые сразу всё-равно не понять и не освоить. Хорошо, если вы уже программировали на каком-нибудь другом языке и умеете выполнять программы на своём компьютере, но в принципе это не обязательно. В некотором отношении даже лучше начинать изучение программирования с освоения такого языка, как Ruby.

Поскольку никакой язык нельзя освоить без составления программ и выполнения их на компьютере, в заключение этой вводной части укажу, что надо сделать, чтобы запустить свою программу:

1.Установить интерпретатор Ruby, который легко найти в интернете. Всё программное обеспечение Ruby бесплатно (это было непременным условием с самого начала жизни языка). Установка сводится к скачиванию файла и двойному щелчку на его названии.

2.Написать свою программу на каком-либо текстовом редакторе (всё-равно каком, но очень удобен *Geany*) и записать её в любую папку с расширением в названии *.rb,* например *proba.rb.*

*3.*В командной строке *(cmd)* перейти в эту папку (команда *cd*).

4.Напечатать: *ruby proba.rb.* Сразу увидите результат выполнения программы, или сообщение об ошибках, если они будут. (При использовании *Geany* программу можно запускать прямо из текстового редактора).

Кроме того, имеется интерактивный режим выполнения кода. Для его вызова — команда *irb*, после чего каждая введённая строка кода будет

сразу выполняться с выводом результатов.

1.Простейшие программы

По стилю программирования различают процедурные, объектно-ориентированные (ООП) и функциональные языки. Ruby универсален и позволяет программировать в любом стиле, но основным для него является ООП. Легче всего освоить процедурный стиль. В этом случае программа выполняется строчка за строчкой сверху вниз. Например:

```
x = gets
x = Float(x)
y = 2 * x**3 / 5  + Math.sin(x)
puts(y)
```

Здесь *gets* - встроенная в язык (стандартная) функция, позволяющая вводить текст с клавиатуры (может быть вам потребуется писать *STDIN.gets*). Строка *x = gets* означает, что переменной *x* присваивается тот текст, который мы напечатаем на клавиатуре. Даже если это число, например 56, всё-равно оно будет рассматриваться, как текст, который по правилам языка записывается в кавычках: *"56"*. Для перевода его в число применяем принудительный перевод типа с помощью строки *x = Float(x)*. Слово *Float* означает, что теперь переменная *x* будет иметь тип "число с плавающей запятой". В следующей строке выполняем вычисления по некоторой формуле, мало отличающейся от обычной математической формулы (*x**3* означает возведение *x* в третью степень). *Math.sin(x)* это вычисление тригонометрического синуса, все математические функции вызываются из отдельной библиотеки с названием *Math*. *puts* - тоже стандартная функция, которая выводит на экран всё, что передаётся ей в списке аргументов (у нас только переменная *y*). После запуска программы появится окно командной строки и программа будет ждать ввода, после чего получим результат.

Даже эту примитивную программу можно записать короче, Ruby всегда предоставляет нам много разных вариантов:

```
include Math
x = Float gets
y = 2 * x**3 / 5  + sin(x)
p y
```

После ввода строчки *include Math* можно писать просто *sin*, *cos*, *log* и так далее. Скобки после *Float* можно опускать, если нравится.

Эту строку можно также написать и в таком виде: *x = gets.to_f*, где *to_f* - встроенная функция (чаще их называют методами), делает то же, что и *Float*. В строке *p y* (можно писать *p(y)) p* - тоже функция вывода, вообще говоря, работает несколько иначе, чем *puts*, но в данном случае не будет никакой разницы. Есть ещё и другие функции вывода: *print, printf, sprintf*, мы с ними ещё встретимся.

Можно создавать свои собственные функции (методы), для этого используется служебное (ключевое) слово *def*

```
def f(x,y)
        z = (x+y)/2.5
        c = x*y
        s = z+c
end
a = f(2.3, 7)
puts a          #получим 19.82
```

Здесь *f* - название функции, а *x* и *y* - аргументы. *end* обозначает конец функции, а всё, что между именем и *end* называют телом функции. Во многих языках тело заключают в фигурные скобки. Строка *a = f(2.3, 7)* осуществляет вызов функции и передаёт ей аргументы, то-есть переменной *x* присваивается значение 2.3, а *y* будет равна 7. Результат присваивается переменной *a*, соответственно *puts a* выводит его на экран.

Запись "*#получим 19.82"* представляет комментарий, отмечаемый знаком *#*. Всё что расположено после этого знака до конца строки не обрабатывается и служит только, как пояснения для лучшей читаемости текста программы.

Функции в Ruby всегда возвращают последнее вычисленное значение, то-есть обращение к нашей функции вернёт нам значение, равное *z+c*. Потому мы и можем написать *a = f(2.3, 7)*. Введённая нами переменная *s* таким образом никак нами не использована, она даже вне тела функции вообще недоступна, поэтому её надо убрать. Да и промежуточные переменные *z* и *c* можно исключить, сгруппировав формулу:

```
def f(x,y)
        (x+y)/2.5 + x*y
end
a = f(2.3, 7)
puts a       # -> 19.82 (знак  -> вместо слов получим или равно)
```

Ясно, что переменную *a* тоже можно было не применять, написав:

puts f(2.3, 7). Бывает, что в функции требуется возвращать несколько значений. Тогда эти значения надо помещать в массив, или применить служебное слово *return*. (О массивах позже).

```
def f(x,y)
   a = "Hello"
   z = (x+y)/2.5 + x*y
   return a, z   # -> можно просто [a, z], без return
end
puts f(2.3, 7)    # -> ["Hello", 19.82]
```

В процедурном стиле можно создавать программы для решения многих задач, используя замечательные возможности Ruby. Для иллюстрации я составил программу для решения систем линейных алгебраических уравнений методом Гаусса с выбором главного элемента. Если проявите интерес и посмотрите, как эта программа выглядит на других языках, например, на Паскале, то увидите, что на Ruby она существенно короче и разобраться в ней постороннему читателю на Ruby проще. Многое тут пока будет непонятно для вас, но об этом мы будем говорить позже.

```
a=[[3.0,1,2,4,11],[8.0,3,5,1,20],[-12.0,2,1,4,-3],[1.0,3,7,2,16]]
def amax b
      ma=0; ima=0
      b.each_index {|i| r=b[i][0].abs; ma, ima=r, i if r>ma;}
       b[0], b[ima]=b[ima], b[0]
       r=b[0][0]
       b[0].map! {|x| x/r}
       for i in (1...b.length) do
             r=b[i][0]
             b[i].each_index {|j| b[i][j]=b[i][j]-b[0][j]*r}
       end
       b
end
b=a; n=a.length
(0...n).each {|i| b=amax b; a[i]=b[0]; b=b-[b[0]];
b.each_index {|j| b[j]=b[j]-[b[j][0]]}; }

a.reverse!

x=[]; x[0]=a[0][1];
```

```
(1...n).each {|i| sum=0; (0...i).each{|j| sum=sum+x[i-j-1]*a[i][j+1]};
x[i]=a[i][i+1]-sum;}

x.reverse!
p x      # -> [1, 2, 1, 1] это решение системы из 4 уравнений (массив a)
```

2.Типизация

Всякая программа обрабатывает какие-то данные (какую-нибудь информацию). Эти данные фигурируют в программе в виде переменных и констант. В зависимости от вида данных применяются переменные разных типов. Для чисел в Ruby имеются следующие типы: целые числа - тип *Integer,* числа с плавающей запятой - тип *Float,* рациональные дроби - тип *Rational,* комплексные числа - тип *Complex.* Число будет считаться целым, если в нём только цифры и знак числа: *x = 567, y = -23*. Числа с плавающей запятой можно писать в виде: *x = 456.055* или *x = 0.456055e3* , (такую форму программисты обычно называют "научной нотацией"), что будет одно и тоже. Ясно, что допустимо и такое число: *y = -0.0035,* или *y = -0.35e-2.* Для того, чтобы 7 было типа *Float* , достаточно написать 7.0 (0 после точки нельзя опускать). Ruby, таким образом, не требует явно указывать тип переменных, как это принято во многих других языках. Но явная типизация возможна и иногда её приходится применять для устранения неопределённости. Можно писать *x = Integer 56, y = Float 34.05*. Для изменения типа переменной можно использовать два способа записи (говорят "две различные нотации"): *a = Float x,* или *a = x.to_f.* Теперь будет *a = 56.0.* Соответственно *b = Integer y,* или *b = y.to_i,* что даст b = *34.* Дробная часть отбрасывается без округления числа. *to_f* и *to_i* стандартные функции (методы), а запись с точкой *x.to_f* означает, что к объекту *x* применяется метод *to_f*, то-есть *x* здесь является аргументом метода *to_f*. Кстати, такая нотация допустима и для методов, создаваемых с помощью слова *def*:

```
def func
    x = self
    puts x
end
45.func    # -> 45
```

Следовательно, аргумент (у нас число 45) доступно в теле метода

через служебное (зарезервированное) слово *self*. Это слово всегда представляет тот объект, для которого вызывается метод - в нашем случае это как раз целое число 45.

Тип *Rational* позволяет работать с рациональными дробями. Запись *x = Rational(3, 5)* задаёт (инициализирует) значение *x = 3/5*. Над рациональными числами можно выполнять арифметические операции, что позволяет проводит расчёты без потери точности (иногда это может быть очень полезно). Подобным же образом используется тип *Complex*, позволяющий работать с комплексными числами. Более подробно эту тему не будем обсуждать.

Обозначения (идентификаторы) переменных всегда должны начинаться с буквы в нижнем регистре (или со знака _). Если первую букву сделать заглавной (в верхнем регистре), то получим константу: *A = 45.87*. Теперь эту величину в программе невозможно изменить.

Важно отметить, что арифметические операции над целыми числами всегда дают в результате целое число, поэтому *9/2* будет равно *4*, а *11/3 даст 3*, то-есть округление не выполняется. Легко можно допустить ошибку, поэтому надо быть внимательным. Чтобы получит результат типа *Float* достаточно в выражении сделать хотя бы одну величину с плавающей точкой: *9.0/2* будет равно *4.5*. Для гарантии иногда бывает полезно применить явное преобразование типа: *x.to_f/y* или *Float(x)/y*. Теперь *x* может иметь любой тип *Integer* или *Float* (или даже тип *String*, то-есть, например, "*56*") результат будет один и тот же и всегда с типом *Float*.

Возможны различные способы присваивания значений переменным (инициализации переменных). *x, y, z = 5, 2.3, -4.88* это так называемое «групповое присваивание», здесь переменные получают соответствующие значения. Легко задать взаимный обмен значений для двух переменных без ввода вспомогательной переменной: *x, y = y, x*. Допустима такая конструкция: *x = y = z = 5* → все переменные получают одно и то же значение. *x+= 3* → то же самое, что и *x = x + 3*, то-есть целая переменная *x* увеличивает своё значение на *3*. Соответственно, и для вычитания: *x-=1 -> x=x-1*.

Как и любой другой язык программирования, Ruby позволяет работать с двоичными, восьмеричными и шестнадцатеричными числами и опытный программист должен знать эту арифметику.

Программы работают не только с числами, но и с текстами. В этом случае используются строковые (или текстовые) переменные,

имеющие тип *String*. Значениями этих переменных является набор знаков, заключённых в кавычки. x = *"Hello"* - здесь текстовая переменная *x* инициирована значением "*Hello"*. Применяется и одинарные кавычки: *x = 'Hello'*. Между этими нотациями есть разница.

Сделаем небольшое отступление. Язык Ruby относится к ООП, в нём всё является объектом какого-то класса (подробнее о классах дальше). Так, целые числа всегда объекты класса Fixnum, если их абсолютное значение не превышает некоторого предела; числа больше этого предела - объекты класса Bignum. Чтобы узнать, к какому классу относится объект, применяется метод *class*. Сделаем в интерактивном режиме *irb* такие действия: *x = 67; x.class -> Fixnum; x = 9999999999999999; x.class -> Bignum*. В классе *Bignum* числа могут быть как угодно большими и Ruby сам выбирает класс для целых чисел, нам не надо об этом заботиться. Для *y = 44.0065* получим *y.class -> Float*, значит числа с плавающей запятой относятся к классу *Float*. Сделаем то же для текстовых переменных: *s = "Hello"; s.class -> String; s1 = 'Hello'; s1.class -> String*. Значит обе этих формы принадлежат к классу *String*. Разница между ними в том, что в одинарных кавычках переменной передаётся весь текст без изменения, так, как он записан, а в двойных кавычках - имеется возможность применять так называемую "интерполяцию", которая очень полезна при организации вывода данных. Покажем это на примере: пусть будет *x = 123.405305*. Запрограммируем вывод этой переменной в такой форме: *puts("Variable x = #{x}")*. Получим: *Variable x = 123.405305*. То-есть, в тексте внутри кавычек для того, что содержится внутри фигурных скобок с предшествующим знаком # происходит вывод значений переменных величин. На этом месте могут быть даже и выражения, которые будут вычислены перед выводом: *puts("Result = #{x*2}")* → *Result = 246.81061*. Метод *printf* позволяет форматировать выводимую информацию: *printf(“x = %7.3f”, “#{x}”)* будет выведено: *x = 123.405*. Здесь после % первая цифра *(7)* указывает, сколько всего отводится позиций для выводимого числа, а вторая (3) — сколько вывести знаков после десятичной точки. При этом выполняется округление требуемое числа, то-есть для *x=123.405703* получим *x = 123.406*. Буква *f* указывает на тип переменной; для целых чисел надо писать *d* , а для текстовых переменных — *s*. Формат можно сохранять в текстовой переменной для повторного использования:

frm = “x = %7.3f”; printf(frm, “#{x}”) → 123.406. Применение формы ”#{}” для вставки данных в текст и называется интерполяцией. Форматирование позволяет придавать выводимой информации нужный порядок и красоту. Отметим, что в Ruby есть ещё и дополнительные средства форматирования. В частности можно вставлять так называемые «управляющие символы», которые записываются в виде обратной косой черты и буквы. Например ”\n” означает перевод строки: *printf(“x = %7.3f\n”, “#{x}”)* → теперь после выведенной строки будет выполнен переход на новую строку.

Имеется очень много средств для работы с текстовой информацией. Например, метод *reverse* позволяет изменить порядок символов на обратный: *x = “Hello”; x.reverse → “olleH”*. Метод *to_s* позволяет числа преобразовывать в текст:
x = 45.098; x.to_s -> "45.098". О методах работы с переменными типа *String* немного поговорим позже. Отметим пока, что Ruby, как и многие другие языки, позволяет использовать для обработки текста так называемые «регулярные выражения». Эта тема весьма обширна и здесь мы не будем её рассматривать, но впоследствии этот мощный инструмент для работы с текстом необходимо изучить. По этой теме есть огромное количество руководств, доступных в интернете.

Существует в Ruby тип, принадлежавший классу *NilClass* (означающий фактически «ничего»), соответствующий неопределённому состоянию переменной: *x = nil; x.class → NilClass*. Такой тип иногда бывает нужен при работе программы; дальше мы это увидим.

3.Массивы, хэши, диапазоны.

Массивы используются во всех языках программирования — это упорядоченные наборы некоторых данных. В Ruby создать массив очень просто: *a = [1, 2, 3, 4, 5]; a.class → Array.* Следовательно, переменная, представляющая массив относится к классу *Array.* Члены массива обычно называют его элементами. В нашем примере элементы — целые числа. Массив на Ruby может содержать элементы любого типа и даже в одном массиве типы элементов могут быть разными: *b = [5, “Peter”, 23.87, 43]*. В массиве *b* есть элементы типа *Integer, String* и *Float*. Каждому элементу ставится в соответствие целое число, называемое индексом элемента. Индексы просто равны номеру элемента, при этом нумерация начинается с нуля. По индексу можно извлечь нужный элемент из массива с помощью такой записи:

b[0] → 5; b[2] → 23.87; b[1] → “Peter”. Чтобы изменить элемент, надо инициализировать его новым значением: *b[3] = “Anna”*. Можно добавлять в массив новые элементы: *b[4] = 0.023*. Если вызвать несуществующий элемент, его значение будет *nil*: *b[7] → nil*. Для объединения двух массивов в один новый массив используется знак +. то-есть тот же самый, что и для сложения чисел:
a=[1,2,3]; b=[4,5]; c=a+b -> [1,2,3,4,5].
Для удаления элемента применяется знак - (минус):
d=c-[c[3]] -> [1,2,3,5]. e=c[2,3] -> [3,4] - так можно получить часть (срез) массива, указав с какого и до какого индекса выделить элементы. Такой срез может быть удалён *g=c-c[2,3] -> [1,2,5]*.
Попробуйте понять самостоятельно, почему при удалении одного элемента надо писать *[c[3]]*, а для удаления среза *c[2,3]*, то-есть не надо заключать срез в квадратные скобки.
Можно удалять любые элементы, при этом удалаются все дублирующиеся: *[1, 1, 2, 2, 3, 3, 4, 5] - [1, 2, 4] # -> [3, 3, 5].*
Добавлять элементы в массив можно с помощью знака <<:
[1, 2] << "c" << "d" << [3, 4] # => [1, 2, "c", "d", [3, 4]] .
Последний элемент в получившемся массиве - другой массив.

При инициализации массива с одними только текстовыми элементами приходится ставить много кавычек:
a=["alpha", "beta", "gamma", "zeta"], что довольно утомительно. Для удобства есть такой способ: *a=%w[alpha beta gamma zeta]*, дающий тот же результат. То-есть если перед квадратными скобками поставить знаки *%w*, то не надо писать кавычки, а вместо запятых надо ставить пробелы. Такого рода фокусы программисты называют "синтаксический сахар", подобного "сахара" в Ruby предостаточно.

Имеется множество различных методов (имею ввиду функции) для работы с массивами, кое-что рассмотрим далее. Отметим, что текстовые переменные в некоторых случаях подобны массивам. Так символы в тексте также имеют индексы и можно извлекать их или делать срезы:
s="Hello, world!"; s[0] -> "H"; s[4] -> "o"; s[2,8] -> "llo, wo"; и так далее.
Также, как и массивы, можно объединенять строки с помощью знака + (сложение), обычно эту операцию называют конкатенацией:
"Anna" + "Bob" -> "AnnaBob" . Отметим, что удалять отдельные символы или срезы с помощью знака вычитания для текстов невозможно.

Приведём пример:

```
def func(x,y)
    x+y
end
puts(func(23, 1.77)) -> 24.77
puts(func([2, 3, 4], [7, 1]))  -> [2, 3, 4, 7, 1]
puts(func("alpha", "beta")) -> "alphabeta"
```

Созданный нами метод *func* работает для чисел, массивов и текстов потому, что к ним применим один и тот же знак сложения.

Как уже видели, элементами массива могут быть другие массивы: *a = [[1,2,3], [7,6,2,4], ["a", "b", "c", "d"]]*. Для извлечения элемента из такого массива надо указывать два индекса: *a[2][3] -> "d"*. Таким способом можно создавать двумерные матрицы, для трёхмерных матриц потребуется три индекса.

Некоторые примеры методов для массивов:

a.first -> [1,2,3] - возвращает первый элемент массива.

a - a.first -> [[7,6,2,4], ["a","b","c","d"]] - так можно удалить первый элемент массива.

Метод *last* делает тоже самое с последним элементом.

a.pop ->["a","b","c","d"] - возвращает последний элемент массива и одновременно удаляет этот элемент из массива.

Очень полезный метод *sort* позволяет отсортировать массив (расставить элементы в порядке возрастания):

a = [9, 7, 56, 3,2]; b = a.sort -> [2, 3, 7, 9, 56];

b.reverse -> [56, 9, 7, 3, 2].

Некоторые методы могут быть записаны с восклицательным знаком на конце, например *a.sort!* Если восклицательного знака нет, то метод возвращает отсортированный массив, которому можно дать другое название, как это сделал я выше. При этом исходный массив *a* не изменяется и может использоваться снова в том же виде. Если знак ! поставить, то отсортированный массив будет на месте исходного массива, то-есть наш массив *a* изменится и его первоначальный вид будет навсегда утерян. Точно также можно использовать метод *reverse!* и многие другие методы. Таким образом, применять методы со знаком ! на конце надо осторожно.

Метод *max* позволяет найти максимальный элемент в массиве: *a.max -> 56*. Есть, разумеется, и метод *min*.

Для того, чтобы узнать размер (число элементов) массива применяется метод *length: a = [0,1,2,3,4,5]; a.length ->6*. Вместо

length можно применять метод *size: a.size -> 6*, эти методы являются синонимами, подобных синонимов в Ruby очень много, дальше я не буду их всех перечислять.

Хеши (часто их называют "ассоциированными массивами" или даже "словарями") имеют более сложную структуру, чем массивы. Обычная форма для них выглядит так:
h = {1 => "Peter", 5 => "Anna", 77 => "Marta"}. (Скобки фигурные). Первый элемент в каждой паре называют "ключ", а второй - "значение". От массива хеш отличается тем, что роль индекса играет не порядковый номер, а ключ: *h[5] -> "Anna"; h[77] -> "Marta"*. В роли ключей и значений могут быть объекты любого типа, например *h1 = {"a" => 5.78, 6 => [1,2,3], "age" => 25}; h1["age"] -> 25* и так далее. В примере одним из значений является массив. В хеш можно добавлять элементы: *h1["c"] = "Hello"*. Для хешей тоже имеется синтаксический сахар: если ключи имеют тип *String*, хеш можно инициализировать так: *h2 = {a:7.89, alpha:"Hello", c:78}*. Если *h2* вывести на экран с помощью *puts*, то получим:
{:a => 7.89, :alpha => "Hello", :c => 78}. Такой синтаксис, когда перед набором знаков стоит двоеточие, определяет объекты особого рода, называемые "символами" (не надо путать с буквами). В руководствах найдёте подробное описание этой замысловатой конструкции, но пока нам это не нужно. Можно бы и вообще о символах пока не упоминать, но они употребляются в другом, очень важном контексте, как мы скоро увидим. Да и в хешах использование символов в качестве ключей очень удобно.

Многие методы для массивов работают и для хешей: *h2.length -> 3*, но метод *reverse*, например, не работает. Метод *to_a* позволяет хеш преобразовать в массив в таком виде:
h.to_a -> [[1, "Peter"], [5, "Anna"], [77,"Marta"]]. Для хешей работает метод *first*, а метод *last* не работает.

Диапазоны (или ранги)- набор натуральных чисел, задаваемых начальной и конечной границей: *d = (2..5)*. Метод *to_a* позволяет преобразовать диапазон в массив: *d.to_a -> [2,3,4,5]*, вторая форма диапазона не включает верхнюю границу: *d1 = (2...5); d1.to_a - > [2,3,4]*, (обратите внимание на количество точек). Диапазоны можно задавать и для букв алфавита: *("a".."g")* - все буквы от "*a*" до "*g*". Как увидим далее, диапазоны бывают очень удобны для организации циклов.

4.Логические операции и ветвление в программах

Совокупность логических операций называют булевой алгеброй. Логические (булевы) переменные могут иметь только два значения: *true* - истина и *false* - ложь. Эти значения возвращают "условия" или "сравнения". В Ruby любое выражение возвращает результат, что справедливо и для сравнения двух величин с помощью знаков == (равно), != (не равно), > (больше), < (меньше), >= (больше - равно) и <= (меньше - равно). Поэтому можем написать:
x=5; y=9; b=(x<y) ->true (скобки на обязательны). *b1= x>y -> false.*
Логические операции над булевыми переменными имеют следующие обозначения: *&&* - логическое "и", || - логическое "или", ! - логическое "нет" (отрицание). *b && b1 -> false; b || b1 -> true; !b -> false; !b1 -> true*.
Отметим, что сравнивать по величине можно не только числа, но и строки, величина которых определяется расположением в алфавите первого знака текстовой величины. Если первые знаки совпадают, сравниваются вторые знаки, и так далее, так же, как мы поступаем при составлении сортированного списка фамилий.
"Anna" > "Marta" -> false; Anna" < "Marta" -> true. По этому правилу сортируются, например, массивы с текстовыми элементами (метод *sort*).

Имеется, кроме того, очень полезный оператор сравнения, для которого принят знак <=>. Например, для двух переменных *a1* и *a2* он работает так: *c = (a1 <=> a2) -> c = -1*, если *a1 < a2*; *c = 0*, если *a1 = a2*, и *c = 1*, если *a1 > a2*. Анализируя величину *c*, мы узнаем, как соотносятся по величине переменные *a1* и *a2*.

Для программирования ветвления в программах используются специальные (условные) операторы. В Ruby есть несколько способов для этого. Прежде всего это оператор *if*:

```
puts "Введите номер дня недели (1 - 7)"
n = gets.to_i
if n<6 then puts("Рабочий день")
    else puts("Выходной")
end
```

Здесь *if, then, else* - служебные слова (*then* не обязательно). Как и у функций, *end* указывает на окончание условного оператора if. Если условие *n<6* выполняется (*true)*, будет выполнен вывод строки "*Рабочий день*" и произойдёт переход на строку программы после *end*. Если условие *n<6* даст *false*, выведется строка "Выходной".

Замечание: может быть, что в конкретных условиях программного обеспечения на вашей машине возникнет проблема с выводом русского текста (кириллицы). Тогда придётся менять кодировку. Эта тема достаточно сложная и несколько запутанная, но если вы решили всерьёз заняться программированием, то впоследствии вам придётся в этом разобраться. А пока, если не сможете выводить русский текст, пользуйтесь одной только латиницей.

Иногда бывают цепочки ветвлений, то-есть после слова *else* может быть новое *if*. Для сокращения записи введено объединённое слово *elsif:*

```
puts "Введите номер дня недели (1 - 7)"
n = gets.to_i
if n<6
   puts("Рабочий день")
   elsif n==6
       puts("Буду отсыпаться")
    else puts("Пойду в театр")
end
```

Если введём число 7, то оба условия *n<6* и *n==6* дадут *false* и значит будет выведено "Пойду в театр".

Поскольку оператор *if* возвращает значение, то нашу программу можно записать и так:

```
puts "Введите номер дня недели (1 - 7)"
n = gets.to_i
text = if n<6
        "Рабочий день"
    elsif n==6
        "Буду отсыпаться"
    else
        "Пойду в театр"
end
puts text
```

Для программирования ветвления почти во всех языках есть конструкция *case*, позволяющая выбирать один вариант из нескольких. Приведём пример с использованием сравнения <=>:

```
puts "ВВедите номер дня недели (1 - 7)"
n = gets.to_i
text = case n<=>6
         when -1
            "Рабочий день"
```

```
        when 0
          "Буду отсыпаться"
        when 1
          "Пойду в театр"
end
puts text
```
Здесь *when* также служебное слово. Принцип работы оператора *case* очевиден.

На месте условия в этом случае может стоят любая величина:
```
s=gets.chop
case s
    when "one"
       puts("Единица")
    when "two"
       puts("Двойка")
    else
      puts("Что-то другое")
end
```
Здесь вводится текст, который затем сравнивается со словами *"one"* и *"two"*. При вводе мы применили метод *chop*, который удаляет хвостовой символ перевода строки; без этого сравнение будет работать неверно. Иногда на это обстоятельство надо обращать внимание. Отметим, что на Ruby имеется большое количество различных вариантов использования *case*.

Интересно, а что будет, если на место условия подставить какую-нибудь величину, например число или строку?
```
if 76.87 then puts "Foo"; end    #-> "Foo"
```
Будет выведено "Foo", потому, что число в этом контексте приравнивается *true*. В Ruby все значения, даже и ноль, дают *true*. И только *nil* и *false* трактуются, как *false*. Иногда это обстоятельство полезно использовать, в руководствах много соответствующих примеров.

Кстати, в последнем примере мы записали два оператора в одной строке. Это допустимо, надо только операторы разделять точкой с запятой. Ею можно заканчивать и все строки программы; в некоторых языках это обязательно, но Ruby позволяет ; в конце строки опускать. Ruby вообще очень демократичен.

Ещё одна форма условного выражения (называется "модификатор") имеет вид:

x=gets.to_i
puts("Выходной") if x >= 6
Вывод на экран будет выполнен, если условие даст *true*. Ветвь *else* в модификаторе недопустима.

Наконец, есть ещё так называемый "тернарный" оператор сравнения:
x = gets.chomp.to_i
z = x==1 ? "Petja" : "Vanja"
p z
Здесь если условие перед знаком вопроса будет выполнено, то оператор возвращает значение, расположенное перед двоеточием, если нет - то значение после двоеточия. В строке *x = gets.chomp.to_i* мы применили два метода последовательно: сначала удаляем хвостовой символ перевода строки, а потом переводим тип из *String* в *Integer*. На Ruby допустимо применять целые цепочки методов. (Вообще-то в данном случае метод *chomp* можно опустить как мы это уже делали выше, так как метод *to_i* сам удаляет хвостовой символ перевода строки).

5.Циклы

Циклы - важнейшее понятие в программировании. На Ruby можно программировать циклы многими способами. Как и в других языках, имеется оператор цикла *for*. Покажем на примере: пусть где-то в программе определён массив:
a = %w[alpha beta Marta monday Vasja] и требуется вывести на экран все элементы массива последовательно друг за другом:
```
for x in a do
    print "#{x} "
end
```
Здесь *in* служебное слово, указывающее, что переменная *x* последовательно принимает значения, равные элементам массива. do и *end* определяют начало и конец тела цикла. Всё, что запрограммировано в теле будет последовательно повторяться, пока не закончится массив. Обратите внимание, что при интерполяции мы ввели пробел между закрывающей фигурной скобкой и кавычками, и это обеспечивает вывод пробела между словами. Если тут поставить запятую *"#{x},"* то выводимые слова будут разделяться запятыми. Напишем программу иначе:

```
a = %w[alpha beta Marta monday Vasja]
for i in (0...a.length) do
  printf("a[%2d] = %7s\n", "#{i}","#{a[i]}")
end
```
Теперь мы ввели переменную *i*, которая будет последовательно принимать значения из диапазона *(0...a.length)* и использовали эту переменную в качестве индекса для массива *a*. Постарайтесь самостоятельно понять, почему в диапазоне надо ставить три точки, а не две. Выполните программу и проанализируйте результаты, попробуйте изменить формат вывода.

Циклы можно создавать с помощью оператора *while*:

```
i =0
while i < a.lehgth do
   printf("a[%2d] = %7s\n", "#{i}","#{a[i]}")
   i+=1
end
```
Теперь мы самостоятельно изменяем переменную цикла *i*, начиная с 0 и каждый раз увеличивая на 1. Оператор *while* повторяет выполнение цикла до тех пор, пока условие не выдаст *false*.

Приведём ещё один пример с использованием оператора бесконечного цикла *loop*:

```
i = 0
n = a.length - 1
loop do
   printf("a[%2d] = %7s\n", "#{i}","#{a[i]}")
   i+=1
   break if i>n
end
```
Мы применили выход из цикла с помощью стандартного оператора *break* с модификатором *if*.

Есть ещё и другие способы создания подобных циклов. Но самые замечательные возможности организации циклических вычислений предоставляются при использовании "итераторов". Итераторы были придуманы уже давно, но в Ruby они доведены до совершенства. Есть целый набор методов, создающих итераторы; основной из них *each*. Вот как он работает для массивов (применим всё тот же массив *a*):

```
a = %w[alpha beta Marta monday Vasja]
a.each { |x| puts(x) }   #-> на печать все элементы массива.
```
То, что располагается в фигурных скобках называется "блок". В

прямых скобках указывается переменная (у нас *x*), которая последовательно приравнивается элементам массива, для которого вызван метод . Дальше в блоке может быть один или несколько операторов (отделять друг от друга знаком точка с запятой), выполняющих какие-то действия над этой переменной. Последний результат этих действий возвращается блоком (у нас вывод на печать элементов массива). Как видим, итератор позволяет весь цикл запрограммировать одной строчкой, просто фантастика!
Есть также итератор *each_index*, работающий несколько иначе:

```
a=[1,2,3,4,5]
a.each_index {|i| puts a[i]}   # → 1, 2, 3, 4, 5
```

В этом случае переменная в прямых скобках принимает значения индекса, а не самих элементов массива.
Приведём ещё пример:

```
ar = [1.2,  4,  5.2,  2.1, 8]
b = []
ar.each { |y| b+= [y**2]}
puts b    # ->[1.44, 16, 27.04, 4.41, 64]
```

Сначала создаём пустой массив *b=[]*, а затем в блоке прибавляем к нему элементы, равные квадрату элементов массива *ar.* То же самое можно сделать с помощью итератора *map*, который автоматически создаёт новый массив:

```
ar = [1.2,  4,  5.2,  2.1, 8]
b=ar.map { |x| x**2}
print b
```

Итератор *map* имеет вариант *map!*; в этом случае новый массив создаётся на месте исходного массива:

```
ar = [1.2,  4,  5.2,  2.1, 8]
ar.map! { |x| x**2}
print ar    # -> [1.44, 16, 27.04, 4.41, 64]
```

Итератор *times* использует блок, в котором может быть произвольный код:

```
x = []; i = 1; n = 5
n.times { x+= [i * i]; i+= 1 }
print x   # -> 1, 4. 9, 16, 25
```

Переменная *n* задаёт число повторений цикла, в каждом из них выполняется блок.
Итераторы можно применять к диапазонам:

```
(1..5).each { |j| puts j**2 }  # -> 1, 4, 9, 16, 25
```

и к хешам:
h = {a:1, b:2, c:3, d:4, e:5}
*h.each { |i, x| printf("%3s => %5d\n", i, x**2) }*
результат будет выведен в таком виде:
a => 1
b => 4
c => 9 - и так далее.

Отметим, что блок в итераторах может быть многострочным и тогда удобнее ограничивать его не фигурными скобками, а служебными словами *do* и *end*, как это мы делали выше. Обе эти формы равнозначны.

Иногда бывает полезен итератор *upto*. Покажем его работу на примере вычисления факториала числа
fact = 1
1.upto(5) { |x| fact= x}*
puts fact #-> 120 (5! = 120)
В выражении *1.upto(5)* единица задаёт начальное значение для *x*, а пятёрка - конечное. Если в первой строке положить *fact = 0*, в результате получим нуль. Напоминаю, что *fact*= x* то же самое, что и *fact = fact*x*. Заменив знак умножения * на знак сложения + и приняв начальное значение равным нулю, получим сумму чисел.

Итератор *step* делает то же, что и *upto*, но позволяет работать с числами типа *Float* .
0.step(Math::PI, 0.1) { |x| puts x}
Здесь 0 - начальное значение для *x*, а *Math::PI* (это число π) - конечное. Параметр 0.1 задаёт шаг изменения для *x* в цикле.

Итератор *select* позволяет выбирать элементы из массива по какому-нибудь признаку:
a = [4,7,3,8,13,14]
ev = a.select { |x| x%2==0 }
puts ev # -> [4,8,14]
x%2 это целочисленное деление, дающее в результате остаток от деления переменной *x* на два, то-есть для чётных чисел получим ноль. Следовательно, наша программа отбирает из массива *a* чётные элементы, помещая их в новый массив. Применив *select!*, результат получим на месте исходного массива, то-есть, массив *a* будет изменён.
А вот как можно вычислить факториал с помощью итератора *inject*:
*fact = (1..5).inject(1.0) { |p, x| p*x }*

```
puts fact    # -> 120.0
```
Разберитесь в этом примере самостоятельно.

Имеются ещё и другие итераторы, понять их работу и использовать на практике совсем не сложно.

6.Классы и объекты

Самый модный стиль в наше время — объектно-ориентированное программирование. Основное понятие в ООП — класс. В руководствах найдёте много научных определений как классов, так и самого ООП. Практически от них нет никакой пользы. Например, основными характеристиками ООП считают инкапсуляцию (сокрытие), наследование и полиморфизм (множественность). Первые два термина определяют очень простые вещи, с которыми мы скоро познакомимся, а третий — трактуется каждым программистом по-своему.

Без заумных рассуждений можно сказать, что класс — это некоторый фрагмент кода программы, в котором определяются общие характеристики какой-нибудь группы предметов, людей, явлений и вообще чего угодно. Лучше всего показать на примере. Будем, например, рассматривать группу студентов. Каждый студент обладает многими параметрами или свойствами. Для простоты ограничимся всего тремя: имя, возраст, а также умение выполнять какое-то действие. Ну, например, пусть каждый студент умеет складывать два числа (надеюсь, что они это на самом деле умеют). При этих предпосылках можно создать класс, характеризующий всех студентов группы, он будет выглядеть так:

```
class Student
    name = ""
    age = 0
    def f(x,y)
       z = x + y
    end
end
```

Класс всегда начинается со слова *class*, после чего идёт данное нами имя класса, всегда с заглавной буквы. Заканчивается класс словом *end*. В других языках тело класса, как и тело функций, заключают в фигурные скобки. Не знаю, что лучше; обычно от обилия фигурных скобок просто рябит в глазах, но и слово *end* в Ruby тоже несколько надоедает. Правда, скобок в два раза больше, так как каждой

закрывающей соответствует открывающая. Для того, чтобы не путаться, какое *end* к чему относится, следует писать текст с отступами, или лесенкой, наподобие стихов Маяковского.

В классе мы ввели текстовую переменную *name* (пока мы её инициировали пустой строкой) и численную *age* (пока приравняли к нулю). В ООП переменные часто называют «полями». Мы также создали метод *f*, принимающий два параметра *x* и *y* и вычисляющий их сумму, обозначенную буквой *z*. Сам по себе класс бесполезен, но можно создавать «объекты» класса с помощью стандартного метода *new*:

```
st = Student.new
```

st — это имя объекта, а тот код, который мы создали в классе определяет функциональность объекта. Теперь мы можем заставить студента, для которого мы создали объект, выполнить сложение двух чисел, то-есть метод *f* доступен:

```
a = st.f(3.6, 12.07)   # → выполнить метод f объекта st
puts a # → 15.37
```

Но если мы попытаемся *присвоить* конкретное имя для этого студента, то-есть написать *st.name = «Peter»;* то ничего не получится. В отличие от метода *f*, ни по записи, ни по чтению поля *name* и *age* вне класса не доступны. Эта недоступность как раз и называется мудрёным словом "инкапсуляция ". Вообще это свойство очень полезно, так как гарантирует невозможность вмешательства в состояние полей объекта извне, но зато, когда доступ необходим, приходится применять специальные меры.

Чаще всего, передаваемые объекту параметры приходится использовать не в одном методе, как в нашем примере, а во многих. Тогда принятый нами способ передачи параметров методу неудобен, так как требует повторно перечислять весь их список для каждого метода. В Ruby имеется специальная функция *initialize*, позволяющая создавать так называемые "переменные экземпляра" или "атрибуты" объекта. Для доступа к полям класса как по чтению, так и по записи применяется специальная директива attr_accessor (есть директивы доступа только по чтению и только по записи). С учётом сказанного, снова напишем наш класс:

```
class Student
   attr_accessor :name, :age
   def initialize(name, age, x, y)
      @name, @age, @x, @y = name, age, x, y
```

```
    end
    def f
        @x + @y
    end
end
st = Student.new("Peter", 19, 45, 9)
puts st.name   # -> "Peter"
st.name = "Anna"
puts st.name   # -> "Anna"
puts st.f      # -> 54
```

В директиве *attr_accessor* указывается список переменных, для которых мы желаем иметь доступ извне. В этом списке названия переменных должны иметь вид "символов", о которых я говорил раньше, то-есть перед названиями надо ставить двоеточие. Почему тут надо было применять такой синтаксис знает, по-видимому, только автор языка японец Мац. Пока не следует над этим ломать голову, надо просто запомнить и пользоваться. Сами переменные экземпляра (атрибуты) имеют обычные идентификаторы с добавлением впереди знака @. Функция *initialize* принимает список параметров, причём параметры передаются функции методом *new*, создающим объект, и происходит это автоматически в момент создания. Функция initialize в Ruby подобна так называемым «конструкторам» в других языках ООП: C++, Java, Perl и так далее. В теле функции требуется присвоить атрибутам значения, равные полученным параметрам. В принципе для атрибутов можно применить новые обозначения, но практически это не имеет смысла. Всё это пока кажется несколько запутанным, но надо просто усвоить, что тут нет никакой мистики, запомнить эти правила и потренироваться, создавая разные варианты программ.

Теперь можно убедиться, что переменные действительно доступны извне. Так строка *puts st.name* возвращает нам введённое значение для переменной *name* и здесь не надо ставить знак @ впереди. Строки

```
st.name = "Anna"
puts st.name   # -> "Anna"
```

убеждают нас, что переменные доступны и по записи. Наконец, мы можем видеть, что переменные *x* и *y* доступны для метода *f*, и их не надо передавать функции в форме исходных параметров.

Кроме переменных экземпляра можно создавать переменные класса, которые доступны для всех объектов класса. Посмотрите на

пример:

```
class Cl
   def f x
      @@x = x
   end
   def g
      @@x
   end
end
a=Cl.new  # → создаём объект класса Cl.
a.f 5        # → инициируем переменную класса @@x.
puts a.g    # → 5
b=Cl.new  # → создаём ещё один объект класса Cl.
puts b.g    # → 5
```

Таким образом, идентификаторы переменных класса имеют в начале два символа @@. С помощью функции *f* мы можем присваивать значение переменной класса, а с помощью функции *g* читать её значение. Можно видеть, что значение переменной класса, присвоенное в объекте *a*, в объекте *b* тоже доступно. Переменные класса иногда полезны, например, с их помощью можно запрограммировать счётчик для числа созданных объектов класса.

Если ограничится созданием одного экземпляра класса и использовать только свойство инкапсуляции, то не стоило бы городить весь этот огород. Тем более, что в современных "процедурных" языках роль таких одиночных "экземпляров класса" вполне могут играть подпрограммы (процедуры), в которых тоже имеет место инкапсуляция данных. Главная ценность ООП в том, что можно создавать много объектов класса, передавая каждому объекту свои, только ему присущие параметры. Совокупность этих объектов представляет собой новый тип данных, совершенно равнозначный стандартным типам, о которых говорилось выше. Так если мы проверим, к какому типу принадлежит объект *st*, созданный в нашем примере, то получим:
st.class # → Student То-есть тип объекта имеет то же название, что и породивший этот объект класс.

Теперь мы можем создать свой объект класса для каждого студента группы и эти объекты будут совершенно независимы друг от

друга, сохраняя неизменной информацию о каждом студенте:
st1 = Student.new("Anna", 18, 2, 5)
st2 = Student.new("Bob", 25, 13, 7.8)
st3 = Student.new("Peter", 21, 6.8, 9.05) # и так далее.
Попробуем как-то использовать эти объекты. Создадим такой хеш:
grp = {st1.name => st1, st2.name => st2, st3.name => st3}
В качестве ключей мы применили имя студентов, а значениями являются объекты с информацией о каждом студенте. Теперь мы можем легко эти данные извлечь для использования:
print grp["Anna"].age, "\n" # → 18
print grp["Anna"].f, "\n" # → 7
print grp["Peter"].age, "\n" # → 21 Ну, и так далее.
Конечно, у нас слишком уж скудные данные о студентах. Но ничто нам не мешает ввести множество параметров, характеризующих студентов, например сведения об их успеваемости. А в методе *f* мы могли бы запрограммировать какие-то расчёты, например, вычисление какого-то среднего балла, характеризующего успехи каждого студента. Наконец, мы могли бы добавить в класс и другие методы. Тогда наш хеш стал бы почти настоящей базой данных. Попробуйте запрограммировать всё это, например, на языке Basic, и тогда поймёте, насколько велика мощь Ruby.

При том способе передачи параметров функциям, который мы применяли до сих пор, надо, чтобы порядок расположения параметров был одинаковым при вызове функции и в списке параметров в самой функции. При большом количестве параметров это может быть не очень удобно. Можно сделать порядок расположения параметров при вызове функции произвольным. Для этого можно, например, использовать хеш. Посмотрите пример:

```
class Foo
   def initialize h
      @x,@y,@z = h[:x], h[:y], h[:z]
   end
   def f
      puts @x,@y,@z   #  какие-то действия над @x,@y,@z
   end
end
hash = {y:43, z:92.7, x:27.05}
ob = Foo.new hash
```

ob.f # -> 27.5, 43, 92.7
Можно, разумеется, при написании хеша использовать нотацию *"y" => 43* и так далее.

При передаче параметров функции можно запрограммировать их значения "по умолчанию". Это бывает полезно, когда при многократном обращении к функции нам надо изменять не все, а только часть параметров. Вот как делается передача "по умолчанию":

```
def fun(x, y, z=76.5, w=56)
   puts(x, y, z, w, "\n")
end
fun(-4.5, 78)  # → -4,5, 78, 76,5, 56
fun(5, 23.01, 89, 123)  # → 5, 23.01, 89, 123
```

Таким образом, если в списке некоторым параметрам присвоены значения, то их можно не указывать при вызове функции. В то же время при необходимости мы можем их изменить. Надо только параметры "по умолчанию" всегда располагать в конце списка.

Приведу здесь ещё один пример на использование хеша:

```
def funk(word)
   c = Hash.new(0)
   for w in word
      c[w] += 1
   end
   c
end
p funk(["world", "hello", "abc", "hello", "on", "home", "hello"])
```

Эта программа выведет такой результат:
{ "word" => 1, "hello" => 3, "abc" => 1, "home" => 1 }
Предлагаю самостоятельно разобраться, как это работает. Если всё будет понятно, можете считать, что работу с хешами вы усвоили.

7.немного о наследовании.

Пусть у нас имеется созданный ранее класс *Animal*, в котором поля и методы содержат информацию об общих свойствах всех животных вообще. Допустим, что мы захотели создать класс для какого-то конкретного животного, например, класс *Elephant*, характеризующий слонов. Ясно, что слон в том числе обладает и свойствами, характерными для всех животных вообще и для того, чтобы в новом классе не повторять описание этих общих свойств, мы можем

запрограммировать наследование класса *Elephant* классу *Animal*. Это будет выглядеть так:

```
class Animal
    # какие-то методы и поля класса Animal
end

class Elephant < Animal
    # какие-то методы и поля класса Elephant
end
```

Знак < означает, что класс *Elephant* ("дочерний", или "подкласс") наследует классу *Animal* ("родительский" или "суперкласс"). При этом все поля и методы родительского класса доступны в дочернем классе. Наследование позволяет использовать программный код, разработанный раньше вами же, или кем-то другим. Имеется также возможность изменять ("переопределять") методы родительского класса, если они нас не удовлетворяют по каким-то причинам. Для этого достаточно в дочернем классе написать новый метод под тем же именем, что и в родительском. Ruby позволяет даже переопределять встроенные в язык (стандартные) методы, что вообще уж очень необычно. Массивы принадлежат классу *Array.* Добавим в этот встроенный класс свой метод *find*:

```
class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
  end
  return nil
  end
end
puts [1, 3, 5, 7, 9].find {|v| v * v > 30 } # => 7
```

Поскольку класс стандартный, не требуется создавать его экземпляр с помощью *new,* так как любой массив уже является объектом класса *Array,* можно сразу к массиву применять созданный нами метод *find*. Метод *size (*или *length)* в цикле *for* автоматически применяется к нашему массиву, а для выбора элементов массива используем ссылку *self*. Кроме того, к методу *find* мы присоединили блок, который вызывается оператором *yield* (об *yield* смотрите дальше). Наконец, мы

применили выход из цикла по условию с помощью директивы *return*, возвращающей найденное значение. Если условие не будет выполнено (например, для массива *[1, 3, 5]*), метод вернёт значение *nil*. Как много различных возможностей Ruby продемонстрировали мы в такой короткой программе.

Существует множество библиотек готовых программ, которые можно использовать без ограничений. Можно также создавать свои собственные библиотеки. Если библиотека оформлена в виде модуля (детали не станем здесь рассматривать), то для её использования надо вставить в свою программу директиву *include* как мы это сделали в примере выше: *include Math*. После слова *include* указывается название модуля без кавычек. Если интересующий нас программный код содержатся в файле с расширением *.rb*, то-есть это просто написанная ранее нами же или кем-то другим программа, то надо использовать директиву *require*, например, *require "prog"* (*prog* – название файла – *prog.rb*). То-есть в этом случае название файла должно быть в кавычках (можно и в одинарных). После директивы *require* весь программный код загружается в нашу программу и может использоваться, как свой собственный.

Замечание: в учебниках по программированию всегда рекомендуют для обозначений переменных (идентификаторов) и для названий функций применять понятные слова, то-есть названия величин или тех действий, которые должны выполнять функции. Например, если вводится переменная для скорости автомобиля, то надо писать что-нибудь вроде *car_speed = 70* или *carSpeed = 87*.9 и тому подобное. В некотором отношении это правильно, поскольку облегчает чтение программы самим её автором или посторонним человеком. (Многие, правда считают, и я, кстати, тоже, что посторонние никогда чужих программ, кроме разве что простейших, не читают, так как разобраться в чужом тексте сложнее, чем самому написать его заново). По моему, злоупотреблять длинными названиями всё-таки не следует; это так сильно загромождает текст, что усложняется восприятие основного каркаса программы. Чаще всего лучше применять обозначение из одной-двух букв. Ту же скорость автомобиля я бы обозначил буквой *v*, ну, или *va*.

8.Динамический аспект.

Ruby, как и некоторые другие языки позволяет вмешиваться в работу программы, меняя её код по ходу выполнения, как говорится "на лету". Имеется несколько возможностей для этого. Рассмотрим только самый простой вариант, как обычно, на примере:

```
def seg(m,n,c)
   i=0
   while i<n do
      yield(i,m,c)
```

```
        i+= 1
    end
end
seg(5,3,1) { |i,m,c| puts(i*m+c) }    # -> 1,6,11
```

Всё здесь нам уже должно быть понятно, кроме загадочной директивы *yield*. Означает она следующее: если где-то в теле функции поставить эту директиву, то к обычному вызову функции (у нас *seg(5,3,1)*) надо приставить блок (у нас *{ |i,m,c| puts(i*m+c) }*). Тогда во время выполнения функции на место этой самой *yield* будет подставлен заданный нами блок. При этом параметры, указанные для *yield* (у нас *yield(i,m,c)*), будут переданы в блок на место переменных в прямых скобках (у нас *|i,m,c|*). Ясно, что тут мы могли бы применить и другие обозначения, например написать: *{ |a,b,d| puts(a*b+d) }*, ничего бы не изменилось. Таким образом, в нашем цикле происходит вычисление по формуле при изменяющейся переменной *i* и печать полученных результатов. Что нам это даёт? Только то, что если мы захотим проводить вычисления по другой формуле, то нам не надо ничего менять в функции, а ведь она может быть расположена где-то, может быть даже в другой программе, достаточно только изменить блок. Напоминаю, что блок может быть большим и сложным, состоять из многих строк (тогда лучше вместо скобок *{}* ограничить его словами *do* и *end*). В принципе есть возможность менять содержимое блока в программе, но для этой цели удобнее второй способ:

```
bloc = [Proc.new {|x| print x, " " }, Proc.new {|x| print x*x, " "}]
def f pr
  for i in (1..5)
     pr.call i
  end
end
print(f bloc[0])   # → 1 2 3 4 5
print "\n"
print(f bloc[1])   # → 1 4 9 16 25
```

В этом случае мы используем встроенный класс *Proc*, объектами которого могут быть блоки, передаваемые функции *new* в качестве аргумента. Мы создали два разных блока, представленные элементами массива *bloc*. Если мы хотим использовать блоки в функции, то их надо передавать ей в качестве аргументов. В теле

функции блок может быть вызван встроенным методом *call*, которому в свою очередь передаются аргументы для блока. В нашем примере мы передаём функции *f* по очереди два блока из массива *bloc*, меняя таким способом её функциональность. (Можно также сохранять блоки в хеше). Обладая такой возможностью, программа становится динамичной, фактически способной видоизменяться в зависимости от внешних обстоятельств. Есть многие другие возможности подобного рода.

9.Функциональный стиль программирования

Для общего впечатления немного о функциональном стиле, который в принципе можно использовать на Ruby. В этом случае основной приём в программе — функция, возвращающая некий результат. Ruby годится для этой цели потому, что в нём всё хоть что-нибудь да возвращает. Для организации циклов в данном случае используют так называемую "рекурсию" — приём, когда функция сама вызывает себя, заставляя повторять одни и те же действия. Поскольку по ходу можно какие-то параметры менять, то можно организовать обычный цикл с выходом из него по условию достижения изменяемых параметров некоторых значений. Напишем для примера программу вычисления суммы элементов массива:

```
def su(a)
    if a == [] then 0
        else a[0] + su(a - [a[0]])
    end
end
puts su([1,2,3,4,5])  # -> 15
```

Функции *su* в качестве аргумента передаём массив. Если он не пустой, то берётся первый (нулевой) элемент массива и происходит новое обращение к функции *su*, которой передаём тот же массив, но уже без первого элемента. Во втором проходе будет получена сумма двух первых элементов, а затем опять всё повторяется. Когда массив сделается пустым, ветвь *else* не будет выполнена и цикл закончится, а в сумму попадёт её начальное значение равное нулю. Можете проверить — если вместо 0 подставить, например, 3, получим 18. Наверняка некоторые детали тут пока вам не понятны, не обращайте внимания.

Есть языки для программирования исключительно в

функциональном стиле. Самый известный из них — Haskell. Когда вполне освоите Ruby и вам захочется попробовать что-нибудь новое, рекомендую познакомиться с Haskell. Увидите, что это нечто совершенно другое. Программировать в этом стиле намного труднее, но зато и очень увлекательно — иногда приходится решать настоящие головоломки.

Заключение

На этом позвольте мне закончить очень краткое знакомство с Ruby. Тема эта огромна, кроме того язык постоянно развивается. Мы не затронули здесь очень многие аспекты языка. Повторюсь ещё раз — на начальном этапе бессмысленно пытаться вникнуть во всё и сразу. Изучение такого языка, как Ruby потребует многих усилий. И главное — тренировка в написании своих собственных программ. Если имеются трудности в подборке задач, возьмите любую книгу по программированию и там найдёте массу всяких задач от простейших до более сложных.