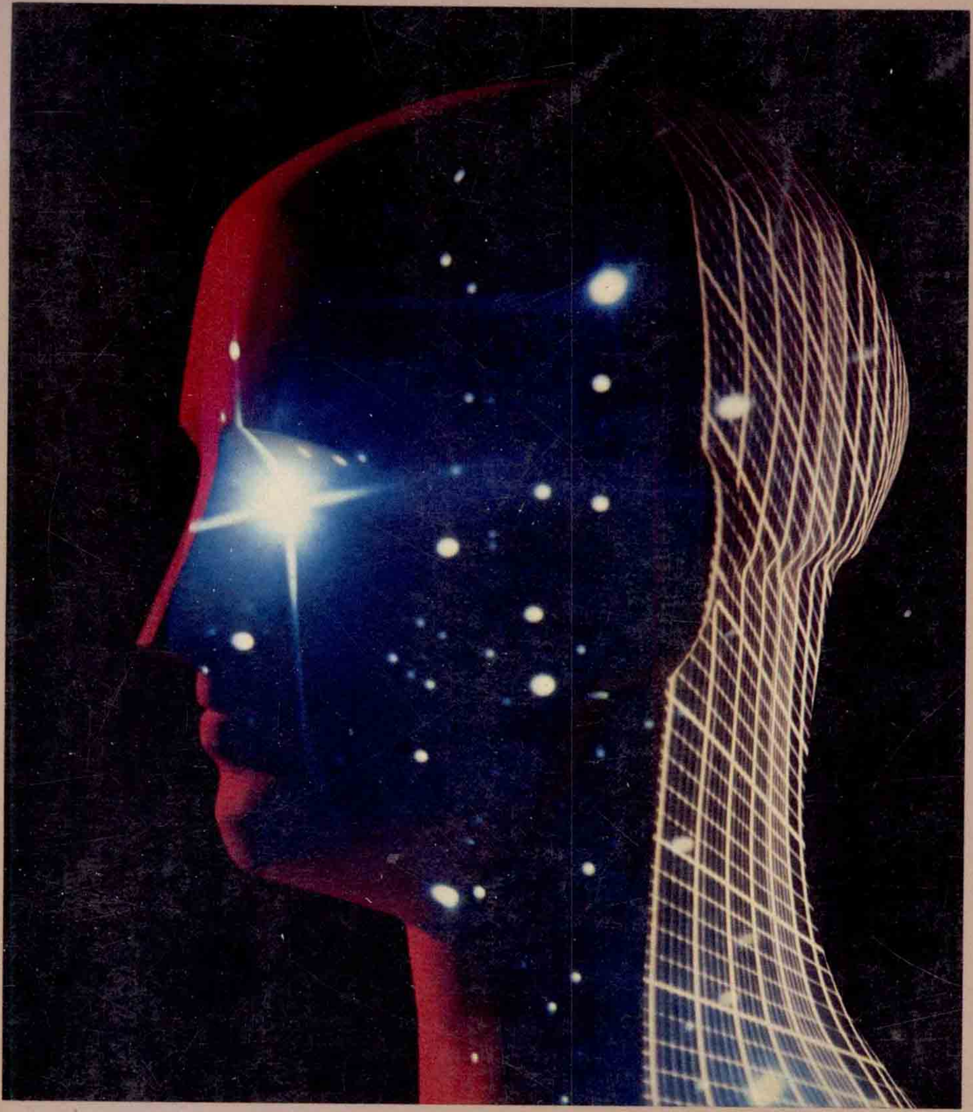
Computing with Logic

LOGIC PROGRAMMING WITH PROLOG



David Maier/David S. Warren

COMPUTING WITH LOGIC

Logic Programming with Prolog

David Maier

Oregon Graduate Center

David S. Warren

State University of New York at Stony Brook

The Benjamin/Cummings Publishing Company, Inc.

Menlo Park, California • Reading, Massachusetts
Don Mills, Ontario • Wokingham, U.K. • Amsterdam
Sydney • Singapore • Tokyo • Madrid • Bogota
Santiago • San Juan

Sponsoring Editor: Alan Apt
Production Editor: Laura Kenney
Copyeditor: Jonas Weisel
Technical Art: Graphic Typesetting Service
Cover Designer: Gary Head
Cover Photo: A. Gescheidt, © The Image Bank
Compositor: Graphic Typesetting Service

The basic text of this book was designed using the Modular Design System, as developed by Wendy Earl and Design Office Bruce Kortebein.

Library of Congress Cataloging-in-Publication Data

Maier, David, 1953–
Computing with logic.

Includes bibliographies and index.
1. Prolog (Computer program language) 2. Logic programming. I. Warren, David S. II. Title.
QA76.73.P76M349 1988 005.13'3 87–23856
ISBN 0-8053-6681-4

BCDEFGHIJ-DO-898

The Benjamin/Cummings Publishing Company, Inc.
2727 Sand Hill Road
Menlo Park, CA 94025

Preface

This text is appropriate for a senior or first-year graduate course on logic programming. It concentrates on the formal semantics of logic programs, automatic theorem-proving techniques, and efficient implementation of logic languages. It is also an excellent reference for the computer professional wishing to do self-study on the fundamentals of logic programming. We have included numerous examples to illustrate all the major concepts and results. No other text deals with implementation in as much detail. Other discussions of implementation techniques do not explain and develop the relationship between interpreter (or compiler) behavior and resolution theorem proving. We stress that connection throughout.

Logic Programming

A course just on logic programming is a reasonable endeavor for several reasons. First, logic programs offer a different way of thinking about problem solving. They have both a declarative and a procedural meaning, so that a person can think about the correctness of a program apart from its operational behavior. Second, logic programming languages have a "stronger" and more natural formal semantics than most other programming languages. They are, therefore, a good vehicle for studying language semantics and meaning-preserving program transformations. Third, logic programming languages are very-high-level languages. They are almost specification languages (languages for specifying what problem is to be solved apart from the means of solution). But for logic programming languages, these specifications can be executed directly. Fourth, detailed knowledge of how these languages are implemented will allow a programmer to program more efficiently and effectively in them. Fifth, the implementation techniques and strategies can be applied to improving the efficiency of other high-level and run-time-intensive languages.

Logic languages, particularly Prolog and its concurrent and parallel variants, have gotten much publicity in connection with their use in proposed "fifth generation" systems, which rely heavily on artificial intelligence and knowledge representation techniques. There are many reasons for the interest. For one, the structure of Prolog resembles that of rule-based expert systems and knowledge bases (databases with inferential components). Rule-based systems typically employ more varied inference strategies than Prolog, but Prolog provides a foundation for understanding the semantics of other kinds of rules and is itself a good language for writing rule processors with various search strategies. Prolog is well suited for symbolic manipulation and information representation tasks. Construction and extraction of data structures are the principal mechanisms for computation in Prolog, and often the same code can be used to do both operations. Also, code can easily be treated as data in Prolog, through the use of "metaprogramming" features, making it a natural choice for writing tools that generate or modify other programs. The integration of data and procedure in Prolog is almost seamless. Whether a particular relationship is represented as a table or a function, or some mixture of the two, is largely irrelevant to other parts of a program that use the relationship. This transparency gives great flexibility in deciding how to represent a particular chunk of knowledge in Prolog. Finally, the declarative semantics of "pure Prolog" make it amenable to implementation on parallel machines, since the meaning of a program is not bound up with a particular model of computation.

The only background absolutely necessary for this book is a first course in data structures. A previous course in compilers is useful for understanding the parsing and symbol table issues. Also, a course with abstract mathematical content that involves theorems (such as finite structures, automata theory, or abstract algebra) is helpful for the more formal material, but we have tried to be self-contained for the topics in mathematical logic.

How to Use This Book

There is more material here than will comfortably fit in a semester. The material on model elimination and on the connection of Datalog to relational algebra can be skipped with no effect on continuity. If Prolog programming is covered in another course, Chapters 8 and 12 are not essential. Also, the chapters on formal logic are fairly independent of the chapters on implementation. A course on formal foundations of logic programming could omit Chapters 3, 6, 10, and 11. A course stressing implementation techniques could leave out portions of Chapters 2, 5, and 9 after the sections on Proplog, Datalog, and Prolog semantics, respectively.

When either of us has taught the course, we have included a programming component—sometimes a course project, sometimes a number of shorter assignments. The appendix presents two ways the course might be structured to include a course project involving Prolog. Shorter programming projects require a bit more planning to integrate, since the book does not get to term structures until fairly far along in the text. One possibility is to introduce lists earlier to permit more interesting Prolog programming assignments sooner. Students in the course will need a companion text, a Prolog programming primer, if there is any substantial programming component. Ours is *not* a text for Prolog programming techniques, or Prolog for certain applications, although those might be reasonable topics for a follow-up course. (We are, however, faithful to DEC-10 Prolog and C-Prolog syntax, and all of the longer examples have actually been run.) We have tried to make the programming examples realistic. Some of the programs (such as the course requirements and fabric examples from Chapter 1 and the poker example from Chapter 7) have to be fairly large to capture a reasonable slice of the real world accurately. Such examples are too unwieldy to use in the classroom in their entirety; instructors should select subsets or alternate examples for presentation.

The material in this book is more important than Prolog programming techniques for a serious computer science student. Most of the computer science in logic programming is not in Prolog applications and programming. However, the material presented here will help a student use Prolog to its best capabilities. Programming in Prolog without knowing its logical foundations means a student probably does not grasp the true nature of the language. Prolog is an impure language—if a student does not recognize the underlying ideal, he or she cannot separate good techniques from bad techniques. A student familiar with the material in this book will be better able to give Prolog programs, or fragments of them, a declarative reading and will be better able to write programs with a declarative meaning. Such programs are not only easier to understand and debug, they almost always are executed more efficiently. A logic programmer has to direct a theorem prover to provide control over the deduction process in order to get reasonable performance. He or she must understand the deduction process to control it intelligently.

The reader may question why we use Prolog and Prolog subsets as the only examples of logic programming languages when we are trying to give a general introduction to the field. First, Prolog is currently the only *real* logic programming language, and the only one in widespread use. Second, it is the logic language for which the most advanced implementations exist. Compilers exist for few other logic programming languages. Third, the choice of language does not much affect the sections on mathematical logic. The material covered in those sections is applicable to any logic programming language. Fourth, Prolog's deduction process is simple and straightforward (top-down, left-to-right) and is hence easier to control. We will take up other logic programming languages in a planned companion volume.

This book is based on notes for a first-year graduate course offered at SUNY Stony Brook and Oregon Graduate Center. Those notes were based, in turn, on a compiler course at Stony Brook in which students implemented a Prolog compiler, and a graduate seminar at OGC in logic programming.

Acknowledgments

Thank you all—particularly our wives, Kaye Van Valkenburg and Paula Pelletier. Also our technical reviewers: John Conery, Bruce Porter, Ray Reiter, and Sharon Salveter (who tested the text in the classroom). Alan Apt, our editor, encouraged the project early on and kept us motivated through the long haul. (You can release the children now, Alan.) Harry Porter gave the whole manuscript a thorough reading, and we avoided incorporating his suggestions only when we were too lazy (or he was too pedantic). Allen Van Gelder showed us how to simplify our arguments on lifting proof trees, and Jean-Louis Lassez helped us see how most general unifiers really work in logic languages. Dick Kieburtz lent us his cottage on the Oregon coast as a comfortable place to write. Quintus helped support the second author during the final throes and gave the use of their impressive Prolog system to produce the example in Chapter 12. Kathy Hammerstrom helped route the right versions of the manuscript to the right places and prepared the table of contents. Laura Kenney patiently led us through the process of turning our manuscript into a book. Thanks to all the students in our courses who helped debug the manuscript, especially Doug Pase (a champion entomologist), Mark Grossman, Phil Miller, Gary Beaver, Bart Schaefer, Deborah Bouchette, Siroos Farahmand-nia, Todd Brunhoff, Phil White, Martin Zimmerman, Brad Needham, Jerry Peak, Hassan Hosseini, Becky Lakey, Mike Rudnick, Larry Morandi, Wroff Courtney, Paul Bober, and Malcolm Printer.

D. M.
D. S. W.

Introduction

A critical property of a programming language is its level of abstraction. We want to program in a more declarative style—saying what a program should compute, rather than how to compute it. In logic programming we define properties and relationships for the objects of interest, and the system determines how to compute with those objects. In this paradigm programming becomes setting up constraints with knowns and unknowns. The system solves for the unknowns, analogously to solving linear equations or to a spreadsheet filling in values of empty cells. There are some key differences, though, between the latter examples and the kind of constraints solving in logic programming.

1. In logic programming the constraints involve data structures and variables representing data structures, rather than algebraic equations and variables representing numbers.
2. There is usually some degree of nondeterminism involved in solving logic programming constraints. We expect constraint solving to yield sets of answers in general. In logic programming we typically have more than one rule for resolving a constraint into a set of subconstraints. Often we want the answers for all the possible ways of resolving the constraint. In a spreadsheet we expect a solution to the constraints to yield a unique value for each unknown cell.
3. There are multiple strategies for solving the constraints defined by logic programs, and many different ways have been proposed for evaluating logic programs, including parallel, data flow, and intelligent-control strategies. This choice of implementation techniques is evidence that we have abstracted away more of the machine details in logic programming languages than in more conventional programming languages.

Why is a declarative style in a programming language better than the traditional procedural style? Such a style encourages the programmer to think about the *intent* of a program, about the *static* description of relationships and properties that are to hold in a program, without having to worry about dynamic changes in the store of a computer. It de-emphasizes the role of time in understanding programs and allows parts of a program to be examined and understood in isolation. Programs in a declarative language are easier to modify because we can add further constraints without having to worry about the timing of checking those constraints. Adding a constraint does not invalidate other constraints already present, so statements are not context dependent. In a procedural language the effect of a statement depends on the statements executed before it, and the statement changes the context for all statements that follow it. For example, encountering the statement X := X + 1 means the value associated with variable X is different for statements that go before and those that come after. The semantics of a statement in a procedural language is quite complex, because that meaning is both dependent on a context and indicates a modification to the context.

Divorcing the meaning of a program from any particular computational model is important. The separation gives the freedom to pick alternative implementations of a program. The fact that there are a number of alternative implementations gives evidence that a language is at a high level of abstraction. For relational database query languages—a particularly simple sort of logic programming language—most query evaluators examine alternative execution strategies for a query and pick the one estimated to be the most time efficient. The split between meaning and the computation model means declarative languages are more amenable to optimization, because such languages can express the intent of a program apart from a particular implementation of that intent. Optimizers need not analyze programs to extract the intent from the implementation, as with say, a vectorizing compiler for FORTRAN. Of course, declarative languages have efficiency penalties. We cannot expect an evaluator always to find as efficient an execution strategy as a human programmer could.

The most widely used logic programming language is Prolog. It does not achieve all the ideals just set forth. We do need to think about control when writing a Prolog program, but at least we may ignore control for a first cut at understanding the program.

Consider what we need for an effective declarative language.

1. A clear statement of the semantics of the language, independent of operational considerations. For logic programming, the semantics is based on formal logic and model theory. For numerical equation solvers, the semantics comes from arithmetic and algebra.
2. A theory of meaning-preserving transformations on programs—that is, a deduction system. In logic programming we have rules and properties relating knowns and unknowns, and we apply transformations to them to solve for the unknowns. An example of such a transformation in the domain of algebra is that adding equals to both sides of an equality or inequality preserves the relationship.
3. Strategies for applying the transformations to yield a solution for the unknowns if it exists, and special forms for statements that support particular strategies. The special forms should make the strategy easy to express. An example is Gaussian elimination. A set of simultaneous linear equations is organized

into a matrix of coefficients. That organization allows certain solution-preserving transformations to be applied to the set of equations through scaling, row swaps, and substraction of rows. It is easy to express a sequence of these transformations that will solve the equations in terms of iterations through the matrix.

4. Suitable data structures and algorithms for implementing the strategy efficiently in a particular machine. In the Gaussian elimination example we have the choice of implementing the coefficient matrix in row-major or column-major order, or perhaps with some kind of linked structure or entry list if we expect a sparse matrix.

This book is organized into three parts, each of which covers the preceding points 1–4 for three successively more powerful logic languages. Part I is on Proplog, a logic language based on propositional logic. Part II is on Datalog, a language for predicate logic (without function symbols). Part III covers Prolog, a language based on functional logic (predicate logic including function symbols). In each part there are at least three chapters—call them A, B, and C.

Chapter A introduces the language via examples to give an intuitive semantics. It also presents a naive interpreter (constraint solver) based on the intuitive semantics.

Chapter B formalizes the semantics using the appropriate logic and models and then looks at the semantics of the full logic (since our languages are based on restricted subsets), deduction, a particular format for formulas (called clauses), and a deduction rule (called resolution) that works on clauses. The resolution rule is the basis for a procedure, known as refutation, for deciding the validity of a formula. We then look at a subclass of clauses, called Horn clauses, and specialize and refine the refutation procedure for that subclass. We show that the specialized procedure is the basis of the naive interpreter of Chapter A, demonstrating the correctness and completeness of the interpreter.

Chapter C takes a naive interpreter, known correct from Chapter B, and optimizes it using special data structures and the Procrastination Principle: put off until later work that might not have to be done, such as concatenating lists or copying data structures.

Part III, on Prolog, includes three additional chapters beyond A, B, and C. One chapter is on procedural extensions to the pure logic form of the language. The second chapter is on further optimization of the interpreter and compilation techniques, the latter being based on symbolic execution and in-line expansion of parts of the interpreter. The third chapter contains an extended example on implementing a database query language in Prolog. It illustrates areas in which Prolog has proved particularly apt: parsing, translation, code generation, and code optimizations.

Why did we choose to develop our topic and recapitulate it twice rather than doing it just once for Prolog? We use a sequence of three increasingly complex languages because we can expound certain concepts more clearly without all the complications of functional logic. We can introduce a few new ideas for each language. Often an argument or development from one language carries over to the next with little or no change. Also, for Prolog, the theorem-proving theory diverges quickly from the efficient interpreter. It is easier to point out the connections in the simpler languages.

Why did we pick the particular sublanguages of Prolog that we did? We chose Proplog because it corresponds to a natural subset of first-order logic, propositional logic, and the structures and models used there are finite and easy to reason about. We chose Datalog because of its connection with query languages for relational databases (which is where the "Data" part comes from). Relational query languages are essentially Datalog without recursion. Such languages are another major example of declarative languages. Datalog also permits us to examine the notion of a logical variable in its simplest form.

We have tried to be obvious rather than clever or succinct. There are no great mysteries to logic programming semantics or Prolog implementation. We present those topics so that any advanced computer science student or practitioner can master them.

Contents