Exploring Languages with Interpreters
and Functional Programming

Chapter 0

H. Conrad Cunningham

26 July 2018

Contents

0 Preface ................................................. 2
  0.1 Dedication ............................................ 2
  0.2 Course 1 and Course 2 .............................. 2
  0.3 Motivation for “Functional Programming” .......... 3
  0.4 Motivation for “Exploring Languages with Interpreters” .... 4
  0.5 Author’s Perspective .................................. 5
  0.6 Textbook Prerequisites ............................... 6
  0.7 Acknowledgements ..................................... 6
    0.7.1 Notes on Functional Programming with Haskell .... 6
    0.7.2 Programming Languages Organization ............. 8
    0.7.3 Exploring Languages with Interpreters and Functional Programming ................. 9
  0.8 References .............................................. 10
  0.9 Terms and Concepts ................................... 11

Copyright (C) 2016, 2017, 2018, H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of July 2018 is a recent version of Firefox from Mozilla.
0 Preface

0.1 Dedication

I dedicate this book to my mother, Mary Alice Gambill Cunningham, and my father, the late Harold “Sonny” Cunningham. I thank them for their love and encouragement throughout my nearly 64 years on this earth. They taught me the importance of hard work, both physical and intellectual. They taught me the importance of personal integrity and faith in God. I hope I have been a good student.

I write this sitting at a desk in my mother’s home in July 2018. I am reminded of a time more than a half-century ago when I was struggling with an elementary school writing assignment. Mother wrote an “example” page that I remember as amazing. I thank her for that encouragement. I still suffer from a deficit of creativity at times, but I was able to write these approximately 400-page textbook.

I also dedicate this book to my wife of nearly 42 years, Diana Ruth Glass Cunningham. This book is an effort that spans more than a quarter century. She has lived it nearly as much as I have. Many times she has urged me to get some sleep, as she did just now.

0.2 Course 1 and Course 2

As the title suggests, I designed this textbook to be used for at least two different kinds of courses:

1. A course on “functional programming” targeted at advanced undergraduate and beginning graduate students who have previously programmed using imperative languages but who have not used functional or relational languages extensively.

   This functional programming course focuses on parts of Chapter 2 and 91 and Chapters 4-30.

   TODO: Update upper end of chapter references appropriately.

   I have been teaching such an elective course at the University of Mississippi since 1991 (CSci 555, Functional Programming). I have been teaching the Haskell programming language since 1993. Some of the content of this textbook evolved from class notes I originally developed for that course in the 1993-6 period.

   My approach to the course was initially motivated the first edition of the excellent Bird and Wadler textbook [Bird 1988].
2. A course on programming language organization targeted at a similar audience.

There are several approaches to teaching the programming languages course. My approach in this textbook focuses on “exploring languages with interpreters”. It seeks to guide students to learn how programming languages work by developing interpreters for simple languages.

This programming language organization course focuses on Chapters 1-3, Chapters 41-49, and parts of Chapters 4-30 as needed.

TODO: Update chapter references above.

Kamin’s textbook *Programming Languages: An Interpreter-Based Approach* [Kamin 1990] motivated my approach. But, instead of using Pascal or C to develop the interpreters as Kamin’s book did, this textbook primarily uses Haskell. Other influences on my approach are the book by Sestoft [Sestoft 2012] and an early manuscript by Ramsey [Ramsey 2013].

I began experimenting with this approach using the Lua language in my graduate Software Language Engineering (CSci 658) course in Fall 2013. I first taught the interpreter approach (using Lua) in the required undergraduate Organization of Programming Languages (CSci 450) course at the University of Mississippi in 2016. I first used Haskell with the interpreter-based approach in 2017.

Of course, students must become familiar with basic functional programming and Haskell for Course 2 to be possible.

Most real courses will likely be a mix of the two approaches.

### 0.3 Motivation for “Functional Programming”

Course type 1 is a course on functional programming.

As a course on *programming*, it emphasizes the analysis and solution of problems, the development of correct and efficient algorithms and data structures that embody the solutions, and the expression of the algorithms and data structures in a form suitable for processing by a computer. The focus is more on the human thought processes than on the computer execution processes.

As a course on *functional* programming, it approaches programming as the construction of definitions for (mathematical) functions and (immutable) data structures. Functional programs consist of *expressions* that use these definitions. The execution of a functional program entails the evaluation of the expressions making up the program. Thus this course’s focus is on problem solving techniques, algorithms, data structures, and programming notations appropriate for the functional approach.
Course 1 is not a course on functional programming languages. In particular, it does not undertake an in-depth study of the techniques for implementing functional languages on computers. (That is partly covered in Course 2.) The focus is on the concepts for programming, not on the internal details of the technological artifact that executes the programs.

Of course, we want to be able to execute our functional programs on a computer and, moreover, to execute them efficiently. Thus we must become familiar with some concrete programming language and use an implementation of that language to execute our programs. To be able to analyze program efficiency, we must also become familiar with the basic techniques that are used to evaluate expressions.

The academic community has long been interested in functional programming. In recent years, the practitioner community has also become interested in functional programming techniques and languages. There is growing use of languages that are either primarily functional or have significant functional subsets—such as Haskell, OCaml, Scala, Clojure, F#, Erlang, and Elixir. Most mainstream languages have been extended with new functional programming features and libraries—for example, Java, C#, Python, JavaScript, and Swift. Other interesting research languages such as Elm and Idris are also generating considerable interest.

In this textbook, we use the Haskell 2010 language. Haskell is a “lazy” functional language whose development began in the late 1980’s. We also use a set of programming tools based on GHC, the Glasgow Haskell Compiler. GHC is distributed in a “batteries included” bundle called the the Haskell Platform. (That is, it bundles GHC with commonly used libraries and tools.)

Most of the concepts, techniques, and skills learned in this Haskell-based course can be applied in other functional and multiparadigm languages and libraries.

More importantly, any time we learn new approaches to problem solving and programming, we become better programmers in whatever language we are working. A course on functional programming provides a novel, interesting, and, probably at times, frustrating opportunity to learn more about the nature of the programming task.

Enjoy the “functional programming” aspects of the course and textbook!

0.4 Motivation for “Exploring Languages with Interpreters”

TODO
0.5 Author’s Perspective

In the 1974 Turing Award Lecture *Computer Programming as an Art*, Donald Knuth said [Knuth 2009]:

The chief goal of my work as an educator and author is to help people learn to write beautiful programs.

In my writing and my teaching, I hope I can emulate Knuth.

I approach writing this textbook, most of my teaching for that matter, from the following (opinionated) perspectives:

- The essence of computing science is programming. Programming is fun!

- To learn a programming paradigm and language well, we should immerse ourselves in the paradigm and language for a period of time. We need to learn to think in that paradigm and language even if it is quite different from our previous experiences.

- Although we learn to program in a particular language and paradigm, we should seek to compare how the new concepts, features, and patterns of thought apply to other approaches to programming that we have learned in the past or will in the future.

- Our programs should be elegant—both conceptually in terms of their design (architecture, algorithms, data structures, use of appropriate abstraction) and physically in terms of their style (use of language features, layout, use of names, appropriate comments).

- We should rigorously describe what our programs must do. For example, we can define a “contract” that specifies what the client of a program must do and what the program must do in response.

- We should construct larger programs as sets of collaborating modules. The modules should be designed and constructed according to the information-hiding and abstract interface principles.

- We should design our programs to be testable and test them thoroughly.

- We should always reflect upon what we have done. What about our successes and failures can we observe and exploit in the future? Did our specific problem reveal or reinforce a general principle? What can we do better next time?

- Many tasks can be viewed as language design or language processing tasks. Language design and processing is fun!

- As much as feasible, we should make instructional materials accessible and available in multiple formats. Toward that end, I have developed most of my new instructional materials using Pandoc’s dialect of Markdown and tools compatible with Pandoc.
0.6 Textbook Prerequisites

This textbook assumes the reader has basic knowledge and skills in programming, algorithms, and data structures at least at the level of a three-semester introductory computer science sequence. It assumes that the reader has programming experience using a language such as Java, C++, Python, or C#; it does not assume any previous experience in functional programming. (For example, successful completion of at least CSci 211, Computer Science III, at the University of Mississippi should be sufficient.)

This textbook also assumes the reader has basic knowledge and understanding of introductory computer architecture from a programmer’s perspective. (For example, successful completion of at least CSci 223, Computer Organization and Assembly Language, at the University of Mississippi should be sufficient.)

In addition, this course assumes the reader has basic knowledge and skills in mathematics at the level of a college-level course in discrete mathematical structures for computer science students. (For example, successful completion of Math 301, Discrete Mathematics, at the University of Mississippi should suffice.)

The “Review of Relevant Mathematics” chapter (appendix) reviews some of the concepts, terminology, and notation used in this course.

0.7 Acknowledgements

0.7.1 Notes on Functional Programming with Haskell

I created the course CSci 555, Functional Programming, at the University of Mississippi and first taught it during the Spring 1991 semester.

I adopted the first edition of Bird and Wadler [Bird 1988] as the initial textbook for the course. I thank Jeremy Gibbons for suggesting that excellent book in a response to an inquiry I posted to a Usenet news group in Summer 1990.

I also used Wentworth’s RUFL (Rhodes University Functional Language) interpreter and his tutorial [Wentworth 1990] in the course. I thank Peter Wentworth for sending me (unsolicited, in response to my Usenet post) his interpreter and tutorial on a floppy disk through snail mail from the then-sanctioned South Africa.

My approach was also shaped by my research on formal methods and my previous teaching on that topic. I created the course Program Semantics and Derivation (CSci 550) and first taught it in Spring 1990. I followed that with the course Theory of Concurrent Programming (Engr 664), which I first taught in Fall 1990. I thank my dissertation advisor Gruia-Catalin Roman for developing my interests in formal methods, Jan Tijmen Udding for teaching a graduate course on program derivation that piqued my interests, and the researchers and authors
who have developed and popularized the big ideas of that field: Edsger Dijkstra, Tony Hoare, David Gries, Mani Chandy, Jayadev Misra, and many others.

For the third offering of CSci 555 in Fall 1993 I switched the course to use the Gofer interpreter for the Haskell language. I thank Mark Jones for developing that system and making it and its successor HUGS widely available.

Because of the need for a tutorial like Wentworth’s and an unexpected delay in getting copies of the Bird and Wadler textbook from Prentice Hall International that semester, I began writing, on an emergency basis, what evolved into my Notes on Functional Programming with Haskell [Cunningham 2014].

Some parts of the Notes were based on my handwritten class notes from the the 1991 and 1992 offerings of the course. Many pages of the Notes were written “just-in-time” in late-night sessions before I taught them the next day. I thank Prentice Hall (now Pearson) for its delay in shipping books across the “big pond”, my wife Diana Cunningham for tolerating my disruptive schedule, and my Fall 1993 students for not complaining too vigorously about a quite raw set of class notes.

I continued to develop the Notes for the Fall 1994 and Fall 1995 offerings of the course. In early 1996, I created a stable version of the Notes that I continued to use in subsequent offerings of CSci 555. I thank the students in the 1993-1996 period who pointed out typos and suggested improvements.

I formatted the Notes using LaTeX augmented by BibTeX for the bibliography and makeIndex for the index. I thank Donald Knuth, Leslie Lamport, and the many others who have developed and maintained TeX, LaTeX, and the other tools and packages over four decades. They form an excellent system for creating scientific documentation.

I used GNU Emacs for writing and editing the source files for the Notes. I thank Richard Stallman and the many others who developed, maintained, and popularized Emacs over more than four decades.

For the Spring 1997 offering of CSci 555, I started using the new HUGS interpreter and the first edition of Thompson’s textbook [Thompson 2011]. I thank Simon Thompson for writing his excellent, comprehensive introductory textbook on Haskell programming.

Over the next 15 years, I corrected a few errors in the Notes but mostly left the Notes alone. However, I created supplementary notes for CSci 555 and related courses that I formatted with HTML, Microsoft Word and Powerpoint, or plain text.

I decided to use Haskell as one of the languages in the Fall 2014 offering of Organization of Programming Languages (CSci 450). But I needed to change the language usage from the Haskell 98 standard and HUGS to the new Haskell 2010 standard and the Glasgow Haskell Compiler (GHC) and its interactive user
interface GHCi. I edited the Notes through chapter 10 on Problem Solving to reflect the changes in Haskell.

0.7.2 Programming Languages Organization

I first taught the required Organization of Programming Languages (CSci 450) course in Fall 1995. I took over that class for another instructor and used the textbook the Department Chair had already selected. The textbook was the 2nd Edition of Sebesta’s book [Sebesta 1993].

Although the Sebesta book, now in its 11th edition, is probably one of the better and more popular books, I found it difficult for me to use that semester. It and its competitors seem to be large, expensive tomes that try to be all things to all instructors and students. I personally find the kind of survey course these books support as a disjointed hodgepodge. There is much more material than I can cover well in one semester. I abandoned the Sebesta book mid-way through the semester and have never wanted to use it again.

I had a copy of Kamin’s textbook [Kamin 1990] and used two of its interpreters after abandoning Sebesta. It seemed to work better than Sebesta. So I ended the semester with a positive view of the Kamin approach.

I began planning to take start teaching CSci 450 again in Fall 2013, but circumstances delayed that until Fall 2014.

I decided to try an experiment. I planned to use the Kamin approach for part of the course but to redevelop the interpreters in the Lua language. Lua is a minimalist dynamic language that can support multiple paradigm and which few students have experience. (Hence, the students are on a level playing field.) I used the Fall 2013 offering of Software Language Engineering (CSci 450) to explore Lua programming and interpreters.

However, in Summer 2014, I did not believe I was ready to undertake the interpreter-based approach early in the large Fall 2014 class. Instead, I planned to try a multiple paradigm survey. I planned to begin with Haskell static functional programming, then cover Lua dynamic multiparadigm programming, and then use a logic language like Prolog. I had taught Haskell, Lua, and Prolog in 500-level elective courses in the past.

I was comfortable with the Haskell part, but I found a required course a more challenging environment in which to teach Haskell than an elective. Covering Haskell took nearly two-thirds of the semester, leaving Lua in one-third, and squeezing out coverage of the logic language.

I was scheduled to teach CSci 450 again in Fall 2016. For this offering, I decided to (a) begin with Lua and then follow with Haskell (instead of the reverse I used in 2014) and (b) to use the interpreter approach with Lua.
Unfortunately, that offering suffered from immature teaching materials for both Lua and for the interpreter approach. I was unable to invest sufficient time in Summer 2016 to prepare course materials. Students, who mostly had experience with Java, had considerable difficulty modifying and debugging Lua programs with 1000+ lines. (For various reasons, I decided to use the new Elm language instead of Haskell in the last three weeks of the semester.)

I thank the students in the Fall 2014 and 2016 CSci 450 classes for giving me valuable feedback on what works and what does not—perhaps more on the latter than the former. I learned that a required course like CSci 450 needs more mature teaching materials and tools than an elective course does. It should have been obvious.

0.7.3 Exploring Languages with Interpreters and Functional Programming

In Summer 2016, I participated in the eLearning Training Course (eTC) at the University of Mississippi to become eligible to teach online. As a part of that course, I was expected to prepare a syllabus and at least one module for some class. I chose to focus on CSci 555, Functional Programming.

This triggered me to begin reorganizing my previous Notes on Functional Programming with Haskell to be a textbook for an online course on Functional Programming. I thank the eTC instructors for pushing me to begin developing this textbook.

Unfortunately, I devoted too much time to this project and not enough to developing Lua-based materials for CSci 450 in Fall 2016, as I discussed above.

As another result of the eTC, I became sensitized to the need to produce accessible instructional materials (e.g. compatibility with screen readers). I decided to expand my use of Pandoc-flavored Markdown and the Pandoc tools for producing materials in a variety of accessible formats (HTML, LaTeX/PDF).

In Summer 2016, I had materials in a variety of formats. The Notes on Functional Programming with Haskell used LaTeX, BibTeX, and makeIndex. This is a great format for producing printed scientific documents, but not as good for display on the Web. Some materials use HTML, which is great for the Web, but not for printed documents. I also had some material in Microsoft Office formats, Pandoc-flavored Markdown, and plain text (e.g. program comments). So Pandoc-flavored Markdown offered a way to achieve both greater flexibility and greater accessibility. Of course, sometimes I have to compromise on the appearance in some formats.

I thank John MacFarlane and many others who have developed and maintained the excellent Pandoc tool.

In Spring and Summer 2017, I combined the efforts from the previous years
and sought to expand the Haskell-based functional programming course materials to include the interpreter-based approach to the programming languages course. I also included some materials I had developed for graduate courses on object-oriented programming, software architecture, multiparadigm programming, and domain-specific languages.

I redirected the work from developing materials for an online course to developing a textbook for the types of courses I describe in the Introduction above.

I thank my Fall 2017 Teaching Assistant (TA) for CSci 450, Kyle Moore, for his suggestions or corrections in a number of areas. I also thank the Fall 2017 students for their feedback.

In 2018, I began restructuring the 2017 version of the textbook to better meet the needs of the course. I choose the title to *Exploring Languages with Interpreters and Functional Programming*.

I am incorporating more materials from the *Notes on Functional Programming with Haskell* and other materials that had not been previously incorporated. I am also developing new chapters on software testing and the Imperative Language interpreter.

As I have noted above, I maintain this preface as text in Pandoc’s dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

## 0.8 References


[Ramsey 2013]: Normam Ramsey (with Samuel L. Kamin). *Programming Languages: Build, Prove, and Compare*, draft manuscript, 2013. (To be published by Cambridge University Press.)


0.9 Terms and Concepts

TODO