Christopher Wood
Reflective Essay #1

If the history of technology has shown any pattern, it is that computing systems and application domains are constantly changing. With this change comes new problems that need to be solved; very often they are tackled by software. Unfortunately, the general purpose programming languages that are emphasized in computer science curricula do not address these domain-specific problems such as concurrency and security.

Therefore, I completely agree with the support for programming language courses in computer science curricula made by all authors in the articles "We Need More Than On" [2] and "Programming Languages are Part of Core Computer Science" [1]. Perhaps the most compelling argument made was the need for students to adopt to new languages with minimal effort. This appealed to me on a personal level because I have experienced the benefits of knowing multiple languages while working in industry. One of my tasks as a software engineer at L-3 Communications was to implement power management routines for a radio battery controller implemented on a CPLD. Unfortunately, the existing code base was written entirely in Verilog and I was only familiar with VHDL. However, given that they solve essentially the same problem of modelling hardware using RTL-level constructs, I was easily able to grasp the syntax and semantics of Verilog in order to quickly implement what needed to be done.

In this example, even though the domain did not change, my previous experience with VHDL, which is outside of the scope of languages taught in both the Computer Science and Software Engineering curricula at RIT, enabled me to adapt to a new way of expressing hardware.

I do not believe that such benefits are limited to languages within the same domain. As mentioned in [2], concurrent systems that utilize instruction- and thread-level parallelism supported by processing units with more than one core are becoming ubiquitous. Programmers cannot expect to see Java and C (with POSIX libraries) continue to adapt to such changes. Instead, we must welcoming of new domain-specific languages like Go and Scala that were specifically designed to address the problems introduced by concurrency, among other issues. As this technology continues to grow and evolve we may even see more domain-specific languages come to life, each with a distinct way of expressing concurrent computations. However, armed with the ability to understand the fundamental models of concurrent computing, including shared-memory models, message passing, and software transactional memory, adopting newer languages will become nothing more than a syntactic hurdle. By exposing students to such issues in a programming languages course, and by forcing them to practice different ways of expressing computation, we can lower the learning curve they will encounter with new languages in the future.

Another surprising argument that I had not fully considered myself is that the design and implementation of programming languages bridge the gap between core computer science issues and software engineering practices. Often times there is a shortage of such courses, with theoretical topics such as data structures and algorithms focusing too much on the mathematical complexity of software development, and standard software engineering lessons focusing too much on the appropriate design and architecture for large-scale systems. What many students might not realize is that programming languages, in addition to the tools, libraries, and IDEs that support them, are perfectly legitimate examples of real-world technologies that are created to solve particular problems. In my experience, comparative analyses are very effective ways to reinforce concepts taught in class and increase retention. Therefore, conducting such analyses on various programming languages that target similar problems will not only expose the different implementation and performance aspects of each language, but will also reinforce the fundamental concepts behind programming language design, such as modularity, abstraction, extensibility, and implementation flexibility.

It is easy to see that teaching programming languages increases student learning aptitude and adaptability to new computing environments and application domains. Perhaps this is best expressed with a metaphor. Effective contractors will seldom carry nothing but a swiss army knife to do their job. Instead, the best ones will carry a variety of tools on their belt, each with a specific purpose, and will be able to easily learn a new tool should a new job call for it.

References

[1] Kim Bruce and Stephen N. Freund. Programming languages as part of core computer science. *SIGPLAN Not.*, 43(11):50–54, November 2008.

[2] Kathleen Fisher. We need more than one: why students need a sophisticated understanding of programming languages. *SIGPLAN Not.*, 43(11):62–65, November 2008.