CMPT383 Assignment 2

Christopher Kassner, ckassner@sfu.ca, 301351308

November 6, 2020

1 Introduction

For this report, I chose to use Julia as my new programming language to explore, since it is still relatively new and it seemed interesting to learn, as well as being potentially useful in the future. The first version of Julia was released in 2012, and the language has steadily grown in popularity over the years, with over 20,000,000 downloads as of September 2020. Relative to it's age, Julia is currently ranked quite highly - at an impressive 30th place, according to the TIOBE index, which ranks the most popular programming languages in the world. For context, Julia is only one position behind Scala, and is many positions ahead of Lisp, Kotlin, Lua, Haskell, and TypeScript, among others. The current version is Julia 1.5.2, released less than two months ago.

Julia is generally considered as a 'high-level', dynamic programming language, and is multi-paradigm - supporting multiple dispatch as it's dominant programming paradigm, while also offering support for procedural, functional, meta, and multi-staged programming. In addition, Julia is also useful for 'low-level' systems programming, web programming, and as a specification language. Given it's wide range of potential uses, Julia could best be classified as a general-purpose language, albeit being originally designed for numerical and technical computations.

Julia supports concurrent, parallel and distributed computing, as well as direct calling of libraries featured in other programming languages such as C or Python. It uses a JIT(just-in-time) compiler, and is garbage collected. Furthermore, it uses eager evaluation, and is dynamically typed.

The official distribution of the language includes an interactive command-line REPL(read-eval-print loop), allowing for quick and simple testing of code. In addition, popular IDE(integrated development environments)'s such as Visual Studio Code feature extensions allowing for the support of Julia.

2 Common Uses

2.1 Projects

Albeit its classification as a general-purpose language, Julia is particularly well-suited, as well as being a popular choice, for data science and numerical analysis, as it features libraries for floating-point calculations, linear algebra, and regex matching, among others. Perhaps most notably, Julia is currently being used by NASA and INPE - the National Institute for Space Research in Brazil, for planning space missions and satellite simulations, attesting to it's merit as a programming language in spite of it's youth. Furthermore, Julia is also being used by several companies including BlackRock, for time-series analytics, and Avila, for risk calculations, as well as being used by The Federal Reserve Bank of New York to create models of the U.S. economy. It is also noteworthy that as of 2017 Julia has been used by the Celeste project - which is attempting to learn an astronomical catalog of the visible universe, for purposes of obtaining optimal performance of 1.54 petaFLOPS(peta floating point operations per second) using 1.3 million threads on 9300 KNL(Knights Landing) nodes of the Cori II(Cray XC40) supercomputer. Given that petaFLOPS computations have only been achieved by a few languages such as C, C++ and Fortran, this is a highly remarkable accomplishment on the part of Julia.

Additional projects include a collaboration between Julia Computing and NVIDIA, aimed at making Julia available as a pre-packaged container on the NGC(NVIDIA GPU Cloud) container registry, as well as it's use by the CMA(Climate Modeling Alliance) for a global climate model. As of 2019, Mozilla has been funding the project "Bringing Julia to the Browser", in order to offer support for the language in Firefox.

2.2 Strengths

After spending a few dozen hours getting familiar with Julia, as well as reading various articles about the language, it is obvious that there are many highly impressive aspects and strengths to the language.

Julia's first strength is its speed. Although generally classified as a 'high-level' language along the lines of Python, Julia's JIT(just-in time) compiler and type declarations - specifically, type-stability through specialization via multiple-dispatch, allow for typical run speeds similar to those of C. Ignoring the ambiguity and lack of a well-defined definition for the distinction between 'high' and 'low' languages, as well as the potential meaninglessness of such categorizations, this is clearly an impressive feat and strength. Albeit featuring a dynamic type system, Julia's performance approaches that of statically-typed languages. Clearly, this strength gives Julia an advantage over other languages geared towards general numerical analysis and computational science, or over other dynamically typed languages in general, in instances where speed and ef-

ficiency is crucial. Whereas languages such as Python are clearly not the most efficient when handling large datasets consisting of potentially millions or billions of elements and their respective computations, Julia is extremely fast and efficient. Although Python, for example, can be expressly optimized and made faster through tools such as Cython, or third-party JIT compilers such as PyPy, Julia's aforementioned JIT compiler and type declarations allow for it to eclipse unoptimized Python in terms of speed.

Another strength of Julia is it's syntax. Similarly to Python, Julia is both highly intuitive to code in and highly readable. This not only allows for a more comfortable user experience, but also lends to easier adoption by new users, as it requires far less time or patience to gain familiarity with the language in comparison with other new languages such as Rust. In particular, the syntax for mathematical operations and functions is highly intuitive, making them easy to use.

Directly related to Julia's strengths in terms of speed and efficiency, as well as it's highly intuitive syntax for both writing and reading code, is that Julia seems to have solved the "two-language problem". The "two-language problem" is essentially defined as a trade-off between languages being relatively easy for humans to write(e.g. 'high-level' languages such as Python or R), and languages relatively easy for the computer to run with high performance in mind (e.g. 'low-level' languages such as C). This is not mere coincidence - the developers designed Julia with this problem in mind.

Yet another strength of Julia, is it's support for concurrency, parallelism, multi-threading, and distributed computing - an area in which it outshines other 'high-level' similar usage languages such as Python. Julia allows for the suspension and resumption of computations for I/O, event handling, and other processes. Tasks are able to synchronize through operations such as "wait" and "fetch", as well as communicate through "Channels". Furthermore, multi-threading functionality in Julia builds on tasks by allowing for them to run simultaneously on more than one thread or CPU core, sharing memory. Regarding distributed computing, Julia allows for running multiple processes with separate memory spaces, and potentially on different machines.

Another strength of Julia, is its interoperability with other programming languages - namely, C. Julia is able to call C functions directly, without needing any wrappers or APIs. Also, Julia packages allow for the ability to interface with many other popular and widely-used programming languages such as Python, R, Java, Scala, and MATLAB. This makes it much easier and more convenient to combine multiple programming languages with Julia in comparison with other languages.

Finally, there are a few more strengths of Julia worth briefly mentioning, such as its open source status, unlike languages such as C or MATLAB. It's also

free, has an active and supportive community of developers, and the official documentation is concise and easy to read. Also, its automatic garbage collection - it manages to achieve performance nearing that of C without forcing manual memory management.

2.3 Weaknesses

Despite the many strengths of Julia highlighted in the previous section, there are also a few weaknesses worth mentioning. First, the relative newness of the language. Since Julia first appeared in 2012, the language's package ecosystem is sorely lacking compared to that of other languages such as Python, which features an incredible variety of libraries used for everything from mathematical computations to machine learning and neural networks, such as NumPy, TensorFlow, and Keras. Currently, the Julia ecosystem contains over 4,000 packages registered in 'General' - Julia's default package registry , whereas that of Python consists of over 270,000, according to PyPI (Python Package Index). This weakness is less of a fundamental language problem than a problem of language maturity, as Python was first released in 1991, and I would expect Julia's package ecosystem to continue growing and expanding in the future.

Related to the previous weakness, the Julia REPL needs to be explicitly closed and restarted each time the user modifies or changes the .jl file in any way (e.g. adding a function, changing a variable, etc), in order to see those changes or use them. This definitely constitutes a minor annoyance when spending significant amounts of time coding, as it takes time and is not ideal when writing new functions, adding variables, or trying to debug code. This problem can be circumvented with the Revise package, which allows users to modify code and use changes without restarting Julia. However, this package must be manually searched for and installed by the user, and it also adds a small latency at Julia startup - further evidencing that there are yet many improvements yet to be made to the Julia language.

A final weakness is related to one of Julia's strengths - the JIT compilation. Although Julia is incredibly fast and efficient, packages may be slow to pre-compile - even taking minutes in some cases, and functions using certain packages such as 'Images', 'ImageView', or 'CSV', may be slow to run the first time they're called. Nevertheless, once the packages have been compiled and functions have been called for the first time, this becomes a non-issue.

3 Other Interesting Features

There are several interesting features in Julia, many of which have been elaborated on earlier in the 'Strengths' section, such as interoperability with many other programming languages. However, there are other interesting features worth discussing, such as macros. Although macros exist in many other programming languages, Julia allows for the creation and use of user-defined macros, simply by using the "macro" keyword. These user-defined macros can then be used by adding the @ symbol before the macro's name, as is illustrated in the following example.

```
macro multiply(a, b)
    a * b
end
```

Another interesting syntactic feature would be that similarly to JavaScript, Julia features both a double equality '==' and a triple equality '===='. Unlike the double equality, the triple equality entails a far stricter operation, by measuring egality instead of value equality. Egality is defined as two objects being egal if they cannot be distinguished from each other by the program itself, with an example as follows.

```
>julia 1 === 1
true
```

Whereas,

```
julia> 1 === 1.0
false
```

The difference is that '1' and '1.0' are different types - and therefore can be distinguished by the program itself.

Yet another interesting feature of this language is that Julia allows interpolation of literals or expressions into quoted expressions, by using the prefix $. Since construction of 'Expr' objects can be unnecessarily verbose or tedious, interpolation is used for simplicity and readability, as is displayed in the example below.

```
julia> x = [1,2,3,4,5,6,7]
7-element Array{Int64, 1}:
1
2
3
4
5
```

```
6
7

julia> "x: \$x"
"x: [1,2,3,4,5,6,7]"
```

A final interesting feature of this language is related to its dominant programming paradigm - multiple dispatch. Simply put, multiple dispatch provides the ability to define function behaviour across many combinations of argument types, by using the equal operator '='. An example is as follows.

```
function _int(n)
    println!("Int!")
end
function _float(n)
    println!("Float!")
end
example(n::Int64) = _int(n)
example(n::Float64) = _float(n)
```

If we then enter our example into console,

```
example(.1)
example(1)
```

the output is as follows:

```
Float.
Int.
```

4 Comparison with Other Languages

This section will discuss the implementations of various functions in Julia, as well as comparisons to previously implemented Haskell functions, starting with the following 'divisors' function.

```
#Haskell implementation of divisors
divisors :: Int -> [Int]
divisors n = [i | i <- [2.. (n `div` 2)], n `mod` i == 0]

#Julia implementation of divisors
function divisors(n)
        divisors = Int64[]
        for i in 1:n
                if n % i == 0
                        push!(divisors, i)
                end
        end
        return(divisors)
end
```

The Haskell implementation of 'divisors' only involves 2 lines of code - the type declaration, and a list comprehension returning a list of divisors. In comparison, 'divisors' in Julia uses a for-loop, using the same logic of checking if 'n % i == 0', and then pushing 'i' to the list of divisors if it does, and returning the list. push!() is similar to the append() method in Python.

```
#Haskell implementation of primes
primes :: Int -> [Int]
primes n = [i | i <- [2..n], divisors i == []]

#Julia implementation of primes
function isPrime(n)
        q = floor(sqrt(n))
        for i in 2:n
                if n % i == 0
                        return false
                end
        return true
        end
end

function primes(n)
        primeList = Int64[]
        push!(primeList, 2)
        for i in 2:n
                if isPrime(i)
```

```
                        push!(primeList, i)
                end
        end
        return(primeList)
end
```

Similarly to the previous example, the Haskell implementation of 'primes' is significantly simpler, using a list comprehension to return a list of primes. The Julia implementation uses a helper function to check if a number is prime or not, and returning either 'true' or 'false'. Then, a for-loop is used to push 'i' into the list of primes(primeList).

```
#Haskell implementation of pythagorean
pythagorean :: Int -> [(Int, Int, Int)]
pythagorean n = [ (a,b,c) | c <- [1..n], a <- [1..c], b <- [1..a],(a^2) + (b^2) == (c^2)
```

```
#Julia implementation of pythagorean
function pythagorean(n)
        for c in 1:n
                for a in 1:c
                        for b in 1:a
                                if a^2 + b^2 == c^2
                                        list = (b,a,c)
                                        print(list)
                                end
                        end
                end
        end
end
```

Again, the Haskell implementation of 'pythagorean' is significantly simpler, using a list comprehension. The Julia implementation follows the same logic but uses a double-nested for-loop.

```
#Haskell implementation of join
join :: String -> [String] -> String
join _ [] = ""
join j xs = foldr1 (\a b->a ++ j ++ b) xs
```

```
#Julia implementation of join
function joiner(j, k)
        k = join(k, j)
        print(k)
end
```

The Julia implementation of 'joiner' uses the built-in string function 'join' to concatenate the string and list of strings, whereas the recursive Haskell implementation uses 'foldr1' and a lambda expression.

```
#Julia implementation of mergeSort
function mergeSort(myArray)
        if length(myArray) > 1
        #trunc() returns nearest integral value of same type
                midPoint = trunc(Int, length(myArray)/2)
                leftHalf = myArray[1:midPoint]
                rightHalf = myArray[midPoint+1:end]
                mergeSort(leftHalf)
                mergeSort(rightHalf)
                i = 1
                j = 1
                k = 1
                while i <= length(leftHalf) && j <= length(rightHalf)
                        if leftHalf[i] < rightHalf[j]
                                myArray[k] = leftHalf[i]
                                i += 1
                        else
                                myArray[k] = rightHalf[j]
                                j += 1
                        end
                        k += 1
                end
                while i <= length(leftHalf)
                        myArray[k] = leftHalf[i]
                        i += 1
                        k += 1
                end
                while j <= length(rightHalf)
                        myArray[k] = rightHalf[j]
                        j += 1
                        k+= 1
                end
                return myArray
        end
end
```

Julia features an extremely standard implementation of mergesort, similar to that in Python. The only notable difference is the necessity of the 'trunc(n)' function, which returns the nearest integral value of the same type as 'n', whose absolute value is $\leq$ to 'n'. Without the 'trunc()' function, e.g., if we were to try merge([5,4,3,2,1]), with midPoint = (Int, length(myArray)/2), the output would be 2.5.

```
#Haskell implementation of isPrimeDay
isPrimeDay :: Day -> Bool
isPrimeDay d
```

```
 | divisors (getDay (toGregorian d)) == [] = True
 | != [] = False

getDay(y, m, d) = d
```

```
#Julia implementation of isPrimeDay
function isPrimeDay()
        dateToday = today()
        dayOfMonth = day(dateToday)
        if isPrime(dayOfMonth)
                return true
                end
        return false
end
```

Using the 'Dates' module, which is a part of the standard library in Julia, isPrimeDay was incredibly easy to implement. The 'today()' function gets the current date, and then the 'day()' function gets the day of the month. Finally, we simply use the previously defined 'isPrime()' helper function to check if dayOfMonth is prime or not, returning either true or false.

```
#Haskell implementation of isFriday
--isFriday
-isFriday :: Day -> Bool
isFriday f
        | snd (mondayStartWeek f) == 5 = True
        | != 5 = False
```

```
#Julia implementation of isFriday
function isFriday()
        currentDay = dayname(now())
        if currentDay == "Friday"
                return true
                end
        return false
end
```

Similarly to 'isPrimeDay', 'isFriday' was trivial to implement in Julia. This function simply uses 'dayname(now())' to get the name of the current day as a string (e.g. "Thursday"), and then checks if that day is equal to "Friday" or not, returning either true or false. The Haskell implementation uses pattern matching, checking if the second item in the tuple returned by 'mondayStartWeek', which belongs to the 'Data.Time.Calendar.OrdinalDate' module in the 'time' library, is equal to 5(Friday), returning either 'true' or 'false'.

4.1 Sample Programs

```
#Gets an English country name and outputs the German name
function englishToOtherLangs()
```

```
        print("Enter a country name: ")
        name = readline()
        #get German and Russian data from .csv files
        germanData = CSV.File("germandatatable.csv")
        russianData = CSV.File("russiandatatable.csv")
        #check for name in German data
        for row in germanData
                if contains(row.English, name)
                        println(name, " is ", row.German, " in German.")
                        end
                end
        #check for name in Russian data
        for row in russianData
                if contains(row.English, name)
                        println(name, " is ", row.Russian, " in Russian.")
                        end
                end
end
```

One of the sample programs that I chose to create was an English to German/Russian translator. This program uses the 'CSV' package, taking the data from two .csv files, both containing two columns - one for English country names, and the second for German country names in the "germandatatable.csv" file, as well as Russian country names in the "russiandatatable.csv" file. It gets the user's input and checks whether the English row in the .csv file contains that country's name or not using the 'contain()' function. If that country's name exists in the English row, it finds the name in German and prints it on the next line.

The following example compares Julia to Python and illustrates one of the current weaknesses of Julia - the lack of support for various image-related functionalities despite several image-related packages.

```
#Julia implementation of displayImage()
using Images
using ImageView

function displayImage()
        #gets the path to the .jpg file
        imgPath = ".julia/sourpatch.jpg"
        #loads the image
        img = load(imgPath)
        #displays the image in ImageView
        imshow(img)
        H = height(img)
        W = width(img)
        totalPixels = H * W
        println("There are a total of ", totalPixels, " pixels in this image")
        bluePixels = 0
        rgbArr(img) = begin
                #creates an array of 0s, of size H by W
                arr = zeros(H, W)
                for w in 1:W
                        for h in 1:H
                                RGB = img[h,w]
                                arr[h,w,1] = RGB.(r)
                                arr[h,w,2] = RGB.(g)
                                arr[h,w,3] = RGB.(b)
                                if RGB.b > 150 && RGB.b > RGB.r && RGB.b > RGB.g
                                        bluePixels += 1
                                end
                        end
                end
        end
        percentBlue = (100 * bluePixels) / totalPixels
        println("There are ", percentBlue, " blue pixels in this image")
end
```

```
#Python implementation of displayImage()
from PIL import Image

def displayImage():
  file = Image.open("sourpatch.jpg")
  sourpatch_image = file.load()
  width = file.width
  height = file.height
  total_pixels = width * height
  print("There are a total of " + str(total_pixels) + " pixels in this image.\n")
  blue_pixels = 0
  for i in range(width):
          for j in range(height):
                  r = sourpatch_image[i, j][0]
                  g = sourpatch_image[i, j][1]
                  b = sourpatch_image[i, j][2]
                  if b > 150 and b > r and b > g:
                        blue_pixels += 1
  percent_blue = 100 * blue_pixels / (width * height)
  print("There are " + str(blue_pixels) + " blue pixels in this image.\n")
```

Another sample program I chose to create was one that gets a .jpg image and displays it, using the 'Images' and 'Imageview' packages. Also, this function gets the height and width of the image and tells the user the number of total pixels in the image. I had originally planned to analyze the percentage of blue, red, and green coloured pixels in the image, but this proved to be surprisingly difficult in comparison with a Python implementation of displayImage(), due to the wide variety of color types available in Julia, and the difficulty in extracting their RGB values. For instance, 'red()', 'blue()', and 'green()' functions can be used to extract the RGB values of colours belonging to the 'AbstractRGB' type, but not One notable difference is that arrays start at 0 in Python and most other languages, whereas arrays start at 1 in Julia.

4.2 main

Interestingly, and unlike the vast majority of languages I've used in the past(e.g. C, C++, Python, Java, etc) Julia doesn't have a main().

**Talk about speed

5 Conclusion

In conclusion, I was left with an extremely favourable impression of the Julia programming language. It is highly intuitive and easy to pick up, making for a truly enjoyable learning experience. Moreover, the combination of being a 'high-level' language offering 'low-level' efficiency and speed - it's approach to the 'two-language problem' makes it stand out in comparison to other programming languages, older and newer alike. With it's steadily increasing popularity and the loyal dedication of a growing community of users and developers, Julia has a bright future ahead of it. As the package ecosystem expands, I could see Julia eventually becoming one of the ten most popular and widespread programming languages in use, and perhaps even replacing Python at some point in the future.