CS 4402-01 Comparative Programming Languages

Instructor: Professor Rajesh Soni

Name: Ryohei Hayashi

Learning Journal 1

**Overview of the Week**

This week, I studied the history and evolution of programming languages and basic computer architectures. Specifically, I learned about the definition of programming languages and their paradigms (imperative, data-oriented, object-oriented, and non-imperative). Additionally, I explored the contributions of John von Neumann, Alan Turing, and Alonzo Church to modern computer science and computational theory. I also examined how Turing Machines and Lambda Calculus are related to computational theory and the Church-Turing thesis.

**Personal Reflections**

My prior experiences with programming languages like Python and Java provided a solid foundation for understanding this week’s content. Learning about the differences between imperative and non-imperative languages was particularly fascinating, as the concept of functional programming offered a new perspective. Functional programming languages, focusing on mathematical function evaluation without state changes, presented an alternative to imperative languages, and their abstraction and simplicity were intriguing.

Additionally, revisiting von Neumann architecture and Turing Machine theory deepened my appreciation for the foundational concepts underlying modern computers. The theoretical simplicity of the Turing Machine as a model of computability left a strong impression. On the other hand, I found the mathematical aspects of Lambda Calculus challenging but valuable in understanding the basis of functional programming languages and their abstraction capabilities.

**Topics Studied in Depth**

1. Comparison of Imperative and Non-Imperative Languages

Imperative languages rely on procedural approaches, with detailed control of program flow and state changes. In contrast, non-imperative languages define problem-solving approaches without requiring programmers to control the computational flow. This week’s study helped me understand the advantages and limitations of both paradigms. Imperative languages are efficient but can be harder to read and maintain, while non-imperative languages excel in abstraction and simplicity but face challenges in implementation efficiency. Understanding these trade-offs provided valuable insights into language selection in practical scenarios.

2. The Role of Turing Machines and Lambda Calculus

Turing Machines are a formal method for defining computability, emphasizing the manipulation of states and data. Their theoretical simplicity serves as a cornerstone of computational theory. Lambda Calculus, on the other hand, uses mathematical functions to model computation and forms the foundation of functional programming languages. These two distinct approaches converge in the Church-Turing thesis, which posits that all computable functions can be expressed using either Turing Machines or Lambda Calculus. This realization was particularly thought-provoking and highlighted the theoretical unity in computation.

**Future Challenges and Goals for Next Week**

Next week, I aim to tackle the following challenges:

1. Understanding BNF and EBNF Models

I plan to study these models as the foundation of syntax parsing and gain a deeper understanding of how programming language syntax is formalized.

2. Comprehending the Importance of Type Checking

I want to explore the differences between static and dynamic typing, as well as strong and weak typing, to clarify the role of type systems in programming language design and debugging.

3. Comparing Virtual Machine Models

In addition to Java Virtual Machines, I will examine Unix and IBM virtual machine models to understand their respective impacts on programming. Specifically, I aim to clarify how virtual machines contribute to portability and efficiency in programming.

By addressing these challenges, I hope to strengthen my theoretical foundation in programming languages and improve my practical application skills.

Word Count: 534

References

1. Ben-Ari, M. (2006). *Understanding programming languages.* Weizman Institute of Science.