

The Little Typer The Mit Press

This book covers elementary discrete mathematics for computer science and engineering. It emphasizes mathematical definitions and proofs as well as applicable methods. Topics include formal logic notation, proof methods; induction, well-ordering; sets, relations; elementary graph theory; integer congruences; asymptotic notation and growth of functions; permutations and combinations, counting principles; discrete probability. Further selected topics may also be covered, such as recursive definition and structural induction; state machines and invariants; recurrences; generating functions. A thorough and accessible introduction to a range of key ideas in type systems for programming language. The study of type systems for programming languages now touches many areas of computer science, from language design and implementation to software engineering, network security, databases, and analysis of concurrent and distributed systems. This book offers accessible introductions to key ideas in the field, with contributions by experts on each topic. The topics covered include precise type analyses, which extend simple type systems to give them a better grip on the run time behavior of systems; type systems for low-level languages; applications of types to reasoning about computer programs; type theory as a framework for the design of sophisticated module systems; and advanced techniques in ML-style type inference. Advanced Topics in Types and Programming Languages builds on Benjamin Pierce's Types and Programming Languages (MIT Press, 2002); most of the chapters should be accessible to readers familiar with basic notations and techniques of operational semantics and type systems—the material covered in the first half of the earlier book. Advanced Topics in Types and Programming Languages can be used in the classroom and as a resource for professionals. Most chapters include exercises, ranging in difficulty from quick comprehension checks to challenging extensions, many with solutions.

The Little Typer MIT Press

How the future has been imagined and made, through the work of writers, artists, inventors, and designers. The future is like an unwritten book. It is not something we see in a crystal ball, or can only hope to predict, like the weather. In this volume of the MIT Press's Essential Knowledge series, Nick Montfort argues that the future is something to be made, not predicted. Montfort offers what he considers essential knowledge about the future, as seen in the work of writers, artists, inventors, and designers (mainly in Western culture) who developed and described the core components of the futures they envisioned. Montfort's approach is not that of futurology or scenario planning; instead, he reports on the work of making the future—the thinkers who devoted themselves to writing pages in the unwritten book. Douglas Engelbart, Alan Kay, and Ted Nelson didn't predict the future of computing, for instance. They were three of the people who made it.

Montfort focuses on how the development of technologies—with an emphasis on digital technologies—has been bound up with ideas about the future. Readers learn about kitchens of the future and the vision behind them; literary utopias, from Plato's Republic to Edward Bellamy's Looking Backward and Charlotte Perkins Gilman's Herland; the Futurama exhibit at the 1939 New York World's Fair; and what led up to Tim Berners-Lee's invention of the World Wide Web. Montfort describes the notebook computer as a human-centered alternative to the idea of the computer as a room-sized “giant brain”; speculative practice in design and science fiction; and, throughout, the best ways to imagine and build the future. The notion that “thinking about computing is one of the most exciting things the human mind can do” sets both *The Little Schemer* (formerly known as *The Little LISPer*) and its new companion volume, *The Seasoned Schemer*, apart from other books on LISP. The authors' enthusiasm for their subject is compelling as they present abstract concepts in a humorous and easy-to-grasp fashion. Together, these books will open new doors of thought to anyone who wants to find out what computing is really about. *The Little Schemer* introduces computing as an extension of arithmetic and algebra; things that everyone studies in grade school and high school. It introduces programs as recursive functions and briefly discusses the limits of what computers can do. The authors use the programming language Scheme, and interesting foods to illustrate these abstract ideas. *The Seasoned Schemer* informs the reader about additional dimensions of computing: functions as values, change of state, and exceptional cases. *The Little LISPer* has been a popular introduction to LISP for many years. It had appeared in French and Japanese. *The Little Schemer* and *The Seasoned Schemer* are worthy successors and will prove equally popular as textbooks for Scheme courses as well as companion texts for any complete introductory course in Computer Science.

How big data is transforming the creative industries, and how those industries can use lessons from Netflix, Amazon, and Apple to fight back. “[The authors explain] gently yet firmly exactly how the internet threatens established ways and what can and cannot be done about it. Their book should be required for anyone who wishes to believe that nothing much has changed.” —The Wall Street Journal “Packed with examples, from the nimble-footed who reacted quickly to adapt their businesses, to laggards who lost empires.” —Financial Times Traditional network television programming has always followed the same script: executives approve a pilot, order a trial number of episodes, and broadcast them, expecting viewers to watch a given show on their television sets at the same time every week. But then came Netflix's *House of Cards*. Netflix gauged the show's potential from data it had gathered about subscribers' preferences, ordered two seasons without seeing a pilot, and uploaded the first thirteen episodes all at once for viewers to watch whenever they wanted on the devices of their choice. In this book, Michael Smith and Rahul Telang, experts on entertainment analytics, show how the success of *House of Cards* upended the film and TV industries—and how companies like Amazon and

Apple are changing the rules in other entertainment industries, notably publishing and music. We're living through a period of unprecedented technological disruption in the entertainment industries. Just about everything is affected: pricing, production, distribution, piracy. Smith and Telang discuss niche products and the long tail, product differentiation, price discrimination, and incentives for users not to steal content. To survive and succeed, businesses have to adapt rapidly and creatively. Smith and Telang explain how. How can companies discover who their customers are, what they want, and how much they are willing to pay for it? Data. The entertainment industries, must learn to play a little "moneyball." The bottom line: follow the data.

The first comprehensive presentation of reduction semantics in one volume, and the first tool set for such forms of semantics. This text is the first comprehensive presentation of reduction semantics in one volume; it also introduces the first reliable and easy-to-use tool set for such forms of semantics. Software engineers have long known that automatic tool support is critical for rapid prototyping and modeling, and this book is addressed to the working semantics engineer (graduate student or professional language designer). The book comes with a prototyping tool suite to develop, explore, test, debug, and publish semantic models of programming languages. With PLT Redex, semanticists can formulate models as grammars and reduction models on their computers with the ease of paper and pencil. The text first presents a framework for the formulation of language models, focusing on equational calculi and abstract machines, then introduces PLT Redex, a suite of software tools for expressing these models as PLT Redex models. Finally, experts describe a range of models formulated in Redex. PLT Redex comes with the PLT Scheme implementation, available free at <http://www.plt-scheme.org/>. Readers can download the software and experiment with Redex as they work their way through the book.

Summary Type-Driven Development with Idris, written by the creator of Idris, teaches you how to improve the performance and accuracy of your programs by taking advantage of a state-of-the-art type system. This book teaches you with Idris, a language designed to support type-driven development. Purchase of the print book includes a free eBook in PDF, Kindle, and ePub formats from Manning Publications. About the Technology Stop fighting type errors! Type-driven development is an approach to coding that embraces types as the foundation of your code - essentially as built-in documentation your compiler can use to check data relationships and other assumptions. With this approach, you can define specifications early in development and write code that's easy to maintain, test, and extend. Idris is a Haskell-like language with first-class, dependent types that's perfect for learning type-driven programming techniques you can apply in any codebase. About the Book Type-Driven Development with Idris teaches you how to improve the performance and accuracy of your code by taking advantage of a state-of-the-art type system. In this book, you'll learn

type-driven development of real-world software, as well as how to handle side effects, interaction, state, and concurrency. By the end, you'll be able to develop robust and verified software in Idris and apply type-driven development methods to other languages. What's Inside Understanding dependent types Types as first-class language constructs Types as a guide to program construction Expressing relationships between data About the Reader Written for programmers with knowledge of functional programming concepts. About the Author Edwin Brady leads the design and implementation of the Idris language. Table of Contents PART 1 - INTRODUCTION Overview Getting started with Idris PART 2 - CORE IDRIS Interactive development with types User-defined data types Interactive programs: input and output processing Programming with first-class types Interfaces: using constrained generic types Equality: expressing relationships between data Predicates: expressing assumptions and contracts in types Views: extending pattern matching PART 3 - IDRIS AND THE REAL WORLD Streams and processes: working with infinite data Writing programs with state State machines: verifying protocols in types Dependent state machines: handling feedback and errors Type-safe concurrent programming

This textbook offers an understanding of the essential concepts of programming languages. The text uses interpreters, written in Scheme, to express the semantics of many essential language elements in a way that is both clear and directly executable. Structure and Interpretation of Computer Programs has had a dramatic impact on computer science curricula over the past decade. This long-awaited revision contains changes throughout the text. There are new implementations of most of the major programming systems in the book, including the interpreters and compilers, and the authors have incorporated many small changes that reflect their experience teaching the course at MIT since the first edition was published. A new theme has been introduced that emphasizes the central role played by different approaches to dealing with time in computational models: objects with state, concurrent programming, functional programming and lazy evaluation, and nondeterministic programming. There are new example sections on higher-order procedures in graphics and on applications of stream processing in numerical programming, and many new exercises. In addition, all the programs have been reworked to run in any Scheme implementation that adheres to the IEEE standard.

How parents have been set up to fail, and why helping them succeed is the key to achieving a fair and prosperous society. Parenting is, by many measures, the largest industry in the United States. Yet it receives little political support, and its many workers--also known as parents--toil in isolation, without recognition or compensation. If they ask for help, they are made to feel guilty. The parenting industry has no centralized organization representing its interests, and it spends almost nothing on research and development. It's almost as if parents are set up to fail. In The Parent Trap, Nate Hilger explains how this inefficient, inequitable, and demoralizing situation has come about and what we can do about it. Parents are expected to do more than care for their children. In the 90 percent of the time that their kids are not in school, parents must help them develop the skills they will

need to survive in today's socioeconomic reality. But most parents, including even the most caring parents on the planet, are not trained in skill development, and many lack the resources to pay for help--a situation that exacerbates inequality and constrains a child's chances for success later in life. How can we fix this? The key, Hilger argues, is to ask less of parents, not more. Parents need the kind of large-scale support that is best supplied by government--because, contrary to myth, government programs are effective at helping people. And a comprehensive program to help families--call it Familycare--would be an investment with a big payoff. To make this happen, parents need to organize--to build an organization that could lobby as effectively for Familycare as AARP does for Medicare.

A new edition of a book, written in a humorous question-and-answer style, that shows how to implement and use an elegant little programming language for logic programming. The goal of this book is to show the beauty and elegance of relational programming, which captures the essence of logic programming. The book shows how to implement a relational programming language in Scheme, or in any other functional language, and demonstrates the remarkable flexibility of the resulting relational programs. As in the first edition, the pedagogical method is a series of questions and answers, which proceed with the characteristic humor that marked *The Little Schemer* and *The Seasoned Schemer*. Familiarity with a functional language or with the first five chapters of *The Little Schemer* is assumed. For this second edition, the authors have greatly simplified the programming language used in the book, as well as the implementation of the language. In addition to revising the text extensively, and simplifying and revising the "Laws" and "Commandments," they have added explicit "Translation" rules to ease translation of Scheme functions into relations.

Key ideas in programming language design and implementation explained using a simple and concise framework; a comprehensive introduction suitable for use as a textbook or a reference for researchers. Hundreds of programming languages are in use today—scripting languages for Internet commerce, user interface programming tools, spreadsheet macros, page format specification languages, and many others. Designing a programming language is a metaprogramming activity that bears certain similarities to programming in a regular language, with clarity and simplicity even more important than in ordinary programming. This comprehensive text uses a simple and concise framework to teach key ideas in programming language design and implementation. The book's unique approach is based on a family of syntactically simple pedagogical languages that allow students to explore programming language concepts systematically. It takes as premise and starting point the idea that when language behaviors become incredibly complex, the description of the behaviors must be incredibly simple. The book presents a set of tools (a mathematical metalanguage, abstract syntax, operational and denotational semantics) and uses it to explore a comprehensive set of programming language design dimensions, including dynamic semantics (naming, state, control, data), static semantics (types, type reconstruction, polymorphism, effects), and pragmatics (compilation, garbage collection). The many examples and exercises offer students opportunities to apply the foundational ideas explained in the text. Specialized topics and code that implements many of the algorithms and compilation methods in the book can be found on the book's Web site, along

with such additional material as a section on concurrency and proofs of the theorems in the text. The book is suitable as a text for an introductory graduate or advanced undergraduate programming languages course; it can also serve as a reference for researchers and practitioners.

An introduction to writing proofs about computer programs, written in an accessible question-and-answer style, complete with step-by-step examples and a simple proof assistant. The Little Prover introduces inductive proofs as a way to determine facts about computer programs. It is written in an approachable, engaging style of question-and-answer, with the characteristic humor of *The Little Schemer* (fourth edition, MIT Press). Sometimes the best way to learn something is to sit down and do it; the book takes readers through step-by-step examples showing how to write inductive proofs. The Little Prover assumes only knowledge of recursive programs and lists (as presented in the first three chapters of *The Little Schemer*) and uses only a few terms beyond what novice programmers already know. The book comes with a simple proof assistant to help readers work through the book and complete solutions to every example.

Riffs, revisions, knockoffs, and homages: artists pay tribute to Ed Ruscha's famous photo-conceptual small books. In the 1960s and 1970s, the artist Ed Ruscha created a series of small photo-conceptual artist's books, among them *Twentysix Gas Stations*, *Various Small Fires*, *Every Building on the Sunset Strip*, *Thirtyfour Parking Lots*, *Real Estate Opportunities*, and *A Few Palm Trees*. Featuring mundane subjects photographed prosaically, with idiosyncratically deadpan titles, these "small books" were sought after, collected, and loved by Ruscha's fans and fellow artists. Over the past thirty years, close to 100 other small books that appropriated or paid homage to Ruscha's have appeared throughout the world. This book collects ninety-one of these projects, showcasing the cover and sample layouts from each along with a description of the work. It also includes selections from Ruscha's books and an appendix listing all known Ruscha book tributes. These small books revisit, imitate, honor, and parody Ruscha in form, content, and title. Some rephotograph his subjects: *Thirtyfour Parking Lots*, *Forty Years Later*. Some offer a humorous variation: *Various Unbaked Cookies* (which concludes, as did Ruscha's *Various Small Fires*, with a glass of milk), *Twentynine Palms* (twenty-nine photographs of palm-readers' signs). Some say something different: *None of the Buildings on Sunset Strip*. Some reach for a connection with Ruscha himself: *17 Parked Cars in Various Parking Lots Along Pacific Coast Highway Between My House and Ed Ruscha's*. With his books, Ruscha expanded the artist's field of permissible subjects, approaches, and methods. With *VARIOUS SMALL BOOKS*, various artists pay tribute to Ed Ruscha and extend the legacy of his books.

A handbook to the Coq software for writing and checking mathematical proofs, with a practical engineering focus. The technology of mechanized program verification can play a supporting role in many kinds of research projects in computer science, and related tools for formal proof-checking are seeing increasing adoption in mathematics and engineering. This book provides an introduction to the Coq software for writing and checking mathematical proofs. It takes a practical engineering focus throughout, emphasizing techniques that will help users to build, understand, and maintain large Coq developments and minimize the cost of code change

over time. Two topics, rarely discussed elsewhere, are covered in detail: effective dependently typed programming (making productive use of a feature at the heart of the Coq system) and construction of domain-specific proof tactics. Almost every subject covered is also relevant to interactive computer theorem proving in general, not just program verification, demonstrated through examples of verified programs applied in many different sorts of formalizations. The book develops a unique automated proof style and applies it throughout; even experienced Coq users may benefit from reading about basic Coq concepts from this novel perspective. The book also offers a library of tactics, or programs that find proofs, designed for use with examples in the book. Readers will acquire the necessary skills to reimplement these tactics in other settings by the end of the book. All of the code appearing in the book is freely available online.

“In a time in which the ways we communicate and connect are constantly changing, and not always for the better, Sherry Turkle provides a much needed voice of caution and reason to help explain what the f*** is going on.” —Aziz Ansari, author of *Modern Romance*

Renowned media scholar Sherry Turkle investigates how a flight from conversation undermines our relationships, creativity, and productivity—and why reclaiming face-to-face conversation can help us regain lost ground. We live in a technological universe in which we are always communicating. And yet we have sacrificed conversation for mere connection. Preeminent author and researcher Sherry Turkle has been studying digital culture for over thirty years. Long an enthusiast for its possibilities, here she investigates a troubling consequence: at work, at home, in politics, and in love, we find ways around conversation, tempted by the possibilities of a text or an email in which we don't have to look, listen, or reveal ourselves. We develop a taste for what mere connection offers. The dinner table falls silent as children compete with phones for their parents' attention. Friends learn strategies to keep conversations going when only a few people are looking up from their phones. At work, we retreat to our screens although it is conversation at the water cooler that increases not only productivity but commitment to work. Online, we only want to share opinions that our followers will agree with – a politics that shies away from the real conflicts and solutions of the public square. The case for conversation begins with the necessary conversations of solitude and self-reflection. They are endangered: these days, always connected, we see loneliness as a problem that technology should solve. Afraid of being alone, we rely on other people to give us a sense of ourselves, and our capacity for empathy and relationship suffers. We see the costs of the flight from conversation everywhere: conversation is the cornerstone for democracy and in business it is good for the bottom line. In the private sphere, it builds empathy, friendship, love, learning, and productivity. But there is good news: we are resilient. Conversation cures. Based on five years of research and interviews in homes, schools, and the workplace, Turkle argues that we have come to a better understanding of where our technology can and cannot take us and that the time is right to reclaim conversation. The most human—and

humanizing—thing that we do. The virtues of person-to-person conversation are timeless, and our most basic technology, talk, responds to our modern challenges. We have everything we need to start, we have each other. Turkle's latest book, *The Empathy Diaries* (3/2/21) is available now.

A new edition of a textbook that provides students with a deep, working understanding of the essential concepts of programming languages, completely revised, with significant new material. This book provides students with a deep, working understanding of the essential concepts of programming languages. Most of these essentials relate to the semantics, or meaning, of program elements, and the text uses interpreters (short programs that directly analyze an abstract representation of the program text) to express the semantics of many essential language elements in a way that is both clear and executable. The approach is both analytical and hands-on. The book provides views of programming languages using widely varying levels of abstraction, maintaining a clear connection between the high-level and low-level views. Exercises are a vital part of the text and are scattered throughout; the text explains the key concepts, and the exercises explore alternative designs and other issues. The complete Scheme code for all the interpreters and analyzers in the book can be found online through The MIT Press web site. For this new edition, each chapter has been revised and many new exercises have been added. Significant additions have been made to the text, including completely new chapters on modules and continuation-passing style. *Essentials of Programming Languages* can be used for both graduate and undergraduate courses, and for continuing education courses for programmers.

This text develops a comprehensive theory of programming languages based on type systems and structural operational semantics. Language concepts are precisely defined by their static and dynamic semantics, presenting the essential tools both intuitively and rigorously while relying on only elementary mathematics. These tools are used to analyze and prove properties of languages and provide the framework for combining and comparing language features. The broad range of concepts includes fundamental data types such as sums and products, polymorphic and abstract types, dynamic typing, dynamic dispatch, subtyping and refinement types, symbols and dynamic classification, parallelism and cost semantics, and concurrency and distribution. The methods are directly applicable to language implementation, to the development of logics for reasoning about programs, and to the formal verification language properties such as type safety. This thoroughly revised second edition includes exercises at the end of nearly every chapter and a new chapter on type refinements.

Everything we need to know about metadata, the usually invisible infrastructure for information with which we interact every day. When “metadata” became breaking news, appearing in stories about surveillance by the National Security Agency, many members of the public encountered this once-obscure term from information science for the first time.

Should people be reassured that the NSA was “only” collecting metadata about phone calls—information about the caller, the recipient, the time, the duration, the location—and not recordings of the conversations themselves? Or does phone call metadata reveal more than it seems? In this book, Jeffrey Pomerantz offers an accessible and concise introduction to metadata. In the era of ubiquitous computing, metadata has become infrastructural, like the electrical grid or the highway system. We interact with it or generate it every day. It is not, Pomerantz tell us, just “data about data.” It is a means by which the complexity of an object is represented in a simpler form. For example, the title, the author, and the cover art are metadata about a book. When metadata does its job well, it fades into the background; everyone (except perhaps the NSA) takes it for granted. Pomerantz explains what metadata is, and why it exists. He distinguishes among different types of metadata—descriptive, administrative, structural, preservation, and use—and examines different users and uses of each type. He discusses the technologies that make modern metadata possible, and he speculates about metadata's future. By the end of the book, readers will see metadata everywhere. Because, Pomerantz warns us, it's metadata's world, and we are just living in it.

An introduction to dependent types, demonstrating the most beautiful aspects, one step at a time. A program's type describes its behavior. Dependent types are a first-class part of a language, and are much more powerful than other kinds of types; using just one language for types and programs allows program descriptions to be as powerful as the programs they describe. *The Little Typer* explains dependent types, beginning with a very small language that looks very much like Scheme and extending it to cover both programming with dependent types and using dependent types for mathematical reasoning. Readers should be familiar with the basics of a Lisp-like programming language, as presented in the first four chapters of *The Little Schemer*. The first five chapters of *The Little Typer* provide the needed tools to understand dependent types; the remaining chapters use these tools to build a bridge between mathematics and programming. Readers will learn that tools they know from programming—pairs, lists, functions, and recursion—can also capture patterns of reasoning. *The Little Typer* does not attempt to teach either practical programming skills or a fully rigorous approach to types. Instead, it demonstrates the most beautiful aspects as simply as possible, one step at a time. Analyzes cognitive, social and technical issues of end user programming. Drawing on empirical research on existing end user systems, this text examines the importance of task-specific programming languages, visual application frameworks and collaborative work practices for end user computing.

An introduction to the scientific consensus on the human role in global warming.

Recounts the story of how a notorious gang of MIT blackjack savants devised and received backing for a system for winning at the world's most sophisticated casinos, an endeavor that earned them more than three million dollars.

Originally published as *Bringing Down the House*. Reissue. (A Columbia Pictures film, written by Peter Steinfeld & Allan Loeb, directed by Robert Luketic, releasing March 2008, starring Kevin Spacey, Kate Bosworth, Laurence Fishburne, Jim Sturgess, & others) (Current Affairs)

The process of user-centered innovation: how it can benefit both users and manufacturers and how its emergence will bring changes in business models and in public policy. Innovation is rapidly becoming democratized. Users, aided by improvements in computer and communications technology, increasingly can develop their own new products and services. These innovating users—both individuals and firms—often freely share their innovations with others, creating user-innovation communities and a rich intellectual commons. In *Democratizing Innovation*, Eric von Hippel looks closely at this emerging system of user-centered innovation. He explains why and when users find it profitable to develop new products and services for themselves, and why it often pays users to reveal their innovations freely for the use of all. The trend toward democratized innovation can be seen in software and information products—most notably in the free and open-source software movement—but also in physical products. Von Hippel's many examples of user innovation in action range from surgical equipment to surfboards to software security features. He shows that product and service development is concentrated among "lead users," who are ahead on marketplace trends and whose innovations are often commercially attractive. Von Hippel argues that manufacturers should redesign their innovation processes and that they should systematically seek out innovations developed by users. He points to businesses—the custom semiconductor industry is one example—that have learned to assist user-innovators by providing them with toolkits for developing new products. User innovation has a positive impact on social welfare, and von Hippel proposes that government policies, including R&D subsidies and tax credits, should be realigned to eliminate biases against it. The goal of a democratized user-centered innovation system, says von Hippel, is well worth striving for. An electronic version of this book is available under a Creative Commons license.

An examination of the uses of data within a changing knowledge infrastructure, offering analysis and case studies from the sciences, social sciences, and humanities. "Big Data" is on the covers of *Science*, *Nature*, the *Economist*, and *Wired* magazines, on the front pages of the *Wall Street Journal* and the *New York Times*. But despite the media hyperbole, as Christine Borgman points out in this examination of data and scholarly research, having the right data is usually better than having more data; little data can be just as valuable as big data. In many cases, there are no data—because relevant data don't exist, cannot be found, or are not available. Moreover, data sharing is difficult, incentives to do so are minimal, and data practices vary widely across disciplines. Borgman, an often-cited authority on scholarly communication, argues that data have no value or meaning in isolation; they exist within a knowledge infrastructure—an ecology of people, practices, technologies, institutions, material objects, and relationships. After laying out the premises of her investigation—six "provocations" meant to inspire discussion about the uses of data in scholarship—Borgman offers case studies of data practices in the sciences, the social sciences, and the humanities, and then considers the implications of her findings for scholarly practice and research policy. To manage and exploit data over the long term, Borgman argues, requires massive investment in knowledge infrastructures; at stake is the future of scholarship.

foreword by Ralph E. Johnson and drawings by Duane Bibby 'This is a book of 'why' not 'how.' If you are interested in the nature of computation and curious about the very idea behind object orientation, this book is for you. This book will engage your brain (if not your tummy). Through its sparkling interactive style, you will learn about three essential OO concepts: interfaces, visitors, and factories. A

refreshing change from the 'yet another Java book' phenomenon. Every serious Java programmer should own a copy.' -- Gary McGraw, Ph.D., Research Scientist at Reliable Software Technologies and coauthor of Java Security Java is a new object-oriented programming language that was developed by Sun Microsystems for programming the Internet and intelligent appliances. In a very short time it has become one of the most widely used programming languages for education as well as commercial applications. Design patterns, which have moved object-oriented programming to a new level, provide programmers with a language to communicate with others about their designs. As a result, programs become more readable, more reusable, and more easily extensible. In this book, Matthias Felleisen and Daniel Friedman use a small subset of Java to introduce pattern-directed program design. With their usual clarity and flair, they gently guide readers through the fundamentals of object-oriented programming and pattern-based design. Readers new to programming, as well as those with some background, will enjoy their learning experience as they work their way through Felleisen and Friedman's dialogue.

[src='/graphics/yellowball.gif' href='/books/FELTP/Java-fm.html'Foreword and Preface](#)

An exploration of how design might be led by marginalized communities, dismantle structural inequality, and advance collective liberation and ecological survival. What is the relationship between design, power, and social justice? “Design justice” is an approach to design that is led by marginalized communities and that aims explicitly to challenge, rather than reproduce, structural inequalities. It has emerged from a growing community of designers in various fields who work closely with social movements and community-based organizations around the world. This book explores the theory and practice of design justice, demonstrates how universalist design principles and practices erase certain groups of people—specifically, those who are intersectionally disadvantaged or multiply burdened under the matrix of domination (white supremacist heteropatriarchy, ableism, capitalism, and settler colonialism)—and invites readers to “build a better world, a world where many worlds fit; linked worlds of collective liberation and ecological sustainability.” Along the way, the book documents a multitude of real-world community-led design practices, each grounded in a particular social movement. Design Justice goes beyond recent calls for design for good, user-centered design, and employment diversity in the technology and design professions; it connects design to larger struggles for collective liberation and ecological survival.

Title and statement of responsibility from page 4 of cover.

The Formal Semantics of Programming Languages provides the basic mathematical techniques necessary for those who are beginning a study of the semantics and logics of programming languages. These techniques will allow students to invent, formalize, and justify rules with which to reason about a variety of programming languages. Although the treatment is elementary, several of the topics covered are drawn from recent research, including the vital area of concurrency. The book contains many exercises ranging from simple to miniprojects. Starting with basic set theory, structural operational semantics is introduced as a way to define the meaning of programming languages along with associated proof techniques. Denotational and axiomatic semantics are illustrated on a simple language of while-programs, and fall proofs are given of the equivalence of the operational and denotational semantics and soundness and relative completeness of the axiomatic semantics. A proof of Godel's incompleteness theorem, which emphasizes the impossibility of achieving a fully complete axiomatic semantics, is included. It is supported by an appendix providing an introduction to the theory of computability based on while-programs. Following a presentation of domain theory, the semantics and methods of proof for several functional languages are treated. The simplest language is that of recursion equations with both call-by-value and call-by-name evaluation. This work is extended to languages with higher and recursive types, including a treatment of the eager and lazy lambda-calculi. Throughout, the relationship between denotational and

operational semantics is stressed, and the proofs of the correspondence between the operation and denotational semantics are provided. The treatment of recursive types - one of the more advanced parts of the book - relies on the use of information systems to represent domains. The book concludes with a chapter on parallel programming languages, accompanied by a discussion of methods for specifying and verifying nondeterministic and parallel programs.

Chaos and complexity explained, with illuminating examples ranging from unpredictable pendulums to London's wobbly Millennium Bridge. The math we are taught in school is precise and only deals with simple situations. Reality is far more complex. Trying to understand a system with multiple interacting components—the weather, for example, or the human body, or the stock market—means dealing with two factors: chaos and complexity. If we don't understand these two essential subjects, we can't understand the real world. In *Everyday Chaos*, Brian Clegg explains chaos and complexity for the general reader, with an accessible, engaging text and striking full-color illustrations. By chaos, Clegg means a system where complex interactions make predicting long-term outcomes nearly impossible; complexity means complex interacting systems that have new emergent properties that make them more than the sum of their parts. Clegg illustrates these phenomena with discussions of predictable randomness, the power of probability, and the behavior of pendulums. He describes what Newton got wrong about gravity; how feedback kept steam engines from exploding; and why weather produces chaos. He considers the stock market, politics, bestseller lists, big data, and London's wobbling Millennium Bridge as examples of chaotic systems, and he explains how a better understanding of chaos helps scientists predict more accurately the risk of catastrophic Earth-asteroid collisions. We learn that our brains are complex, self-organizing systems; that the structure of snowflakes exemplifies emergence; and that life itself has been shown to be an emergent property of a complex system.

For anyone who has ever wondered how computers solve problems, an engagingly written guide for nonexperts to the basics of computer algorithms. Have you ever wondered how your GPS can find the fastest way to your destination, selecting one route from seemingly countless possibilities in mere seconds? How your credit card account number is protected when you make a purchase over the Internet? The answer is algorithms. And how do these mathematical formulations translate themselves into your GPS, your laptop, or your smart phone? This book offers an engagingly written guide to the basics of computer algorithms. In *Algorithms Unlocked*, Thomas Cormen—coauthor of the leading college textbook on the subject—provides a general explanation, with limited mathematics, of how algorithms enable computers to solve problems. Readers will learn what computer algorithms are, how to describe them, and how to evaluate them. They will discover simple ways to search for information in a computer; methods for rearranging information in a computer into a prescribed order (“sorting”); how to solve basic problems that can be modeled in a computer with a mathematical structure called a “graph” (useful for modeling road networks, dependencies among tasks, and financial relationships); how to solve problems that ask questions about strings of characters such as DNA structures; the basic principles behind cryptography; fundamentals of data compression; and even that there are some problems that no one has figured out how to solve on a computer in a reasonable amount of time.

A single line of code offers a way to understand the cultural context of computing. This book takes a single line of code—the extremely concise BASIC program for the Commodore 64 inscribed in the title—and uses it as a lens through which to consider the phenomenon of creative computing and the way computer programs exist in culture. The authors of this collaboratively written book treat code not as merely functional but as a text—in the case of 10 PRINT, a text that

appeared in many different printed sources—that yields a story about its making, its purpose, its assumptions, and more. They consider randomness and regularity in computing and art, the maze in culture, the popular BASIC programming language, and the highly influential Commodore 64 computer.

Umberto Eco's wise and witty guide to researching and writing a thesis, published in English for the first time. By the time Umberto Eco published his best-selling novel *The Name of the Rose*, he was one of Italy's most celebrated intellectuals, a distinguished academic and the author of influential works on semiotics. Some years before that, in 1977, Eco published a little book for his students, *How to Write a Thesis*, in which he offered useful advice on all the steps involved in researching and writing a thesis—from choosing a topic to organizing a work schedule to writing the final draft. Now in its twenty-third edition in Italy and translated into seventeen languages, *How to Write a Thesis* has become a classic. Remarkably, this is its first, long overdue publication in English. Eco's approach is anything but dry and academic. He not only offers practical advice but also considers larger questions about the value of the thesis-writing exercise. *How to Write a Thesis* is unlike any other writing manual. It reads like a novel. It is opinionated. It is frequently irreverent, sometimes polemical, and often hilarious. Eco advises students how to avoid “thesis neurosis” and he answers the important question “Must You Read Books?” He reminds students “You are not Proust” and “Write everything that comes into your head, but only in the first draft.” Of course, there was no Internet in 1977, but Eco's index card research system offers important lessons about critical thinking and information curating for students of today who may be burdened by Big Data. *How to Write a Thesis* belongs on the bookshelves of students, teachers, writers, and Eco fans everywhere. Already a classic, it would fit nicely between two other classics: *Strunk and White* and *The Name of the Rose*. Contents The Definition and Purpose of a Thesis • Choosing the Topic • Conducting Research • The Work Plan and the Index Cards • Writing the Thesis • The Final Draft

with a foreword by Robin Milner and drawings by Duane Bibby Over the past few years, ML has emerged as one of the most important members of the family of programming languages. Many professors in the United States and other countries use ML to teach courses on the principles of programming and on programming languages. In addition, ML has emerged as a natural language for software engineering courses because it provides the most sophisticated and expressive module system currently available. Felleisen and Friedman are well known for gently introducing readers to difficult ideas. *The Little MLer* is an introduction to thinking about programming and the ML programming language. The authors introduce those new to programming, as well as those experienced in other programming languages, to the principles of types, computation, and program construction. Most important, they help the reader to think recursively with types about programs.

Since it was first published in 1995, Photonic Crystals has remained the definitive text for both undergraduates and researchers on photonic band-gap materials and their use in controlling the propagation of light. This newly expanded and revised edition covers the latest developments in the field, providing the most up-to-date, concise, and comprehensive book available on these novel materials and their applications. Starting from Maxwell's equations and Fourier analysis, the authors develop the theoretical tools of photonics using principles of linear algebra and symmetry, emphasizing analogies with traditional solid-state physics and quantum theory. They then investigate the unique phenomena that take place within photonic crystals at defect sites and surfaces, from one to three dimensions. This new edition includes entirely new chapters describing important hybrid structures that use band gaps or periodicity only in some directions: periodic waveguides, photonic-crystal slabs, and photonic-crystal fibers. The authors demonstrate how the capabilities of photonic crystals to localize light can be put to work in devices such as filters and splitters. A new appendix provides an overview of computational methods for electromagnetism. Existing chapters have been considerably updated and expanded to include many new three-dimensional photonic crystals, an extensive tutorial on device design using temporal coupled-mode theory, discussions of diffraction and refraction at crystal interfaces, and more. Richly illustrated and accessibly written, Photonic Crystals is an indispensable resource for students and researchers. Extensively revised and expanded Features improved graphics throughout Includes new chapters on photonic-crystal fibers and combined index-and band-gap-guiding Provides an introduction to coupled-mode theory as a powerful tool for device design Covers many new topics, including omnidirectional reflection, anomalous refraction and diffraction, computational photonics, and much more.

An introduction to dependent types, demonstrating the most beautiful aspects, one step at a time. A program's type describes its behavior. Dependent types are a first-class part of a language, and are much more powerful than other kinds of types; using just one language for types and programs allows program descriptions to be as powerful as the programs they describe. The Little Typer explains dependent types, beginning with a very small language that looks very much like Scheme and extending it to cover both programming with dependent types and using dependent types for mathematical reasoning. Readers should be familiar with the basics of a Lisp-like programming language, as presented in the first four chapters of The Little Schemer . The first five chapters of The Little Typer provide the needed tools to understand dependent types; the remaining chapters use these tools to build a bridge between mathematics and programming. Readers will learn that tools they know from programming--pairs, lists, functions, and recursion--can also capture patterns of reasoning. The Little Typer does not attempt to teach either practical programming skills or a fully rigorous approach to types. Instead, it demonstrates the most beautiful aspects as simply as possible, one step at a time.

A comprehensive introduction to type systems and programming languages. A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute. The study of type systems—and of programming languages from a type-theoretic perspective—has important applications in software engineering, language design, high-performance compilers, and security. This text provides a comprehensive introduction both to type systems in computer science and to the basic theory of programming languages. The approach is pragmatic and operational; each new concept is motivated by programming examples and the more theoretical sections are driven by the needs of implementations. Each chapter is accompanied by numerous exercises and solutions, as well as a running implementation, available via the Web. Dependencies between chapters are explicitly identified, allowing readers to choose a variety of paths through the material. The core topics include the untyped lambda-calculus, simple type systems, type reconstruction, universal and existential polymorphism, subtyping, bounded quantification, recursive types, kinds, and type operators. Extended case studies develop a variety of approaches to modeling the features of object-oriented languages.

[Copyright: b22e0882de3f376e6261af84bdd5db34](#)