CSci 450: Organization of Programming Languages
Exploring Languages using Interpreters and Functional Programming

H. Conrad Cunningham

20 March 2018

Contents

0 Introduction to Functional Programming **2**
0.1 Motivation 2
0.2 Course prerequisites 3
0.3 Course goals 3
0.4 Desired Student Outcomes 3

Acknowledgements: I adapted and revised this introductory module from chapter 1 of my *Notes on Functional Programming with Haskell* and the revised accreditation and assessment planning document I developed for the course.

In 2017, I continued to develop this module. The student outcomes have **not** been updated to match the changes for either the Fall 2017 Programming Languages course or the Spring 2017 Multiparadigm Programming course.

In Spring 2018, I linked in modified Programming Paradigms and Abstraction documents that will be included in a future draft of this material. I also changed to my new working title for this work.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of March 2018 is a recent version of Firefox from Mozilla.

TODO:

- Update objectives for the Programming Languages course

0 Introduction to Functional Programming

0.1 Motivation

This 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.

This is not a course on functional programming *languages*. In particular, the course does not undertake an in-depth study of the techniques for implementing functional languages on computers. 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 version of this course, 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 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 course!

0.2 Course prerequisites

This course assumes the reader has basic knowledge and skills in programming, algorithms, and data structures at the level of a three-semester introductory computer science sequence or above. 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, completion of CSci 211, Computer Science III, at the University of Mississippi should suffice.)

This course also assumes that 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, completion of Math 301, Discrete Mathematics, at the University of Mississippi should suffice.) The “Review of Relevant Mathematics” section reviews some of the concepts, terminology, and notation used in this course.

0.3 Course goals

The course has the following general goals.

1. This course enables undergraduate seniors and beginning graduate students to develop nontrivial computer programs within the functional programming paradigm using the chosen language. The programs should correctly meet their requirements, be algorithmically efficient, and be developed using appropriate design and programming methods.

2. The students should learn to think and to solve problems using the concepts of functional programming and be able to apply those concepts in languages other than the chosen language for the course.

For this version of the course, we choose to use the Haskell programming language.

0.4 Desired Student Outcomes

Upon successful completion of the course, students will be able to:

1. describe the concepts of the functional programming paradigm

    - Core Concepts – related mathematics (functions, recursion, induction, operations, etc.), programming paradigms (imperative, declarative, functional, logic), explicit versus implicit state, side effect, command versus expression, referential transparency, procedural and data abstraction, functions, recursion, mutable and immutable data structures, first class functions, higher-order functions

    - Advanced Concepts – anonymous functions, algebraic data types, pattern matching, polymorphic typing, modules, abstract data types, information hiding, sequence comprehensions, type inference, type classes, subtyping

2. recognize syntactically and semantically correct functional programs written in the chosen language

    - Core Concepts – syntax and semantics of the chosen language

3. explain the execution (i.e., evaluation) of functional programs using appropriate (substitution) models

    - Core Concepts – substitution model, strict versus nonstrict evaluation, string and graph reduction models, lazy and eager evaluation, time and space complexity, termination, partial versus total functions, preconditions and postconditions

    - Advanced Concepts – infinite data structures, corecursion, productivity

4. develop (i.e., design, implement, execute, and test) syntactically and semantically correct functional programs written in the chosen language

    - Core Concepts – compilation and interpretation, Read-Evaluate-Print Loop (REPL), debugging, testing, use of features of chosen language

5. analyze problems to formulate the requirements for functional programming solutions

6. apply appropriate design and programming techniques and idioms to develop effective functional programs in the chosen language

    - Core Concepts – tail recursion, auxiliary functions, accumulating parameters, modules, information hiding, data abstraction and abstract data types, top-down refinement, separation of data from control, following "types" to implementations, problem-solving strategies

- Advanced Concepts – currying, partial evaluation, composition, combinators, patterns of computation, generic functions

7. evaluate alternative functional programs in the chosen language to determine which are better according to selected criteria
8. relate the concepts and methods for functional programming in the chosen language to their previous programming knowledge and experiences
9. be more confident as computer programmers using a variety of languages and techniques
10. appreciate the ability to exploit the abstraction and mathematical properties of a purely functional programs and languages
11. study and learn a programming paradigm and language with which they previously had little knowledge
12. (optionally) construct reusable libraries by abstracting recurring aspects using advanced features of the chosen language
    - Example Concepts – algebraic design, monads, functors, advanced design of abstract data types and modules
13. (optionally) exploit the mathematical nature of purely functional programs to prove properties, transform programs, and synthesize (i.e., derive) programs from their specifications
    - Example Concepts – algebraic properties, equational reasoning, structural induction, program synthesis
14. (optionally) develop concurrent programs using selected features of the chosen programming language
    - Example Concepts – persistent data structures, parallel map-reduce operations, actors, transactional memory
15. (optionally) generate new or modify existing programs or special purpose languages by using selected programming and metaprogramming features of the chosen programming language
    - Example Concepts – parser generators, parser combinators, quotations, macros, reflection, domain-specific languages

Outcomes 12 through 15 are optional aspects that may vary based on the chosen language and the particular term.

For this version of the course, we choose to use the Haskell programming language. We address the optional component associated with outcome 13.