
Preface

A career in computer science is a commitment to a lifetime of learning. You will not be taught every detail you will need in your career while you are a student. The goal of a computer science education is to give you the tools you need so you can teach yourself new languages, frameworks, and architectures as they come along. The creativity encouraged by a lifetime of learning makes computer science one of the most exciting fields today.

There are engineering and theoretical aspects to the field of computer science. Theory often is a part of the development of new programming languages and tools to make programmers more productive. Computer programming is the process of building complex systems with those tools. Computer programmers are program engineers, and this process is sometimes called software engineering. No matter what kind of job you end up doing, understanding the tools of computer science, and specifically the programming languages you use, will help you become a better programmer.

As programmers it is important that we be able to *predict* what our programs will do. Predicting what a program will do is easier if you understand the way the programming language works. Programs execute according to a computational model. A model may be implemented in many different ways depending on the targeted hardware architecture. While there are currently a number of popular hardware architectures, most can be categorized into one of two main areas: register-based central processing units and stack-based virtual machines. While these two types of architectures are different in some ways, they also share a number of characteristics when used as the target for programming languages. This text develops a stack-based virtual machine based on the Python virtual machine called JCoCo.

Computer scientists differentiate programming languages based on three paradigms or ways of thinking about programming: *object-oriented/imperative programming*, *functional programming*, and *logic programming*. This text covers these three paradigms while using each of them in the implementation of a non-trivial programming language.

It is expected that most readers of this text will have had some prior experience with object-oriented languages. JCoCo is implemented in Java (hence the J), providing a chance to learn Java in some detail and see it used in a larger software project like the JCoCo implementation. The text proceeds in a bottom-up fashion by implementing extensions to JCoCo using Java. Then, a full-featured functional language called *Small* is implemented on top of the JCoCo virtual machine. The *Small* language is a subset of Standard ML. Standard ML is first introduced in this text and then used to implement the *Small* subset of the Standard ML language, which really is not that small afterall. Finally, late in the text a type inference system for *Small* is developed and implemented in Prolog. Prolog is an example of a logic programming language.

The text is meant to be used interactively. You should read a section, and as you read it, do the practice exercises. Each of the exercises is meant to give you a goal in reading a section of the text.

The text Web site <http://www.cs.luther.edu/~leekent/PL> includes code and other support files that may be downloaded. These include the JCoCo virtual machine and the MLComp compiler/type inference system.

I hope you enjoy reading the text and working through the exercises and practice problems. Have fun with it and get creative!

Acknowledgements

I have been fortunate to have good teachers throughout high school, college, and graduate school. Ken Slonneger was my advisor in graduate school, and this book came into being because of him. He inspired me to write a text that supports the same teaching style he used in his classroom. I'd also like to thank Eric Manley of Drake University for working with me by trying the projects with his students and for the valuable feedback he provided to me during the development of this text. Thanks, Eric.

I'm also fortunate to have good students working with me. Thanks go to Jonathan Opdahl for his help in building the Java version of CoCo, a virtual machine used throughout this text, and named JCoCo both because it is implemented in Java and because Jonathan helped me build it. Thank you Jonathan for your work on this project. It is greatly appreciated.

For Teachers

This book was written to fulfill two goals. The first is to introduce students to three programming paradigms: object-oriented/imperative, functional, and logic programming. To be ready for the content of this book, students should have some background in an imperative language, probably an object-oriented language such

as Python, Java, or C++. They should have had an introductory course and a course in data structures as a minimum. While the prepared student will have written several programs, some of them fairly complex, most probably still struggle with predicting exactly what their program will do. It is assumed that ideas such as polymorphism, recursion, and logical implication are relatively new to students reading this book. The text assumes that students have little or no experience with the functional and logic programming paradigms.

The object-oriented language presented in this book is Java. C++ has many nuances that are worthy of several chapters in a textbook. The first edition of this text did cover C++ as the object-oriented language, but Java is better suited to the JCoCo virtual machine implementation presented in this text. However, significant topics of C++ are contrasted to Java in this text. Notably, the pass-by-value and pass-by-reference mechanisms in C++ create considerable complexity in the language. In addition, the ability of C++ programs to create objects both on the run-time stack and in the heap is contrasted to Java. Of course the standard object-oriented concepts including polymorphism and inheritance and a comparison of templates from C++ and interfaces from Java are covered in this text.

The text uses Standard ML as the functional language. ML has a polymorphic type inference system to statically type programs of the language. In addition, the type inference system of ML is formally proven sound and complete. This has some implications in writing programs. While ML's cryptic compiler error messages are sometimes hard to understand at first, once a program compiles it will often work correctly the first time. That's an amazing statement to make if your past experience is in a dynamically typed language such as Lisp, Scheme, Ruby, or Python.

The logic language used in this text is Prolog. While Prolog has traditionally been an Artificial Intelligence language, it originated as a metalanguage for expressing other languages. The text concentrates on using Prolog to implement a type inference system. Students learn about logical implication and how a problem they are familiar with can be re-expressed in a logic programming language.

The second goal of the text is to be interactive. This book is intended to be used in and outside of class. It is my experience that we almost all learn more by doing than by seeing. To that end, the text encourages teachers to actively teach. Each chapter follows a pattern of presenting a topic followed by a practice exercise or exercises that encourage students to try what they have just read. These exercises can be used in class to help students check their understanding of a topic. Teachers are encouraged to take the time to present a topic and then allow students time to reflect and practice the concept just presented. In this way, the text becomes a lecture resource. Students get two things out of this. It forces them to be interactively engaged in the lectures, not just passive observers. It also gives them immediate feedback on key concepts to help them determine whether they understand the material or not. This encourages them to ask questions when they have difficulty with an exercise. Tell students to bring the book to class along with a pencil and paper. The practice exercises are easily identified.

This book presents several projects to reinforce topics outside the classroom. Each chapter of the text suggests several non-trivial programming projects that accompany the paradigm being covered to drive home the concepts covered in that chapter. The projects and exercises described in this text have been tested in practice, and documentation and solutions are available upon request.

Decorah, USA

Kent D. Lee

<http://www.springer.com/978-3-319-70789-1>

Foundations of Programming Languages

Lee, K.D.

2017, XIV, 370 p. 189 illus., 39 illus. in color., Softcover

ISBN: 978-3-319-70789-1