Assignment 3 - Programming Languages

Scala, Java, and Clojure

by Jose Miguel Mendez

Language Discussion

Java

- Using generics, I was able to polymorph the function to accept any object given to it. Java's Type inferencing meant that the compiler figured out the type of object being passed in without having to explicitly defining it. The main problem came when making an array of generic objects. The compile did not like my implementation and told me multiple times that what I was doing was unsafe. After looking on stackoverflow.com I found a way using a code hack to get around the compiler's warnings however it was not optimal.
- The 3 test comparators for String, Float and Int were inherited from Java.util.Comparator. These were turned into concrete objects and passed each time the Sort function was run. I used statics heavily to avoid more boilerplate code.

Scala

- Using generics again, I was able to polymorph the function to accept any object given to it. The main problem again occured when making an array of generics. However in this case, I was pleasently suprised that Scala had a system built in for my use case. Leanring about Manifests made it simple to create an object and even get the names of generic objects using the erasure library. Very cool.
- The 3 test comparators for String, Float and Integer were inherited from Java.util.Comparator. In this case instead of using it as an interface it was used as a trait which was much easier to work with. I found out that in the back end its all interfaces but still nice.

Clojure

- Polymorphism was ridiculously easy in **Clojure**. I didn't even think about it until this question. Dynamic typing is already performed so well.
- The compiler took care of evertyhing and working with multiple classes was so simple it felt like cheating. Most of the

datatypes are already supported with the `(compare x y)` macro. Reading the documentation actually suggested NOT using a comparator. **Clojure** compare example Interestingly **Clojure** has a built-in macro for creating `comparators`. Just supply the predicate in the call `(comparator pred)`.

Metrics

Lines of Code

	Scala	Java	Clojure
Bubblesort	50	48	22
MergeSort	96	79	37
QuickSort	95	86	42
Main	155	146	111
Average	98	89	53

Lines of Code is fairly similar for Scala and Java however, I didn't dig deep into scala to use the functional portions. I stuck to imperative because I knew it well.

The biggest difference is **Clojure** with almost half the lines of code required. There is however a little biase because I did a better job of refactoring after reading **Clean Code** by Robert Cecil Martin.

Completely changed the way I write code.

But still, half is insane; I ❤️ **Clojure**.

The average is close at 98 vs 89 for Scala and java but half for **Clojure** with arguably cleaner and more extendable functionality.

Runtime Errors vs Compile Errors

	Scala	Java	Clojure
Bubble Sort Runtime Errors	6	9	0
Bubble Sort Compile Errors	127	98	92
Merge Sort Runtime Errors	0	26	10
Merge Sort Compile Errors	69	28	6
Quick Sort Runtime Errors	0	1	1
Quick Sort Compile Errors	8	7	8

In general I had ALOT more trouble with scala because I don't know the language well enough.

In each the longest time was taken getting everything up and running which started with Bubblesort in Java.

In general I find mergesort has the most room for errors. Quick sort is much easier to implement.

The area that gave me the most trouble in each language was figuring out generics and the package structure.

Java was failry straight forward and running `javac Main.java` recursively went out and compiled all the packages.

In Scala, you had to run `scalac Main.scala /bubblesort/Sort.scala ...` this was a huge pain in the butt to realize.

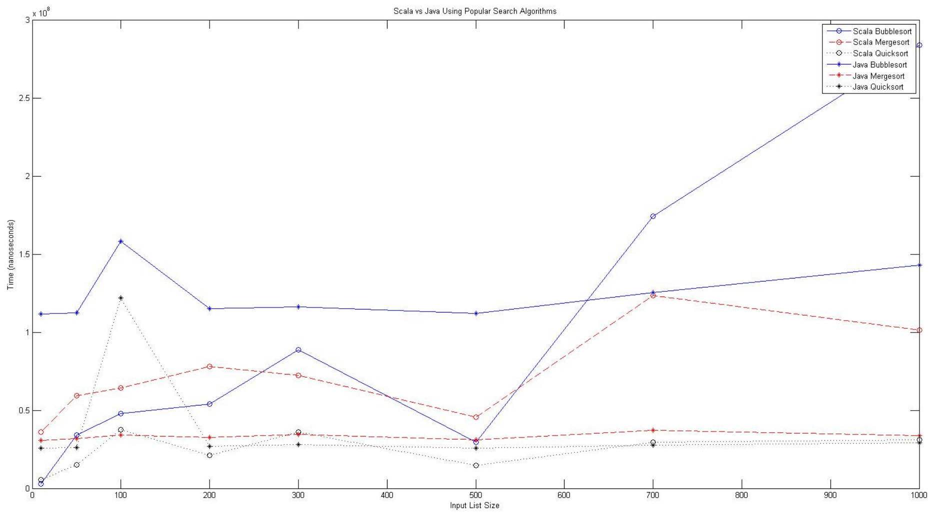
Clojure is undoubtedly the easiest to work with once you get past the rough points. Its tough to say because we've done this now 3 times, so each time you get better. However, **Clojure** was definately easier on paper.

The main difference in using **Clojure** is you spend alot of time starring at the screen and less time writing, which I dont know if is good or bad.

Its tough to show someone what you've done when there isn't alot of code written.

Performance

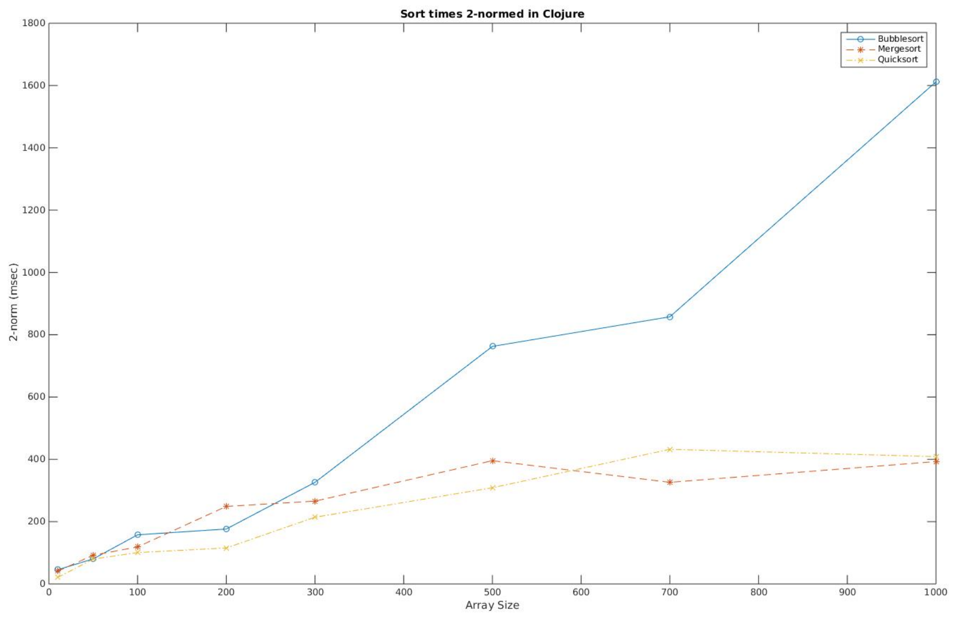
As you can see below, java was actually faster than scala in all the tests I ran.



I believe this has to do with java's maturity and the fact that I used mainly an imperative approach which is not scala's strong suite.

Doing this all in functional vs imperative I expect scala to come out on top.

I did `10, 50, 300, 100, 200, 500, 700, 1000` for the list sizes, running each case 15 times. The 2 norm was taken of each case of String, Integer, and Int, Ascending, and Descending to get a rough average.



Clojure's graph is really interesting. Unfortunately I formatted my computer for Linux, so I lost the original matlab files but comparing the 2 graphs (convert from ms to nanoseconds) we can see an interesting trend:

- Scala is by far the slowest in all sorts
- Java is the absolute fastest
- Clojure is right in the middle

Interestingly however, they all converge to around 400 milliseconds for mergesort and quicksort. The biggest difference is in bubble sort where Java kills it.

However, for developer time, clojure should be the lowest on the graph. Its up to the developer to weight the tradeoffs of developer time vs performance.

Overall Programming Experience

Even though I've been using Java alot, I really enjoyed Scala. In my group of friends, nobody liked Scala, however I may be biased. My favorite parts of Scala are:

- `object` symbol makes a singleton. Very fast and simple.
- for loops with `i <- 0 until array.length` is very welcome and easy to read
- Being able to dip into functional when required.
- Having a mutable vs immutable versions of each collection type is amazing for concurrency

Least Favorite Part is the Compile Time.

Holy smokes, only using 4 files it took about 3-4 seconds to compile. I'm scaled of what it might take when scaling up to a bigger project.

I know sbt has an incremental build daemon but doing it without any tools seems nuts.

2nd least favorite is the crazy symbols. Different objects have different converstions.

Example: `76.toString` vs `SomeClass.toString()`

One is a function the other is a member variable..

Clojure is still fairly new to me but I did use it alot earlier in the year. You really have to work at it to be able to think like a functional programmer. Having to context switch between imperative and functional is really tough.

What I'm suprised is how quickly I did it even still. I had alot of frustration but the previous assignment took me 2-3 days of solid work. **Clojure** took a day even with all the frustration.

If i had to choose a language to rapidly prototype in, **Clojure** would be number 1 ☝.

Favorite parts of **Clojure**:

- Almost no boiler plate, no creating objects everywhere
- Functional programming forces you to think cleanly (almost annoyingly so..)
- Macros are so handy, anything that manipulates lists can be done in 1-2 functions
- Being able to show your friends that their 50 line java program can be done in 3-4 lines in **Clojure**.

Least Favorite parts of **Clojure**:

- Brackets...vim-fireplace helps but very annoying to quickly add a function

Clojure! its the future