Preface (to the instructor)

This text's approach

This text provides a distinct way of teaching discrete mathematics. Since discrete mathematics is crucial for rigorous study in computer science, many texts include applications of mathematical topics to computer science or have selected topics of particular interest to computer science. This text fully integrates discrete mathematics with programming and other foundational ideas in computer science.

In fact, this text serves not only the purpose of teaching discrete math. It is also an introduction to programming, although a non-traditional one. *Functional programming* is a paradigm in which the primary language construct is the function—and *function* here is essentially the same as what it is in mathematics. In the functional paradigm we conceive the program as a collection of composed functions, as opposed to a sequence of instructions (in the imperative paradigm) or a set of interacting objects (in the object-oriented paradigm). Dominant computer science curricula emphasize object-oriented and imperative programming, but competence in all paradigms is important for serious programmers—and functional programming in particular may be appropriate for many casual programmers, too. For our purposes, the concepts underlying functional programming are especially grounded in those of discrete mathematics.

Discrete mathematics and *functional programming* are equal partners in this endeavor, with the programming topics giving concrete applications and illustrations of the mathematical topics, and the mathematics providing the scaffolding for explaining the programming concepts. The two work together in mutual illumination. (The time spent on the math and programming content streams, however, is closer to a 60-40 split.)

Discrete math courses and texts come in many flavors. This one emphasizes sets, using the set concept as the building block for discussions on propositional logic, relations, recursion, functions, graphs, complexity classes, lattices, groups, and automata. Of course the specific topics themselves are subservient to our real object: teaching how to write proofs.

The presentation of functional programming is based on the classic approach embodied in Abelson and Sussman's *Structure and Interpretation of Computer Programs* (SICP) [1]. In short, it is SICP-light, modernized and adapted to the discrete math context, and translated into the ML programming language.

Rationale

Why teach discrete mathematics—or functional programming—this way? This approach is proposed as a solution to a variety of curricular problems.

Serving the constituencies. As a rookie professor, the author taught a discrete math course designed for math majors but filled with math-averse computer science majors fulfilling their math requirement and, in the same semester, an introductory programming course designed for computer science majors but filled with math majors fulfilling their computing requirement. Needless to say, much effort went into convincing each population that the course was relevant to them.

If the supporting courses really are so important, than why do these need to be separate tasks? Why not offer a single course intertwining programming and proof-based discrete mathematics, making their mutual relevancy explicit? The math content will motivate the math majors and make the programming more familiar and palatable. The programming content will sweeten the math pill for the computer science majors.

Moreover, bringing the two populations together will allow them to learn from and help each other. This text's premise is

> Math majors should learn to write programs
> and
> computer science majors should learn to write proofs
> *together*.

Further still, this text is appropriate for non-majors who are interested in a compact experience in both fields. Accordingly, liberal arts colleges will find this approach particularly useful for interdisciplinary cross-pollination.

Theory vs practice. Where do the foundations of computing belong in undergraduate curriculum? Some departments have faced battles between, on one hand, those wanting to use the foundations and theory of computer science as a gateway for incoming majors and, on the other, those who want to ensure students develop programming skills immediately.

With this text, you do not have to choose. While this is not explicitly a course on the theory of computation or other foundations of computer science, discrete mathematics provides the right context for previewing and introducing these ideas. Moreover, the programming content keeps the theory grounded in practice.

Coverage of functional programming. Despite the paradigm's importance, finding the right place for functional programming in an undergraduate curriculum has been tricky. From the 80's through the early 2000's, several good texts were produced for a *functional-first* approach to teaching programming, but the approach never became dominant. Most students receive a brief exposure to functional programming in a sophomore-level programming languages course, if at all. As the undergraduate curriculum becomes crowded with new topics vying for attention and old, elective topic being pushed earlier (mobile computing and concurrency, for two examples), the programming languages course absorbs much of the pressure. The temptation to jettison functional programming altogether is strong.

This would be a great mistake. Many design patterns in object-oriented programming are imports of ideas from functional programming. Features from functional languages are appearing in languages not traditionally thought of as functional (Python, C#, and Java 8). And functional programming is a handy way to reason about questions in the foundations of computer science.

The solution proposed here is to put functional programming in the discrete math course. This way functional programming is seen early (ideally freshman or sophomore year) and, for computer science majors, is taught in parallel with a traditional introduction to programming, such as would teach imperative, object-oriented, and concurrent programming.

Why ML? The ML programming language—to be specific, Standard ML or SML—is chosen as the language for this book, but the approach is not inherently tied to ML. In choosing a language, our first goal should be to do no harm—the language, especially its syntax, should not get in the way. ML was chosen for its simplicity, particularly its similarity to mathematical notation. Students used to mathematics should find the syntax natural. For students with prior programming experience, it is not much of a transition syntactically from, say, Java, either.

With a little effort, this text could be used with programming examples and exercises in F#, OCaml, or Haskell. With a little *more* effort, a Lisp dialect or even Python could be used.

Themes

A diverse set of topics fall under the discrete mathematics umbrella, but certain themes come up throughout the book.

Proof and program. This text strives to make skill in proof-writing and programming transferable from one to the other. The interplay between the two content streams shows how the same thinking patterns for writing a rigorous proof should be employed to write a correct and useful program.

Thinking recursively. Many students have found recursion to be one of the most mind-boggling parts of learning programming. The functional paradigm and the discrete math context make recursion much more natural. Recursive thinking in both algorithms and data structures is a stepping stone to proofs using structural induction and mathematical induction.

Formal definitions. Precision in proofs hangs on precision in the definitions of things the proof is about. Informal definitions are useful for building intuition, but this text also calls students' attention to the careful definitions of the terms.

Analysis and synthesis. *Analysis* is taking something apart. *Synthesis* is putting things together. Whether we are proving a proposition or writing a program, thinking through the problem often comes into two phases: breaking it down into its components and assembling them into a result. In a proof this manifests itself in a first section of the proof where the given information is dissected using formal definitions and a second section where the known information is assembled into the proposition being proven, again using formal definitions. (We call that the analytical and synthetic use of the definitions.) A similar pattern is seen in operations on lists—analytical operations that take lists apart and synthetic operations that construct them.

How to use this text

Audience. Ideally this text can be used by first-year undergraduates who are well-prepared mathematically—that is, able to reason abstractly and willing to think in new ways. Weaker students mathematically may be better served by taking such a course in their sophomore or junior years. The only hard prerequisites are high school algebra and pre-calculus. Occasionally the text uses examples from differential or integral calculus, but these can be skipped without harming the general flow the material.

The structure of each chapter. The names of the chapters are singular nouns: Set, Proof, Function, Graph, etc. This is to draw attention to each chapter's primary object of study. The opening sections of a chapter provide foundational definitions and examples, followed by the most important properties, propositions (and proofs) about the object of study, and finally applications of them, especially computational applications.

All sections except special topics or those providing a very general introduction end with a selection of exercises. Almost all chapters end with an extended programming example (including a project) and a special topic that condenses an advanced mathematical or computational idea to an accessible level based on the contents of the chapter.

The structure of the book. The chapters are collected into three parts. Set, List, Proposition, and Proof constitute the Foundations—ideas and skills for a student to master before seeing the other material which is built on them. The Core part of the text is the chapters on Relation, Self Reference, and Function. The Foundations and Core chapters build on each other in sequence.

The Elective chapters—Graph, Complexity Class, Lattice, Group, and Automaton—represent more advanced topics, building on both the foundations and the core. These chapters are almost completely independent of each other. The instructor may pick and choose from these and even reorder them with very little difficulty. A new recurring theme emerges in Graph, Lattice, and Group, that of **isomorphism**, structural equivalence between mathematical objects.

Pacing and planning. There is a wide range of difficulty in this material, and accordingly it can be adapted to courses at several levels. The Electives part naturally contains more advanced material, but particularly challenging sections pop up at various points in the book. These include Sections 4.9, 5.9, 6.7–6.10, 7.9, and 10.4. These (and any of the extended examples or special topics) can be omitted without harming the flow of the text.

For average to strong students in their freshman or sophomore year, most of Chapters 1–7 can be covered in one semester—the instructor is encouraged to choose some of the challenging or extended example sections to cover, but not all. The chapters in the Elective part can be used in a second semester. There may be time at the end of the semester to cover select topics from the Elective part. The author has used this material for six years in such a course with this approximate outline (mapping class days to sections, assuming a four-hour course meeting three days a week):

Day 1	1.1–1.3	Day 15	4.1–4.2	Day 27	6.1–6.2
Day 2	1.4–1.6	Day 16	4.3–4.4	Day 28	6.4
Day 3	1.7–1.9	Day 17	4.5–4.8	Day 29	6.5–6.6
Day 4	1.10–1.13	Day 18	4.9	Day 30	6.9–6.10
Day 5	2.1–2.2	Day 19	4.10–4.11	Day 31	6.10–6.11
Day 6	2.3–2.4	Day 20	5.1–5.3	Day 32	6.12
Day 7	2.5–2.6	Day 21	5.4	Day 33	7.1–7.3
Day 8	3.1–3.4	Day 22	5.5	Day 34	7.4–7.5
Day 9	3.5–3.7	Day 23	5.6–5.7	Day 35	7.6–7.8
Day 10	3.8–3.9	Day 24	5.8–5.9	Day 36	7.9
Day 11	3.10–3.13	Day 25	Review; 4.12	Day 37	Review; 7.15
Day 12	3.14	Day 26	Test	Day 38	Test
Day 13	Review				
Day 14	Test				

This schedule leaves about two weeks at the end of the semester for topics chosen from later chapters.

For a class of students who are less mathematically prepared, this text can be used in a more slowly paced course that covers Chapters 1–3 and roughly the first half of each of Chapters 4–7.

For a class of very experienced students, this text can be adapted to an advanced course where Chapters 1–4 are treated very quickly, leaving time for significant coverage of material from Chapters 8–12.

Supplemental materials and other related resources can be found at `http://cs.wheaton.edu/~tvandrun/dmfp`.