МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Национальный исследовательский Нижегородский государственный университет
им. Н.И. Лобачевского

А.В. Шиндин

ЯЗЫК ПРОГРАММИРОВАНИЯ МАТЕМАТИЧЕСКИХ ВЫЧИСЛЕНИЙ JULIA. БАЗОВОЕ РУКОВОДСТВО

Учебно-методическое пособие

Рекомендовано методической комиссией радиофизического факультета
для студентов ННГУ, обучающихся по специальностям
03.03.03 «Радиофизика», 02.03.02 «Фундаментальная информатика и
информационные технологии», 10.05.02 «Информационная безопасность
телекоммуникационных систем»

Нижний Новгород
2016

УДК 519.682
ББК 22.19
Ш-62

Рецензент: д.ф.-м.н., профессор **А.В. Якимов**

Ш-62 Шиндин А.В. ЯЗЫК ПРОГРАММИРОВАНИЯ МАТЕМАТИЧЕСКИХ ВЫЧИСЛЕНИЙ JULIA. БАЗОВОЕ РУКОВОДСТВО: Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2016. – 24 с.

Пособие посвящено новому свободно распространяемому интерпретируемому языку программирования Julia, созданному для математических вычислений. Julia обладает рядом преимуществ по сравнению с традиционно используемыми в математике и физике языками (MATLAB, PYTHON, FORTRAN, C). Среди них высокая скорость, богатая библиотека функций, поддержка большого числа команд распределенных вычислений и др.

Пособие предназначено для студентов радиофизического факультета ННГУ, курсовые и дипломные работы которых связаны с задачами математического моделирования или обработки экспериментальных данных. Предлагаемое к ознакомлению руководство содержит сведения по установке интерпретатора языка Julia, принципах работы, основных конструкциях и возможностях языка.

Ответственный за выпуск:
заместитель председателя методической комиссии радиофизического факультета ННГУ, д.ф.-м.н., доцент **Е.З. Грибова**

УДК 519.682
ББК 22.19

ВВЕДЕНИЕ

Большинство задач, которые встают перед исследователем-физиком в настоящее время так или иначе сводятся либо к обработке экспериментальных данных, либо к математическому моделированию физического процесса. Для решения обеих этих задач имеется широкий спектр различных аппаратных средств, начиная от персональных компьютеров и заканчивая вычислительными кластерами и суперкомпьютерами. Что же касается программного обеспечения, то можно выделить целый класс языков программирования, предназначенных специально для численных расчетов. По сравнению с языками общего назначения они обеспечивают простой (зачастую интуитивно понятный) синтаксис программ, а также большую библиотеку специализированных функций. Все они являются интерпретируемыми, что ускоряет реализацию и отладку алгоритмов, но негативно сказывается на скорости работы программ. Наиболее популярными среди таких языков являются MATLAB, а также его свободные реализации Octave и Scilab. Основная особенность этих языков — широкие возможности по работе с матрицами. Также постепенно завоевывает популярность в научной среде язык программирования общего назначения Python вместе с дополнительными модулями NumPy и SciPy. Обладая минималистичным синтаксисом, автоматическим управлением памятью, а также рядом других особенностей, он обеспечивает широкие возможности для использования различных парадигм программирования.

Общей чертой упомянутых выше программных пакетов является тот факт, что большинство специализированных функций реализовано в них через вызовы предварительно скомпилированных библиотек, написанных на языке C. Причиной этому служит желание разработчиков повысить производительность программ. Зачастую, если перед исследователем стоит задача обеспечить максимально возможную скорость работы своей программы, ему приходится переписывать программу на одном из традиционно применяемых статических языков (C/C++, Fortran). В итоге, если исследователь не является профессиональным программистом, как правило используется следующий цикл разработки: 1) создание прототипа программы в одной из специализированных сред программирования (MATLAB/Octave/Scilab, модули Numpy и Scipy языка Python, Ruby, R и другие); 2) отладка и тестирование программы с возможной модификацией алгоритма; 3) переписывание программы на C/C++ или Fortran для повышения производительности. Полученную C-программу уже можно использовать на любом доступном вычислительном оборудовании, включая кластеры и суперкомпьютеры. Очевидно, что необходимость переписывания программы создает дополнительные сложности для исследователя.

Первая версия языка Julia была опубликована в феврале 2012 года. Концепция языка состоит в том, что в рамках одного языка можно решать как задачи прототипирования, так и глубокой оптимизации кода (включая просмотр ассемблерного кода, поддержку параллельных и распределенных вычислений, а

также другие возможности). По утверждению авторов языка, сотрудников MIT Стефана Карпински, Джеффа Безансона и Вирала Шаха, приложения, полностью написанные на языке, практически не уступают в производительности приложениям, написанным на статически компилируемых языках вроде C или C++. В языке реализован прямой вызов C/Fortran-функций без дополнительных надстроек. Кроме того, существует возможность транслировать код на языке Julia в C-код. При этом синтаксис языка близок к MATLAB/Octave.

Язык программирования Julia дополняет плеяду замечательных бесплатных свободнораспростроняемых программных инструментов, поддерживаемых и развивающихся научным сообществом. Доказательством востребованности Julia может служит тот факт, что уже спустя 4 года после выхода первой версии, в высших учебных заведениях всего мира (среди них MIT, Stanford University, University of Edinburgh, Sciences Po Paris, Sapienza" University of Rome и другие) насчитывается более 35 курсов так или иначе включающих изучения этого языка программирования (см. http://julialang.org/teaching/).

Настоящее пособие содержит базовые сведения о новом высокопроизводительном языке программирования Julia (абсолютно бесплатном и свободно распространяемом под лицензией MIT) и может рассматриваться как справочник основных команд языка с примерами и описанием принципов работы. Пособие предназначено для студентов и сотрудников радиофизического факультета ННГУ, желающих освоить новый перспективный язык или повысить эффективность уже созданных программных решений.

1. ОСНОВНЫЕ СВЕДЕНИЯ О JULIA

Julia – это открытый свободный высокопроизводительный динамический язык высокого уровня, созданный специально для технических (математических) вычислений. Его синтаксис близок к синтаксису других сред технических вычислений, таких как Matlab и Octave. Он имеет в своем составе сложный компилятор, обеспечивает распределенное параллельное выполнение инструкций, вычислительную точность и обширную библиотеку математических функций. Базовая библиотека языка в основном написана на том же языке Julia, но также включает в себя развитые, лучшие в своем классе открытые библиотеки для линейной алгебры, генерации случайных чисел, обработки сигналов и строковых переменных, написанные на C и Fortran. Кроме того, есть возможность быстро установить через встроенный менеджер ряд внешних пакетов, подготовленных сообществом разработчиков Julia. Модуль языка IJulia, разработанный совместно сообществами Jupyter и Julia, обеспечивает мощный веб-интерфейс так называемого графического блокнота (graphical notebook), хорошо знакомый пользователям Python, который

объединяет код, форматированный текст, математические вычисления и мультимедийные ресурсы в одном документе.

Возможности языка:

- Множественная диспетчеризация (мультиметод): обеспечение возможности определять поведение функции при различных комбинациях типов аргументов.
- Динамическая типизация.
- Хорошая производительность, приближающаяся к производительности статических языков.
- Встроенный менеджер пакетов.
- Макросы и другие объекты метапрограммирования.
- Функции обработки вызовов Python: пакет PyCall.
- Функции прямой обработки вызовов C без надстроек.
- Мощные возможности оболочки для управления другими процессами.
- Имеются возможности обеспечения параллелизма и распределенных вычислений.
- Эффективная поддержка кодировки Unicode, включая, но не ограниченная, UTF-8.
- Лицензия MIT: распространяется бесплатно вместе с исходными кодами.

2. УСТАНОВКА И НАЧАЛО РАБОТЫ С JULIA

Официальный сайт языка Julia находится по адресу www.julialang.org. В разделе Downloads находятся все доступные для загрузки сборки языка. На текущий момент поддерживаются следующие компьютерные платформы: GNU/Linux, Darwin/OS X, FreeBSD, а также Windows. Для всех перечисленных операционных систем поддерживаются процессорные архитектуры x86/64 (64 бита) и x86 (32 бита). В тестовом режиме доступна также поддержка архитектуры ARM. Следующие дистрибутивы Linux включают Julia, таким образом установка Julia в них осуществляется через встроенные менеджеры пакетов: Arch Linux, Debian, Fedora (RHEL, CentOS, Scientific Linux), Gentoo, openSUSE, NixOS, Ubuntu. Официальный дистрибутив Julia включает интерактивную оболочку исполнения команд Julia’s REPL (Read-eval-print loop). Она позволяет вводить выражения, которые среда тут же будет вычислять, а результат вычисления отображать пользователю. Такая оболочка очень удобна при изучении языка и тестировании кода, так как предоставляет пользователю быструю обратную связь. Есть возможность выполнить системную команду (system shell) не покидая REPL. Для этого достаточно нажать «**;**» и затем ввести команду операционной системы. Чтобы получить доступ к справке нужно ввести «**?**» в REPL, а затем ввести оператор, по которому требуется справка. REPL сохраняет историю команд, в том числе и между сессиями. Предварительно подготовленный исходный код (скрипт) можно исполнить,

сохранив его в файл с расширением «**.jl**», а затем выполнить следующую команду в системной командной строке:

$ julia <имя_файла>

Описания дополнительных пакетов, которые можно добавить к ядру языка содержатся по адресу www.pkg.julialang.org. Добавление пакета осуществляется через REPL путем выполнения команды: «**Pkg.add(<имя_пакета>)**», автоматическое обновление всех установленных пакетов — «**Pkg.update()**». Ниже приводится список дополнительных пакетов, используемых в примерах (язык Julia чувствителен к регистру выражений):

IJulia, PyPlot, Gadfly, Interact, MAT, Debug.

После установки пакета IJulia становится доступным режим работы с интерактивным блокнотом (interactive notebook). Данный режим объединяет код на языке Julia, форматированный текст, математические выражения, а также мультимедиа в одном документе. Чтобы запустить IJulia notebook сервер, требуется ввести в REPL две строчки:

```
using IJulia
notebook()
```

Чтобы прервать вычисления в режиме REPL, нужно нажать на клавиатуре «**CTRL+C**». Размещение «**;**» после выражения отключает вывод его значения Чтобы выйти из режима REPL, нужно ввести «**exit()**» или нажать «**CTRL+D**». Комментарии выделяются символом «#».

Далее приведено несколько полезных команд REPL (они также могут быть включены в скрипты):

```
whos() # список глобальных переменных
cd("D:/") # сменить рабочую директорию
pwd() # текущая рабочая директория
include("file.jl") # Запустить скрипт
clipboard([1,2]) # скопировать данные в буфер обмена
clipboard() # загрузить данные из буфера в строку
workspace() # очистить область переменных
```

Ниже приводится пример программы (функции), реализующей алгоритм «Решето Эратосфена» нахождения всех простых чисел до некоторого целого числа «**n**»:

```
function es(n::Int) # функция es имеет один целый входной параметр
    isprime = ones(Bool, n) # n-элементный вектор булевых значений
    isprime[1] = false # 1 — не является простым числом
    for i in 2:round(Int, sqrt(n)) # цикл от 2 to sqrt(n)
    if isprime[i] # conditional evaluation
    for j in (i*i):i:n # последовательность с шагом i
    isprime[j] = false
```

```
    end
    end
    end
    return filter(x -> isprime[x], 1:n) # фильтрация с
использованием анонимной функции
    end

    println(es(100)) # выводит все простые числа меньшие
или равные 100
    @time length(es(100)) # выводит время исполнения
функции и объем используемой памяти.
```

3. БАЗОВЫЕ ТИПЫ ПЕРЕМЕННЫХ

Пример объявления переменной с указанием типа данных — **x::<Тип>**.

Список базовых скалярных типов данных:

```
1::Int64 # 64-битное целое
1.0::Float64 # 64-битное вещественное число с
плавающей запятой, включая NaN, -Inf, Inf
true::Bool # булевое число, может быть либо «true»
либо «false»
'c'::Char # символ из кодовой таблицы Unicode
"s"::AbstractString # строка из символов Unicode
```

Объявление типа необязательно, но в случае объявления типа переменных может увеличить производительность программы. Если тип не объявлен, то Julia выберет тип по умолчанию. Необходимо отметить, что значения по умолчанию могут отличаться для 32- и 64-битных версии Julia. Наиболее существенная разница имеет место для типа по умолчанию целых чисел — **Int32** и **Int64** соответственно. Кроме того, объявление **1::Int32** окончится приведет к ошибке в 64-х битной версии, а **1::Int64** — в 32-х битной версии Julia. Чтобы исключить Использование при объявлении **Int** будет означать либо **Int32** либо **Int64** в зависимости от версии Julia (то же касается **Uint**).

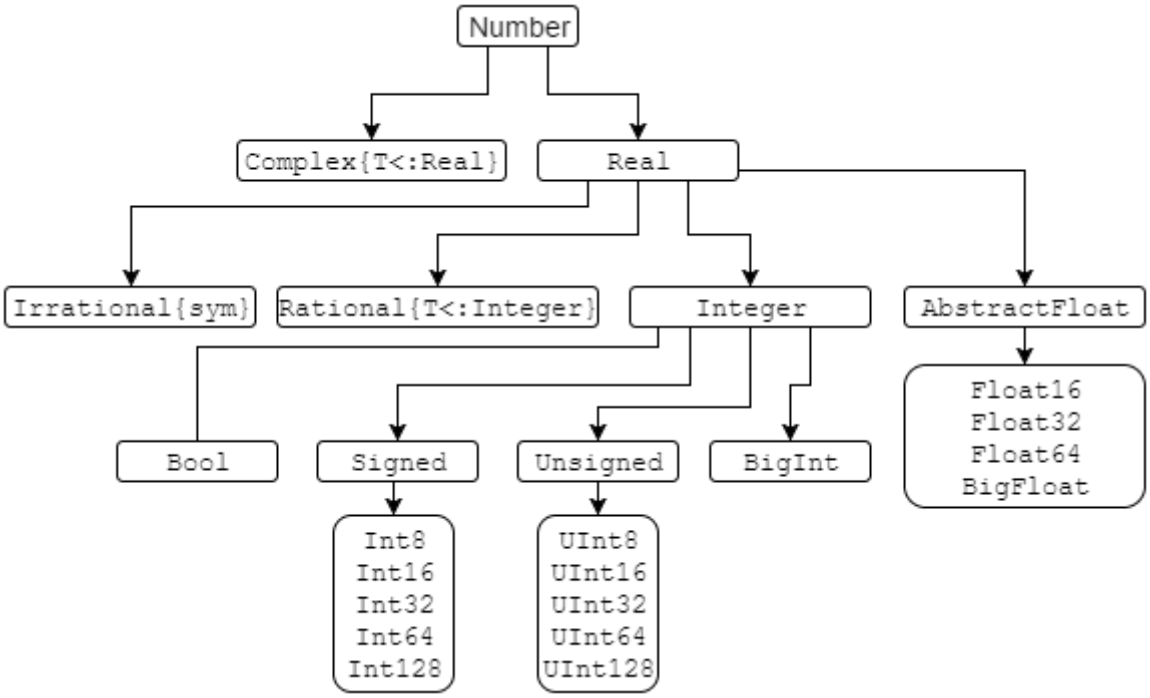

Рисунок 1. Иерархия числовых типов

В Julia нет автоматического преобразования типов (что особенно важно в вызовах функций). Примеры преобразований:

```
Int64('a') # из символа в целое
Int64(2.0) # из вещественного в целое
Int64(1.3) # приведет к ошибке
Int64("a") # приведет к ошибке
Float64(1) # из целого в вещественное
Bool(1) # преобразует в булевое true
Bool(0) # преобразует в булевое false
Bool(2) # ошибка преобразования
Char(89) # из целого в символ
string(true) # преобразует булевое число в строку
(работает и с другими типами данных)
string(1,true) # оператор string может принимать
более одного аргумента, объединяя вывод
zero(10.0) # ноль того же типа, что и 10.0
one(Int64) # единица того же типа, что и Int64
```

Существует оператор обобщенного преобразования типов:

```
convert(Int64, 1.0) # преобразует вещественное в
целое
```

Разбор строки может быть осуществлен с помощью оператора **`parse(Type, str)`**:

```
parse(Int64, "1") # разбор строки "1" в тип Int64
```

Автоматическое приведение нескольких аргументов к одному типу (если это возможно), используя оператор **`promote`**:

promote(true, BigInt(1)//3, 1.0) # создаст кортеж из нескольких чисел типа BigFloat, true приводится к 1.0

promote("a", 1) # приведение к одному общему типу невозможно.

Многие операторы в том числе арифметические и присвоения, обеспечивают автоматическое преобразование типа.

Тип аргумента можно проверить с помощью следующих операторов:

typeof("abc") # возвратит ASCIIString, являющийся подтипом AbstractString

isa("abc", AbstractString) # возвратит true

isa(1, Float64) # возвратит false, так как тип целое и тип вещественное — не одно и то же

isa(1.0, Float64) # возвратит true

isa(1.0, Number) # возвратит true, так как Number — абстрактный тип

super(Int64) # возвратит абстрактный тип, включающий в себя Int64

subtypes(Real) # возвратит подтипы абстрактного типа Real

Есть возможность производить вычисления, используя арифметику произвольной точномти и рациональные числа:

BigInt(10)^1000 # возвратит большое целое

BigFloat(10)^1000 # возвратит большое вещественное число

123//456 # рациональные числа создаются с помощью оператора //

Иерархия всех стандартных числовых типов представлена на рисунке 1.

4. СЛОЖНЫЕ ТИПЫ ПЕРЕМЕННЫХ

Специальные типы:

Any # все объекты имеют этот тип

Union{} # подтип всех типов, ни один объект не может иметь этот тип

Void # ни на что не указывающий тип, подтип типа Any

nothing # единственный экземпляр типа Void

4.1 Кортежи

Кортежи представляют собой неизменяемые последовательности, индексируемые, начиная с 1:

() # пустой кортеж

(1,) # кортеж из одного элемента

```
("a", 1) # кортеж из двух элементов
('a', false)::Tuple{Char, Bool} # явное объявление типов составляющих кортежа
x = (1, 2, 3)
x[1] # возвратит 1 (элемент кортежа)
x[1:2] # возвратит кортеж (1, 2)
x[4] # возвратит ошибку
x[1] = 1 # возвратит ошибку, так как кортеж нельзя изменять
a, b = x # распаковка кортежа a=1, b=2
```

4.2 Массивы

В отличие от кортежей, массивы изменяемы, а обращение к ним осуществляется по ссылке. Создание массивов:

```
Array(Char, 2, 3, 4) # массив символов размером 2x3x4
Array{Int64}(0, 0) # вырожденный 0x0 массив из Int64
cell(2, 3) # массив 2x3 из элементов любого типа Any
zeros(5) # вектор нолей типа Float64
ones(5) # вектор единиц типа Float64
ones(Int64, 2, 1) # массив 2x1 из Int64 единиц
trues(3), falses(3) # кортеж из булевых векторов
eye(3) # единичная матрица 3x3 из Float64
x = linspace(1, 2, 5) # итератор из пяти равномерно распределенных элементов от 1 до 2 включительно
collect(x) # конвертирует итератор в вектор
1:10 # создает итерируемый объект от 1 до 10
1:2:10 # итерируемый объект от 1 до 9 с шагом 2
reshape(1:12, 3, 4) # массив 3x4 заполненный значениями 1:12
fill("a", 2, 2) # массив 2x2 заполненный "a"
repmat(eye(2), 3, 2) # единичная матрица 2x2, повторенная 3x2 раз
x = [1, 2] # вектор из двух элементов
resize!(x, 5) # изменить размер вектора x на месте для хранения пяти значений (будут заполнены мусором)
[1] # вектор (не скаляр) из одного элемента
[x * y for x in 1:2, y in 1:3] # списковое включение, генерирующее массив 2x3
Float64[x^2 for x in 1:4] # приведение результата спискового включения к Float64
[1 2] # матрица 1x2 (то же, что функция hcat)
[1 2]' # матрица 2x1 (после транспонирования)
[1, 2] # вектор (то же, что функция vcat)
```

```
[1; 2] # вектор(то же, что функция hvcat)
[1 2 3; 1 2 3] # матрица 2x3 (функция hvcat)
[1; 2] == [1 2]' # возвратит false, так как массивы имеют разные размеры
[(1, 2)] # вектор из одного элемента-кортежа
collect((1, 2)) # двухэлементный вектор из распаковки кортежа
[[1 2] 3] # добавление элемента к вектору-строке (hcat)
[[1; 2]; 3] # добавление элемента к вектору-столбцу (vcat)
```

Векторы (одномерные массивы) рассматриваются в качестве вектора столбца. Часто используемые функции по работе с массивами:

```
a = [x * y for x in 1:2, y in 1, z in 1:3] # массив 2x1x3 из Int64
ndims(a) # количество измерений в a
eltype(a) # тип элементов в a
length(a) # количество элементов в a
size(a) # кортеж содержащий размеры a в каждом измерении
vec(a) # преобразование массива к вектору (к одному измерению)
squeeze(a, 2) # удаляет второе измерение, так как размер a в нем равен 1
sum(a, 3) # вычисляет сумму вдоль 3-его измерения матрицы, аналогичный синтаксис у операторов mean, std, prod, minimum, maximum, any, all
count(x -> x > 0, a) # расчитывает сколько раз утверждение истинно, аналогичный синтаксис возможен с операторами all, any
```

Функции доступа:

```
a = linspace(0, 1) # объект типа LinSpace{Float64} длиной 50
a[1] # возратит скаляр 0.0
a[end] # возвратит скаляр 1.0 (последний элемент массива)
a[1:2:end] # каждый второй элемент из диапазона LinSpace{Float64}
a[repmat([true, false], 25)] # выбрать каждый второй элемент, тип Array{Float64,1}
a[[1, 3, 6]] # 1-й, 3-й и 6-й элементы из a, Array{Float64,1}
sub(a, 1:2:50) # выводит подмассив a
endof(a) # последний индекс массива a
```

Вывод всех элементов вдоль выбранного измерения:

```
a = reshape(1:12, 3, 4) матрица 3x4
a[:, 1:2] # матрица 3x2
a[:, 1] # вектор из 3-х элементов
a[1, :] # матрица 1x4
```

Присваивание элементам массива:

```
x = reshape(1:8, 2, 4) матрица 2x4
x[:,2:3] = [1 2] # ошибка, несоответствие размеров
x[:,2:3] = repmat([1 2], 2) # ошибки нет
x[:,2:3] = 3 # ошибки нет
```

Присваивание и обращение к массивам осуществляется по ссылкам:

```
x = cell(2)
x[1] = ones(2)
x[2] = trues(3)
a = x # создание новой ссылки на массив
b = copy(x) # поверхностная копия
c = deepcopy(x) # абсолютная копия
x[1] = "Bang"
x[2][1] = false
a # идентична x
b # по сравнению с x изменился только элемент x[2][1]
c # содержит оригинальное содержимое x
```

Создание массивов с указанием типа:

```
cell(2)::Array{Any, 1} # вектор типа Any
[1 2]::Array{Int64, 2} # 2-х мерный массив из Int64
[true; false]::Vector{Bool} # булевый вектор
[1 2; 3 4]::Matrix{Int64} # матрица из Int64
```

4.3 Составные типы

Составные типы изменяемы, обращение к ним осуществляется по ссылке. Определение и доступ к составным типам осуществляется следующим образом:

```
type Point
  x::Int64
  y::Float64
  meta
end
p = Point(0, 0.0, "Origin")
p.x # доступ к полю x
p.meta = 2 # изменение значение поля
p.x = 1.5 # ошибка, неправильный тип данных
p.z = 1 # ошибка, нет такого поля z
fieldnames(p) # вывести имена полей объекта
fieldnames(Point) # вывести имена полей типа
```

Есть возможность определить неизменяемый тип заменив «type» на «immutable».

4.4 Словари

Ассоциативные коллекции (словари ключ-значение):

```
x = Dict{Float64, Int64}() # пустой словарь отображающий вещественные числа на целые
y = Dict("a"=>1, "b"=>2) # заполненный словарь
y["a"] # возвращение значения по ключу a
y["c"] # ошибка, нет такого ключа
y["c"] = 3 # добавление элемента в словарь
haskey(y, "b") # проверка наличия в словаре y ключа "b"
keys(y), values(y) # кортеж итераторов, возвращающих ключи и значения словаря y
delete!(y, "b") # удалить ключ из коллекции (см. также «pop!»)
get(y,"c","default") # возвратит y["c"] или "default", в случае если такого ключа нет (т. е. Если haskey(y,"c"))==false)
```

5. СТРОКИ

Операции над строками:

```
"Hi " * "there!" # объединение строк
"Ho " ^ 3 # повтор строки 3 раза
string("a= ", 123.3) # создание строки с использованием функции «print»
repr(123.3) # преобразует значение любого числового типа в строку
contains("ABCD", "CD") # проверка, содержится ли вторая строка в первой
"\"\n\t\$" # C-подобное экранирование спецсимволов
x = 123
"$x + 3 = $(x+3)" # не экранированный $ используется для подстановки значений переменных
"\$199" # чтобы получить символ $, его нужно экранировать
```

6. КОНСТРУКЦИИ ЯЗЫКА

Присваивание — простейший способ создания новой переменной:

```
x = 1.0 # x имеет тип Float64
x = 1 # теперь x имеет тип Int32 на 32-х битной системе и Int64 на 64-х битной
y::Float64 = 1.0 # y должна быть типа Float64
```

Выражения могут быть объединены с помощью «;» или с помощью блока «begin end»:

```
x = (a = 1; 2 * a) # в итоге получится: x = 2; a = 1
y = begin
    b = 3
    3 * b
end # в итоге получится: y = 9; b = 3
```

Стандартные языковые конструкции:

```
if false # конструкция if-else требует проверки булевого условия
    z = 1
elseif 1==2
    z = 2
else
    a = 3
end # в итоге получится: a = 3 , a z — не определено

1==2 ? "A" : "B" # стандартная тернарная условная операция (возвратит «B»)

i = 1
while true
    i += 1
    if i > 10
        break
    end
end

for x in 1:10 # синтаксис: x «in» коллекция, вместо «in» может быть использовано «=»
    if 3 < x < 6
        continue # пропуск одной итерации
    end
    println(x)
end
```

Объявление собственных функций:

```
f(x, y = 10) = x + y # объявление новой функции f со значением по умолчанию для y — 10, функция возвращает последний результат
f(3, 2) # простой вызов, вернет 5
```

```
    f(3) # вернет 13
    function g(x::Int, y::Int) # ограничения по типу
аргументов
        return y, x # явное возвращение кортежа
    end
    g(x::Int, y::Bool) = x * y # вводим перегрузку
функции g для других типов входных аргументов
    g(2, true) # такой вызов подразумевает второе
определение g
    methods(g) # выводит все методы определенные для g
    (x -> x^2)(3) # анонимная функция объединенная с
вызовом
    () -> 0 # анонимная функция без аргументов
    h(x...) = sum(x)/length(x) - mean(x) # функция с
неопределенным числом аргументов; x - кортеж
    h(1, 2, 3) # результат 0
    x = (2, 3) # кортеж
    f(x) # возвратит ошибку, так как кортеж нужно
распаковать
    f(x...) # все нормально — кортеж распакован
    s(x; a = 1, b = 1) = x * a / b # функция с
аргументами-ключами a и b
    s(3, b = 2) # вызов функции с аргументом-ключом
    t(; x::Int64 = 2) = x # один аргумент-ключ
    t() # возвратит 2
    t(; x::Bool = true) = x # для функций с аргументами-
ключами перегрузка невозможна, она будет переписана
    t() # возвратит true; старая функция была переписана
    q(f::Function, x) = 2 * f(x) # простая обертка для
функции
    q(x -> 2x, 10) # возвратит 40; нет необходимости
использовать «*» в 2x (это эквивалентно 2*x)
    q(10) do x # создание анонимной функции с
использованием конструкции «do», полезно при операциях
ввода/вывода
        2 * x
    end
    m = reshape(1:12, 3, 4)
    map(x -> x ^ 2, m) # возвратит массив 3x4
обработанных данных
    filter(x -> bits(x)[end] == '0', 1:12) # причудливый
способ выбора четных значений в заданном диапазоне
```

Функции конверсии с именами, оканчивающимися на «!» изменяют свой аргумент на месте (см. например функцию `resize!`).

Работа с заданными по умолчанию аргументами функций:

```
y = 10
f1(x=y) = x; f1() # возвратит 10
f2(x=y,y=1) = x; f2() # тоже возвратит 10
f3(y=1,x=y) = x; f3() # возвратит 1
f4(;x=y) = x; f4() # возвратит 10
f5(;x=y,y=1) = x; f5() # ошибка - y не определена
f6(;y=1,x=y) = x; f6() # возвратит 1
```

7. ОБЛАСТИ ВИДИМОСТИ ПЕРЕМЕННЫХ

В двух разных областях видимости переменных могут существовать две переменные с одинаковыми именами, ссылающиеся на разные данные. Использование следующих конструкций неявно образует новую область видимости переменных: **function**, **while**, **for**, **try/catch**, **let**, **type**. Существует возможность объявить переменные как:

- **global**: объявление переменной из глобальной области видимости (глобальная переменная);
- **local**: объявление переменной в текущей области видимости;
- **const**: обеспечение неизменности переменной (только для глобальных переменных).

Особые случаи:

```
t # ошибка, переменная не существует
f() = global t = 1
f() # после вызова функции f, переменная t объявляется как глобальная

function f1(n)
    x = 0
    for i = 1:n
        x = i
    end
    x
end
f1(10) # возвратит 10, так как внутри цикла мы используем локальную переменную, объявленную вне цикла

function f2(n)
    x = 0
    for i = 1:n
        local x
        x = i
    end
```

```
        x
    end
    f2(10) # возвратит 0, так как внутри цикла мы
используем новую локальную переменную

    function f3(n)
        for i = 1:n
            local x # в данном случае явное
использование local необязательно; использование for
автоматически создает новую область видимости переменных
            x = i
        end
        x
    end
    f3(10) # возвратит ошибку, так как x не определена во
внешней области переменных

    const x = 2
    x = 3 # предупреждение, значение константы
переопределено
    x = 3.0 # ошибка — неправильный тип

    function fun() # no warning
        const x = 2
        x = true
    end
    fun() # возвратит true без предупреждений
```

Глобальные константы ускоряют исполнение программ. Оператор **let** обеспечивает повторное присваивание переменной:

```
    Fs = cell(2)
    i = 1
    while i <= 2
        j = i
        Fs[i] = () -> j
        i += 1
    end
    Fs[1](), Fs[2]() # (2, 2); то же самое присвоение для
j
    Fs = cell(2)
    i = 1
    while i <= 2
        let j = i
            Fs[i] = () -> j
```

```
        end
        i += 1
    end
    Fs[1](), Fs[2]() # (1, 2); новое присвоение для j

    Fs = cell(2)
    i = 1
    for i in 1:2
        j = i
        Fs[i] = () -> j
    end
    Fs[1](), Fs[2]() # (1, 2); циклы for и однострочные
выражения переприсваивают переменные
```

8. МОДУЛИ

Модули инкапсулируют код. Могут быть перезагружены, что может использоваться для переопределения функций и типов.

```
    module M # имя модуля
        export x # что модуль экспортирует во внешнее
пространство
        x = 1
        y = 2 # скрытая переменная
    end

    whos(M) # список экспортированных переменных
    x # переменная не найдена в глобальной области
видимости
    M.y # возможен прямой доступ к переменной

    using M # импорт всех экспортированных переменных;
таким же образом загружаются стандартные пакеты языка

    import M.y # импорт переменной y в глобальную область
(даже если она не была экспортирована)
```

9. ОПЕРАТОРЫ

Julia имеет следующие особенности при использовании стандартных операторов:

```
    true || false # бинарный оператор ИЛИ (только для
переменных единичного размера), аналогично применяется
бинарный оператор И - &&
    [1 2] & [2 1] # побитовый оператор И
```

1 < 2 < 3 # допускаются цепочки условий (только для переменных единичного размера)

[1 2] .< [2 1] # применяя операторы к векторам необходимо добавить '.' перед ним

x = [1 2 3]

2x + 2(x+1) # оператор умножения может быть опущен перед переменной или перед левой скобкой

y = [1, 2, 3]

x + y # возвратит ошибку

x .+ y # возвратит матрицу 3x3, автоматическое расширение размерности

x + y' # возврати матрицу 1x3

x * y # произведение массивов, возврати одноэлементный вектор (не скаляр)

x .* y # по элементное произведение массивов, возвратит матрицу 3x3

x == [1 2 3] # возвратит true, так как объекты выглядят одинаково

x === [1 2 3] # возвратит false, так как объекты не идентичны

z = reshape(1:9, 3, 3)

z + x # возвратит ошибку

z .+ x # для осуществления операции x расширится вертикально

z .+ y # y расширится горизонтально

broadcast(+, [1 2], [1; 2]) # явное расширение размерности; оператор сложения вызывается для каждого элемента массива

Также доступно множество операторов реализующих типичные матричные преобразования (см. документацию к языку).

10. НЕОБХОДИМЫЕ ФУНКЦИИ ОБЩЕГО ПОЛЬЗОВАНИЯ

show(collect(1:100)) # выводит текстовое представление объекта

eps() # расстояние от 1.0 до следующего числа типа Float64

nextfloat(2.0) # следующее за 2.0 число типа float; аналогично доступна функция prevfloat

isequal(NaN, NaN) # возвратит true

NaN == NaN # возвратит false

NaN === NaN # возвратит true

isequal(1, 1.0) # возвратит true

1 == 1.0 # возвратит true

1 === 1.0 # возвратит false

isfinite(Inf) # возвратит false, доступны также: isinf, isnan

fld(-5, 3), mod(-5, 3), div(-5, 3) # доступны различные варианты целочисленного деления

rem(-5, 3) # остаток от деления

find(x -> mod(x, 2) == 0, 1:8) # поиск индексов, для которых функция возвращает истинное значчение

x = [1 2]; identity(x) === x # возвратит true, функция идентичности

info("Info") # вывод информации, аналогичны в использовании warn and error

ntuple(x->2x, 3) # создает кортеж вызовом x->2x со значениями 1, 2 and 3

isdefined(:x) # возвратит x, если переменная x определена (:x - символ)

1:5 |> exp |> sum # последовательное применение функций

enumerate("abc") # создает итератор из кортежей (индекс, элемент коллекции)

isempty("abc") # проверка на пустоту коллекции

'b' in "abc" # проперка: имеется ли элемент в коллекции

findin("abc", "abrakadabra") # [1, 2] ('c' не найдена)

unique("abrakadabra") # возвратит уникальные элементы

issubset("abc", "abcd") # проверка: является ли каждый элемент первой коллекции еще и элементом второй коллекции

indmax("abrakadabra") # количество вхождений самого частого элемента (3 - 'r' в данном случае)

findmax("abrakadabra") # кортеж: самый частый элемент и его индекс

filter(x->mod(x,2)==0, 1:10) # сохраняет только те элементы колллекции, которые удовлетворяют утверждению

dump(1:2:5) # выводит все доступные пользователю поля структуры объекта

sort(rand(10)) # сортировка 10-ти равномерно распределенных случайных величин

11. ЧТЕНИЕ И ЗАПИСЬ ДАННЫХ

Базовые операции для работы с текстовыми файлами:

- `readdlm`, `readcsv`: чтение из тектовых файлов

- `writedlm`, `writecsv`: запись в текстовый файл

Необходимо отметить, что множественные пробелы в текстовых файлах не поглощаются, если `delim=' '`.

За детальным описанием реализации ввода/вывода данных в Julia необходимо обратиться к документации.

12. СЛУЧАЙНЫЕ ЧИСЛА

Базовые операции:

```
rand() # генерация случайного числа из полуинтервала [0,1)
rand(3, 4) # генерация матрицы 3x4 случайных чисел из полуинтервала [0,1]
rand(2:5, 10) # генерация вектора из 10 целых случайных чисел в диапазоне от 2 до 5
randn(10) # генерация вектора из 10 случайных чисел (стандартное нормальное распределение)
```

Некоторые операторы из пакета **`Distributions`**:

```
using Distributions # загрузка пакета
b = Beta(0.4, 0.8) # Beta распределение с параметрами 0.4 и 0.8
# Julia поддерживает широкий спектр распределений (см. документацию)
mean(b) # математическое ожидание распределения b
rand(b, 100) # 100 независимых случчайных отсчетов из распределения b
```

13. ПОСТРОЕНИЕ ГРАФИКОВ

Для Julia существует несколько пакетов для построения графиков: `Winston`, `Gadfly` и `PyPlot`. Ниже представлен пример работы с пакетом Winston.

```
using Winston # загрузка пакета Winston
x = linspace(0, 1, 100)
y = sin(4x*pi) .* exp(-5x)
p = FramedPlot(title="4x\\pi",xlabel="x", ylabel="f(x)")
add(p, Curve(x, y))
savefig(p, "fun.pdf") # запись графика в pdf файл

# второй график (двухмерный)
```

```
srand(1)
x, y = randn(10000), randn(10000)
plothist2d([x y], 100)
```

14. ЗАМЕРЫ ВРЕМЕНИ И ИСПОЛЬЗОВАННОЙ ПАМЯТИ

Ниже приводятся макросы и операторы, позволяющие замерять время и память, затраченные на выполнение программ:

```
@time [x for x in 1:10^6] # вывод затраченных времени и объема памяти
@timed [x for x in 1:10^6] # возвращает значение, затраченные время и объем памяти
@elapsed [x for x in 1:10^6] # возвращает затраченное время
@allocated [x for x in 1:10^6] # возвращает затраченный объем памяти
tic() # старт таймера
toc() # остановка таймера и вывод затраченного времени
toq() # остановливает таймера и возвращает затраченное время
```

ЗАКЛЮЧЕНИЕ

Настоящее руководство не претендует на полное описание языка. Для этого существует официальная документация, а также подробные издания-руководства. Необходимо отметить важные темы не затронутые настоящим пособием:

- параметрические типы;
- параллельные и распределенные вычисления;
- продвинутые операторы ввода.вывода;
- работа с дополнительными пакетами (см. команду Pkg);
- взаимодействие с системной консолью (см. команду run);
- создание сопрограмм (см. команду Task);
- двухсторонняя интеграция с языками C и Fortran.

С другой стороны, пособие поможет сделать первые шаги тем, кто хочет быстро освоить базовые возможности языка.Julia, а затем перейти к самостоятельному изучению.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Julia - официальная документация http://docs.julialang.org/en/latest/

2. Bogomił Kamiński. The Julia Express.
http://bogumilkaminski.pl/files/julia_express.pdf
3.Leah Hanson. Learn Julia in Y minutes.
http://learnxinyminutes.com/docs/julia/
4. Julia: A fresh approach to numerical computing. Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah (2014), http://arxiv.org/abs/1411.1607
5. Julia: A fast dynamic language for technical computing. Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman (2012), http://arxiv.org/abs/1209.5145

Алексей Владимирович **Шиндин**

ЯЗЫК ПРОГРАММИРОВАНИЯ МАТЕМАТИЧЕСКИХ ВЫЧИСЛЕНИЙ JULIA. БАЗОВОЕ РУКОВОДСТВО

Учебно-методическое пособие

Компьютерная верстка – А.В. Шиндин

Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский нижегородский государственный университет
им. Н.И. Лобачевского».
603950, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать . Формат 60×84 1/16.
Бумага офсетная. Печать офсетная. Гарнитура Таймс.
Усл. печ. л. . Уч.-изд. л.
Заказ № . Тираж 100 экз.

Отпечатано в типографии Нижегородского госуниверситета
им. Н.И. Лобачевского.
603600, г. Нижний Новгород, ул. Большая Покровская, 37.