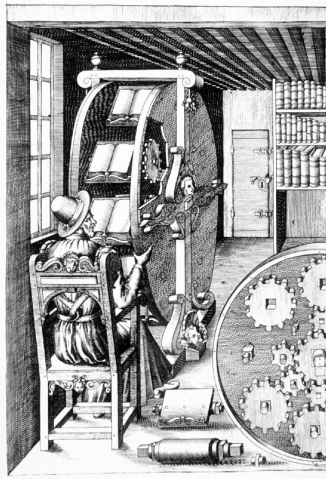


# Structure and Interpretation of Computer Programs



**SECOND EDITION**

Unofficial Texinfo Format 2.andresraba5.6

Harold Abelson and  
Gerald Jay Sussman  
with Julie Sussman  
foreword by Alan J. Perlis

©1996 by The Massachusetts Institute of Technology

Structure and Interpretation of Computer Programs,  
second edition

Harold Abelson and Gerald Jay Sussman  
with Julie Sussman, foreword by Alan J. Perlis



This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License  
([CC BY-SA 4.0](#)). Based on a work at [mitpress.mit.edu](#).

The MIT Press  
Cambridge, Massachusetts  
London, England

McGraw-Hill Book Company  
New York, St. Louis, San Francisco,  
Montreal, Toronto

Unofficial Texinfo Format [2.andresraba5.6](#) (February 2, 2016),  
based on [2.neilvandyke4](#) (January 10, 2007).

# Contents

Unofficial Texinfo Format	ix
Dedication	xii
Foreword	xiii
Preface to the Second Edition	xix
Preface to the First Edition	xxi
Acknowledgments	xxv
<b>1 Building Abstractions with Procedures</b>	<b>1</b>
1.1 The Elements of Programming . . . . .	6
1.1.1 Expressions . . . . .	7
1.1.2 Naming and the Environment . . . . .	10
1.1.3 Evaluating Combinations . . . . .	12
1.1.4 Compound Procedures . . . . .	15
1.1.5 The Substitution Model for Procedure Application	18
1.1.6 Conditional Expressions and Predicates . . . .	22
1.1.7 Example: Square Roots by Newton’s Method . .	28

1.1.8	Procedures as Black-Box Abstractions . . . . .	33
1.2	Procedures and the Processes They Generate . . . . .	40
1.2.1	Linear Recursion and Iteration . . . . .	41
1.2.2	Tree Recursion . . . . .	47
1.2.3	Orders of Growth . . . . .	54
1.2.4	Exponentiation . . . . .	57
1.2.5	Greatest Common Divisors . . . . .	62
1.2.6	Example: Testing for Primality . . . . .	65
1.3	Formulating Abstractions with Higher-Order Procedures . . . . .	74
1.3.1	Procedures as Arguments . . . . .	76
1.3.2	Constructing Procedures Using <code>lambda</code> . . . . .	83
1.3.3	Procedures as General Methods . . . . .	89
1.3.4	Procedures as Returned Values . . . . .	97
<b>2</b>	<b>Building Abstractions with Data</b>	<b>107</b>
2.1	Introduction to Data Abstraction . . . . .	112
2.1.1	Example: Arithmetic Operations for Rational Numbers . . . . .	113
2.1.2	Abstraction Barriers . . . . .	118
2.1.3	What Is Meant by Data? . . . . .	122
2.1.4	Extended Exercise: Interval Arithmetic . . . . .	126
2.2	Hierarchical Data and the Closure Property . . . . .	132
2.2.1	Representing Sequences . . . . .	134
2.2.2	Hierarchical Structures . . . . .	147
2.2.3	Sequences as Conventional Interfaces . . . . .	154
2.2.4	Example: A Picture Language . . . . .	172
2.3	Symbolic Data . . . . .	192
2.3.1	Quotation . . . . .	192

2.3.2	Example: Symbolic Differentiation . . . . .	197
2.3.3	Example: Representing Sets . . . . .	205
2.3.4	Example: Huffman Encoding Trees . . . . .	218
2.4	Multiple Representations for Abstract Data . . . . .	229
2.4.1	Representations for Complex Numbers . . . . .	232
2.4.2	Tagged data . . . . .	237
2.4.3	Data-Directed Programming and Additivity . . . . .	242
2.5	Systems with Generic Operations . . . . .	254
2.5.1	Generic Arithmetic Operations . . . . .	255
2.5.2	Combining Data of Different Types . . . . .	262
2.5.3	Example: Symbolic Algebra . . . . .	274
<b>3</b>	<b>Modularity, Objects, and State</b>	<b>294</b>
3.1	Assignment and Local State . . . . .	296
3.1.1	Local State Variables . . . . .	297
3.1.2	The Benefits of Introducing Assignment . . . . .	305
3.1.3	The Costs of Introducing Assignment . . . . .	311
3.2	The Environment Model of Evaluation . . . . .	320
3.2.1	The Rules for Evaluation . . . . .	322
3.2.2	Applying Simple Procedures . . . . .	327
3.2.3	Frames as the Repository of Local State . . . . .	330
3.2.4	Internal Definitions . . . . .	337
3.3	Modeling with Mutable Data . . . . .	341
3.3.1	Mutable List Structure . . . . .	342
3.3.2	Representing Queues . . . . .	353
3.3.3	Representing Tables . . . . .	360
3.3.4	A Simulator for Digital Circuits . . . . .	369
3.3.5	Propagation of Constraints . . . . .	386
3.4	Concurrency: Time Is of the Essence . . . . .	401

3.4.1	The Nature of Time in Concurrent Systems . . .	403
3.4.2	Mechanisms for Controlling Concurrency . . .	410
3.5	Streams . . . . .	428
3.5.1	Streams Are Delayed Lists . . . . .	430
3.5.2	Infinite Streams . . . . .	441
3.5.3	Exploiting the Stream Paradigm . . . . .	453
3.5.4	Streams and Delayed Evaluation . . . . .	470
3.5.5	Modularity of Functional Programs and Modularity of Objects . . . . .	479
<b>4</b>	<b>Metalinguistic Abstraction</b>	<b>487</b>
4.1	The Metacircular Evaluator . . . . .	492
4.1.1	The Core of the Evaluator . . . . .	495
4.1.2	Representing Expressions . . . . .	501
4.1.3	Evaluator Data Structures . . . . .	512
4.1.4	Running the Evaluator as a Program . . . . .	518
4.1.5	Data as Programs . . . . .	522
4.1.6	Internal Definitions . . . . .	526
4.1.7	Separating Syntactic Analysis from Execution .	534
4.2	Variations on a Scheme — Lazy Evaluation . . . . .	541
4.2.1	Normal Order and Applicative Order . . . . .	542
4.2.2	An Interpreter with Lazy Evaluation . . . . .	544
4.2.3	Streams as Lazy Lists . . . . .	555
4.3	Variations on a Scheme — Nondeterministic Computing	559
4.3.1	Amb and Search . . . . .	561
4.3.2	Examples of Nondeterministic Programs . . . .	567
4.3.3	Implementing the amb Evaluator . . . . .	578
4.4	Logic Programming . . . . .	594
4.4.1	Deductive Information Retrieval . . . . .	599

4.4.2	How the Query System Works . . . . .	615
4.4.3	Is Logic Programming Mathematical Logic? . .	627
4.4.4	Implementing the Query System . . . . .	635
4.4.4.1	The Driver Loop and Instantiation . .	636
4.4.4.2	The Evaluator . . . . .	638
4.4.4.3	Finding Assertions by Pattern Matching . . . . .	642
4.4.4.4	Rules and Unification . . . . .	645
4.4.4.5	Maintaining the Data Base . . . . .	651
4.4.4.6	Stream Operations . . . . .	654
4.4.4.7	Query Syntax Procedures . . . . .	656
4.4.4.8	Frames and Bindings . . . . .	659
<b>5</b>	<b>Computing with Register Machines</b>	<b>666</b>
5.1	Designing Register Machines . . . . .	668
5.1.1	A Language for Describing Register Machines .	672
5.1.2	Abstraction in Machine Design . . . . .	678
5.1.3	Subroutines . . . . .	681
5.1.4	Using a Stack to Implement Recursion . . . . .	686
5.1.5	Instruction Summary . . . . .	695
5.2	A Register-Machine Simulator . . . . .	696
5.2.1	The Machine Model . . . . .	698
5.2.2	The Assembler . . . . .	704
5.2.3	Generating Execution Procedures for Instructions . . . . .	708
5.2.4	Monitoring Machine Performance . . . . .	718
5.3	Storage Allocation and Garbage Collection . . . . .	723
5.3.1	Memory as Vectors . . . . .	724
5.3.2	Maintaining the Illusion of Infinite Memory . .	731

5.4	The Explicit-Control Evaluator . . . . .	741
5.4.1	The Core of the Explicit-Control Evaluator . . .	743
5.4.2	Sequence Evaluation and Tail Recursion . . . .	751
5.4.3	Conditionals, Assignments, and Definitions . .	756
5.4.4	Running the Evaluator . . . . .	759
5.5	Compilation . . . . .	767
5.5.1	Structure of the Compiler . . . . .	772
5.5.2	Compiling Expressions . . . . .	779
5.5.3	Compiling Combinations . . . . .	788
5.5.4	Combining Instruction Sequences . . . . .	797
5.5.5	An Example of Compiled Code . . . . .	802
5.5.6	Lexical Addressing . . . . .	817
5.5.7	Interfacing Compiled Code to the Evaluator . .	823
	<b>References</b>	<b>834</b>
	<b>List of Exercises</b>	<b>844</b>
	<b>List of Figures</b>	<b>846</b>
	<b>Index</b>	<b>848</b>
	<b>Colophon</b>	<b>855</b>



# Unofficial Texinfo Format

This is the second edition SICP book, from Unofficial Texinfo Format.

You are probably reading it in an Info hypertext browser, such as the Info mode of Emacs. You might alternatively be reading it T<sub>E</sub>X-formatted on your screen or printer, though that would be silly. And, if printed, expensive.

The freely-distributed official HTML-and-GIF format was first converted personally to Unofficial Texinfo Format (UTF) version 1 by Lytha Ayth during a long Emacs lovefest weekend in April, 2001.

The UTF is easier to search than the HTML format. It is also much more accessible to people running on modest computers, such as donated '386-based PCs. A 386 can, in theory, run Linux, Emacs, and a Scheme interpreter simultaneously, but most 386s probably can't also run both Netscape and the necessary X Window System without prematurely introducing budding young underfunded hackers to the concept of *thrashing*. UTF can also fit uncompressed on a 1.44MB floppy diskette, which may come in handy for installing UTF on PCs that do not have Internet or LAN access.

The Texinfo conversion has been a straight transliteration, to the extent possible. Like the T<sub>E</sub>X-to-HTML conversion, this was not without some introduction of breakage. In the case of Unofficial Texinfo Format,

figures have suffered an amateurish resurrection of the lost art of ASCII. Also, it's quite possible that some errors of ambiguity were introduced during the conversion of some of the copious superscripts (‘<sup>^</sup>’) and subscripts (‘<sub>^</sub>’). Divining *which* has been left as an exercise to the reader. But at least we don't put our brave astronauts at risk by encoding the *greater-than-or-equal* symbol as `<u>&gt;</u>`.

If you modify `sicp.texti` to correct errors or improve the ASCII art, then update the `@set utfversion 2.andresraba5.6` line to reflect your delta. For example, if you started with Lytha's version 1, and your name is Bob, then you could name your successive versions `1.bob1`, `1.bob2`, ... `1.bobn`. Also update `utfversiondate`. If you want to distribute your version on the Web, then embedding the string “`sicp.texti`” somewhere in the file or Web page will make it easier for people to find with Web search engines.

It is believed that the Unofficial Texinfo Format is in keeping with the spirit of the graciously freely-distributed HTML version. But you never know when someone's armada of lawyers might need something to do, and get their shorts all in a knot over some benign little thing, so think twice before you use your full name or distribute Info, DVI, PostScript, or PDF formats that might embed your account or machine name.

*Peath, Lytha Ayth*

**Addendum:** See also the SICP video lectures by Abelson and Sussman: at [MIT CSAIL](#) or [MIT OCW](#).

**Second Addendum:** Above is the original introduction to the UTF from 2001. Ten years later, UTF has been transformed: mathematical symbols and formulas are properly typeset, and figures drawn in vector graphics. The original text formulas and ASCII art figures are still there in

the Texinfo source, but will display only when compiled to Info output. At the dawn of e-book readers and tablets, reading a PDF on screen is officially not silly anymore. Enjoy!

*A.R, May, 2011*

## Dedication

**T**HIS BOOK IS DEDICATED, in respect and admiration, to the spirit that lives in the computer.

“I think that it’s extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don’t think we are. I think we’re responsible for stretching them, setting them off in new directions, and keeping fun in the house. I hope the field of computer science never loses its sense of fun. Above all, I hope we don’t become missionaries. Don’t feel as if you’re Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don’t feel as if the key to successful computing is only in your hands. What’s in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more.”

—Alan J. Perlis (April 1, 1922 – February 7, 1990)

## Foreword

EDUCATORS, GENERALS, DIETICIANS, psychologists, and parents program. Armies, students, and some societies are programmed. An assault on large problems employs a succession of programs, most of which spring into existence en route. These programs are rife with issues that appear to be particular to the problem at hand. To appreciate programming as an intellectual activity in its own right you must turn to computer programming; you must read and write computer programs—many of them. It doesn't matter much what the programs are about or what applications they serve. What does matter is how well they perform and how smoothly they fit with other programs in the creation of still greater programs. The programmer must seek both perfection of part and adequacy of collection. In this book the use of “program” is focused on the creation, execution, and study of programs written in a dialect of Lisp for execution on a digital computer. Using Lisp we restrict or limit not what we may program, but only the notation for our program descriptions.

Our traffic with the subject matter of this book involves us with three foci of phenomena: the human mind, collections of computer programs, and the computer. Every computer program is a model, hatched in the mind, of a real or mental process. These processes, arising from

human experience and thought, are huge in number, intricate in detail, and at any time only partially understood. They are modeled to our permanent satisfaction rarely by our computer programs. Thus even though our programs are carefully handcrafted discrete collections of symbols, mosaics of interlocking functions, they continually evolve: we change them as our perception of the model deepens, enlarges, generalizes until the model ultimately attains a metastable place within still another model with which we struggle. The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate. If art interprets our dreams, the computer executes them in the guise of programs!

For all its power, the computer is a harsh taskmaster. Its programs must be correct, and what we wish to say must be said accurately in every detail. As in every other symbolic activity, we become convinced of program truth through argument. Lisp itself can be assigned a semantics (another model, by the way), and if a program's function can be specified, say, in the predicate calculus, the proof methods of logic can be used to make an acceptable correctness argument. Unfortunately, as programs get large and complicated, as they almost always do, the adequacy, consistency, and correctness of the specifications themselves become open to doubt, so that complete formal arguments of correctness seldom accompany large programs. Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure—we call them idioms—and learn to combine them into larger structures using organizational techniques of proven value. These techniques are treated at length in this book, and understanding them is essential to participation in the Promethean enterprise called programming. More than anything

else, the uncovering and mastery of powerful organizational techniques accelerates our ability to create large, significant programs. Conversely, since writing large programs is very taxing, we are stimulated to invent new methods of reducing the mass of function and detail to be fitted into large programs.

Unlike programs, computers must obey the laws of physics. If they wish to perform rapidly—a few nanoseconds per state change—they must transmit electrons only small distances (at most  $1\frac{1}{2}$  feet). The heat generated by the huge number of devices so concentrated in space has to be removed. An exquisite engineering art has been developed balancing between multiplicity of function and density of devices. In any event, hardware always operates at a level more primitive than that at which we care to program. The processes that transform our Lisp programs to “machine” programs are themselves abstract models which we program. Their study and creation give a great deal of insight into the organizational programs associated with programming arbitrary models. Of course the computer itself can be so modeled. Think of it: the behavior of the smallest physical switching element is modeled by quantum mechanics described by differential equations whose detailed behavior is captured by numerical approximations represented in computer programs executing on computers composed of . . .!

It is not merely a matter of tactical convenience to separately identify the three foci. Even though, as they say, it’s all in the head, this logical separation induces an acceleration of symbolic traffic between these foci whose richness, vitality, and potential is exceeded in human experience only by the evolution of life itself. At best, relationships between the foci are metastable. The computers are never large enough or fast enough. Each breakthrough in hardware technology leads to more massive programming enterprises, new organizational principles, and

an enrichment of abstract models. Every reader should ask himself periodically “Toward what end, toward what end?”—but do not ask it too often lest you pass up the fun of programming for the constipation of bittersweet philosophy.

Among the programs we write, some (but never enough) perform a precise mathematical function such as sorting or finding the maximum of a sequence of numbers, determining primality, or finding the square root. We call such programs algorithms, and a great deal is known of their optimal behavior, particularly with respect to the two important parameters of execution time and data storage requirements. A programmer should acquire good algorithms and idioms. Even though some programs resist precise specifications, it is the responsibility of the programmer to estimate, and always to attempt to improve, their performance.

Lisp is a survivor, having been in use for about a quarter of a century. Among the active programming languages only Fortran has had a longer life. Both languages have supported the programming needs of important areas of application, Fortran for scientific and engineering computation and Lisp for artificial intelligence. These two areas continue to be important, and their programmers are so devoted to these two languages that Lisp and Fortran may well continue in active use for at least another quarter-century.

Lisp changes. The Scheme dialect used in this text has evolved from the original Lisp and differs from the latter in several important ways, including static scoping for variable binding and permitting functions to yield functions as values. In its semantic structure Scheme is as closely akin to Algol 60 as to early Lisps. Algol 60, never to be an active language again, lives on in the genes of Scheme and Pascal. It would be difficult to find two languages that are the communicating coin of two more dif-



ferent cultures than those gathered around these two languages. Pascal is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms—imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. The organizing principles used are the same in both cases, except for one extraordinarily important difference: The discretionary exportable functionality entrusted to the individual Lisp programmer is more than an order of magnitude greater than that to be found within Pascal enterprises. Lisp programs inflate libraries with functions whose utility transcends the application that produced them. The list, Lisp’s native data structure, is largely responsible for such growth of utility. The simple structure and natural applicability of lists are reflected in functions that are amazingly nonidiosyncratic. In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures. As a result the pyramid must stand unchanged for a millennium; the organism must evolve or perish.

To illustrate this difference, compare the treatment of material and exercises within this book with that in any first-course text using Pascal. Do not labor under the illusion that this is a text digestible at MIT only, peculiar to the breed found there. It is precisely what a serious book on programming Lisp must be, no matter who the student is or where it is used.

Note that this is a text about programming, unlike most Lisp books, which are used as a preparation for work in artificial intelligence. After all, the critical programming concerns of software engineering and artificial intelligence tend to coalesce as the systems under investigation

become larger. This explains why there is such growing interest in Lisp outside of artificial intelligence.

As one would expect from its goals, artificial intelligence research generates many significant programming problems. In other programming cultures this spate of problems spawns new languages. Indeed, in any very large programming task a useful organizing principle is to control and isolate traffic within the task modules via the invention of language. These languages tend to become less primitive as one approaches the boundaries of the system where we humans interact most often. As a result, such systems contain complex language-processing functions replicated many times. Lisp has such a simple syntax and semantics that parsing can be treated as an elementary task. Thus parsing technology plays almost no role in Lisp programs, and the construction of language processors is rarely an impediment to the rate of growth and change of large Lisp systems. Finally, it is this very simplicity of syntax and semantics that is responsible for the burden and freedom borne by all Lisp programmers. No Lisp program of any size beyond a few lines can be written without being saturated with discretionary functions. Invent and fit; have fits and reinvent! We toast the Lisp programmer who pens his thoughts within nests of parentheses.

Alan J. Perlis

New Haven, Connecticut

## Preface to the Second Edition

Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to always see it as a soap bubble?

—Alan J. Perlis

THE MATERIAL IN THIS BOOK has been the basis of MIT's entry-level computer science subject since 1980. We had been teaching this material for four years when the first edition was published, and twelve more years have elapsed until the appearance of this second edition. We are pleased that our work has been widely adopted and incorporated into other texts. We have seen our students take the ideas and programs in this book and build them in as the core of new computer systems and languages. In literal realization of an ancient Talmudic pun, our students have become our builders. We are lucky to have such capable students and such accomplished builders.

In preparing this edition, we have incorporated hundreds of clarifications suggested by our own teaching experience and the comments of colleagues at MIT and elsewhere. We have redesigned most of the major programming systems in the book, including the generic-arithmetic system, the interpreters, the register-machine simulator, and the compiler; and we have rewritten all the program examples to ensure that

choice) by translating the explicit-control evaluator of [Section 5.4](#) into C. In order to run this code you will need to also provide appropriate storage-allocation routines and other run-time support.

**Exercise 5.52:** As a counterpoint to [Exercise 5.51](#), modify the compiler so that it compiles Scheme procedures into sequences of C instructions. Compile the metacircular evaluator of [Section 4.1](#) to produce a Scheme interpreter written in C.

## References

**Abelson**, Harold, Andrew Berlin, Jacob Katzenelson, William McAllister, Guillermo Rozas, Gerald Jay Sussman, and Jack Wisdom. 1992. The Supercomputer Toolkit: A general framework for special-purpose computing. *International Journal of High-Speed Electronics* 3(3): 337-361. →

**Allen**, John. 1978. *Anatomy of Lisp*. New York: McGraw-Hill.

**ANSI X3.226-1994**. *American National Standard for Information Systems—Programming Language—Common Lisp*.

**Appel**, Andrew W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters* 25(4): 275-279. →

**Backus**, John. 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8): 613-641. →

**Baker**, Henry G., Jr. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21(4): 280-293. →

**Batali**, John, Neil Mayle, Howard Shrobe, Gerald Jay Sussman, and Daniel Weise. 1982. The Scheme-81 architecture—System and chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI*, edited by Paul Penfield, Jr. Dedham, MA: Artech House.

**Borning**, Alan. 1977. ThingLab—An object-oriented system for building simulations using constraints. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*. →

**Borodin**, Alan, and Ian Munro. 1975. *The Computational Complexity of Algebraic and Numeric Problems*. New York: American Elsevier.

**Chaitin**, Gregory J. 1975. Randomness and mathematical proof. *Scientific American* 232(5): 47-52. ->

**Church**, Alonzo. 1941. *The Calculi of Lambda-Conversion*. Princeton, N.J.: Princeton University Press.

**Clark**, Keith L. 1978. Negation as failure. In *Logic and Data Bases*. New York: Plenum Press, pp. 293-322. ->

**Clinger**, William. 1982. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 226-234.

**Clinger**, William, and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pp. 155-162. ->

**Colmerauer** A., H. Kanoui, R. Pasero, and P. Roussel. 1973. Un système de communication homme-machine en français. Technical report, Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy. ->

**Cormen**, Thomas, Charles Leiserson, and Ronald Rivest. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.

**Darlington**, John, Peter Henderson, and David Turner. 1982. *Functional Programming and Its Applications*. New York: Cambridge University Press.

**Dijkstra**, Edsger W. 1968a. The structure of the "THE" multiprogramming system. *Communications of the ACM* 11(5): 341-346. ->

**Dijkstra**, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43-112. ->

**Dinesman**, Howard P. 1968. *Superior Mathematical Puzzles*. New York: Simon and Schuster.

**deKleer**, Johan, Jon Doyle, Guy Steele, and Gerald J. Sussman. 1977. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pp. 116-125.

→

**Doyle**, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12: 231-272. →

**Feigenbaum**, Edward, and Howard Shrobe. 1993. The Japanese National Fifth Generation Project: Introduction, survey, and evaluation. In *Future Generation Computer Systems*, vol. 9, pp. 105-117. →

**Feeley**, Marc. 1986. Deux approches à l'implantation du language Scheme. Masters thesis, Université de Montréal. →

**Feeley**, Marc and Guy Lapalme. 1987. Using closures for code generation. *Journal of Computer Languages* 12(1): 47-66. →

**Feller**, William. 1957. *An Introduction to Probability Theory and Its Applications*, volume 1. New York: John Wiley & Sons.

**Fenichel**, R., and J. Yochelson. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11): 611-612. →

**Floyd**, Robert. 1967. Nondeterministic algorithms. *JACM*, 14(4): 636-644. →

**Forbus**, Kenneth D., and Johan deKleer. 1993. *Building Problem Solvers*. Cambridge, MA: MIT Press.

**Friedman**, Daniel P., and David S. Wise. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming: Third International Colloquium*, edited by S. Michaelson and R. Milner, pp. 257-284. →

**Friedman**, Daniel P., Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages*. Cambridge, MA: MIT Press/McGraw-Hill.

- Gabriel**, Richard P. 1988. The Why of Y. *Lisp Pointers* 2(2): 15-25. ->
- Goldberg**, Adele, and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley. ->
- Gordon**, Michael, Robin Milner, and Christopher Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, volume 78. New York: Springer-Verlag.
- Gray**, Jim, and Andreas Reuter. 1993. *Transaction Processing: Concepts and Models*. San Mateo, CA: Morgan-Kaufman.
- Green**, Cordell. 1969. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 219-240. ->
- Green**, Cordell, and Bertram Raphael. 1968. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the ACM National Conference*, pp. 169-181. ->
- Griss**, Martin L. 1981. Portable Standard Lisp, a brief overview. Utah Symbolic Computation Group Operating Note 58, University of Utah.
- Guttag**, John V. 1977. Abstract data types and the development of data structures. *Communications of the ACM* 20(6): 396-404. ->
- Hamming**, Richard W. 1980. *Coding and Information Theory*. Englewood Cliffs, N.J.: Prentice-Hall.
- Hanson**, Christopher P. 1990. Efficient stack allocation for tail-recursive languages. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pp. 106-118. ->
- Hanson**, Christopher P. 1991. A syntactic closures macro facility. *Lisp Pointers*, 4(3). ->
- Hardy**, Godfrey H. 1921. Srinivasa Ramanujan. *Proceedings of the London Mathematical Society* XIX(2).
- Hardy**, Godfrey H., and E. M. Wright. 1960. *An Introduction to the Theory of Numbers*. 4th edition. New York: Oxford University Press. ->



**Havender**, J. 1968. Avoiding deadlocks in multi-tasking systems. *IBM Systems Journal* 7(2): 74-84.

**Hearn**, Anthony C. 1969. Standard Lisp. Technical report AIM-90, Artificial Intelligence Project, Stanford University. ->

**Henderson**, Peter. 1980. *Functional Programming: Application and Implementation*. Englewood Cliffs, N.J.: Prentice-Hall.

**Henderson**, Peter. 1982. Functional Geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 179-187. ->, 2002 version ->

**Hewitt**, Carl E. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 295-301. ->

**Hewitt**, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3): 323-364. ->

**Hoare**, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1(1).

**Hodges**, Andrew. 1983. *Alan Turing: The Enigma*. New York: Simon and Schuster.

**Hofstadter**, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books.

**Hughes**, R. J. M. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, edited by David Turner. Reading, MA: Addison-Wesley, pp. 17-42. ->

IEEE Std 1178-1990. 1990. *IEEE Standard for the Scheme Programming Language*.

**Ingerman**, Peter, Edgar Irons, Kirk Sattley, and Wallace Feurzeig; assisted by M. Lind, Herbert Kanner, and Robert Floyd. 1960. THUNKS: A way of compiling procedure statements, with some comments on procedure declarations. Unpublished manuscript. (Also, private communication from Wallace Feurzeig.)

**Kaldewaij**, Anne. 1990. *Programming: The Derivation of Algorithms*. New York: Prentice-Hall.

**Knuth**, Donald E. 1973. *Fundamental Algorithms*. Volume 1 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley.

**Knuth**, Donald E. 1981. *Seminumerical Algorithms*. Volume 2 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley.

**Kohlbecker**, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University. →

**Konopasek**, Milos, and Sundaresan Jayaraman. 1984. *The TK!Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Berkeley, CA: Osborne/McGraw-Hill.

**Kowalski**, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh. →

**Kowalski**, Robert. 1979. *Logic for Problem Solving*. New York: North-Holland. →

**Lamport**, Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7): 558-565. →

**Lampson**, Butler, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek. 1981. Report on the programming language Euclid. Technical report, Computer Systems Research Group, University of Toronto. →

**Landin**, Peter. 1965. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2): 89-101.

**Lieberman**, Henry, and Carl E. Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6): 419-429. →

**Liskov**, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* 1(1): 7-19. →

**McAllester**, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory. →

**McAllester**, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory. →

**McCarthy**, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4): 184-195. →

**McCarthy**, John. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland. →

**McCarthy**, John. 1978. The history of Lisp. In *Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages*. →

**McCarthy**, John, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. 1965. *Lisp 1.5 Programmer's Manual*. 2nd edition. Cambridge, MA: MIT Press. →

**McDermott**, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259, MIT Artificial Intelligence Laboratory. →

**Miller**, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3): 300-317. →

**Miller**, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory. →

**Moon**, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science. →

**Moon**, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory. →

**Morris**, J. H., Eric Schmidt, and Philip Wadler. 1980. Experience with an applicative string processing language. In *Proceedings of the 7th Annual ACM SIGACT/SIGPLAN Symposium on the Principles of Programming Languages*.

**Phillips**, Hubert. 1934. *The Sphinx Problem Book*. London: Faber and Faber.

**Pitman**, Kent. 1983. The revised MacLisp Manual (Saturday evening edition). Technical report 295, MIT Laboratory for Computer Science. →

**Rabin**, Michael O. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12: 128-138.

**Raymond**, Eric. 1993. *The New Hacker's Dictionary*. 2nd edition. Cambridge, MA: MIT Press. →

**Raynal**, Michel. 1986. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press.

**Rees**, Jonathan A., and Norman I. Adams IV. 1982. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114-122. →

**Rees**, Jonathan, and William Clinger (eds). 1991. The revised<sup>4</sup> report on the algorithmic language Scheme. *Lisp Pointers*, 4(3). →

**Rivest**, Ronald, Adi Shamir, and Leonard Adleman. 1977. A method for obtaining digital signatures and public-key cryptosystems. Technical memo LCS/TM82, MIT Laboratory for Computer Science. →

**Robinson**, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1): 23.

**Robinson**, J. A. 1983. Logic programming—Past, present, and future. *New Generation Computing* 1: 107-124.

**Spafford**, Eugene H. 1989. The Internet Worm: Crisis and aftermath. *Communications of the ACM* 32(6): 678-688. →

**Steele**, Guy Lewis, Jr. 1977. Debunking the “expensive procedure call” myth. In *Proceedings of the National Conference of the ACM*, pp. 153-62. →

**Steele**, Guy Lewis, Jr. 1982. An overview of Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 98-107.

**Steele**, Guy Lewis, Jr. 1990. *Common Lisp: The Language*. 2nd edition. Digital Press. ->

**Steele**, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory. ->

**Steele**, Guy Lewis, Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow. 1983. *The Hacker's Dictionary*. New York: Harper & Row. ->

**Stoy**, Joseph E. 1977. *Denotational Semantics*. Cambridge, MA: MIT Press.

**Sussman**, Gerald Jay, and Richard M. Stallman. 1975. Heuristic techniques in computer-aided circuit analysis. *IEEE Transactions on Circuits and Systems* CAS-22(11): 857-865. ->

**Sussman**, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints—A language for expressing almost-hierarchical descriptions. *AI Journal* 14: 1-39. ->

**Sussman**, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257: 256-262. ->

**Sussman**, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory. ->

**Sutherland**, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory. ->

**Teitelman**, Warren. 1974. Interlisp reference manual. Technical report, Xerox Palo Alto Research Center. ->

**Thatcher**, James W., Eric G. Wagner, and Jesse B. Wright. 1978. Data type specification: Parameterization and the power of specification techniques. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 119-132.

**Turner**, David. 1981. The future of applicative languages. In *Proceedings of the 3rd European Conference on Informatics*, Lecture Notes in Computer Science, volume 123. New York: Springer-Verlag, pp. 334-348.

**Wand**, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27(1): 164-180. ->

**Waters**, Richard C. 1979. A method for analyzing loop programs. *IEEE Transactions on Software Engineering* 5(3): 237-247.

**Winograd**, Terry. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report AI TR-17, MIT Artificial Intelligence Laboratory. ->

**Winston**, Patrick. 1992. *Artificial Intelligence*. 3rd edition. Reading, MA: Addison-Wesley.

**Zabih**, Ramin, David McAllester, and David Chapman. 1987. Non-deterministic Lisp with dependency-directed backtracking. *AAAI-87*, pp. 59-64. ->

**Zippel**, Richard. 1979. Probabilistic algorithms for sparse polynomials. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT.

**Zippel**, Richard. 1993. *Effective Polynomial Computation*. Boston, MA: Kluwer Academic Publishers.

# List of Exercises

## Chapter 1

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
1.11	1.12	1.13	1.14	1.15	1.16	1.17	1.18	1.19	1.20
1.21	1.22	1.23	1.24	1.25	1.26	1.27	1.28	1.29	1.30
1.31	1.32	1.33	1.34	1.35	1.36	1.37	1.38	1.39	1.40
1.41	1.42	1.43	1.44	1.45	1.46				

## Chapter 2

2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10
2.11	2.12	2.13	2.14	2.15	2.16	2.17	2.18	2.19	2.20
2.21	2.22	2.23	2.24	2.25	2.26	2.27	2.28	2.29	2.30
2.31	2.32	2.33	2.34	2.35	2.36	2.37	2.38	2.39	2.40
2.41	2.42	2.43	2.44	2.45	2.46	2.47	2.48	2.49	2.50
2.51	2.52	2.53	2.54	2.55	2.56	2.57	2.58	2.59	2.60
2.61	2.62	2.63	2.64	2.65	2.66	2.67	2.68	2.69	2.70
2.71	2.72	2.73	2.74	2.75	2.76	2.77	2.78	2.79	2.80
2.81	2.82	2.83	2.84	2.85	2.86	2.87	2.88	2.89	2.90
2.91	2.92	2.93	2.94	2.95	2.96	2.97			

### Chapter 3

3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
3.11	3.12	3.13	3.14	3.15	3.16	3.17	3.18	3.19	3.20
3.21	3.22	3.23	3.24	3.25	3.26	3.27	3.28	3.29	3.30
3.31	3.32	3.33	3.34	3.35	3.36	3.37	3.38	3.39	3.40
3.41	3.42	3.43	3.44	3.45	3.46	3.47	3.48	3.49	3.50
3.51	3.52	3.53	3.54	3.55	3.56	3.57	3.58	3.59	3.60
3.61	3.62	3.63	3.64	3.65	3.66	3.67	3.68	3.69	3.70
3.71	3.72	3.73	3.74	3.75	3.76	3.77	3.78	3.79	3.80
3.81	3.82								

### Chapter 4

4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10
4.11	4.12	4.13	4.14	4.15	4.16	4.17	4.18	4.19	4.20
4.21	4.22	4.23	4.24	4.25	4.26	4.27	4.28	4.29	4.30
4.31	4.32	4.33	4.34	4.35	4.36	4.37	4.38	4.39	4.40
4.41	4.42	4.43	4.44	4.45	4.46	4.47	4.48	4.49	4.50
4.51	4.52	4.53	4.54	4.55	4.56	4.57	4.58	4.59	4.60
4.61	4.62	4.63	4.64	4.65	4.66	4.67	4.68	4.69	4.70
4.71	4.72	4.73	4.74	4.75	4.76	4.77	4.78	4.79	

### Chapter 5

5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10
5.11	5.12	5.13	5.14	5.15	5.16	5.17	5.18	5.19	5.20
5.21	5.22	5.23	5.24	5.25	5.26	5.27	5.28	5.29	5.30
5.31	5.32	5.33	5.34	5.35	5.36	5.37	5.38	5.39	5.40
5.41	5.42	5.43	5.44	5.45	5.46	5.47	5.48	5.49	5.50
5.51	5.52								



# List of Figures

## Chapter 1

1.1 1.2 1.3 1.4 1.5

## Chapter 2

2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 2.10  
2.11 2.12 2.13 2.14 2.15 2.16 2.17 2.18 2.19 2.20  
2.21 2.22 2.23 2.24 2.25 2.26

## Chapter 3

3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10  
3.11 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 3.20  
3.21 3.22 3.23 3.24 3.25 3.26 3.27 3.28 3.29 3.30  
3.31 3.32 3.33 3.34 3.35 3.36 3.37 3.38

## Chapter 4

4.1 4.2 4.3 4.4 4.5 4.6

## Chapter 5

5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10
5.11	5.12	5.13	5.14	5.15	5.16	5.17	5.18		

# Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

—Donald E. Knuth, *Fundamental Algorithms*  
(Volume 1 of *The Art of Computer Programming*)

## Symbols

$k$ -term finite continued fraction . . . 95  
 $n$ -fold smoothed function . . . . . 104

## A

abstract models . . . . . 123  
abstract syntax . . . . . 495  
abstraction barriers . . . . . 111, 119  
accumulator . . . . . 156, 303  
acquired . . . . . 421  
action . . . . . 677  
additive . . . . . 243  
additively . . . . . 112, 230  
address . . . . . 724  
address arithmetic . . . . . 724  
agenda . . . . . 379  
algebraic specification . . . . . 123  
aliasing . . . . . 316

and-gate . . . . . 370  
applicative-order . . . . . 542  
applicative-order evaluation . . . . . 21  
arbiter . . . . . 424  
arguments . . . . . 8  
assembler . . . . . 698  
assertions . . . . . 600  
assignment operator . . . . . 297  
atomically . . . . . 423  
automatic storage allocation . . . . . 723  
average damping . . . . . 94

## B

B-trees . . . . . 213  
backbone . . . . . 361  
backquote . . . . . 779  
backtracks . . . . . 564  
balanced . . . . . 151

barrier synchronization	427
base address	725
Bertrand's hypothesis	447
bignum	726
bindings	321
binds	36
binomial coefficients	54
block structure	39
bound variable	36
box-and-pointer notation	132
breakpoint	722
broken heart	735
bugs	2

## C

cache-coherence	406
call-by-name	439, 545
call-by-name thunks	439
call-by-need	439, 545
call-by-need thunks	439
capturing	37
Carmichael numbers	69
case analysis	22
cell	422
chronological backtracking	564
Church numerals	126
Church-Turing thesis	524
clauses	23
closed world assumption	632
closure	111
closure property	133
code generator	773
coerce	270
coercion	263
combinations	8
comments	168

compacting	734
compilation	768
compile-time environment	819
composition	103
compound data	108
compound data object	109
compound procedure	16
computability	523
computational process	1
concurrently	402
congruent modulo	67
connectors	387
consequent expression	23
constraint networks	387
constructors	113
continuation procedures	579
continued fraction	94
control structure	628
controller	668
conventional interfaces	111, 154
current time	383

## D

data	1, 122
data abstraction	109, 112
data paths	668
data-directed	231
data-directed programming	112, 243
deadlock	425
deadlock-recovery	426
debug	2
deep binding	517
deferred operations	44
delayed argument	472
delayed evaluation	296, 430
delayed object	434

dense .....	281
dependency-directed backtracking	565
depth-first search .....	564
deque .....	360
derived expressions .....	507
digital signals .....	370
dispatching on type .....	242
displacement number .....	818
dotted-tail notation .....	141
driver loop .....	520

## E

empty list .....	137
encapsulated .....	300
enclosing environment .....	321
entry points .....	673
enumerator .....	156
environment .....	11
environment model .....	295
environments .....	321
Euclid's Algorithm .....	63
Euclidean ring .....	288
evaluating .....	7
evaluator .....	489
event-driven simulation .....	369
evlis tail recursion .....	748
execution procedure .....	535
explicit-control evaluator .....	741
expression .....	7

## F

failure continuation .....	579
FIFO .....	354
filter .....	82, 156
first-class .....	102
fixed point .....	92

fixed-length .....	219
forcing .....	545
forwarding address .....	735
frame .....	616
frame coordinate map .....	182
frame number .....	818
framed-stack .....	746
frames .....	321
free .....	37
free list .....	729
front .....	353
full-adder .....	372
function boxes .....	370
functional programming .....	312
functional programming languages	484

## G

garbage .....	732
garbage collection .....	724, 732
garbage collector .....	342
garbage-collected .....	550
generic operations .....	112
generic procedures .....	224, 231
glitches .....	2
global .....	41, 321
global environment .....	11
golden ratio .....	49
grammar .....	572

## H

half-adder .....	370
half-interval method .....	89
Halting Theorem .....	526
headed list .....	361
hiding principle .....	300
hierarchical .....	134

hierarchy of types .....	266
higher-order procedures .....	75
Horner's rule .....	162

## I

imperative programming .....	317
indeterminates .....	275
index .....	725
indexing .....	617
instantiated with .....	604
instruction counting .....	721
instruction execution procedure ..	701
instruction sequence .....	775
instruction tracing .....	721
instructions .....	667, 673
integerizing factor .....	290
integers .....	7
integrator .....	465
interning .....	728
interpreter .....	3, 489
invariant quantity .....	60
inverter .....	370
iterative improvement .....	105
iterative process .....	44

## K

key .....	217
-----------	-----

## L

labels .....	673
lazy evaluation .....	542
lexical address .....	818
lexical addressing .....	517
lexical scoping .....	39
linear iterative process .....	44
linear recursive process .....	44

linkage descriptor .....	774
list .....	135, 141
list structure .....	135
list-structured .....	116
list-structured memory .....	723
local evolution .....	40
local state variables .....	297
location .....	724
logic-programming .....	491
logical and .....	370
logical deductions .....	612
logical or .....	370

## M

machine language .....	768
macro .....	507
map .....	156
mark-sweep .....	733
Memoization .....	368
memoization .....	53
memoize .....	545
merge .....	485
message passing .....	125, 253
message-passing .....	303
metacircular .....	492
Metalinguistic abstraction .....	489
Miller-Rabin test .....	73
modular .....	295
modulo .....	67
modus ponens .....	627
moments in time .....	402
Monte Carlo integration .....	309
Monte Carlo simulation .....	306
mutable data objects .....	341
mutators .....	341
mutex .....	421

mutual exclusion . . . . . 421

## N

native language . . . . . 768  
needed . . . . . 777  
networks . . . . . 488  
Newton's method . . . . . 98  
nil . . . . . 137  
non-computable . . . . . 526  
non-strict . . . . . 543  
nondeterministic . . . . . 409  
nondeterministic choice point . . . 563  
nondeterministic computing . 491, 559  
normal-order . . . . . 542  
normal-order evaluation . . . . 21, 491

## O

obarray . . . . . 727  
object program . . . . . 768  
objects . . . . . 295  
open-code . . . . . 815  
operands . . . . . 8  
operator . . . . . 8, 533  
or-gate . . . . . 370  
order of growth . . . . . 54  
ordinary . . . . . 256  
output prompt . . . . . 520

## P

package . . . . . 245  
painter . . . . . 173  
pair . . . . . 115, 116  
parse . . . . . 571  
Pascal's triangle . . . . . 53  
pattern . . . . . 602  
pattern matcher . . . . . 616

pattern matching . . . . . 616  
pattern variable . . . . . 602  
pipelining . . . . . 402  
pointer . . . . . 132  
poly . . . . . 276  
power series . . . . . 450  
predicate . . . . . 23  
prefix . . . . . 219  
prefix code . . . . . 219  
prefix notation . . . . . 8  
pretty-printing . . . . . 9  
primitive constraints . . . . . 387  
probabilistic algorithms . . . . . 70  
procedural abstraction . . . . . 35  
procedural epistemology . . . . . xxiii  
procedure . . . . . 45  
procedure definitions . . . . . 15  
procedures . . . . . 5  
process . . . . . 45  
program . . . . . 2  
programming languages . . . . . 2  
prompt . . . . . 520  
pseudo-random . . . . . 306  
pseudodivision . . . . . 290  
pseudoremainder . . . . . 290

## Q

quasiquote . . . . . 779  
queries . . . . . 598  
query language . . . . . 598  
queue . . . . . 353  
quote . . . . . 193

## R

Ramanujan numbers . . . . . 464  
rational functions . . . . . 287

RC circuit	466
read-eval-print loop	10
reader macro characters	657
real numbers	7
rear	353
recursion equations	3
Recursion theory	524
recursive	12, 33
recursive process	44
red-black trees	213
referentially transparent	315
register machine	667
register table	700
registers	667
released	421
remainder of	67
resolution principle	596
ripple-carry adder	376
robust	191
RSA algorithm	70
rules	599, 608

## S

satisfy	604
scope	37
selectors	113
semaphore	421
separator code	219
sequence	134
sequence accelerator	455
sequences	81
serializer	411
serializers	412
series RLC circuit	475
shadow	321
shared	348

side-effect bugs	317
sieve of Eratosthenes	443
smoothing	104
source language	768
source program	768
sparse	281
special forms	14
stack	45, 689
state variables	44, 296
statements	777
stop-and-copy	733
stratified design	190
stream processing	22
streams	295, 429
strict	543
subroutine	683
substitution	20
substitution model	19
subtype	266
success continuation	579
summation of a series	77
summer	465
supertype	267
symbolic expressions	111
syntactic sugar	15
syntax	494
systematically search	564
systems	488

## T

tableau	456
tabulation	53, 368
tagged architectures	726
tail-recursive	46, 754
target	774
thrashing	ix



thunk .....	545
thunks .....	545
time .....	401
time segments .....	382
tower .....	267
tree accumulation .....	13
tree recursion .....	47
trees .....	147
truth maintenance .....	565
Turing machine .....	523
type field .....	726
type tag .....	237
type tags .....	231
type-inferencing .....	478
typed pointers .....	725

## U

unbound .....	321
unification .....	596, 616, 622
unification algorithm .....	596

univariate polynomials .....	275
universal machine .....	523
upward-compatible extension ....	554

## V

value .....	10
value of a variable .....	321
values .....	193
variable .....	10
variable-length .....	219
vector .....	724

## W

width .....	128
wires .....	370
wishful thinking .....	114

## Z

zero crossings .....	467
----------------------	-----

## Colophon

ON THE COVER PAGE is Agostino Ramelli's bookwheel mechanism from 1588. It could be seen as an early hypertext navigation aid. This image of the engraving is hosted by J. E. Johnson of [New Gottland](#).

The typefaces are Linux Libertine for body text and Linux Biolinum for headings, both by Philipp H. Poll. Typewriter face is Inconsolata created by Raph Levien and supplemented by Dimosthenis Kaponis and Takashi Tanigawa in the form of Inconsolata LGC. The cover page typeface is Alegreya, designed by Juan Pablo del Peral.

Graphic design and typography are done by Andres Raba. Texinfo source is converted to LaTeX by a Perl script and compiled to PDF by XeLaTeX. Diagrams are drawn with Inkscape.