A Series of Toy Programming Languages for Educational Use
(Technical Paper)

How Does Industry Steer Programming-Language Research, and Vice-Versa?
(STS Paper)

A Thesis Prospectus Submitted to the

Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements of the Degree
Bachelor of Science, School of Engineering

James Huang
Spring 2023

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signature _____ Date _____
James Huang

Approved _____ Date _____
Daniel Graham, Department of Computer Science

Approved _____ Date _____
Kent Wayland, Department of Engineering and Society

Introduction

Coders were once those that manually inputted machine codes–numbers that each represent some kind of machine instruction–into neat machines called digital *computers*. Thanks to historical developments in computing, humans (in general) do not have to manually write codes; they can use abstract notations (usually designed for ease of human thought) called *programming languages* to instruct computers as they please (Friedman, 1992). Now to write *programs* run on digital computers, *programmers* can use a wide variety of tools from different rungs of the ladder of abstraction, e.g. coding directly with *machine language*, writing *assembly language* that can be translated into *machine language*, or maybe even writing in a *high-level language* that can be translated into *assembly language* and further into *machine language*.

A Series of Toy Programming Languages for Educational Use

How can a series of "toy" programming languages be used to teach programming? "Toy" programming languages are programming languages designed specifically for educational use. For example, most are abnormally simple so that they are readily digestible by students; features deemed unimportant to the learning process are stripped from toy languages.

Take for example the *toy machine language* designed for use in University of Virginia's Computer Systems and Organization class (Tychonievich, 2022). It consists of only eight or fourteen unique instructions (compare to a modern instruction set like x86-64 which has some 800 or more (Mahoney & McDonald, p. 3)). It also lacks basic functionalities found in other instruction sets like *push*, *pop*, arithmetic flags, subroutine instructions, etc. because it is a toy language: its value is not practical but educational; its simplicity allows students to grasp the

language in its entirety, and by implementing it themselves, they learn something about how all (non-toy) machine languages work.

However, most such toy languages exist in isolation; I have not yet seen a *toy hardware description language* used to implement a *toy machine language* used to write a *toy assembly language* used to write a *toy higher-level language*. I argue that such a *series* of programming languages could be a highly effective teaching tool and thus wish to show it could be developed. The development of such a suite would involve

1. A review of other toy languages
2. A specification of the toy language suite
3. An example implementation of the suite
4. A revised specification
5. Conclusions

What would ideally come out of all of this is a specification of some handful of (toy) programming languages that all build upon each other, designed so that a student can implement all and consequently gain a firm grasp of some kind of full computing 'ladder of abstraction' as it currently stands. This would hopefully provide students a foundation for further, more detailed study of individual rungs of the ladder, and also of new/alternative rungs.

How Does Industry Steer Programming-Language Research, and Vice-Versa?

In 1975, Edsger W. Dijkstra wrote: "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence."

Dijkstra was a mathematician and theoretical physicist who stumbled into early, pioneering computing work (Apt, 2002). As one of the founding figures of computer science, their name lives on with contributions such as the now plainly named *Dijkstra's algorithm*.

COBOL was a programming language designed with industry/business in mind (its name stands for *Common Business Oriented Language*). As exemplified by Dijkstra, COBOL had a fairly negative reputation among computer scientists. As explained by Ben Shneiderman (1985, pp. 350-351), COBOL was not quite a novel academic or scientific endeavor and aimed to solve problems different from the typical computer scientist's, leading many to rebuke or actively ignore it as something familiar yet uniquely unfamiliar.

COBOL is only one example of what seems like a broader theme of computer scientists being disillusioned with industrial applications of their craft. In their 1980 Turing Award lecture, computer scientist C.A.R. Hoare recounts witnessing the development of the PL/1 programming language (also described by Dijkstra as "the fatal disease"):

> At first I hoped that such a technically unsound project would collapse but I soon realized it was doomed to success. Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars.

Yet despite their disillusions, software engineering (Dijkstra, 1988: "the doomed discipline"), has continued to grow to this day (Patel, 2011, p. 28), leading to the question: how have developments in industry affected the academic programming-language research field? How have the computer scientists responded to what industry has used their work for? And

conversely, how has industry developed in the face of programming-language research? How has industry developed on its own, and how has it interacted with the academic community?

"Programming" as an activity has come a long way from its roots as an activity of manually inputting codes into a mainframe computer. Understandably, its long journey to the present must have influenced who, for what reasons, and with what technologies, engages in programming, and in turn what programming represents today. This STS research aims to analyze specifically how *industry* – companies and their employees using programming to solve their needs, and *academia* – researchers largely focused on pushing the boundaries of programming-language knowledge, have interacted to affect the development of programming languages throughout history.

It seems at this point that my most sensible plan of research would be to perform essentially a large literature review. A brief, initial overview revealed interesting bits of history on both sides, like how software engineering developed its own formal methods after struggling to keep up with its own growth (Boehm, 2006, pp. 13-15), or how programming languages were once said to "have been steadily progressing toward their present condition of obesity" (Backus, 1978, p. 614). These bits are interesting but do not offer much to say about how the two fields have interacted. Some other sources, like on education, offer interesting things to think about, like how programming education has changed itself to please industry and potential students (Sajaniemi & Kuittinen, 2008, p. 75). My approach would be to review these kinds of sources – sources written by parties concerning their interests in programming languages (e.g. industry developing its software-development methods, academia deciding what to do next, educators trying to decide what's best for their students) – and try to piece them together as the more 'social' side of the technical history/development of programming languages. The goal would

essentially be to flesh out a more technical, deterministic programming-language history with the harder-to-see social factors to create basically 'a more STS-ey history of programming languages'.

By analyzing sources on both sides of this division–of their histories, their methodologies, their attitudes, etc.–I hope to understand exactly what each side has seen in programming, has wanted out of programming, and has done to affect programming. I hope to tell some kind of history of how these two parties have worked to create programming as it stands today and maybe describe some kind of broader patterns that could be carried to the present. Understanding the forces that drive programming-language research is important to understand how and why programming languages and related computational problems have developed as they have and thus in what ways they have *not* developed–what problems industry or academia have chosen–purposefully or not–to steer away from.

Conclusion

Computers are super mega fast compared to fifty years ago, but human thought does not grow in proportion to Moore's law. With our fancy new computers, we run software written in programming languages fifty years old. How did we get here? What do we not know? Where are we headed? and other questions I wish to answer in my proposed research.

References

8080 Assembly Language Programming Manual. (1975). Intel. https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf

Apt, K. R. (2002). *Edsger Wybe Dijkstra (1930–2002): A portrait of a genius*. Springer.

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, *21*(8), 613–641. https://doi.org/10.1145/359576.359579

Boehm, B. (2006). A view of 20th and 21st century software engineering. *Proceedings of the 28th International Conference on Software Engineering*, 12–29. https://doi.org/10.1145/1134285.1134288

Dijkstra, E. W. (1975). *How do we tell truths that might hurt?* [Manuscript] http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF

Dijkstra, E. W. (1988). *On the cruelty of really teaching computing science*. [Manuscript] http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF

Friedman, L. W. (1992). From Babbage to Babel and beyond: A brief history of programming languages. *Computer Languages*, *17*(1), 1–17. https://doi.org/10.1016/0096-0551(92)90019-J

Mahoney, W., & McDonald, J. T. (n.d.). *Enumerating x86-64 – It's Not as Easy as Counting*. 7.

Patel, P. (2011). Where the jobs are: Software engineering [Careers]. *IEEE Spectrum*, *48*(9), 28–28.

Sajaniemi, J., & Kuittinen, M. (2008). From procedures to objects: A research agenda for the psychology of object-oriented programming education. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*.

Schneiderman, B. (1985). The relationship between COBOL and computer science. *Annals of the History of Computing*, *7*(4), 348–352.

Tychonievich, L. (2022). *CSO1—Simulator* [Course website]. CSO1. http://cs.virginia.edu/tychonievich/CSO1/S2022/lab03-simulator.html