2025

CS 4402

Comparative Programming Languages

LEARNING JOURNAL 7

SANA UR REHMAN

INSTRUCTOR: JIM CASALE

WEEK 7 LEARNING JOURNAL: FUNCTIONAL PROGRAMMING CONCEPTS

This week in Comparative Programming Languages, my focus was on **Functional Programming (FP)**, including the **Lambda Calculus**, the core differences between functional and imperative styles, and specific FP techniques like **Higher-Order Functions**, **Currying**, and **Lazy Evaluation**. I engaged with the provided learning materials, took the self-quizzes, and submitted a detailed discussion post on lazy evaluation and currying in Haskell.

WHAT I DID AND MY REACTIONS

I began by reviewing the fundamental concepts of FP, starting with the **Lambda Calculus**, which I found to be a beautiful, minimalist foundation for computation. This led to understanding how FP, exemplified by languages like **Haskell**, differs from the imperative programming I'm more familiar with. The discussion post was particularly helpful for consolidating my learning on two powerful concepts: **lazy evaluation** and the combination of higher-order functions and currying.

To write the post, I first had to precisely define **lazy evaluation** as the strategy of delaying an expression's computation until its value is truly needed. I then analyzed its benefits, such as **modularity** through the separation of concern—allowing for the use of potentially **infinite structures** like streams—and the potential for **performance improvement** by avoiding unnecessary work. Similarly, I defined **higher-order functions** as functions that take or return other functions, and **currying** as the transformation of a multi-argument function into a sequence of single-argument functions (Bird & Wadler, 1988).

My initial reaction was a mix of fascination and caution. The elegance of lazy evaluation, which enables producers and consumers to be developed independently, was captivating. However, I was immediately struck by the accompanying trade-offs, particularly the complexity of reasoning about **space usage** and the potential for **space leaks** due to accumulating postponed computations, or "thunks". This highlighted that elegance in one dimension (concise code) can lead to complexity in another (performance/memory management) (Hutton, 2016).

LEARNING AND REALIZATIONS

The most significant thing I learned this week was the deep, complementary relationship between lazy evaluation, higher-order functions, and currying. The power of **partial application** resulting from currying—for example, easily creating a specialized function `add 2` from a general `add` function—demonstrates how these concepts raise the level of abstraction in a program. I realized that these techniques make it easy to create small, composable building blocks, allowing for the expression of control patterns as data (Bird & Wadler, 1988).

What particularly caused me to wonder was the trade-off of **readability** versus **conciseness**, which was also the subject of the peer question I included: "How do you balance the readability cost of heavy **point-free style** and deep currying against the benefits of concise abstraction...?". While functional composition is concise, I can see how its overuse could reduce readability for a development team. This felt challenging because it is a practical, not purely theoretical, hurdle, forcing a balance between the purity of the FP style and the need for maintainable, collaborative code (Hutton, 2016).

I recognize that I am gaining a solid conceptual understanding of the **Haskell** paradigm. I am realizing that I, as a learner, tend to focus heavily on the trade-offs and practical challenges of

new concepts. I find myself immediately considering the debugging, performance, and team collaboration implications, not just the abstract benefits. This week's topics are directly applicable to my experience by providing new, declarative ways to approach problem-solving, contrasting sharply with the imperative mindset I usually employ.

An important thing I am thinking about is how to effectively document and structure a codebase that leverages both **laziness** and **currying** to mitigate the potential **maintainability trade-offs**. The power of FP is clear, but its implementation in a team setting requires discipline and a commitment to shared understanding of its nuances.

References

Bird, R., & Wadler, P. (1988). *Introduction to functional programming*. Prentice Hall. https://usi-pl.github.io/doc/Bird_Wadler.%20Introduction%20to%20Functional%20Programming.1ed.pdf

Hutton, G. (2016). *Programming in Haskell* (2nd ed.). Cambridge University Press. https://assets.cambridge.org/97813166/26221/frontmatter/9781316626221_frontmatter.pdf

Wordcount: 614