The Julia Express
Bogumił Kamiński
July 13, 2017

Contents

1 Introduction 2
2 Getting around 2
3 Basic literals and types 3
4 Complex literals and types 4
  4.1 Tuples 4
  4.2 Arrays 5
  4.3 Composite types 6
  4.4 Dictionaries 6
5 Strings 7
6 Programming constructs 7
7 Variable scoping 9
8 Modules 10
9 Operators 10
10 Essential general usage functions 11
11 Reading and writing data 11
12 Random numbers 12
13 Statistics and machine learning 12
14 Plotting 12
15 Macros 12
16 Plotting 13
1 Introduction

The Purpose of this document is to introduce programmers to Julia programming by example. This is a simplified exposition of the language. 1

It is best to execute these examples by copying them to a file and next running them using include function.

If some packages are missing on your system use Pkg.add to require installing them. There are many add-on packages which you can browse at http://pkg.julialang.org/.

Major stuff not covered (please see the documentation):

1) parametric types;
2) parallel and distributed processing;
3) advanced I/O operations;
4) package management; see Pkg;
5) interaction with system shell; see run;
6) exception handling; see try;
7) creation of coroutines;
8) integration with C, Fortran, Python and R.


Julia Express was tested using the following 64-bit Julia version:

```plaintext
versioninfo()
# Julia Version 0.6.0
# Commit 903644385b* (2017-06-19 13:05 UTC)
# Platform Info:
# OS: Windows (x86_64-w64-mingw32)
# CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
# WORD_SIZE: 64
# BLAS: libopenblas (USE64BITINT DYNAMIC_ARCH NO_AFFINITY Haswell)
# LAPACK: libopenblas64_
# LIBM: libopenlibm
# LLVM: libLLVM-3.9.1 (ORCJIT, broadwell)
```

Remember that you can expect every major version of Julia to introduce breaking changes. Check https://github.com/JuliaLang/julia/blob/master/NEWS.md for release notes.

All suggestions how this guide can be improved are welcomed. Please contact me at bkamins@sgh.waw.pl.

2 Getting around

Running julia invokes interactive (REPL) mode. In this mode some useful commands are:

1) ^D (exits Julia);
2) ^C (interrupts computations);
3) ? (enters help mode)
4) ; (enters system shell mode)
5) putting ; after the expression will disable showing of its value.

Examples of some essential functions in REPL (they can be also invoked in scripts):

```plaintext
apropos("apropos")    # search documentation for "apropos" string
@less(max(1,2))        # show the definition of max function when invoked with arguments 1 and 2
whos()                 # list of global variables and their types
cd("D:/")             # change working directory to D:/ (on Windows)
pwd()                 # get current working directory
include("file.jl")    # execute source file
exit(1)                # exit Julia with code 1 (exit code 0 by default)
clipboard([1,2])       # copy data to system clipboard
clipboard()            # load data from system clipboard as string
workspace()            # clear workspace - create new Main module (only to be used interactively)
```

You can execute Julia script by running julia script.jl.

Try saving the following example script to a file and run it (more examples of all the constructs used are given in following sections):

```plaintext
1The rocket ship clip is free for download at http://www.clipartlord.com/free-cartoon-rocketship-clip-art-2/.
```
"Sieve of Eratosthenes function docstring"

```julia
function es(n::Int) # accepts one integer argument  
    isprime = ones(Bool, n) # n-element vector of true-s  
    isprime[1] = false # 1 is not a prime  
    for i in 2:round(Int, sqrt(n)) # loop integers from 2 to sqrt(n)  
        if isprime[i] # conditional evaluation  
            for j in (i*i):i:n # sequence with step 1  
                isprime[j] = false  
            end  
        end  
    end  
    return filter(x -> isprime[x], 1:n) # filter using anonymous function  
end
```

println(es(100)) # print all primes less or equal than 100
@time length(es(10^7)) # check function execution time and memory usage

3 Basic literals and types

Basic scalar literals (x::Type is a literal x with type Type assertion):

1::Int64 # 64-bit integer, no overflow warnings, fails on 32 bit Julia
1.0::Float64 # 64-bit float, defines NaN, -Inf, Inf
true::Bool # boolean, allows "true" and "false"
'c'::Char # character, allows Unicode
"s"::AbstractString # strings, allows Unicode, see also Strings

All basic types are immutable. Specifying type assertion is optional (and usually it is not needed, but I give it to show how you can do it). Type assertions for variables are made in the same way and may improve code performance.

If you do not specify type assertion Julia will choose a default. Note that defaults might be different on 32-bit and 64-bit versions of Julia. A most important difference is for integers which are Int32 and Int64 respectively. This means that 1::Int32 assertion will fail on 64-bit version. Notably Int is either Int64 or Int32 depending on version (the same with UInt).

There is no automatic type conversion (especially important in function calls). Has to be explicit:

```julia
Int64('a') # character to integer
Int64(2.0) # float to integer
Int64(1.3) # inexact error
Int64("a") # error no conversion possible
Float64(1) # integer to float
Bool(1) # converts to boolean true
Bool(0) # converts to boolean false
Bool(2) # conversion error
Char(89) # integer to char
string(true) # cast bool to string (works with other types, note small caps)
string(1,true) # string can take more than one argument and concatenate them
zero(10.0) # zero of type of 10.0
one(Int64) # one of type Int64
```

General conversion can be done using `convert(Type, x)`:

```julia
convert(Int64, 1.0) # convert float to integer
```

Parsing strings can be done using `parse(Type, str)`:

```julia
parse(Int64, "1") # parse "1" string as Int64
```

Automatic promotion of many arguments to common type (if any) using `promote`:

```julia
promote(true, BigInt(1)//3, 1.0) # tuple (see Tuples) of BigFloats, true promoted to 1.0
promote("a", 1) # promotion to common type not possible
```

Many operations (arithmetic, assignment) are defined in a way that performs automatic type promotion. One can verify type of argument:
It is possible to perform calculations using arbitrary precision arithmetic or rational numbers:

```plaintext
BigInt(10)^1000  # big integer
BigFloat(10)^1000  # big float, see documentation how to change default precision
123/456           # rational numbers are created using // operator
```

Type hierarchy of all standard numeric types is given in Figure 1.

4 Complex literals and types

Type beasts:

```plaintext
Any       # all objects are of this type
Union()   # subtype of all types, no object can have this type
Void      # type indicating nothing, subtype of Any
nothing   # only instance of Void
```

Additionally `undef` indicates an incompletely initialized instance (see documentation for details).

4.1 Tuples

Tuples are immutable sequences indexed from 1:

```plaintext
()             # empty tuple
(1,)           # one element tuple
("a", 1)       # two element tuple
(\'a\', false)::Tuple{Char, Bool} # tuple type assertion
x = (1, 2, 3)
x[1]           # 1 (element)
x[1:2]         # (1, 2) (tuple)
x[4]           # bounds error
x[1] = 1       # error - tuple is not mutable
a, b = x       # tuple unpacking a==1, b==2
```
4.2 Arrays

Arrays are mutable and passed by reference. Array creation:

- `Array(Char)(2, 3, 4)` # 2x3x4 array of Char
- `Array(Int64)(0, 0)` # degenerate 0x0 array of Int64
- `Array(Any)(2, 3)` # 2x3 array of Any
- `zeros(5)` # vector of Float64 zeros
- `ones(5)` # vector of Float64 ones
- `ones(Int64, 2, 1)` # 2x1 array of Int64 ones
- `trues(3), falses(3)` # tuple of vector of trues and of falses
- `eye(3)` # 3x3 Float64 identity matrix
- `x = linspace(1, 2, 5)` # iterator having 5 equally spaced elements
- `collect(x)` # converts iterator to vector
- `1:10` # iterable from 1 to 10
- `1:2:10` # iterable from 1 to 9 with 2 skip
- `reshape(1:12, 3, 4)` # 3x4 array filled with 1:12 values
- `fill("a", 2, 2)` # 2x2 array filled with "a"
- `repmat(eye(2), 3, 2)` # 2x2 identity matrix repeated 3x2 times
- `x = [1, 2]` # two element vector
- `x'` # two element RowVector sharing the same memory
- `resize!(x, 5)` # resize x in place to hold 5 values (filled with garbage)
- `[1]` # vector with one element (not a scalar)
- `[x + y for x in 1:2, y in 1:3]` # comprehension generating 2x3 array
- `Float64[x^2 for x in 1:4]` # casting comprehension result to Float64
- `[1, 2]` # 1x2 matrix (hcat function)
- `[1, 2]'` # 2x1 matrix (after transposing)
- `[1, 2]` # vector (concatenation)
- `[[1, 2]]` # 2x1x3 array of Int64; singelton dimension is not dropped
- `[(1, 2)]` # 1-element vector
- `collect((1, 2))` # 2-element vector by tuple unpacking
- `[1 2; 3 4]` # append to a column vector (vcat)
- `view(a, 1:2:50)` # view into subarray of a
- `endof(a)` # last index of the collection a

Vectors (1D arrays) are treated as column vectors.

Julia offers sparse and distributed matrices (see documentation for details).

Commonly needed array utility functions:

- `a = [x * y for x in 1:2; y in 1, z in 1:3]` # 2x3 array of Int64; singelton dimension is dropped
- `a = [x * y for x in 1:2; y in 1:1, z in 1:3]` # 2x1x3 array of Int64; singelton dimension is not dropped

Access functions:
Observe the treatment of trailing singleton dimensions:

```julia
a = reshape(1:12, 3, 4)
a[:, 1:2] # 3x2 matrix
a[:, 1]   # 3 element vector
a[1, :]    # 4 element vector
a[1:1, :]  # 1x4 matrix
a[:, :, 1, 1] # works 3x4 matrix
a[:, :, :, [true]] # works 3x4x1 matrix
a[1, 1, [false]] # works 0-element Array{Int64,1}
```

Array assignment:

```julia
x = collect(reshape(1:8, 2, 4))
x[:,2:3] = [1 2] # error; size mismatch
x[:,2:3] = repmat([1 2], 2) # OK
x[:,2:3] = 3 # OK
```

Arrays are assigned and passed by reference. Therefore copying is provided:

```julia
x = Array{Any}(2)
x[1] = ones(2)
x[2] = trues(3)
a = x
b = copy(x) # shallow copy
   c = deepcopy(x) # deep copy
x[1] = "Bang"
x[2][1] = false
a # identical as x
b # only x[2][1] changed from original x
c # contents to original x
```

Array types syntax examples:

```julia
[1 2]::Array{Int64, 2} # 2 dimensional array of Int64
[true; false]::Vector{Bool} # vector of Bool
[1 2; 3 4]::Matrix{Int64} # matrix of Int64
```

4.3 Composite types

Composite types are mutable and passed by reference.
You can define and access composite types:

```julia
mutable struct Point
    x::Int64
    y::Float64
    meta
end
p = Point(0, 0.0, "Origin")
p.x # access field
p.meta = 2 # change field value
p.x = 1.5 # error, wrong data type
p.z = 1 # error - no such field
fieldnames(p) # get names of instance fields
fieldnames(Point) # get names of type fields
```

You can define type to be immutable by removing `mutable`. There are also union types (see documentation for details).

4.4 Dictionaries

Associative collections (key-value dictionaries):

```julia
x = Dict(Float64, Int64)() # empty dictionary mapping floats to integers
y = Dict("a"=>1, "b"=>2) # filled dictionary
```
Julia also supports operations on sets and deques, priority queues and heaps (please refer to documentation).

## 5 Strings

String operations:

```
"Hi " * "there!"      # string concatenation
"Ho " ^ 3            # repeat string
string("a= ", 123.3)  # create using print function
repr(123.3)          # fetch value of show function to a string
contains("ABCD", "CD") # check if first string contains second
"\n\t\$"                 # C-like escaping in strings, new \$ escape
x = 123
"$x + 3 = $(x+3)"    # unescaped $ is used for interpolation
"\$199"              # to get a $ symbol you must escape it
```

PCRE regular expressions handling:

```
r = r"A|B"              # create new regexp
ismatch(r, "CD")        # false, no match found
m = match(r, "ACBD")    # find first regexp match, see documentation for details
```

There is a vast number of string functions — please refer to documentation.

## 6 Programming constructs

The simplest way to create new variable is by assignment:

```
x = 1.0                 # x is Float64
x = 1                   # now x is Int32 on 32 bit machine and Int64 on 64 bit machine
```

Expressions can be compound using ; or begin end block:

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

There are standard programming constructs:

```
if false # if clause requires Bool test
    z = 1
elseif l=2
    z = 2
else
    a = 3
end       # after this a = 3 and z is undefined
l=2 ? "A" : "B" # standard ternary operator
i = 1
while true
    i += 1
    if i > 10
```
```julia
break
end
end

for x in 1:10  # x in collection, can also use = here instead of in
    if 3 < x < 6
        continue  # skip one iteration
    end
    println(x)
end  # if x was defined in loop outer scope it is changed

You can define your own functions:

```julia
f(x, y = 10) = x + y  # new function f with y defaulting to 10
f(3, 2)  # simple call, 5 returned
f(3)  # 13 returned
function g(x::Int, y::Int)  # type restriction
    return y, x  # explicit return of a tuple
end
g(x::Int, y::Bool) = x * y  # add multiple dispatch  
g(2, true)  # second definition is invoked
methods(g)  # list all methods defined for g
(x -> x^2)(3)  # anonymous function with a call
() -> 0  # anonymous function with no arguments
h(x...) = sum(x)/length(x) - mean(x) # vararg function; x is a tuple
h(1, 2, 3)  # result is 0
x = (2, 3)  # tuple
f(x)  # error
f(x...)  # OK - tuple unpacking
s(x; a = 1, b = 1) = x * a / b  # function with keyword arguments a and b
s(3, b = 2)  # call with keyword argument
t(; x::Int64 = 2) = x  # single keyword argument
t()  # 2 returned
t(; x::Bool = true) = x  # no multiple dispatch for keyword arguments; function overwritten
t()  # true; old function was overwritten
q(f::Function, x) = 2 * f(x)  # simple function wrapper
q(x -> 2x, 10)  # 40 returned, no need to use * in 2x (means 2*x)
q(10) do x  # creation of anonymous function by do construct, useful eg. in IO
    2 * x
end
m = reshape(1:12, 3, 4)
map(x -> x ^ 2, m)  # 3x4 array returned with transformed data
filter(x -> bits(x)[end] == '0', 1:12)  # a fancy way to choose even integers from the range

As a convention functions with name ending with ! change their arguments in-place. See for example resize! in this document.

Default function argument beasts:

```julia
y = 10
f1(x=y) = x; f1()  # 10
f2(x=x, y=1) = x; f2()  # 10
f3(y=1, x=y) = x; f3()  # 1
f4(;x=y) = x; f4()  # 10
f5(;x=y, y=1) = x; f5()  # error - y not defined yet :(
f6(;y=1, x=y) = x; f6()  # 1
```
7 Variable scoping

The following constructs introduce new variable scope: function, while, for, try/catch, let, type. You can define variables as:

- global: use variable from global scope;
- local: define new variable in current scope;
- const: ensure variable type is constant (global only).

Special cases:

```julia
function f1(n)
    x = 0
    for i = 1:n
        x = i
    end
    x
end
f1(10)  # 10; inside loop we use outer local variable

function f2(n)
    x = 0
    for i = 1:n
        local x
        x = i
    end
    x
end
f2(10)  # 0; inside loop we use new local variable

function f3(n)
    for i = 1:n
        local x
        # this local can be omitted; for introduces new scope
        x = i
    end
    x
end
f3(10)  # error; x not defined in outer scope

const x = 2
x = 3  # warning, value changed
x = 3.0  # error, wrong type

function fun()  # no warning
    const x = 2
    x = true
end
fun()  # true, no warning
```

Global constants speed up execution.
The let rebinds the variable:

```julia
Fs = Array{Any}(2)
i = 1
while i <= 2
    j = i
    Fs[i] = () -> j
```
8 Modules

Modules encapsulate code. Can be reloaded, which is useful to redefine functions and types, as top level functions and types are defined as constants.

```julia
module M # module name
    export x # what module exposes for the world
    x = 1
    y = 2 # hidden variable
end

whos(M) # list exported variables
x      # not found in global scope
M.y    # direct variable access possible

# import all exported variables
# load standard packages this way
using M

#import variable y to global scope (even if not exported)
import M.y
```

9 Operators

Julia follows standard operators with the following quirks:

- `true || false` # binary or operator (singeltons only), `||` and `&&` use short-circuit evaluation
- `[1 2] .& [2 1]` # bitwise and operator (vectorized by `.`)
- `1 < 2 < 3` # chaining conditions is OK (singeltons only without `.`)
- `[1 2] .< [2 1]` # for vectorized operators need to add '.' in front
- `x = [1 2 3]` # multiplication can be omitted between a literal and a variable or a left parenthesis
- `y = [1, 2, 3]` # error
- `x .+ y` # 3x3 matrix, dimension broadcasting
- `x + y'` # 1x3 matrix
- `x * y` # array multiplication, 1-element vector (not scalar)
x .+ y # element-wise multiplication, 3x3 array

x == [1 2 3] # true, object looks the same
x === [1 2 3] # false, objects not identical

z = reshape(1:9, 3, 3)
z + x # error
z .+ x # x broadcasted vertically
z .+ y # y broadcasted horizontally

# explicit broadcast of singleton dimensions
# function + is called for each array element
broadcast(+, [1 2], [1; 2])

# broadcasting using . operator
length((randstring(10) for i in 1:5)) # 5
length.((randstring(10) for i in 1:5)) # 5-element array of 10s

Many typical matrix transformation functions are available (see documentation).

## 10 Essential general usage functions

show(collect(1:100)) # show text representation of an object
eps() # distance from 1.0 to next representable Float64
nextfloat(2.0) # next float representable, similarly provided prevfloat
isequal(NaN, NaN) # false
NaN !== NaN # true
1 == 1.0 # true
1 === 1.0 # false

fld(-5, 3), mod(-5, 3) # (-2, 1), division towards minus infinity
div(-5, 3), rem(-5, 3) # (-1, -2), division towards zero
find(x -> mod(x, 2) == 0, 1:8) # find indices for which function returns true
x = [1 2]; identity(x) === x # true, identity function
info("Info") # print information, similarly warn and error (raises error)

ntuple(x->2x, 3) # create tuple by calling x->2x with values 1, 2 and 3
isdefined(:x) # if variable x is defined (:x is a symbol)
y = Array{Any}(2); y[3] # if position 3 in array is assigned (not out of bounds or #undef)
fieldtype(typeof(1:2),:start) # get type of the field in composite type (passed as symbol)
fieldnames(typeof(1:2)) # get field names of a type

zip(1:3, 1:3) |> collect # convert iterables to iterable tuple and pass it to collect
enumerate("abc") # create iterator of tuples (index, collection element)

isempty("abc") # check if collection is empty
'b' in "abc" # check if element is in a collection
indexin(collect("abc"), collect("abrakadabra")) # [11, 9, 8] ('c' not found), needs arrays

findin("abc", "abrakadabra") # [1, 2] ('c' was not found)

"abrakadabra" # return unique elements

issubset("abc", "abcd") # check if every element in first collection is in the second

indmax("abrakadabra") # index of maximal element (3 - 'r' in this case)
findmax("abrakadabra") # tuple: maximal element and its index

filter(x->mod(x,2)==0, 1:10) # retain elements of collection that meet predicate

dump(1:2:5) # show all user-visible structure of an object

sort(rand(10)) # sort 10 uniform random variables

## 11 Reading and writing data

For I/O details refer documentation. Basic operations:
The Julia Express

• readdlm, readcsv: read from file
• writedlm, writecsv: write to a file

Warning! Trailing spaces are not discarded if \texttt{delim=' '} in file reading.

12 Random numbers

Basic random numbers:

\begin{verbatim}
srand(1)  # set random number generator seed to 1
rand()   # generate random number from U[0,1)
rand(3, 4) # generate 3x4 matrix of random numbers from U[0,1]
rand(2:5, 10) # generate vector of 10 random integer numbers in range from 2 to 5
randn(10) # generate vector of 10 random numbers from standard normal distribution
\end{verbatim}

Advanced randomness form Distributions package:

\begin{verbatim}
using Distributions # load package
sample(1:10, 10) # single bootstrap sample from set 1-10
b = Beta(0.4, 0.8) # Beta distribution with parameters 0.4 and 0.8
mean(b) # expected value of distribution b
rand(b, 100) # 100 independent random samples from distribution b
\end{verbatim}

13 Statistics and machine learning

Visit \url{http://juliastats.github.io/} for the details (in particular R-like data frames).

Starting with Julia version 0.4 there is a core language construct \texttt{Nullable} that allows to represent missing value (similar to Haskell \texttt{Maybe}).

\begin{verbatim}
u1 = Nullable(1) # contains value
u2 = Nullable{Int64}() # missing value
get(u1) # OK
get(u2) # error - missing
isnull(u1) # false
isnull(u2) # true
\end{verbatim}

14 Plotting

There are several plotting packages for Julia: Plots, Gadfly and PyPlot. Here we show how to use on PyPlot.

\begin{verbatim}
using PyPlot # have to wait a bit due to compilation
srand(1) # second plot
x, y = randn(100), randn(100)
scatter(x, y)
\end{verbatim}

15 Macros

You can define macros (see documentation for details). Useful standard macros.

Assertions:

\begin{verbatim}
@assert 1 == 2 "ERROR" # 2 macro arguments; error raised
using Base.Test # load Base.Test module
@test 1 == 2 # similar to assert; error
@test_throws DomainError sqrt(-1) # passed, sqrt(-1) is not possible
\end{verbatim}

Function broadcasting:
The Julia Express

\[
t(x::\text{Float64}, y::\text{Float64} = 1.0) = x * y
\]

\[
t(1.0, 2.0)
\]
# OK

\[
t(\{1.0 2.0\})
\]
# error

\[
t.\{1.0 2.0\}
\]
# OK

\[
t(\{1.0 2.0\}, 2.0)
\]
# error

\[
t.\{1.0 2.0\}, 2.0\}
\]
# OK

\[
t(2.0, \{1.0 2.0\})
\]
# OK

\[
t.\{1.0 2.0\}, \{1.0 2.0\}\}
\]
# OK

\[
t.\{1.0, 2.0\}, \{1.0 2.0\}\}
\]
# OK

Benchmarking:

\[
\text{@time} [x \text{for } x \text{in } 1:10^6].'
\]
# print time and memory

\[
\text{@timed} [x \text{for } x \text{in } 1:10^6].'
\]
# return value, time and memory

\[
\text{@elapsed} [x \text{for } x \text{in } 1:10^6]
\]
# return time

\[
\text{@allocated} [x \text{for } x \text{in } 1:10^6]
\]
# return memory

\[
tic()
\]
# start timer

\[
toc()
\]
# stop timer and print time

\[
tic();\text{toq()}
\]
# stop timer and return time

16 Plotting

A simple example adapting

https://matplotlib.org/1.2.1/examples/pylab_examples/histogram_demo.html:

\[
\text{using Distributions}
\]
\[
\text{using PyPlot}
\]
\[
\text{mu, sigma} = 100, 15
\]
\[
x = \text{mu} + \text{sigma} \cdot \text{randn}(10000)
\]
\[
n, \text{bins}, \text{patches} = \text{plt}[\text{:hist}](x, 50, \text{normed}=1,
\]
\[
\text{facecolor=\"green\", alpha=0.75})
\]
\[
y = \text{pdf}(\text{Normal(mu, sigma)}, \text{bins})
\]
\[
\text{plot(bins, y, \"r--", linewidth=1})
\]
\[
\text{xlabel\("Smarts\")}
\]
\[
\text{ylabel\("Probability\")}
\]
\[
\text{title(\"Histogram of IQ: \(\mu=100, \sigma=15\)\")}
\]
\[
\text{axis([40, 160, 0, 0.03])}
\]
\[
\text{grid(true)}
\]

producing: