A Study of Julia

November 22nd, 2022

CSIS 420
Structures of Programming Languages
Fall 2022
Cassandra Wischhoefer
George Fox University

Table of Contents:

Intro to Julia

Julia is a relatively new, high-level, dynamic programming language that was designed for the scientific community to be fast and general purpose. It easily integrates libraries from other languages as well as Julia libraries equipped for various mathematical solutions [2]. It was designed to give users the speed of C/C++, the statistical capabilities of R, and the easy-to-learn structure of Python [4]. As the data science and machine learning industries have grown tremendously over recent years, Julia has been found sufficiently equipped to handle problems faster and more effectively. Julia is being used in self-driving cars, 3-D printers, applications in precision medicine, augmented reality, genomics, machine learning, and risk management, along with many other areas in the technology industry. Among data scientists, Python is the most popular programming language but Julia has been dubbed the “future language” of data science and artificial intelligence [12].

The History of Julia

The Julia language was invented in 2009 by MIT research scientists Stefan Karpinski, Jeff Bezanson, Viral B. Shah, and Alan Edelman. Julia today is ranked among the top programming languages, being used by Amazon, Apple, Facebook, and NASA, but when the founders began building it, their goal was much smaller. Karpinski stated, “We were really just building something for ourselves,”. The initial drive behind creating Julia was a high-level language that held the functionality of MATLAB and R with the speed of C or Ruby, the “best of both worlds” as Karpinski put it. The fact that such a language hadn’t existed, frustrated Karpinski, and this sparked the beginning of huge success in the computer language world [1].

A few years before the release of Julia, Karpinski was working as a data scientist and software engineer at Etsy, creating algorithms using MATLAB, C, and R that made personalized recommendations for shoppers. In his free time, he tried creating a system that would end up replacing the programming languages he used at his day job [1].

Karpinski expressed his idea to Viral Shah - who had previously worked at Interactive Supercomputing for world-renowned mathematician Alan Edelman - and he agreed to help. Shah then reached out to his colleagues Jeff Bezanson and Alan Edelman. The four covered all areas of expertise. Bezanson had a mind for compilers that paired nicely with Shah’s computational science background. Edelman had significant experience in this area, and Karpinski had the drive needed to see it through to the end. At this time, object-oriented languages were becoming very popular, so a large focus was on Julia’s object-oriented design, as it would hopefully make the language more popular. Karpinski expressed that the four wanted “a Goldilocks programming language—one that was high level and low level at the same time, depending on how you used it,”. This gave way to the Julia project: an open-source, dynamic programming language that since has spawned the consulting company - Julia Computing. The company has since raised $4.6 million in funding [1].

Compiler

Julia uses an LLVM (low-level virtual machine) based JIT (Just In Time) compiler which is a combination of AOT (Ahead Of Time) and interpretation. The code is dynamically compiled during the execution of the program rather than before execution. The Julia compiler is compatible with macOS, Windows, and Linux/Unix which makes it widely accessible. Programming code can be written in many ways. The first is through Julia's interactive command-line REPL the is built into the Julia executable. Most of the examples shown later in the paper use this format. The second is through Desktop IDEs such as Juno, VS-Code, Julia Studio, and JuliaBox. Lastly, Julia can be written in web-based environments such as Jupiter Notebooks and Google Colaboratory [4].

To evoke the Julia REPL, simply typing "julia" in the Terminal will bring up the interactive command line. There, the user can write code that is then added in the global scope of that opened instance. To compile a Julia file, the syntax is "julia <filename.jl>".

Control Flow

Julia provides most control flow mechanisms that are standard to high-level programming languages - compound expressions, conditional evaluation, short-circuit evaluation, repeated evaluation, and exception handling [11].

For basic mechanisms, most of what is used in Julia is very similar to other high-level languages. One big difference is the use of the "end" keyword to finish a block, therefore unlike Python, indentation does not play as large of a role in Julia [11]. Here is the basic syntax of an if, elseif, and else statement.

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

Julia's short-circuit evaluation is similar to other high-level languages. The logical AND is supported with "&&" and logical OR with "||". They don't necessarily evaluate their second argument, so if the first argument in a logical OR evaluates to false, the second side is not even considered. These operators can be used to shorten if statements and this behavior is frequently used in Julia. An If <condition> then <statement> can be replaced by a <condition> && <statement> [11]. Below is an example of this.

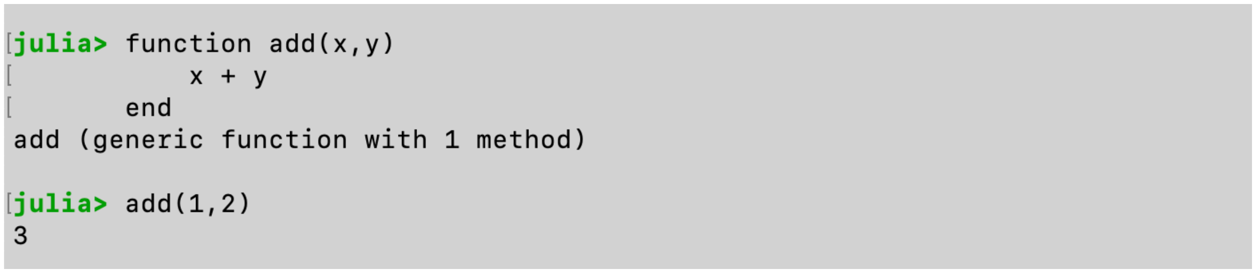
```
[julia> function less(x,y)
[            x < y && print("x is less than y")
[        end
less (generic function with 1 method)

[julia> less(1,2)
x is less than y
```

Julia supports looping with the while, for, and foreach blocks very similar to the way Python implements them, except for Julia's use of "end". Many built-in exceptions are incorporated into Julia: checked, error, and runtime exceptions. It also provides the user the ability to define unique exceptions [11].

Functions

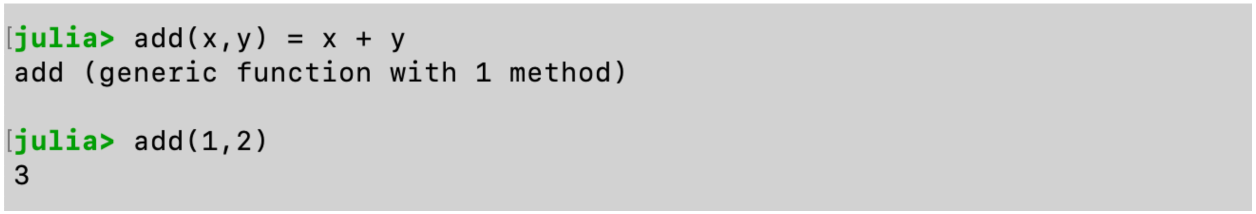
It is important to begin this section by stating that Julia functions are very important to the structure and effectiveness of the program. All code that is performance critical should be within a function. This is because code inside of functions tends to run faster than top-level code, due to the way Julia's compiler works. Not only this, but functions are more reusable and testable. "Write functions, not just scripts" is a recommendation of Julia's style guidelines [11].

A Julia function is an object that maps a tuple of arguments to a return value. They can be altered and affected by the global state of the program. Here is the basic syntax for defining a function:



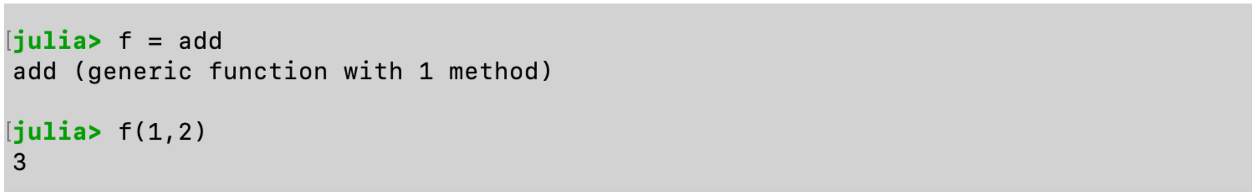
```
[julia> function add(x,y)
[           x + y
[       end
add (generic function with 1 method)

[julia> add(1,2)
3
```

Another way to define a function that uses a more terse syntax is the simple function definition, demonstrated below. In this form, the value of the function must be a single expression or compound expression. This function declaration is very common and recommended because it increases readability and reduces typing for the programmer [11]. This example is equivalent to the function definition above.



```
[julia> add(x,y) = x + y
add (generic function with 1 method)

[julia> add(1,2)
3
```

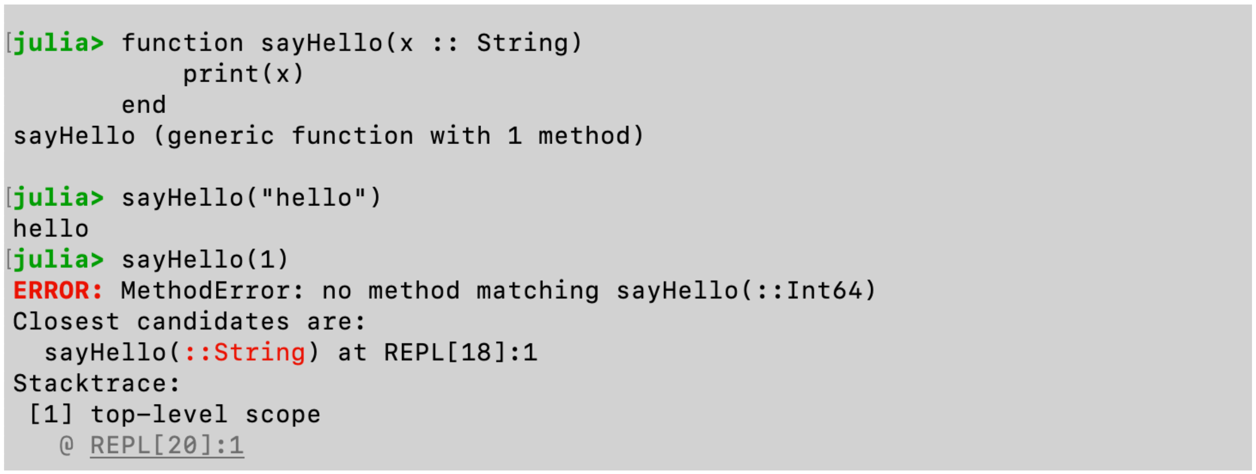
When a function is called without parenthesis, the variable stores the function object. Therefore, in the example below, the add function can also be called using the variable f [11].



```
[julia> f = add
add (generic function with 1 method)

[julia> f(1,2)
3
```

Along with other languages such as Python and Ruby, Julia functions pass arguments using the “pass-by-sharing” (also known as “pass-by-assignment”) convention [11]. This means that every variable is a reference to an object and the actual parameter value is assigned to the formal parameter [10]. When accessing a variable in a function that is defined in the global state, “global” must be declared before the name [5].

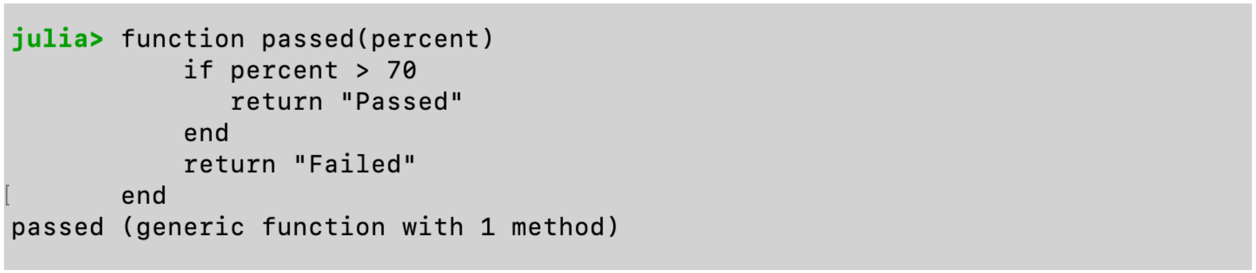
While it is not required to specify parameter types, declaring types is done by appending “::” with the type. This specification means that it will only be callable when the passed value is a subtype of the abstract of that type [11]. For example, the below function will only pass if the type passed is of type String, and it fails on type Int.



```
[julia> function sayHello(x :: String)
            print(x)
        end
sayHello (generic function with 1 method)

[julia> sayHello("hello")
hello
[julia> sayHello(1)
ERROR: MethodError: no method matching sayHello(::Int64)
Closest candidates are:
  sayHello(::String) at REPL[18]:1
Stacktrace:
 [1] top-level scope
   @ REPL[20]:1
```

When choosing not to specify the argument type, it does little to change the performance. When the program is run, Julia compiles a specialized version of the function for the actual argument types passed by the caller. For example, back in the first add() function, a compiled version of add(x :: Integer, y :: Integer) will be created because it is later called with those same types. This is then re-used for all future calls using those types [11].

The value returned by a Julia function is the last line in that block unless otherwise specified with the “return” keyword. Because of this, the return keyword is often pointless to include unless it is within a conditional statement. For example in the function below, if the percent is above 70, then the return must be specified to quit. If this conditional statement isn’t met and the program proceeds, the return keyword is not needed, because it is located on the last line and that value will be returned regardless of the specification [11].



```
julia> function passed(percent)
            if percent > 70
               return "Passed"
            end
            return "Failed"
[       end
passed (generic function with 1 method)
```

Return types can be specified in a similar way to function parameters. By using the "::" and type name, this will convert the returned value to the specified type. For example, the "convertToFloat" function below takes in an integer and converts it into a float as it returns. This is a great functionality for mathematical programs as no extra converting code is required. For functions that do not need to return, the "nothing" keyword is used [11].

```
[julia> function convertToFloat(x::Int64)::Float32
[           return x
       end
convertToFloat (generic function with 2 methods)

[julia> convertToFloat(5)
5.0f0
```

Functions can also return multiple values as shown below. The return values are returned in the form of a tuple, a common use of tuples in this language.

```
[julia> function addMultiply(x,y)
            add = x + y
            mult = x * y
            return add, mult
       end
addMultiply (generic function with 1 method)
```

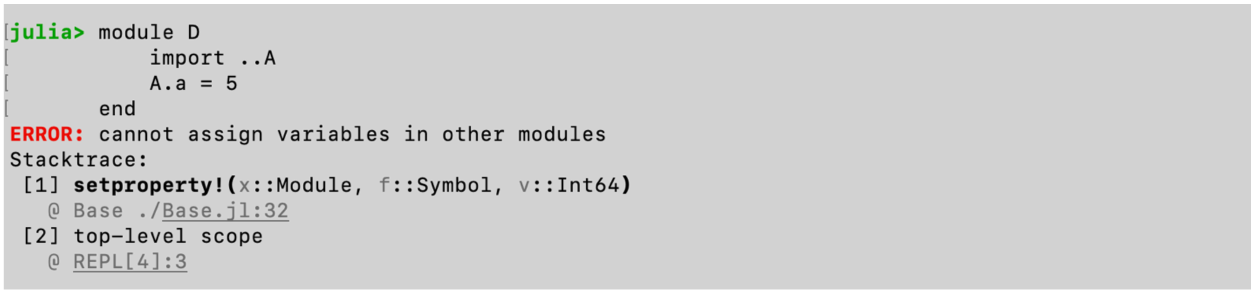
One final thing to note is Julia's convention to append a bang (!) to all function names that change one or more of its arguments. This warns the user that the function is not pure and can have dangerous side effects if the computation is not expected [3].

Scoping

Julia uses lexical scoping [11], meaning that a variable's scope can be statically determined [10]. A scope nested inside of another scope can see the variables in all outer scopes that it is within, while outer scopes cannot see variables in their inner scopes [11]. In Julia, there is no all-encompassing global scope, rather each block introduces its own global scope. The "using" and "import" statements or Julia's dot notation is used to introduce the global scope of another module into the current module [11].

```
[julia> module A
[           a = 1
[       end
Main.A

[julia> module B
[           b = a
[       end
ERROR: UndefVarError: a not defined
Stacktrace:
 [1] top-level scope
   @ REPL[2]:2

[julia> module C
[           import ..A
[           c = A.a
[       end
Main.C
```

While this makes accessing other scopes possible and provides easier access to needed variables, one barrier is that variables cannot be changed inside a module that is importing them. To avoid allowing other modules to import the current block, the word “local” can be used in the variable declaration. This locks that variable within the scope it is declared [11].



```
[julia> module D
[           import ..A
[           A.a = 5
[       end
ERROR: cannot assign variables in other modules
Stacktrace:
 [1] setproperty!(x::Module, f::Symbol, v::Int64)
   @ Base ./Base.jl:32
 [2] top-level scope
   @ REPL[4]:3
```

Explicit declaration in Julia is supported but often tedious and unnecessary. Declaring a variable as local declares that in the local scope, regardless of an existing variable in the outer scope. As stated before, this is tedious because Julia does not require this specification. If the current scope is global, the new variable will be global and if the current scope is local, that new variable is local to that innermost local scope [11].

Data Types

Julia, as mentioned earlier, supports dynamic typing, but also has the advantages of static typing because it is possible to assign specific types to values. When types are omitted at variable declaration, Julia allows assigns the type to Any which is how Julia supports polymorphism. One drawback to this is it may lead to a small cost in computation time, but in most cases, it is insignificant. Julia values have types and variables do not; variables are simply names that are bound to values [11].

Julia supports a variety of numeric types such as Int8, Int16, Int32, Int64, Int128, Float16, Float32, and Float64. There are also UInt options for all the existing Ints listed above. The main difference between the way Julia and Python store numbers is in Julia, an Int corresponds to the machine integer type (Int32 or Int64) whereas a Python Int is an arbitrary-length integer. Therefore, in Julia, the Int type can overflow if the correct type isn’t used, which is where Int128, BigInt, or Float64 are useful. BigInt and BigFloat are used for arbitrary precision integer and floating point numbers [11].

“String” is the concrete type used for Julia’s strings and supports the full range of Unicode characters. They represent a finite sequence of characters and similar to Java, they are immutable. The characters of an AbstractString cannot be changed without converting, for example, it to an array of chars [11]. Strings are concatenated within the *, unlike Python’s use of + [3]. Lastly, the transcode function can be used to convert to other Unicode encodings [11].

The type “Symbol" behaves similarly to that of a String. The main difference is that it is not created with quotation marks, but rather semi-colons. A simple benefit of this over Strings is that one less character is required on creation [3]. Below is an example of this.

```
julia> newSymbol = :text
:text

julia> print(newSymbol)
text
```

Char values represent a single character as a 32-bit primitive type. It is very similar to other high-level languages but one useful functionality is the ability to quickly convert from char to the equivalent integer value, very easily [11]. Shown below is an example of this.

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> print(Int(c))
120
```

Bool is also supported but they do not differ much from other high-level languages so it will not be explained in much depth here. To find a variable's data type, typeOf( ) can be used [11].

```
julia> typeof(5)
Int64

julia> typeof(5.0)
Float64
```

Abstract types in Julia cannot be instantiated and only serve as nodes describing sets of related concrete types. They are the backbone of the types system and build Julia's hierarchy of data types. To declare a new abstract type, the format is

```
julia> abstract type «name» <: «supertype» end
```

and the supertype portion is optional if the newly created type is a subtype of an existing type. When it is omitted, the type "Any" is given. Below are some abstract types that makeup Julia's hierarchy:

```
abstract type Number end
abstract type Real          <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer       <: Real end
abstract type Signed        <: Integer end
abstract type Unsigned      <: Integer end
```

Julia allows the user to declare custom primitive types in addition to the standard primitive types that are already defined in the language itself. The format to declare a primitive type is listed below, again, with the supertype section as optional.

The number of bits indicates how much storage the specific types requires to store the value [11].

```
julia> primitive type «name» <: «supertype» «bits» end
```

Composite types are the more commonly used user-defined types in Julia. In other object-oriented languages, values are objects whether or not they are composite types or some values are not objects while user-defined composite types are true objects. In Julia however, all values are objects but functions are not combined with the object they operate on. This is helpful because Julia uses multiple-dispatch to choose which method of function to use [11].

Composite types are created using the "struct" keyword which is then followed by field names and their types with a "::" [11]. Shown below is the composite type created for the infoSorter assignment:

```
    struct Person
        name :: String
        age :: Int
    end
```

Fields with no type with be given the default type Any. New objects of this composite type would be created, for example, with Person("Emma", 10). Two constructors are created by Julia. One accepts any arguments and converts them to the types listed in the struct, and the other accepts arguments matching the exact fields. The name fieldnames can be used to return the field names of the composite, and access to those are accessible through the dot notation [11].

```
[julia> fieldnames(Person)
(:name, :age)

[julia> person1.name
"Emma"
```

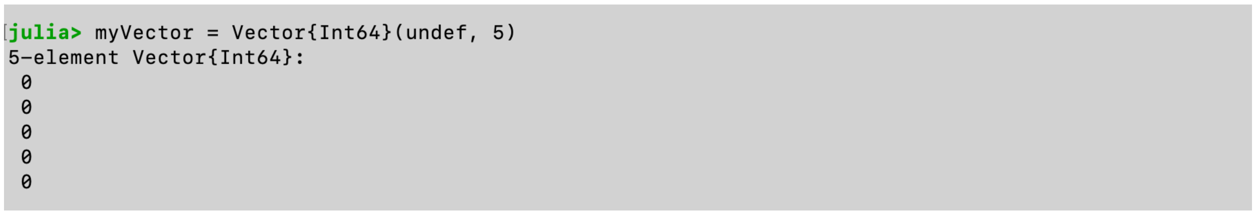
Data Structures

Julia supports most of the common data structures found in many high-level languages; one of the most used in Julia are tuples. They're especially special in Julia because they are often related to functions [11]. Tuples are created with parentheses as shown below.

```
[julia> newTuple = (1, "element", 6.5)
(1, "element", 6.5)
```

A tuple is an immutable type that holds multiple values of differing types. Named tuples are also supported if the user wishes to name the values in the tuple. Tuples can be used with functions that return multiple values, as described earlier because the values it is returning are structured as a tuple [11]. This type of method is explained above under the Functions section.

An array, much like other languages, is a collection of objects. A differing aspect is that Julia allows objects of different data types to be stored in a single array, although it is suggested to only store objects of the same type for better performance. They are also mutable which is why they have been labeled the bread and butter for data scientists. Arrays can be represented as one-dimensional or two-dimensional arrays. These are both Array types in Julia, but the aliases Vector and Matrix can be used for simpler terms. If the contents aren't known when constructing the array, "undef" is used to assign all values Null/zero [3].



```
julia> myVector = Vector{Int64}(undef, 5)
5-element Vector{Int64}:
 0
 0
 0
 0
 0
```

One main difference from other programs is that Julia is not zero-based. Therefore, access by index always begins with 1, while using 0 would result in an out-of-index error [5].

Pairs are another handy data structure that maps two objects to each other. Pairs are used a lot in data manipulation and visualization because both DataFrames.jl and Makie.jl take objects of the type Pair. Most other data structures are supported in Julia and they are similar to Python [3].

Parallel Computing

Julia supports 4 categories of parallel computing: Tasks, Multi-threading, distributed computing, and GPU computing.

Tasks provide asynchronous programming when operations in the program may need to happen in an unpredictable order. Tasks are also known by other names such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations [11]. When a Julia program needs to access the outside world, some tasks may need to wait for completion before they can execute. One example is when downloading a file. After initiating the download operation, Tasks allow the program to perform other operations and resume the code that needs the downloaded file when it is available. When a piece of code is designed as a Task, it is possible to interrupt it by switching to another Task, and the original Task can be resumed and picked up where it last left off. While this seems familiar to a function call, there are two key differences: switching tasks does not use any space and switching can occur in any order, as opposed to a function that must complete before the control flow returns to the calling function [11].

Julia supports two different types of multithreaded programming: loop parallelism and task parallelism [8]. Multithreaded provides the ability to schedule tasks simultaneously on multiple threads or CPU core shared memory. Julia's multithreading is composable - parallel tasks can be started that call library functions that themselves start parallel tasks. When this happens, Julia schedules the threads globally without oversubscribing [11].

The distributed library supports the remote execution of a Julia function. With this, other packages can be used such as MPI.js and Elemental.js which provide access to existing MPI libraries.

The GPU compiler offers the ability to run code natively on GPUs. Many of Julia's packages target GPUs [11]. There are several platforms with differences between all to best fit the simulation. The best supported platform is NVIDIA CUDA; it supports all versions of Julia and this functionality is used for many applications. More supported platforms are Intel GPUs with oneAPI, AMD GPUs, and Apple GPUs [6].

AI and Machine Learning

Because of Julia's ability for scientific computing support, it is a good candidate for AI simulations and machine learning models [11]. It is designed for parallelism and is very fast, which gives advantages over other popular machine learning languages. It is also easy to switch from a background in Python or R, since the syntax is similar and quick to pick up. One downside about using Julia over other popular languages for AI/ML is how recently the language was released. Though it is relatively new, there are many libraries and packages that are useful for machine learning. One issue programmers run into, is that these resources are distributed across different packages so it takes time to find what is needed for a specific purpose, but there are initiatives to regroup the models in larges libraries. Currently, there are two libraries most suitable for machine learning: MLJ and Scikit Learn [3].

Hands-On Experience

My experience with Julia has been mostly positive. I began the first program without doing any research to see how intuitive it is, and I was pleased with how easy it was to pick up. With background knowledge of Python, the syntax felt very similar and the compiler errors generally give clear detail and explanation of where the errors persist and possible ways to fix them. I began my projects by installing Julia on my system from Homebrew and used Visual Studio Code's official Julia (by julialang) extension. Because the structure looks very similar to Python, the programs were not very difficult to write.

Language Critique

The language has generally good readability and writability in my option. While I score these high from my experience, there are differing thoughts on this subject. As most know, computer scientists have very strong opinions on their preferred language, so it is often difficult to determine the true overall opinion on a language's features. Many data scientists swear by Julia and others don't ever want to touch the syntax.

Some of the pros I see in using the Julia language is that it is fast. It contains the speed of C but users don't have to touch low-level code. The two-language problem can be avoided when adopting this language. With Julia, programmers can write easily-understandable code that is magnitudes faster than in other high-level languages. Julia is the sweet spot for performance and productivity [7].

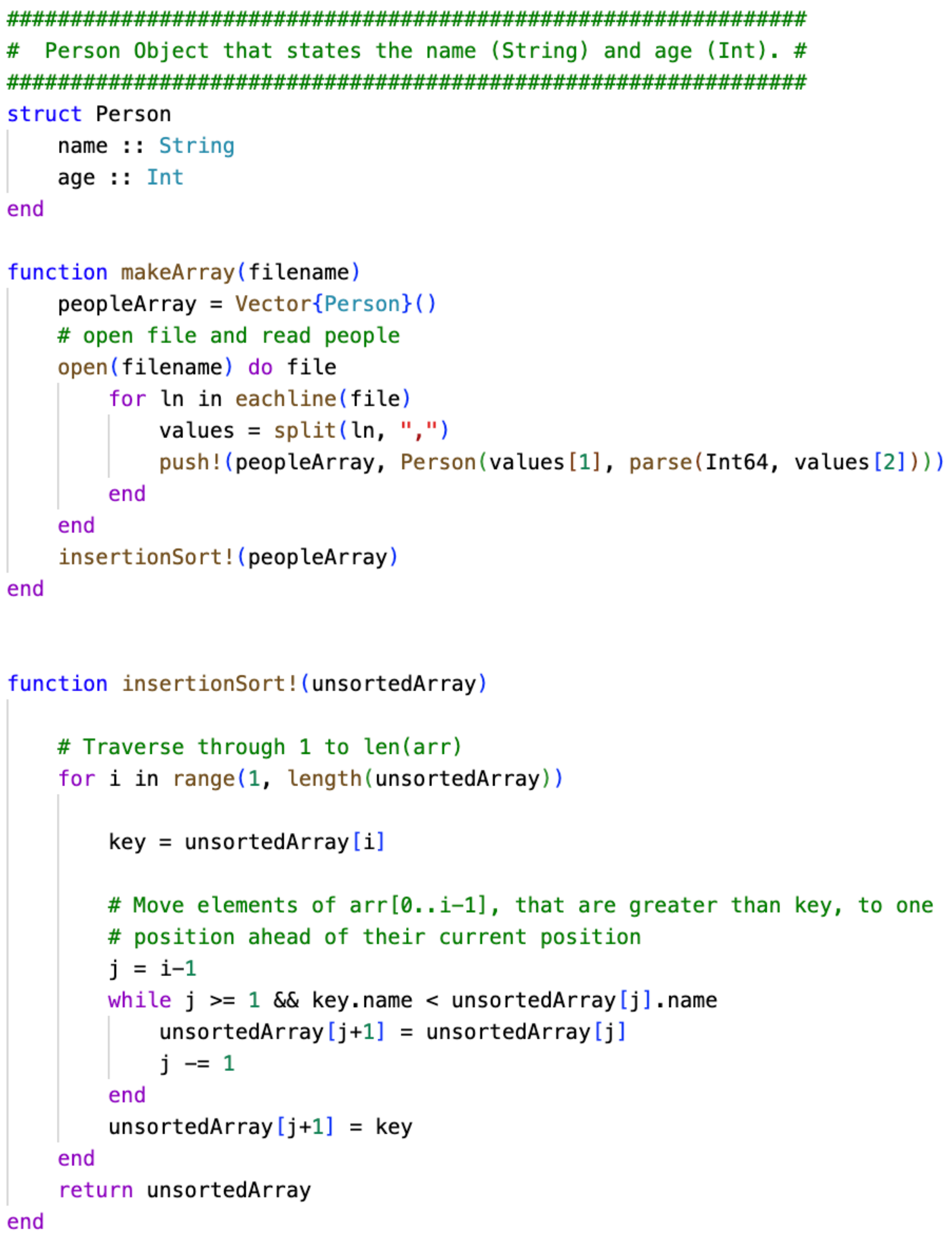
While this is all great, there are also some downsides. The first and probably most obvious con to using Julia is that it is still so new. With the release occurring around a decade ago, there has not been as much improvement or cleanup as say, Python that was released in 199. As stated earlier in the Machine Learning section, many of the resources are distributed across different packages, so it is often difficult to access one package for all related inquiries [9] . The libraries are a bit rough around the edges and haven't had the time needed to improve and create new functionalities [7]. While Julia also promotes itself as easily integrated into other languages, many programmers complain that it is not as simplistic as made out to be [11].

From the research I've conducted and my own coding experience, I don't see many obvious downsides to using this new language. It seems that a common theme is Julia is preferable to other low-level languages but has not yet surpassed its competing high-level languages [7]. Once time has passed and many of the current issues are resolved, I can see Julia becoming very popular. It's fast, straightforward, and easy to learn. I enjoyed working with it and I hope that all reading this paper will be intrigued to give Julia a try.

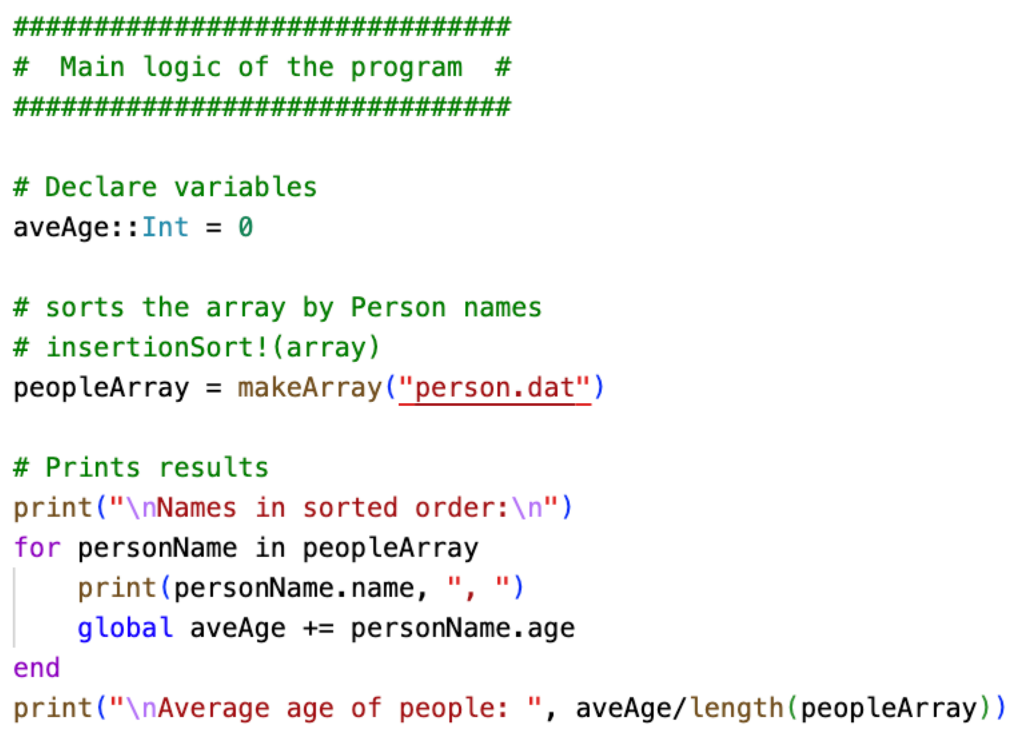
Program I - Factorial.jl

```julia
#######################################################
#   Julia function that computes the factorial of n   #
#######################################################
function computeFactorial!(n::Int64)::Int64
    # base case
    if n == 1
        return n
    else
        return n * computeFactorial!(n-1)
    end
end


#################
#  Main logic  #
#################

# converts the command line argument from string to int
n = parse(Int, ARGS[1])
println(computeFactorial!(n))
```

Program II - InfoSorter.jl



```julia
############################################################
#  Person Object that states the name (String) and age (Int). #
############################################################
struct Person
    name :: String
    age :: Int
end

function makeArray(filename)
    peopleArray = Vector{Person}()
    # open file and read people
    open(filename) do file
        for ln in eachline(file)
            values = split(ln, ",")
            push!(peopleArray, Person(values[1], parse(Int64, values[2])))
        end
    end
    insertionSort!(peopleArray)
end


function insertionSort!(unsortedArray)

    # Traverse through 1 to len(arr)
    for i in range(1, length(unsortedArray))

        key = unsortedArray[i]

        # Move elements of arr[0..i-1], that are greater than key, to one
        # position ahead of their current position
        j = i-1
        while j >= 1 && key.name < unsortedArray[j].name
            unsortedArray[j+1] = unsortedArray[j]
            j -= 1
        end
        unsortedArray[j+1] = key
    end
    return unsortedArray
end
```



```
################################
#  Main logic of the program  #
################################

# Declare variables
aveAge::Int = 0

# sorts the array by Person names
# insertionSort!(array)
peopleArray = makeArray("person.dat")

# Prints results
print("\nNames in sorted order:\n")
for personName in peopleArray
    print(personName.name, ", ")
    global aveAge += personName.age
end
print("\nAverage age of people: ", aveAge/length(peopleArray))
```

Program III - RandomTextWriter.jl

```
using Random

##############################################################
#   This function parses the .g file given in argv[1] and   #
#   returns a dictionary with the organized file contents   #
##############################################################
function parseGrammarFile()
    grammarDict = Dict{String, Vector{String}}()
    currentKey = ""
    currentArray = []
    flag = 0

    # open file
    open(ARGS[1]) do f

        # read till end of file
        while ! eof(f)

           # read a new for every iteration
           line = readline(f)

            # if { is found, then begin a new dict entry
            if (occursin('{', line))
                flag = 1
            # if } is found, finish array and add into the dictionary
            elseif (occursin('}', line))
                # add array to dict values
                grammarDict[currentKey] = currentArray
                currentArray = []
                flag = 0

            # if flag is 2, add all values to data
            elseif (flag == 2)
                append!(currentArray, (strip(line, ['\n', ';'])))

            # if flag is 1, add first value to struct
            elseif (flag == 1)
                currentKey = strip(line, '\n')
                flag = 2
            end
        end
    end
    print(grammarDict)
    grammarDict
end
```

```
#################################################################################
#   Begins with a the start sentence and generates random sentences based off   #
#   of the grammer sentences.                                                   #
#################################################################################
function generateText(grammar, full_story)
    nonterm = findfirst("(<.+?>)", full_story)
    if (nonterm)
        replace(full_story, nonterm.group(1) => rand(grammar[nonterm.group(1)]), 1)
        generateText(grammar, full_story)
    else
        print(full_story)
    end
end

#############
#   Main    #
#############
sortedGrammer = parseGrammarFile()
begin_story = sortedGrammer["<start>"]
generateText(sortedGrammer, begin_story)
```

Bibliography

C. Stokel-Walker, “Julia: The Goldilocks language – increment: Programming languages,” Increment, 26-Apr-2018. [Online]. Available: https://increment.com/programming-languages/goldilocks-language-history-of-julia/. [Accessed: 12-Nov-2022].

Derek Banas, “Julia Tutorial,” YouTube, 28-Oct-2018. [Online]. Available: https://www.youtube.com/watch?v=sE67bP2PnOo. [Accessed: 11-Nov-2022].

J. Storopoli, R. Huijzer, and L. Alonso, “Julia Data Science,” Native data structures - julia data science. [Online]. Available: https://juliadatascience.io/data_structures. [Accessed: 18-Nov-2022].

“Julia (programming language): Home,” Research Guides. [Online]. Available: https://guides.libraries.uc.edu/julia. [Accessed: 11-Nov-2022].

“Julia global keyword: Creating a global variable in julia,” GeeksforGeeks, 26-Mar-2020. [Online]. Available: https://www.geeksforgeeks.org/julia-global-keyword-creating-a-global-variable-in-julia/#:~:text='global'%20keyword%20in%20Julia%20is,global%20variable%20of%20that%20name.&text=With%20the%20use%20of%20global,from%20anywhere%20in%20the%20code. [Accessed: 16-Nov-2022].

“Juliagpu,” RSS. [Online]. Available: https://juliagpu.org/. [Accessed: 18-Nov-2022].

“MatecDev,” Is the julia language worth learning? (pros and cons). [Online]. Available: https://www.matecdev.com/posts/julia-worth-learning.html. [Accessed: 19-Nov-2022].

“Notes on multithreading with julia,” Faculty of Computer Science. [Online]. Available: http://www.cs.unb.ca/~aubanel/JuliaMultithreadingNotes.html. [Accessed: 18-Nov-2022].

“Python® – the language of today and Tomorrow,” About Python. [Online]. Available: https://pythoninstitute.org/about-python#:~:text=Python%20was%20created%20by%20Guido,released%20on%20February%2020%2C%201991. [Accessed: 19-Nov-2022].

R. W. Sebesta, Concepts of programming languages, 12th ed. Upper Saddle River: Pearson, 2022.

S. K. Jeff Bezanson, “The julia programming language,” The Julia Programming Language. [Online]. Available: https://julialang.org/. [Accessed: 11-Nov-2022].

Subin, “Know the history of Julia Programming language,” CloudQ, 21-Jul-2022. [Online]. Available: https://cloudq.net/history-of-julia-programming-language/#:~:text=Julia%20was%20introduced%20to%20the,of%20people%20around%20the%20world. [Accessed: 11-Nov-2022].