First year project
Project 99: Who's Julia?

Group: 99a
Simon Lehmann Knudsen, simkn15
Sonni Hedelund Jensen, sonje15
Asbjørn Mansa Jensen, asjen15
DM501

May 26, 2016

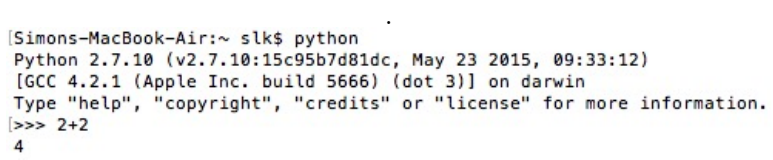
Contents

1 Introduction

Julia is a new open source object orientated programming language which shares many similarities with python. The language is made for high performance and scientific computations while still supporting general purpose programming. Programming in Julia can be done in the terminal like python:

(a) Julia

.



(b) Python

.

Figure 1: Julia in terminal

Making longer programs in the terminal might not be an optimal solution. The text editor Atom supports the Julia language. Atom has a package, **uber-juno**, which sets up Atom to act like an IDE, integrated development environment.

1.1 Syntax

Make examples of syntax between all the languages here.

```
1   Julia:     Python:     Java:           c++:
2   a = 10     a = 10      int a = 10;     int a = 10;
```

Figure 2: Declaring variables

```
for i = 1 : 10
    println(i)
end
```

(a) Julia

```
for i in range(1, 11)
    print i
```

(b) Python

```
for (int i = 1; i < 11; i++){
    System.out.println(i);
}
```

(c) Java

```
for (unsigned int i = 1; i <1 11; i++){
    std::cout << i << "\n";
}
```

(d) C++

Figure 3: For-loops: Incrementing

1.2 Features

Julia supports the following: Multiple dispatch and dynamic typing. The dynamic type system is similar to python. A property of dynamic typing is the type checks which are performed at runtime, mostly - opposed to static typing which makes the type checks at compile time. In Julia it is possible to specify types but for the most part it is not necessary because of the dynamic type system. Julia makes decisions about what size to assign to the variables which does not change at run time and cannot be changed after initialization. The types that are supported in Julia:

· Signed / unsigned integers of; 8, 16, 32, 64 and 128 bits.

· Floating points of 16, 32 and 64 bits.

· Boolean – 8 bits

· Char – 32 bits

Julia will by default assign 32 bits to an integer or floating if no more memory is needed at initialization. Julia has abstract types – an abstract

```
for i = 10 : -1 : 1
    println(i)
end
```

(a) Julia

```
for i in range(10, 0, -1)
    print i
```

(b) Python

```
for (int i = 10; i > 0; i--){
    System.out.println(i);
}
```

(c) Java

```
for (unsigned int i = 10; i > 0; i--){
    std::cout << i << "\n";
}
```

(d) C++

Figure 4: For-loops: Decrementing

type is just some from of generalization of a certain type. For example 16, 32 and 64 bit floats are an AbstractFloat. This is especially useful when the programmer doesn't know how much memory a certain variable may need. Too little memory will result in a bug and too much memory is a waste. Multiple dispatch is used to determine which function to call by the type of one or more of the parsed argument(s). This is useful when, e.g. two functions with the same name are declared:

```
function a(arg1::Int8)
    println("Int")
end

function a(arg1::Float16)
    println("Float")
end
```

If a variable is declared with a type of Int8 and parsed to the function a(...), then it will print "Int", and if the variable is declared as Float16

```julia
function hello(name)
    println("Hello $name")
end
```

(a) Julia

```python
def hello(name):
    print "Hello", name
```

(b) Python

```java
public static void hello(String name){
    System.out.println("Hello " + name);
}
```

(c) Java

```cpp
void hello(std::string name)
{
    std::cout << "Hello " << name << "\n"
        ;
}
```

(d) C++

Figure 5: Declaring functions

it will print "Float". It is also possible to use abstract types here, so arg1::Float16 and arg1::Int8 could be changed to arg1::AbstractFloat and arg1::AbstractInt – this will make the function 'a' accept any kinds of floats and ints. User defined types: User defined types, also known as composite types which gives the option to define new types. A composite type in Julia may look something like the following:

```
type person
    age::Int16
    name
end
```

If a variable in a composite type is not specified with any type, the default is ::Any which accepts any kinds of types. To initialize a composite type you can treat it like an object. Person = person(21, "Carl"), and the values can be changed with "Person.age = 25". The developers of Julia states that user defined types are as fast and compact as built in types.

1.2.1 Garbage Collector

Julia uses a garbage collector to automatically free the memory when needed. There is no guarantee when the garbage collector will run, but is it possible to force garbage collection with a function call: **gc()**. One thing to keep in mind is that once a name is defined in Julia, it will always be present till termination. The garbage collector doesn't free the memory of unreachable object but rather reallocates memory for objects when memory size has changed. So the only way the garbage collector can free memory is when an object has been reduced in size, for example:

```
A = rand(float32, 10000, 10000)
A = 0
Gc()
```

This code will generate a 10000x10000 matrix filled with random 32 bit floating points which consumes a bit of memory. **A** will be set to 0 but will keep the memory size of the 10000x10000 matrix until garbage collection is done either manually as in this example or somewhere in the future when it is done automatically.

1.2.2 Built-in package manager

Julia comes with a built in package manager which keeps track of which packages that needs to be included in the users program to run, so is it not necessary to state which packages that need to be included. For example when the user calls the sort function, **sort()**, no package include statement or **math.sort()** is needed – this is done automatically by the package manager.

1.2.3 Lightweight green threading

A thread is lightweight when it shares address space with other threads opposed to heavyweight threads which has its own address space. If threads share the same address space the communication between them is faster and much simpler. The communication between heavyweight threads have to go through pipes or sockets. Green threads are threads that are scheduled not by the operating system but rather by the runtime library or virtual machine. Green threads does not have to relay on the operating system and can be controlled much better – the threads are in user space and not kernel space. As of Julia 0.4.5 multithreading is not available but in the unstable version 0.5.0 an experimental support of multithreading is available.

1.2.4 Meta-programming and Macros

Meta-programming is a way to write programs in programs and let them use program code as data. In Julia meta-programming can be done by defining macros. For example, the @time macro was used extensively in the project. The @time is in the standard Julia library. The @time takes a function as an argument and sets a timer in the top of the code from the argument and stops the timer in the bottom and prints the time passed and memory allocated. Meta-programming and macros are known from Lisp, a programming language, and have been used in early AI research.

1.2.5 Implementing code from other languages

Since Julia is a newer language there are not many written libraries yet. Julia has an import feature for both Python and C libraries to avoid this disadvantage. Import the library PyCall with the "using" operation to use python and import Python libraries via a macro @pyimport. To use C libraries or coding, simply run the function ccall(). The programming language C is a

well known and used language. Many systems are based on C, which makes C an advantage for hardware programming. Another feature is to execute shell commands. Execute shell commands using run(“). An example would be running run(‘echo Hello World‘) which would return the output "Hello World".

2 Benchmarking

Benchmarking will depend on the running time of each individual problem. When looking at running time the most common terms are wall time (real time) and CPU time. Wall time is the time it took the program to terminate, from start to finish. CPU time is how much time the program used on the CPU. Wall time will always be equal or greater than the CPU time. Difference on time between wall time and CPU time is the fact that the program being benchmarked might get interrupted by the operating system, because other tasks needs to be handled. These tasks could be other programs, or something that the OS needs to get done. This time will be included in the wall time but excluded in the CPU time. In Unix based system, the time command is available from the terminal, used like this:

```
1 time <COMMAND>
```

Where <COMMAND> can be any terminal command. The time command will return three different times.

```
real    0m0.627s
user    0m0.462s
sys     0m0.147s
```

Figure 6: Output of time

.

· Real: This is the wall time mentioned above.

· User: User time is how much time is spent in user mode.

· Sys: System time is how much time is spent in kernel mode.

The difference between user- and kernel mode is the following: In most memory protected system there are some locked operations for security reasons.

E.g. allocation of memory or accessing hardware requires kernel mode and cannot be done in user mode. Processes like malloc and reading / writing to file are done somewhere in the process as a request was sent to the kernel to do certain things and this is counted as system time. This doesn't necessarily mean that, all time is spent in kernel mode when reading a file, since the only time that is spent in kernel mode is accessing the requested data in memory. User time + system time is an estimation of how much time is actually spent by a program on the CPU. To get an even better indication of the CPU time, multiple test are done and the average is calculated. The average run time is considered since there can be fluctuations in run time for every test. Another way to measure time spent on the CPU is to use the built in libraries for example System.currentTimeMillis() in Java. This is especially useful for profiling and figuring out which part of a program is time consuming. Benchmarking will be done by comparing CPU time of the same problem in different languages. The problems will be scaled to have smaller and larger inputs. However, in some situations a large input can cause memory problems, which will be addressed in the specific problem. The memory problems is unfortunately making an upper bound for how large our tests will be on a given problem. The algorithm used for a given problem is the same used in all compared languages. Extended tests will be included for which the algorithm is optimized for a given language.

3 Projecteuler 11

In the 20x20 grid below, four numbers along a diagonal line have been marked in red. The product of these numbers is 26 x 63 x 78 x 14 = 1788696. What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 20x20 grid?

```
4851 3238 1738 4414 2453 3781
1058 2209 2485 2302 1601 4652
4791 3553 706 2520 4236 3842
1210 4707 2305 497 4336 3310
2611 3945 2743 1781 4386 3920
2476 1321 1870 471 1313 3326
```

The program runs from command-line/terminal in order to be able to test on different inputs and amount of adjacent numbers multiplied. To run

the Julia version from the terminal: **Julia euler11.jl 100 4**. This would make the program run a matrix of size 100 and calculating product of 4 adjacent numbers. There was made a matrix generator, found in appendix, which makes a file with data to test. The file would be named **mat100.txt** for data to a $100 \cdot 100$ matrix. The numbers in the file ranging from 100-9999, 3-4 digit numbers. The original problem states two digit numbers in the data, but this would make a lot of duplicated numbers when testing with larger inputs, e.g. a $5000 \cdot 5000$ matrix. Some programming languages optimizes code during run time if it can predict what a given result will be. Having many duplicated numbers in the dataset could have an influence of the benchmarking between languages. To avoid this situation, to some extent, the digits have been increased. The algorithm first reads through a file with the input and makes a matrix. To calculate products in all directions needed in the problem, the algorithm goes through the matrix a total of 3 times. A nested for loop is needed to go though a matrix, lines 1-2 at figure 7. The outer loop iterates through the rows, and the inner loop iterates through the columns. Figure 7 calculates the horizontal and vertical directions, firgure 8 diagonally from left to right(downwards) and figure 9 going diagonally from right to left(downwards). The algorithm starts in the most upper left cell, and iterates through all cells in the matrix. Lets have the first cell as (1,1), first number is row(**i**) and second number is column(**j**). Looking at figure 7, **matLength** is the size of the matrix, size = 100 would mean a matrix of size $100 \cdot 100$. **numProd** is the number of adjacent numbers multiplied together. For matrix with size 100, and multiplying four adjacent numbers, the algorithm would do the following in figure 7:

Starting at cell (1,1) to the end which is cell (100, 100 - 4).

Line 5-7: Loops over the adjacent cells and multiplies the numbers.

Line 8-10: Sets current product to max product, if current is larger than previously max product.

```
for i = 1 : matLength
  for j = 1 : matLength - numProd
    #right/left
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[i, j + k]
    end
    if prod > maxProd
      maxProd = prod
    end
    #up/down
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[j + k, i]
    end
    if prod > maxProd
      maxProd = prod
```

Figure 7: Horizontal and vertical

Figure 8 shows the loop that iterates over the matrix and calculating the product of the diagonal going downwards from left to right. Figure 9 shows the diagonal product going upwards from left to right. The loop starts in the first row, and starting column is the last, not the first like the previously loops.

```
#diagonal left->right
for i = 1 : matLength - numProd
  for j = 1 : matLength - numProd
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[i + k, j + k]
    end
    if prod > maxProd
      maxProd = prod
```

Figure 8: Diagonal downwards left to right

```
#diagonal right->left
for i = 1 : matLength - numProd
  for j = matLength : -1 : numProd
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[i + k, j - k]
    end
    if prod > maxProd
      maxProd = prod
```

Figure 9: Diagonal upwards left to right

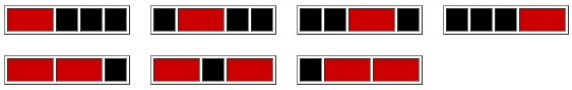
4 Projecteuler 116

Description: A row of five black square tiles is to have a number of its tiles replaced with coloured oblong tiles from red(length two), green(length three), or blue(length four). If red tiles are chosen there are exactly seven ways. If green tiles are chosen there are three ways. And if blue tiles are chosen there are two ways. Figure 10 is a visualization of how the tiles can be lain. Assuming that colours cannot be mixed there are $7 + 3 + 2 = 12$ ways of replacing the black tiles in a row measuring five units in length. How many different ways can the black tiles in a row measuring fifty units in length be replaced if colours cannot be mixed and at least one coloured tile must be used?

5 Quicksort

Quicksort applies the divide-and-conquer paradigm to sort numbers. Divide-and-conquer has three steps. **Divide** the problem into a number of subproblems that are smaller instances of the same problem. **Conquer** the subproblems by solving them recursively. If the subproblems sizes are small enough, however, just solve the subproblems in a straightforward manner. **Combine** the solutions to the subproblems into the solution for the original problem [1, chapter 4, page 65]. Description of quicksort: [1, chapter 7, page 170-171]

> **Divide**: Partition (rearrange) the array A[p..r] into two (possible empty) subarrays A[p..q-1] and A[q+1..r] such that each element of



(a) Red tiles

.

(b) Green tiles

.

(c) Blue tiles

.

Figure 10: Projecteuler: 116

A[p–q1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p..r] is now sorted.

Figure 11 shows a visualization of quicksort.

6 Statistics

7 Learning / Personal experience

Some of the design choices, in Julia, can be really frustrating. The indexes starts at 1 - this could be debatable because of the statement that Julia is used in a lot of scientific computations. As a programmer who is used to indexes starting at 0, this can cause a lot of bugs. The developers of Julia made a weird design choice to the for-loop. A incrementing for loop looks like:

```
1 Julia:
2 for i = 1 : n
3     Some code ...
```

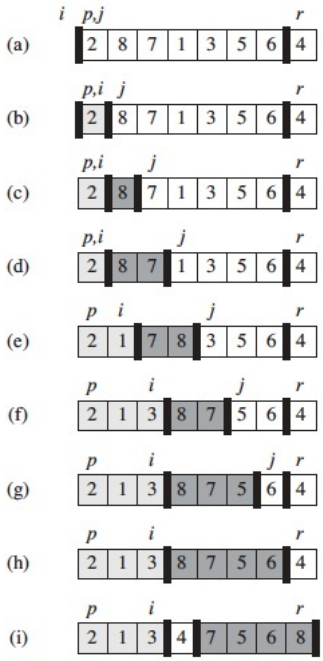


Figure 11: Quicksort

```
4 end
5
6 General languages:
7 for i = 1; i < n; i++
8     Some code ...
```

This is straightforward like many other languages, but take a look at the decrementing for-loop:

```
1 Julia:
2 for i = n : -1 : 0
3     Some code ...
4 end
5
6 General languages:
7 for i = n; i > 0; i--
8     Some code
```

In Julia it is needed to swap the decrement declaration in to the middle of the arguments which is needed both versions, incrementing and decrementing.

The decrementing for-loops both start at n, and stops at 0. This kind of syntax can be the cause of many bugs, and is not that easy to get eyes on. The documentation of Julia is far from being great, which is a huge bump on the road to learning Julia. It feels far from complete. The documentation doesn't explain much but instead gives some examples. Essential information are often lacking, e.g. what types is needed to parse for a given function. It might describe arguments are needed, but lacking to describe the arguments. An example from the documentation is the function **rand()** which is used to generate random numbers:

```
1 rand( [ rng ] [ ,S ] [ , dims... ] )
2
3     Pick a random element or array of random elements from the set
          of values specified by S; S can be:
4     - an indexable collection (for example 1:n or ['x','y','z']), or
5     - a type: the set of values to pick from is then equivalent to
          typemin(S):typemax(S) for integers (this is not applicable to
           BigInt), and to [0,1) for floating point numbers;
6     S defaults to Float64.
```

The argument **S** is explained, but does not mention anything about **rng** or **dims**. Simple things like how to access data within an array at a specific index. This should be one of the first things in the Array section, but is described after about two pages of text. If you have experience in other programming languages, you might know what is needed to type a[index] but this isn't so obvious for new programmers. It is clear that something has to be done to the documentation. A downside to Julia is its consistency. The developers has tried to remove some of the burden from the programmer with dynamic typing, package manager system etc., but with memory allocation and garbage collector it does not make much sense. The garbage collector does not free memory from unreachable object. Sizes of variables cannot be changed after declaration, and is not automatically increased if needed.

8 Conclusion

It can be difficult to make any conclusions by the benchmarking but it may give an idea of how Julia performs, compared to Java, Python and C++. One thing to keep in mind is that as of writing this report, Julia is in version

0.4.5 - it has yet to reach version 1.0.0 while the other languages are much older than Julia. This could mean that Julia might get even faster in the future.

9 Appendix (source code)

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein *Introduction to Algorithms*, Cambridge, Massachusetts, third edition, 2009.