## Danny Oxby

## B9017792

Functional Programming Report

## Functional Programming Evaluation

The functional paradigm focuses on being pure, side effectless and stateless with little importance on execution order, while emphasizing functions and recursions. This means functions don’t change external properties, and that data structures are immutable (Clojure, n.d.). Used by Python and Clojure. This differs to the imperative paradigm that focuses on how to complete a process, emphasizing execution order classes and structures, using state modification to achieve this. Used in C++ and Java (Microsoft, 2021).

This difference is shown in Images 1 and 2 using an extendable list.

|  |  |
| --- | --- |
| Functional using Clojure | Imperative using C++ |
| **Example Image 1** | **Example Image 2** |

This shows the mutability difference, both update their ‘intlist’ using length, and loop to add values to the list. The Imperative example is mutable, and the list changes inside ‘createlist’, alternatively the Functional example the ‘intlist’ changes are only internal. This is seen where internally both methods output ‘0 1 2 3 4’, while externally ‘intlist’s values are different.

### Positives and Negatives

To decide the positives and negatives we will look at the paradigm from a programmer’s perspective focusing on ease of use, application for production, and for testing. (Posa, 2022)

#### Immutability

Immutability is a positive when programming applications that use a central data point, such as a local array in Image 1. When a variable’s data cannot change during runtime you guarantee its value, reducing checks, making more robust, consistent code, and making variables simpler for a programmer to use. This makes testing methods easier since external data is guaranteed to always have the expected values. Moreover, immutability supports concurrent and parallel programming, as issues caused by updating variables are removed.

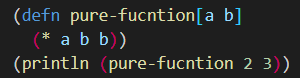
However, this is also a negative. Using the local array example, the contained data may require updates depending on the program’s state, if these changes cannot be reflected the program would need to recreate the array to store these changes. Therefor in production where state matters, immutability is harmful.

#### Recursion and Flow control

The functional paradigm manages flow control though functional calls, ifs and recursion. For programmers’ fewer options make the paradigm simpler to learn, since they don’t have to consider multiple options.

However, because of memory recursion is a negative. Recursive functions require increasing memory on the stack. For production with limited memory e.g., phones, this increases difficulty and potential memory overflow. (WOJCIECHOWSKI, 2019)

#### Modularity and Purity



**Example Image 3**

One advantage is that pure, modular code is easy to understand. Purity makes error finding and testing more direct, and modularity allows functionality to be easily isolated for testing. A pure example is image 3.

However, this purity is a negative when input and output are required. I/O is inherently non-pure as it relies on side-effect to work, if a method needs I/O it must be isolated to maintain purity. (WOJCIECHOWSKI, 2019)

Looking at ease of use, despite making code more testable maintaining purity and having to create multiple objects is difficult, for beginner programmers and large-scale projects this is a negative.

#### Conclusion

From the points above, using my criteria, I conclude that the functional paradigm is more “positive” than “negative”

## Clojure

The Clojure language is a high-level dynamic general purpose functional programming language. It used a dialect of Lisp on the Java platform and access’ Java frameworks using type inference to avoid reflection. It uses immutable, persistent data structures. (Clojure, n.d.)

### Usage example

Clojure’s java interoperation and concurrency make it useful in complex software, such as Atlassian using Clojure as a collaborative editing platform. For example we can link a Clojure backend to an SQL database through the clojure.java.jdbc, [found-here](https://devcenter.heroku.com/articles/clojure-web-application#connecting-to-postgresql-with-clojure-java-jdbc).

### Java integration

Clojure can integrate with Java using multiple ways, some examples are as follows; (Clojure, n.d.)

#### Import

**Example Image 4**



Importing a library makes it available to the code using its unqualified name. Shown in Image 4.

#### Aliases and Primitives

Clojure provides aliases for Java primitive types and arrays, such as int, float and long. Shown in image 5. They are represented using the Java Field Descriptors.

**Example Image 5**



#### Dot Form and Static Methods



**Example Image 6**

Clojure used a ‘.’ Dot form to access java functions and methods. Image 6 shows Clojure using the java ‘ToUpperCase’ function. Alternatively for java static methods like PI, these methods are accessed using a class-/-method structure, Shown by the math function in Image 6.

### Language Comparison

#### Clojure vs C++ (Imperative comparison)

The first difference is the paradigm, such as mutability and emphasis, this can be seen in Images 1 and 2. Next while both are strongly typed, Clojure is dynamically typed with a Lisp base and uses the JVM while C++ is statically typed with a C base. Finally, semantics, for example in mathematical equations Clojure uses the (comparator number number) structure (Image 3) compared to C++’s (number comparator number).

#### Clojure vs Python (functional comparison)

Python, unlike Clojure, doesn’t completely follow the functional paradigm despite being a functional language. Clojure uses only immutable data structures, in comparison Python uses a mix of mutable and immutable data e.g., its List, are mutable while the datatypes are not. Moreover, Clojure used Lisp base while Python has a C base. (Yordanov, 2019)

# Works Cited

Clojure. (n.d.). *Clojure*. Retrieved from The Clojure Programming Language: https://clojure.org/

Clojure. (n.d.). *Java Interop*. Retrieved from Clojure: https://clojure.org/reference/java\_interop

Microsoft. (2021, 09 15). *Functional programming vs. imperative programming.* Retrieved from Microsoft : https://learn.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming

Posa, R. (2022, 08 03). *Compare Functional Programming, Imperative Programming and Object Oriented Programming*. Retrieved from DigitalOcean.: https://www.digitalocean.com/community/tutorials/functional-imperative-object-oriented-programming-comparison

WOJCIECHOWSKI, J. (2019, 08 29). *Functional Primer, Part 3 – When and When Not to Use It*. Retrieved from Atomic Object: https://spin.atomicobject.com/2019/08/29/functional-prog-pros-cons/

Yordanov, V. (2019, 02 03). *Python Basics: Mutable vs Immutable Objects*. Retrieved from Towards Data Science : https://towardsdatascience.com/https-towardsdatascience-com-python-basics-mutable-vs-immutable-objects-829a0cb1530a