The Julia Express

Bogumił Kamiński

June 9, 2016

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 Taking it all together example 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; see `Task`;
8) two-way integration with C and Fortran.

You can find current Julia documentation at http://julia.readthedocs.org/en/latest/manual/.

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

```
versioninfo()
# Julia Version 0.4.0
# Commit 0ff703b* (2015-10-08 06:20 UTC)
# Platform Info:
# System: Windows (x86_64-w64-mingw32)
# CPU: Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
# WORD_SIZE: 64
# BLAS: libopenblas (USE64BITINT DYNAMIC_ARCH NO_AFFINITY Sandybridge)
# LAPACK: libopenblas64_
# LIBM: libopenlibm
# LLVM: libLLVM-3.3
```

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 sugestions 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):

```
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 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 worskspace - 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):

[1] The rocket ship clip is free for download at http://www.clipartlord.com/free-cartoon-rocketship-clip-art-2/.

```
"Sieve of Eratosthenes function docstring"
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 i
                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:

```
  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)`:

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

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

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

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

```
  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:

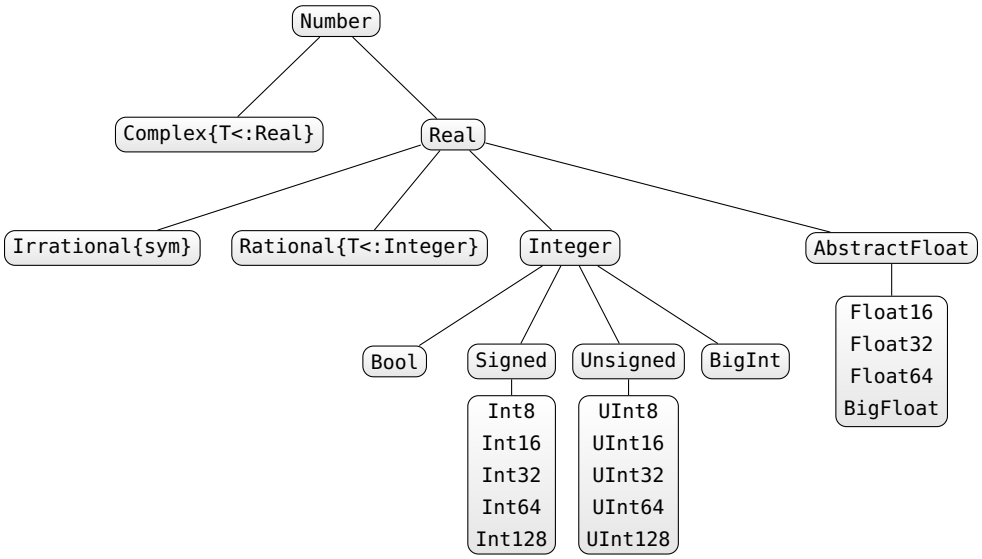


Figure 1: Hierarchy of numeric types

```
typeof("abc")              # ASCIIString returned which is a AbstractString subtype
isa("abc", AbstractString) # true
isa(1, Float64)            # false, integer is not a float
isa(1.0, Float64)          # true
isa(1.0, Number)           # true, Number is abstract type
super(Int64)               # supertype of Int64
subtypes(Real)             # subtypes of bastract type Real
```

It is possible to perform calculations using arbitrary precision arithmetic or rational numbers:

```
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:

```
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:

```
()            # 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 Chars
Array{Int64}(0, 0)   # degenerate 0x0 array of Int64
cell(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
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 (vcat function)
[1; 2]               # vector (hvcat function)
[1 2 3; 1 2 3]       # 2x3 matrix (hvcat function)
[1; 2] == [1 2]'     # false, different array dimensions
[(1, 2)]             # 1-element vector
collect((1, 2))      # 2-element vector by tuple unpacking
[[1 2] 3]            # append to a row vector (hcat)
[[1; 2]; 3]          # append to a column vector (vcat)
```

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] # 2x1x3 array of Int64
ndims(a)             # number of dimensions in a
eltype(a)            # type of elements in a
length(a)            # number of elements in a
size(a)              # tuple containing dimension sizes of a
vec(a)               # cast array to vetor (single dimension)
squeeze(a, 2)        # remove 2nd dimension as it has size 1
sum(a, 3)            # calculate sums for 3rd dimensions, similarly: mean, std,
                     # prod, minimum, maximum, any, all
count(x -> x > 0, a) # count number of times a predicate is true, similar: all, any
```

Access functions:

```
a = linspace(0, 1)   # LinSpace{Float64} of length 50
a[1]                 # get scalar 0.0
a[end]               # get scalar 1.0 (last position)
a[1:2:end]           # every second element from range, LinSpace{Float64}
a[repmat([true, false], 25)] # select every second element, Array{Float64,1}
a[[1, 3, 6]]         # 1st, 3rd and 6th element of a, Array{Float64,1}
sub(a, 1:2:50)       # view into subarray of a
endof(a)             # last index of the collection a
```

Observe the treatment of trailing singleton dimensions:

```
a = reshape(1:12, 3, 4)
a[:, 1:2]          # 3x2 matrix
a[:, 1]            # 3-element vector
a[1, :]            # 1x4 matrix
a[:, :, 1, 1]      # works 3x4 matrix
a[:, :, :, [true]] # wroks 3x4x1x1 matrix
a[1, 1, [false]]   # works 3x4x0 matrix
```

Array assignment:

```
x = 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:

```
x = cell(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:

```
cell(2)::Array{Any, 1}      # vector of Any
[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:

```
type 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 replacing `type` by `immutable`. There are also union types (see documentation for details).

4.4 Dictionaries

Associative collections (key-value dictionaries):

```
x = Dict{Float64, Int64}()        # empty dictionary mapping floats to integers
y = Dict("a"=>1, "b"=>2)          # filled dictionary
y["a"]                            # element retrieval
y["c"]                            # error
y["c"] = 3                        # added element
haskey(y, "b")                    # check if y contains key "b"
keys(y), values(y)                # tuple of iterators returning keys and values in y
delete!(y, "b")                   # delete key from a collection, see also: pop!
get(y,"c","default")              # return y["c"] or "default" if not haskey(y,"c")
```

Julia also supports operations on sets and dequeues, 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
y::Float64 = 1.0 # y must be Float64, not possible in global scope
                 # if in global scope performs assertion on y type when it exists
```

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 1==2
    z = 2
else
    a = 3
end         # after this a = 3 and z is undefined

1==2 ? "A" : "B" # standard ternary operator
```

```
i = 1
while true
    i += 1
    if i > 10
      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              # x is introduced in loop outer scope
```

You can define your own functions:

```
f(x, y = 10) = x + y           # new function f with y defaulting to 10
                               # last result returned
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:

```
y = 10
f1(x=y) = x; f1()      # 10
f2(x=y,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:

```
t                  # error, variable does not exist
f() = global t = 1
f()                # after the call t is defined globally

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:

```
Fs = cell(2)
i = 1
while i <= 2
  j = i
  Fs[i] = () -> j
```

```
  i += 1
end
Fs[1](), Fs[2]() # (2, 2); the same binding for 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); new binding for j

Fs = cell(2)
i = 1
for i in 1:2
  j = i
  Fs[i] = () -> j
end
Fs[1](), Fs[2]() # (1, 2); for loops and comprehensions rebind variables
```

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.

```
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-circut evaluation
[1 2] & [2 1]    # bitwise and operator
1 < 2 < 3        # chaining conditions is OK (singeltons only)
[1 2] .< [2 1]   # for vectorized operators need to add '.' in front
x = [1 2 3]
2x + 2(x+1)      # multiplication can be omitted between a literal and a variable or a left parenthesis
y = [1, 2, 3]
x + y  # 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 singelton dimensions
# function + is called for each array element
broadcast(+, [1 2], [1; 2])
```

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) # true
NaN == NaN        # false
NaN === NaN       # true
isequal(1, 1.0)   # true
1 == 1.0          # true
1 === 1.0         # false
isfinite(Inf)     # false, similarly provided: isinf, isnan
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 = cell(2); isassigned(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
1:5 |> exp |> sum # function application chaining
zip(1:5, 1:3) |> collect # convert iterables to iterable tuple and pass it to collect
enumerate("abc")  # create iterator of tuples (index, collection element)
collect(enumerate("abc"))
isempty("abc")    # check if collection is empty
'b' in "abc"      # check if element is in a collection
indexin(collect("abc"), collect("abrakadabra")) # [11, 9, 0] ('c' not found), needs arrays
findin("abc", "abrakadabra") # [1, 2] ('c' was not found)
unique("abrakadabra") # return unique elements
issubset("abc", "abcd") # check if every element in fist 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:

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

Warning! Trailing spaces are not discarded if `delim=' '` in file reading.

12 Random numbers

Basic random numbers:

```
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 form 2 to 5
randn(10)     # generate vector of 10 random numbers from standard normal distribution
```

Advanced randomness form `Distributions` package:

```
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
                    # see documentation for supported distributions
mean(b)             # expected value of distribution b
                    # see documentation for other supported statistics
rand(b, 100)        # 100 independent random samples from distribution b
```

13 Statistics and machine learning

Visit 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 `Nullable` that allows to represent missing value (similar to Haskell Maybe).

```
x1 = Nullable(1)       # contains value
x2 = Nullable{Int64}() # missing value
get(x1)                # OK
get(x2)                # error - missing
isnull(x1)             # false
isnull(x2)             # true
```

14 Plotting

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

```
using Winston # load Winston plotting package
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")

using Gadfly
srand(1) # second plot
x, y = randn(100), randn(100)
plot(x = x, y = y) # need to give module name as Winston is also loaded
```

15 Macros

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

```
@assert 1 == 2 "ERROR"        # 2 macro arguments; error raised
using Base.Test               # load Base.Test module
@test 1 == 2                  # similar to assert; error
@test_approx_eq 1 1.1         # error
@test_approx_eq_eps 1 1.1 0.2 # no error
```

Function vectorization:

```
t(x::Float64, y::Float64 = 1.0) = x * y
t(1.0, 2.0)               # OK
t([1.0 2.0])              # error
@vectorize_1arg Float64 t # vectorize first argument
t([1.0 2.0])              # OK
t([1.0 2.0], 2.0)         # error
@vectorize_2arg Float64 t # vectorize two arguments
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
```

Benchmarking:

```
@time [x for x in 1:10^6].'    # print time and memory
@timed [x for x in 1:10^6].'   # return value, time and memory
@elapsed [x for x in 1:10^6]   # return time
@allocated [x for x in 1:10^6] # return memory
tic()                          # start timer
toc()                          # stop timer and print time
tic(); toq()                   # stop timer and return time
```

16 Taking it all together example

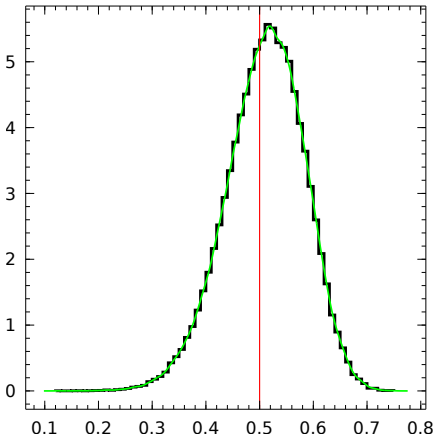
```
using Winston
using Distributions
using KernelDensity

# generate 100 observations from correlated normal variates
srand(1)
n = 100
dist = MvNormal([0.0; 0.0], [1.0 0.5; 0.5 1.0])
r = rand(dist, n)'

# create 100 000 bootstrap replications and fetch time and memory used
@time bootcor = Float64[cor(r[sample(1:n, n),:])[1, 2] for i in 1:10^5]
# calculate kernel density estimator
kdeboot = KernelDensity.kde(bootcor)

h = hist(bootcor, 100)
p = plothist((h[1], h[2]/1000), linewidth=5)
add(p, Curve(kdeboot.x, kdeboot.density,
             color=(0,1,0), linewidth=3)) # color in RGB
add(p, Slope(Inf, (0.5, 0), color=(1,0,0)))
savefig(p, "hist.pdf")
```

This is what you should obtain:


5
4
3
2
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8