Names: Riley Campbell, Riley Kopp, Caelan Klein, Timothy Ford

Language: Julia 1.0.3

Section 1: Comparison to C++, Java, and Python

1. **Q1**

Julia is designed to be a flexible, dynamically typed language for scientific and numerical computations. The goal is to have performance more similar to statically typed languages but with the added ease and advantages of being dynamic. Performance is achieved using a JIT compiler (just-in-time compiler), implemented with LLVM, and by placing a strong focus on concurrent/parallel programming.

Julia is/has:

- Free and open source
- No performance difference between user-defined types and built-ins
- Designed for parallelism/distributed computation
- Call C functions directly
- Integration into Jupyter toolkit

2. **Q2**

Julia is mainly designed to be used in numerical/scientific applications. Thus, it compares best to languages such as Python and R. However, it compares and competes most closely to Python.

Julia	Python
● Faster ● Math friendly syntax ● Powerful language features like Multiple Dispatch	● 0-indexed (like it should be) ● Much better support/stability ● More packages that are generally better supported

3. **Q3**

- (Con) Multiple File management was not intuitive
- (Con) Package management recompiles on runtime
- (Con) Package management suppresses errors on failed install so debugging is hard
- (Con) Concatenation is a * (violation of orthogonality)
- (Con) 1 indexed
- (Con) Required adding working directory to environment path to find modules
- (Con) No way to delete variables. Recommendation is to replace with smaller sized var
- (Pro) Much faster than similar dynamic languages
- (Pro) Allows Unicode
- (Pro) Powerful language features like Multiple Dispatch
- (Pro) Math friendly syntax

4. **Q4**

Portability:

C++: Similar to C++ in that it is a desktop language and is thus about as portable

Java: Java is substantially more portable due to the JVM being nearly everywhere

Python: As both Julia and Python require their interpreter, they are nearly equally portable However, due to Pythons popularity, subsets such as MicroPython are more likely to be made.

Simplicity:

Views on simplicity are based on our teams' experience of having worked with Julia.

C++: About the same as C++

Java: About the same as Java

Python: Python is more simple than Julia due to cleaner/easier syntax. Since Julia is performance focused, more of the nuts and bolts are exposed and thus syntax can be cluttered.

Orthogonality:

C++: Julia is more orthogonal than C++.

Java: Julia is more orthogonal than Java.

Python: Python is very orthogonal and Julia follows close suit since the syntax is largely similar to Python with slightly more verbosity. However, certain things like concatenation are not as orthogonal as they are in Python (discussed above).

Reliability:

C++: C++ being an mature language will be much more reliable than Julia especially since Julia's focus is on numerical computing/data science whle C++ focuses more on desktop applications.

Java: Java will also be more reliable than Julia for the same reasons as above.

Python: While reliability should be closer to Python, Julia focuses much more on being High performance and productivity and thus reliability falters slightly compared to python

Section 2: Syntax, OOP

1. **Q1**

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0; in particular, Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols), and a few other letter-like characters (e.g. a subset of the Sm math symbols) are allowed. Subsequent characters may also include ! and digits (0-9 and other characters in categories Nd/No), as well as other Unicode code points: diacritics and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Operators like + are also valid identifiers but are parsed specially. In some contexts, operators can be used just like variables; for example (+) refers to the addition function, and (+) = f will reassign it. Most of the Unicode infix operators (in category Sm), such as ⊕, are parsed as infix operators and are available for user-defined methods (e.g. you can use const ⊗ = kron to define ⊗ as an infix Kronecker product). Operators can also be suffixed with modifying marks, primes, and sub/superscripts, e.g. +̂ₐ″ is parsed as an infix operator with the same precedence as +. The only explicitly disallowed names for variables are the names of built-in statements. (https://docs.julialang.org/en/v1/manual/variables/)

$$
\begin{aligned}
t &::= \texttt{Any} \mid \texttt{Union}\{t_1, .., t_n\} \mid \texttt{Tuple}\{a_1, .., a_n\} \mid name\{a_1, .., a_n\} \mid t \texttt{ where } t_1 \texttt{<:T<:} t_2 \\
&\quad \mid \texttt{T} \mid \texttt{Type}\{a\} \mid \texttt{DataType} \mid \texttt{Union} \mid \texttt{UnionAll} \\
a &::= t \mid v
\end{aligned}
$$

Fig. 2.1 EBNF example

2. Q2

Julia doesn’t use classes in the traditional sense. It used modules that can be nested inside of other modules to simulated an extension. The relationship between module B and module C in figure 2.2 shows an example of this.

```
julia> module A
           a = 1 # a global in A's scope
       end;

julia> module B
           module C
               c = 2
           end
           b = C.c    # can access the namespace of a nested global scope
                      # through a qualified access
           import ..A # makes module A available
           d = A.a
       end;

julia> module D
           b = a # errors as D's global scope is separate from A's
       end;
ERROR: UndefVarError: a not defined

julia> module E
           import ..A # make module A available
           A.a = 2    # throws below error
       end;
ERROR: cannot assign variables in other modules
```

Fig. 2.2 extensions example

3. Q3

Modules in Julia are separate variable workspaces. Modules allow you to create top-level definitions (aka global variables) without worrying about name conflicts when your code is used together with somebody else's. Within a module, you can control which names from other modules are visible (via importing) and specify which of your names are intended to be public (via exporting). Functions and variables share the same namespace in Julia, as well as each module is a namespace. Julia's package manager is a little unusual in that it is declarative rather than imperative. This means that you tell it what you want, and it figures out what versions to install (or remove) to satisfy those requirements optimally – and minimally. So rather than installing a package, you just add it to the list of requirements and then "resolve" what needs to be installed. This means that if some package had been installed because it was needed by a previous version of something you wanted, and a newer version doesn't have that requirement anymore, updating will remove that package. Julia uses the dot notation for namespace, see fig 2.2 for an example of this syntax (https://docs.julialang.org/en/v1/manual/modules/index.html).

4. **Q4**

Julia allows for function overloading by allowing the variation of parameters. This is called Dispatch in Julia. You could write functions to calculate the volume of shape and modify the parameters to handle different types of shapes.(https://docs.julialang.org/en/v1/manual/methods/#)

For a sphere:
calcVolume(r::Float64) = 0.75 * π * r * r * r
For a box:
calcVolume(width::Float64, depth::Float64, height:Float) = width * depth * height

Section 3: Parsing, Binding, Types

1. **Q1**

Julia is dynamic but it does support some static typing in order to make the code more efficient:
EX:

```
julia> x::Int8 = 30 // Statically typed
julia> x = 30 // Dynamically typed
```

2. **Q2**

Julia is statically scoped:
EX:

```
julia> module Example:
           x = 1
           run() = x
       end

julia> x = -1;

julia> Example.run()
       1
```

3. **Q3**

Julia is parsed from left to right. For A && B, B is only evaluated if A is true. For A || B, B is only evaluated if A is false.

4. **Q4**

Julia has many of the same data types as C (Most of the Julia packages are written in C)
Uint<8, 16, 64, 128> : 0 - 255 for uint8
Int<8, 16, 64, 128> : -127 - 127 for int8

Float<16,32,64> : +/- 6.55 x 10^4 for 16 bits

String: (Unicode)
Tuples: (Any, Any)

Section 4: Control flow, Function, Specialties

1. **Q1**

A typical selection structure in Julia is the if-elseif-else statement.

```
if x < y
   println("x is less than y")
elseif x > y
   println("x is greater than y")
else
   println("x is equal to y")
end
```

The first construct to evaluate to true is executed, and all others are ignored. The if blocks do not introduce local scope. They can also return values. The value returned is the value of the last executed statement in the branch that was chosen. The selection statements in Julia can only evaluate expressions that return a true or false value. For short conditional expressions ternary operators, *a* ? *b* : *c*, can be used. Where a is the condition expression

and *b* and *c* are executed based on *a*. If a is true then *b* is executed, if *a* is false then *c* is executed.

Julia contains two repetition structures the while loop and the for loop. If the iterative variable has not been declared in the current scope it will only be visible within the for loop. The for loop can iterate over any container. In this case, the keyword in or ∈ is typically used instead of =, to make the code easier to read. A loop can be terminated before the evaluation of the test condition by using a break statement. The continue statement can be used to skip to the iteration of the loop.

Below are some examples of the looping structures:

```
julia> i = 1;
julia> while true
                    println(i)
if i >= 3
                    break
end
global i += 1
 End
1
2
3

julia> for j = 1:1000
         println(j)
         if j >= 3
            break
         end
      end
1
2
3

julia> for i in [1,4,0]
         println(i)
      end
1
```

```
4
0
```

Multiple nested for loops can be combined into a single outer loop. Using this syntax, iterables may still refer to outer loop variables. However, a break statement inside this kind of loop exits the entire nest of loops.

```
julia> for i = 1:2, j = 3:4
           println((i, j))
       end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

2. **Q2**

Julia function arguments follow a convention called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable bindings, but the values they refer to are identical to the passed values. Modifications to mutable values made within a function will be visible to the caller.

3. **Q3**

Tasks (aka Coroutines):

Tasks allow computations to be suspended and resumed in a flexible manner. When a piece of computing work such as a function is designated as a Task, it becomes possible to interrupt it by switching to another Task. The Task can then be resumed at a later time. Once resumed it will pick up right where it left off. Switching tasks does not use any space, so any number of task switches can occur without consuming the call stack. Switching among tasks can also occur in any order. A good use case for this feature is the producer-consumer problem. Tasks, allow the producer and consumer to both run, as long as they need to, passing values back and forth as necessary using channels. A Channel is a waitable first-in-first-out queue which can have multiple tasks reading from and writing to it. In the below example we create a function to produce values and add them to a Channel using the **put!** call. The channel constructor allows us to run the task that it is bound to. We then use **take!** to obtain values from the producer.

```
julia> function producer(c::Channel)
          put!(c, "start")
          for n=1:4
             put!(c, 2n)
          end
          put!(c, "stop")
       end;

julia> chnl = Channel(producer);
julia> take!(chnl)
"start"
julia> take!(chnl)
2
julia> take!(chnl)
4
julia> take!(chnl)
6
julia> take!(chnl)
8
julia> take!(chnl)
"stop"
```

Between calls to **put!**, the producer's execution is suspended and the consumer has control. The returned channel can also be used as an iterable object in a for loop, in which case the loop variable takes on all the produced values. The loop is terminated when the channel is closed. Multiple channels can be bound to a task, and vice-versa.

```
julia> for x in Channel(producer)
          println(x)
       end
start
2
4
6
8
Stop
```

4. **Q4**

Calling C and Fortran Code:

Julia makes it simple and efficient to use the many high-quality, mature libraries for numerical computing already written in C and Fortran. This is accomplished just by making an appropriate call with **ccall** syntax, which looks like an ordinary function:

```
julia> t = ccall(:clock, Int32, ())
2292761
```

The arguments to **ccall** are:

1. A (:function, "library") pair (most common), or a :function name symbol or "function" name string (for symbols in the current process or libc),or a function pointer.
2. The function's return type.
3. A tuple of input types, corresponding to the function signature.
4. The actual argument values to be passed to the function, if any; each is a separate parameter.

Some things to note are:

- The (:function, "library") pair, return type, and input types must be literal constants. Julia automatically inserts calls to the **Base.cconvert** function to convert each argument to the specified type.
- The code to be called must be available as a shared library.
- Memory allocation and deallocation must be handled in the libraries being used. Passing an object allocated in Julia to be freed by an external library is also invalid.

C wrapper example:

```
mutable struct gsl_permutation
end

# The corresponding C signature is
#     gsl_permutation * gsl_permutation_alloc (size_t n);
function permutation_alloc(n::Integer)
    output_ptr = ccall(
        (:gsl_permutation_alloc, :libgsl), # name of C function  and library
        Ptr{gsl_permutation},              # output type
```

```
        (Csize_t,),                 # tuple of input types
        n                           # name of Julia variable to pass in
    )
    if output_ptr == C_NULL # Could not allocate memory
        throw(OutOfMemoryError())
    end
    return output_ptr
end
```

Fortran wrapper example:

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
    n = length(DX)
    incx = incy = 1
    product = ccall((:ddot_, "libLAPACK"),
                Float64,
                        (Ref{Int32}, Ptr{Float64}, Ref{Int32}, Ptr{Float64},
                    Ref{Int32}),
                n, DX, incx, DY, incy)
    return product
end
```