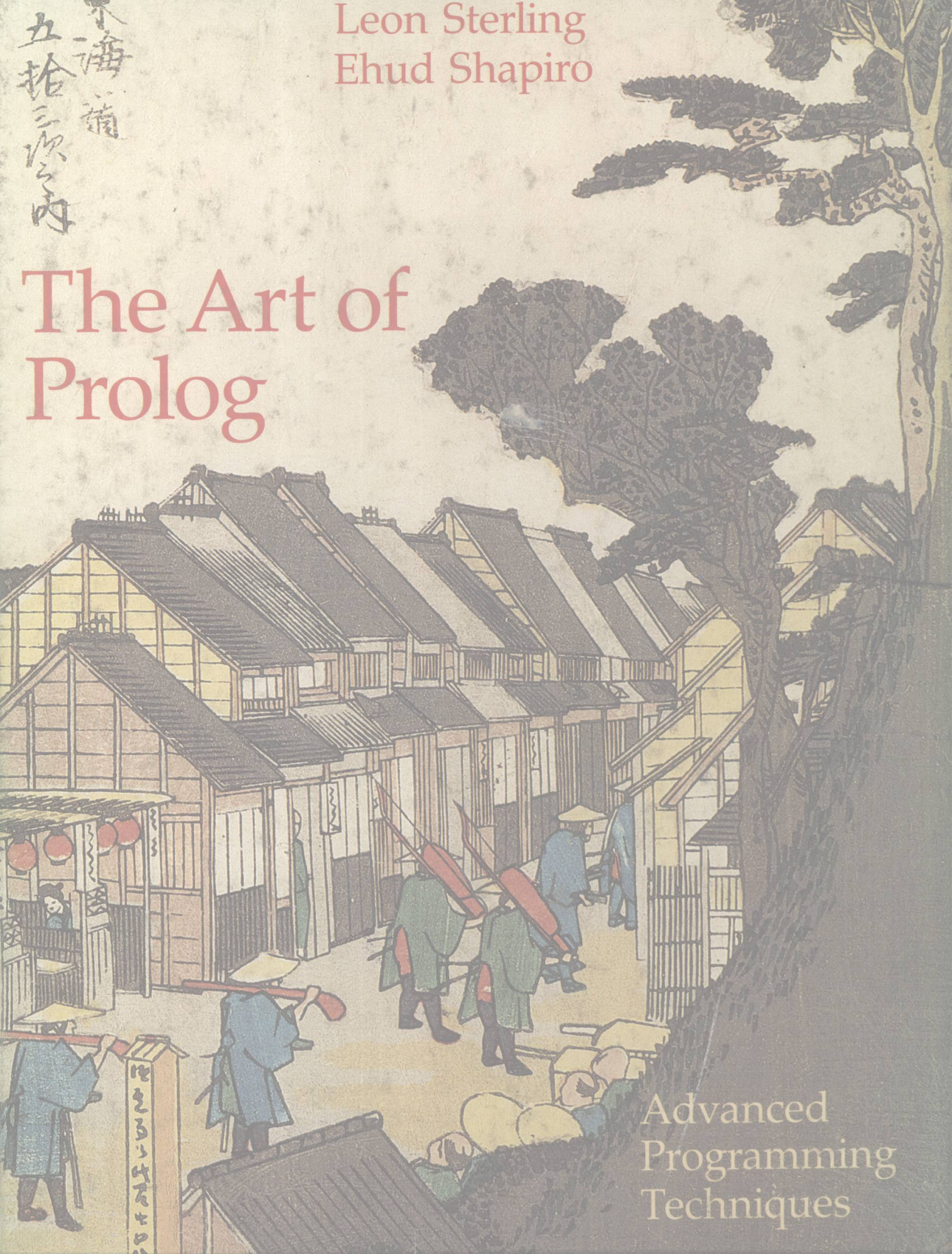
Leon Sterling
Ehud Shapiro

The Art of Prolog



Advanced Programming Techniques

The Art of Prolog

Advanced Programming Techniques

Leon Sterling

Ehud Shapiro

The MIT Press
Cambridge, Massachusetts
London, England

PUBLISHER'S NOTE

This format is intended to reduce the cost of publishing certain works in book form and to shorten the gap between editorial preparation and final publication. Detailed editing and composition have been avoided by photographing the text of this book directly from the authors' prepared copy.

This book was set in TEX by Sarah Fliegelmann
at the Weizmann Institute of Science
and printed and bound by The MIT Press
in the United States of America

Library of Congress Cataloging-in-Publication Data

Sterling, Leon.
The art of Prolog.

(MIT Press series in logic programming)
Includes index.
1. Prolog (Computer program language) I. Shapiro, Ehud Y. II. Title. III. Series.
QA76.73.P76S74 1986 005.13'3 86-10529
ISBN 0-262-19250-0 (hard)
0-262-69105-1 (paper)

The Art of Prolog

To Ruth, Miriam, Michal, and Danya

Preface

The origins of this book lie in graduate student courses aimed at teaching advanced Prolog programming. There is a wealth of techniques that has emerged in the fifteen years since the inception of Prolog as a programming language. Our intention in this book has been to make accessible the programming techniques that kindled our own excitement, imagination and involvement in this area.

The book fills a general need. Prolog, and more generally logic programming, have received wide publicity in recent years. Currently available books and accounts, however, typically describe only the basics. All but the simplest examples of the use of Prolog have remained essentially inaccessible to people outside the Prolog community.

We emphasize throughout the book the distinction between logic programming and Prolog programming. Logic programs can be understood and studied, using two abstract, machine independent concepts: truth and logical deduction. One can ask whether an axiom in a program is true, under some interpretation of the program symbols; or whether a logical statement is a consequence of the program. These questions can be answered independently of any concrete execution mechanism.

On the contrary, Prolog is a programming language, borrowing its basic constructs from logic. Prolog programs have precise operational meaning: they are instructions for execution on a computer — a Prolog machine. Prolog programs in good style can almost always be read as logical statements, thus inheriting some of the abstract properties of logic programs. Most important, the result of a computation of such a Prolog program is a logical consequence of the axioms in it. Effective Prolog programming requires an understanding of the theory of logic programming.

The book consists of four parts: logic programming, the Prolog language, advanced techniques, and applications. The first part is a self-contained introduction to logic programming. It consists of five chapters. The first chapter

introduces the basic constructs of logic programs. Our account differs from other introductions to logic programming by explaining the basics in terms of logical deduction. Other accounts explain the basics from the background of resolution from which logic programming originated. We have found the former to be a more effective means of teaching the material, which students find intuitive, and easy to understand.

The second and third chapters of Part I introduce the two basic styles of logic programming: database programming and recursive programming. The fourth chapter discusses the computational model of logic programming, introducing unification, while the fifth chapter presents some theoretical results without proofs. In developing this part to enable the clear explanation of advanced techniques, we have introduced new concepts, and reorganized others. In particular in the discussion of types and termination. Other issues such as complexity and correctness are concepts whose consequences have not yet been fully developed in the logic programming research community.

The second part is an introduction to Prolog. It consists of Chapters 6 through 13. Chapter 6 discusses the computational model of Prolog as opposed to logic programming, and gives a comparison between Prolog and conventional programming languages such as Pascal. Chapter 7 discusses the differences between composing Prolog programs and logic programs. Examples are given of basic programming techniques.

The next five chapters introduce system-provided predicates that are essential to make Prolog a practical programming language. We classify Prolog system predicates into four categories: those concerned with efficient arithmetic, structure inspection, meta-logical predicates that discuss the state of the computation, and extra-logical predicates that achieve side-effects outside the computational model of logic programming. One chapter is devoted to the most notorious of Prolog extra-logical predicates, the cut. Basic techniques using these system predicates are explained. The final chapter of the section gives assorted pragmatic programming tips.

The main part of the book is Part III. We describe advanced Prolog programming techniques that have evolved in the Prolog programming community, illustrating each with small yet powerful example programs. The examples typify the applications for which the technique is useful. The six chapters cover nondeterministic programming, incomplete data structures, parsing with DCGs, second-order programming, search techniques, and the use of meta-interpreters.

The final part consists of four chapters that show how the material in the rest of the book can be combined to build application programs. A common request of Prolog newcomers is to see larger applications. They understand how

to write elegant short programs but have difficulty in building a major program. The applications covered are game-playing programs, a prototype expert system for evaluating requests for credit, a symbolic equation solver and a compiler.

During the development of the book, it has been necessary to reorganize the foundations and basic examples existing in the folklore of the logic programming community. Our structure constitutes a novel framework for the teaching of Prolog.

Material from this book has been used successfully for several courses on logic programming and Prolog: in Israel, the United States and Scotland. The material more than suffices for a one semester course to first-year graduate students or advanced undergraduates. There is considerable scope for instructors to particularize a course to suit a special area of interest.

A recommended division of the book for a 13-week course to senior undergraduates or first-year graduates is as follows: 4 weeks on logic programming, encouraging students to develop a declarative style of writing programs, 4 weeks on basic Prolog programming, 3 weeks on advanced techniques, and 2 weeks spent on applications. The advanced techniques should include some discussion of nondeterminism, incomplete data structures, basic second-order predicates, and basic meta-interpreters. Other sections can be covered instead of applications. Application areas that can be stressed are search techniques in artificial intelligence, building expert systems, writing compilers and parsers, symbol manipulation, and natural language processing.

There is considerable flexibility in the order of presentation. The material from Part I should be covered first. The material in Part III and IV can be interspersed with the material in Part II to show the student how larger Prolog programs using more advanced techniques are composed in the same style as smaller examples.

Our assessment of students has usually been 50% by homework assignments throughout the course, and 50% by project. Our experience has been that students are capable of a significant programming task for their project. Examples of projects are prototype expert systems, assemblers, game-playing programs, partial evaluators, and implementations of graph theory algorithms.

For the student who is studying the material on her own, we strongly advise reading through the more abstract material in Part I. A good Prolog programming style develops from thinking declaratively about the logic of a situation. The theory in Chapter 5, however, can be skipped until a later reading.

The exercises in the book range from very easy and well-defined to difficult and open-ended. Most of them are suitable for homework exercises. Some of the

more open-ended exercises were submitted as course projects.

The code in this book is essentially in Edinburgh Prolog. The course has been given where students used several different variants of Edinburgh Prolog, and no problems were encountered. All the examples run on Wisdom Prolog, which is discussed in the appendixes.

We acknowledge and thank the people who contributed directly to the book. We also thank, collectively and anonymously, all those who indirectly contributed by influencing our programming styles in Prolog. Improvements were suggested by Lawrence Byrd, Oded Maler, Jack Minker, Richard O'Keefe, Fernando Pereira, and several anonymous referees.

We appreciate the contribution of the students who sat through courses as material from the book was being debugged. The first author acknowledges students at the University of Edinburgh, the Weizmann Institute of Science, Tel Aviv University, and Case Western Reserve University. The second author taught courses at the Weizmann Institute, Hebrew University of Jerusalem and other short courses to industry.

We are grateful to many people for assisting in the technical aspects of producing a book. We especially thank Sarah Fliegelmann who produced the various drafts and camera-ready copy, above and beyond the call of duty. This book may not have appeared without her tremendous efforts. Arvind Bansal prepared the index and helped with the references. Yehuda Barbut drew most of the figures. Max Goldberg and Shmuel Safra prepared the appendix. The publishers, MIT Press, were helpful and supportive.

Finally, we acknowledge the support of family and friends without which nothing would get done.

Introduction

The inception of logic is tied with that of scientific thinking. Logic provides a precise language for the explicit expression of one's goals, knowledge, and assumptions. Logic provides the foundation for deducing consequences from premises; for studying the truth or falsity of statements given the truth or falsity of other statements; for establishing the consistency of one's claims; and for verifying the validity of one's arguments.

Computers are relatively new in our intellectual history. Similar to logic, they are both the object of scientific study, and a powerful tool for the advancement of scientific endeavor in general. Like logic, computers require a precise and explicit statement of one's goals and assumptions. Unlike logic, which has developed with the power of the human thinking as the only external consideration, the development of computers has been governed from the start by severe technological and engineering constraints. Although computers were intended for use by humans, the difficulties in constructing them were so dominant, that the language for expressing problems to the computer and instructing it how to solve them was designed from the perspective of the engineering of the computer alone.

Almost all modern computers are based on the early concepts of von Neumann and his colleagues, which emerged during the 1940's. The von Neumann machine is characterized by a large uniform store of memory cells, and a processing unit with some local cells, called registers. The processing unit can load data from memory to registers, perform arithmetic or logical operations on registers, and store values of registers back into memory. A program for a von Neumann machine consists of a sequence of instructions to perform such operations, and an additional set of control instructions, which can affect the next instruction to be executed, possibly depending on the content of some register.

As the problems of building computers were gradually understood and solved, the problems of using them mounted. The bottleneck ceased to be the inability of the computer to perform the human's instructions, but rather the inability of the human to instruct, or program, the computer. A search for programming

languages convenient for humans to program in has begun. Starting from the language understood directly by the computer, the machine language, better notations and formalisms were developed. The main outcome of these efforts was languages that were easier for humans to express themselves in, but still mapped rather directly to the underlying machine language. Although increasingly abstract, the languages in the mainstream of development, starting from assembly language, through Fortran, Algol, Pascal, and Ada, all carried the mark of the underlying machine — the von Neumann architecture.

To the uninitiated intelligent person, who is not familiar with the engineering constraints that lead to its design, the von Neumann machine seems an arbitrary, even bizzare, device. Thinking in terms of its constrained set of operations is a non-trivial problem, which sometimes stretches the adaptiveness of the human mind to its limits.

These characteristic aspects of programming von Neumann computers have lead to a separation of work: there were those who thought how to solve the problem, and designed the methods for its solution, and there were the coders, who performed the mundane and tedious task of translating the instructions of the designers to instructions a computer can digest.

Both logic and programming require the explicit expression of one's knowledge and methods in an acceptable formalism. The task of making one's knowledge explicit is tedious. However, formalizing one's knowledge in logic is often an intellectually rewarding activity, and usually reflects back on or adds insight to the problem under consideration. In contrast, formalizing one's problem and method of solution using the von Neumann instruction set rarely has these beneficial effects.

We believe that programming can be, and should be, an intellectually rewarding activity; that a good programming language is a powerful conceptual tool — a tool for organizing, expressing, experimenting with, and even communicating one's thoughts; that treating programming as "coding," the last, mundane, intellectually trivial, but time-consuming and tedious phase of solving a problem using a computer system, is perhaps at the very roots of what has been known as the "software crisis."

Rather, we think that programming can be, and should be, part of the problem solving process itself; that thoughts should be organized as programs, so that consequences of a complex set of assumptions can be investigated by "running" the assumptions; that a conceptual solution to a problem should be developed hand-in-hand with a working program that demonstrates it and exposes its different aspects. Suggestions in this direction have been made under the title "rapid prototyping."

To achieve this goal in its fullest — to become true mates of the human thinking process — computers have still a long way to go. However, we find it both appropriate and gratifying from a historical perspective that logic, a companion to the human thinking process since the early days of human intellectual history, has been discovered as a suitable stepping-stone in this long journey.

Although logic has been used as a tool for designing computers, and for reasoning about computers and computer programs since almost their beginning, the use of logic directly as a programming language, termed *logic programming*, is quite recent.

Logic programming, as well as its sister approach, functional programming, departs radically from the mainstream of computer languages. Rather then being derived, by a series of abstractions and reorganizations, from the von Neumann machine model and instruction set, it is derived from an abstract model, which has no direct relationship or dependency to one machine model or another. It is based on the belief that instead of the human learning to think in terms of the operations of a computer, which some scientists and engineers at some point in history happened to find easy and cost-effective to build, the computer should perform instructions that are easy for humans to provide. In its ultimate and purest form, logic programming suggests that even explicit instructions for operation not be given but, rather, the knowledge about the problem and assumptions that are sufficient to solve it be stated explicitly, as logical axioms. Such a set of axioms constitutes an alternative to the conventional program. The program can be executed by providing it with a problem, formalized as a logical statement to be proved, called a goal statement. The execution is an attempt to solve the problem, that is, to prove the goal statement, given the assumptions in the logic program.

A distinguishing aspect of the logic used in logic programming is that a goal statement typically is existentially quantified: it states that there exist some individuals with some property. An example of a goal statement is that there exists a list X such that sorting the list $[3,1,2]$ gives X. The mechanism used to prove the goal statement is constructive: if successful, it provides the identity of the unknown individuals mentioned in the goal statement, which constitutes the output of the computation. In the example above, assuming that the logic program contains appropriate axioms defining the *sort* relation, the output of the computation would be $X=[1,2,3]$.

These ideas can be summarized in the following two metaphorical equations:

program = set of axioms

computation = constructive proof of a goal statement from the program

The ideas behind these equations can be traced back as far as intuitionistic mathematics and proof theory of the early century. They are related to Hilbert's program, to base the the entire body of mathematical knowledge on logical foundations, to provide mechanical proofs for its theories, starting from the axioms of logic and set theory alone. It is interesting to note that the fall of this program, which ensued the incompleteness and undecidability results of Gödel and Turing, also marks the beginning of the modern age of computers.

The first use of this approach in practical computing is a sequel to Robinson's unification algorithm and resolution principle, published in 1965. Several hesitant attempts were made to use this principle as a basis of for a computational mechanism, but they did not gain any momentum. The beginning of logic programming can be attributed to Kowalski and Colmerauer. Kowalski formulated the procedural interpretation of Horn clause logic. He showed that an axiom

A if B_1 and B_2 and ... and B_n

can be read, and executed, as a procedure of a recursive programming language, where A is the procedure head and the B_i's are its body. In addition to the declarative reading of the clause, A is true if the B_i's are true, it can be read as follows: to solve (execute) A, solve (execute) B_1 and B_2 and ... and B_n. In this reading, the proof procedure of Horn clause logic is the interpreter of the language, and the unification algorithm, which is at the heart of the resolution proof procedure, performs the basic data manipulation operations of variable assignment, parameter passing, data selection, and data construction.

At the same time, early 1970's, Colmerauer and his group at the University of Marseille-Aix developed a specialized theorem prover, written in Fortran, which they used to implement natural language processing systems. The theorem prover, called Prolog (for Programation et Logique), embodied Kowalski's procedural interpretation. Later, van Emden and Kowalski developed a formal semantics for the language of logic programs, showing that its operational, model-theoretic, and fixpoint semantics are the same.

In spite of all the theoretical work and the exciting ideas, the logic programming approach seemed unrealistic. At the time of its inception, researchers in the U.S. began to recognize the failure of the "next-generation AI languages," such as Micro-Planner and Conniver, which developed as a substitute for Lisp. The main claim against these languages was that they were hopelessly inefficient, and

very difficult to control. Given their bitter experience with logic-based high-level languages, it is no great surprise that U.S. AI scientists, when hearing about Prolog, thought that the Europeans are over-excited over what we, Americans, have already suggested, tried, and discovered not to work.

In that atmosphere the Prolog-10 compiler was almost an imaginary being. Developed in the mid to late 1970's by David H.D. Warren and his colleagues, this efficient implementation of Prolog dispelled all the myths about the impracticality of logic programming. That compiler, which is still one of the finest implementations of Prolog around, delivered on pure list-processing programs performance comparable to the best Lisp systems available at the time. Furthermore, the compiler itself was written almost entirely in Prolog, suggesting that fairly classical programming tasks, not just sophisticated AI applications, can benefit from the power of logic programming.

The impact of this implementation cannot be over-exaggerated. Without it, the accumulated experience that has lead to this book would not have existed.

In spite of the promise of the ideas, and the practicality of their implementation, most of the Western computer science and AI research community was ignorant, outwardly hostile, or, at best, indifferent to logic programming. By 1980, the number of researchers actively engaged in logic programming were only a few dozens in the U.S., and about one hundred around the world.

No doubt logic programming would have remained a fringe activity in computer science for quite a little longer were it not for the announcement of the Japanese Fifth Generation Project, which took place in October 1981. Although the research program the Japanese have presented was rather baggy, faithful to their tradition of achieving consensus at almost all cost, the important role of logic programming in the next generation of computer systems was presented loud and clear.

Since that time the Prolog language has undergone a rapid transit from adolescence to maturity. There are numerous commercially available Prolog implementations on most widespread computers. There is a large number of Prolog programming books, directed to different audiences and emphasizing different aspects of the language. And the language itself has more-or-less stabilized, having a *de facto* standard, the Edinburgh Prolog family.

The maturity of the language means that it is no longer a concept for scientists yet to shape and define, but rather a given object, with all its vices and virtues. It is time to recognize that, on the one hand, Prolog is falling short of the high goals of logic programming, but that, on the other hand, it is a powerful, productive, and practical programming formalism. Given the standard life cy-

cle of computer programming languages, the next few coming years will witness whether these properties will show their merit only in the classroom or will also be proven useful in the field, where people pay money to solve problems they care about.

So what are the current active subjects of research in logic programming and Prolog? The answer to this question can be found in the regular scientific journals and conferences of the field. The *Logic Programming Journal*, the *Journal of New Generation Computing*, the *International Conference on Logic Programming*, and the *IEEE Symposium on Logic Programming*, as well as in the general computer science journals and conferences.

Clearly, one of the dominant areas of interest is the relationship between logic programming, Prolog, and parallelism. The promise of parallel computers, combined with the parallelism that seems to be available in the logic programming model, have lead to numerous attempts, which are still ongoing, to execute Prolog in parallel, and to devise novel concurrent programming languages based on the logic programming computation model. This, however, is a subject for another book.

Contents