

# Preface

---

Computer Science has matured into a very large discipline. You will not be taught everything you will need in your career while you are a student. The goal of a Computer Science education is to prepare you for a life of learning. The creativity encouraged by a lifetime of learning makes Computer Science one of the most exciting fields today according to Money Magazine's Best Jobs in America in 2006.

The words *Computer Science* don't really reflect what most computer programmers do. The field really should be named something like Computer Program Engineering. It is more of an engineering field than a science. That's why programmers are often called Software Engineers. There *is* science in Computer Science generally relating to the gathering of empirical evidence pertaining to performance of algorithms and hardware. There are also theoretical aspects of the discipline, some of which this text will explore. However, the bulk of what computer scientists do is related to the creative process of building programs. As computer scientists we know some things about how to write good programs, but it is still a very creative process.

Given that programming is such a creative process, it is imperative that we be able to *predict* what our programs will do. To predict what a program will do, you must understand how the program works. The programs of a language execute according to a model of computation. A model may be implemented in many different ways depending on the targeted hardware architecture. However, it is not necessary to understand all the different architectures out there to understand the model of computation used by a language.

For several years in the late 1980's and 1990's I worked at IBM on the operating system for the AS/400 (what is now called System i and the i5/OS). My understanding of compilers and language implementation helped me make better decisions about how to write code and several times helped me find problems in code I was testing. An understanding of the models of computation used by the languages I programmed in aided me in predicting what my programs would do. After completing a course in programming languages, you should understand some of the basics of language implementation. This book is not intended to be a complete text on compiler or interpreter implementation, but there are aspects of language implementation that are included in it. We are better users of tools when we understand how the tools we use work.

I hope you enjoy learning from this text and the course you are about to take. The text is meant to be used interactively. You should read a section and as you read it, do the practice exercises listed in the gray boxes. Each of the exercises are meant to give you a goal in reading a section of the text.

The text has a website where code and other support files may be downloaded. See <http://www.cs.luther.edu/~leekent/ProgrammingLanguages> for all support files.

***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 like Python or Java. 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 like polymorphism, recursion, and logical implication are relatively new to the student reading this book. The functional and logic paradigms, while not the mainstream, have their place and have been successfully used in interesting applications.

The Object-Oriented languages presented in this book are C++ and Ruby. Teachers may choose between the chapter on Ruby or the chapter on C++, or may assign both. It might be useful to read both chapters if you wish to compare and contrast a statically typed, compiled language to a dynamically typed, interpreted language. The same project is presented in both chapters with the C++ chapter requiring a little more explanation in terms of the compiler and organization of the code. Either language is interesting to choose from and the chapters do cross-reference each other to compare and contrast the two styles of programming so if you only have time to cover one or the other, that is possible too.

C++ has many nuances that are worthy of several chapters in a programming languages book. Notably the pass by value and pass by reference mechanisms in C++ create considerable complexity in the language. Polymorphism is another interesting aspect of Object-Oriented languages that is studied in this text.

Ruby is relatively new to the programming language arena, but is widely accepted and is a large language when compared to Python and Java. In addition, its object-centered approach is very similar to Smalltalk. Ruby is also interesting due to the recent development of “Ruby on Rails” as a code generation tool.

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 like Lisp, Scheme, Ruby, or Python.

The logic language is Prolog. While Prolog is an Artificial Intelligence language, it originated as a meta-language for expressing other languages. The text concentrates on using Prolog to implement other languages. Students learn about logical implication and how a problem they are familiar with can be re-expressed in a different paradigm.

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 practice with 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 if 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. Look for the light gray practice problem boxes.

The 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 described in this text have been tested in practice and documentation and solutions are available upon request.

Finally, it is expected that while teaching a class using this text, lab time will be liberally sprinkled throughout the course as the instructor sees fit. Reinforcing lectures with experience makes students appreciate the difficulty of learning new paradigms while making them stronger programmers, too.

Supplementary materials including sample lecture notes, lecture slides, answers to exercises, and programming assignment solutions are available to instructors upon request.

## *Acknowledgments*

I have been fortunate to have good teachers throughout high school, college, and graduate school. Good teachers are a valuable commodity and we need more of them. 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 uses in his classroom. Encouraging students to interact during lecture by giving them short problems to solve that reflect the material just covered is a very effective way to teach. It makes the classroom experience active and energizes the students. Ken graciously let me use his lecture notes on Programming Languages when I started this book and some of the examples in this text come from those notes. He also provided me with feedback on this text and I appreciate all that he did. Thank you very much, Ken!

Other great teachers come to mind as well including Dennis Tack who taught me the beauty of a good proof, Karen Nance who taught me to write, Alan Macdonald who introduced me to programming languages as a field of study, Walt Will who taught me how to write my first assembler, and last but not least Steve Hubbard who still inspires me with his ability to teach complex algorithms and advanced data structures to Computer Science majors at Luther College! Thanks to you all.



<http://www.springer.com/978-0-387-79421-1>

Programming Languages  
An Active Learning Approach  
Lee, K.D.  
2008, XIV, 282 p., Hardcover  
ISBN: 978-0-387-79421-1